Adaptability of Model Transformations

Ivan Kurtev

Dissertation Committee:

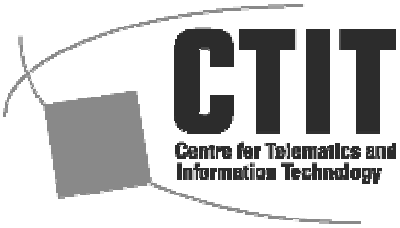
Prof. Dr. A. Bagchi (chairman, University of Twente)

Prof. Dr. Ir. M. Aksit (promotor, University of Twente)
Dr. Ir. K.G. van den Berg (assistant-promotor, University of Twente)

Prof. Dr. P.M.G. Apers (University of Twente)
Prof. Dr. C. Atkinson (University of Mannheim)
Prof. Dr. J. Bezivin (University of Nantes)
Drs. A.G. Kleppe (Klasse Objecten)
Dr. Ir. M.J. van Sinderen (University of Twente)

Ivan Kurtev

Adaptability of Model Transformations
PhD Thesis, University of Twente, 2005
ISBN 90-365-2184-X



IPA Dissertation Series 2005-08
CTIT PhD Thesis Series (ISSN 1381-3617) 05-71

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and within the context of the Centre for Telematics and Information Technology (CTIT)

Printed by Febodruk BV, Enschede, the Netherlands
Cover design by

ADAPTABILITY OF MODEL TRANSFORMATIONS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 19 mei 2005 om 16.45 uur

door

Ivan Kurtev Ivanov

geboren op 9 juni 1972
te Yambol, Bulgarije

Dit proefschrift is goedgekeurd door:

Prof. Dr. Ir. M. Aksit (promotor)
Dr. Ir. K.G. van den Berg (assistent-promotor)

Contents

Epilogue

Reflections

Holding at hand this book marks the end of the PhD period. That is why this initial part of the book is called epilogue. But it also marks a new beginning, which I believe, may be even more exciting.

For me, working on the PhD project means to be in doubt. In the first months you are trying to convince yourself that the vague ideas running in the mind are feasible and deserve some effort. Later on, you doubt that the rest of the world may be convinced as you are. Eventually, everything may finish with a happy end.

However, to be in doubt is not bad. It means that you are asking yourself questions and do not have immediate answers. Ideally, this should be the natural state of the human mind.

Acknowledgements

I would like to thank the people that were together with me during the last four years and that helped me performing my work.

I start by thanking my promotor Mehmet Aksit that gave me the opportunity to carry out this PhD project. My daily supervisor Klaas van den Berg provided me with a constant support in the research and teaching activities. These two people were patient, understanding, and tolerant enough to watch the development of a young researcher with all the ups and downs. From them I learnt a lot. Apart from the interesting research discussions that we had, it was a pleasure for me to discuss with them on many other issues.

My work with Mehmet and Klaas strengthen my conservative view on the learning process. No course and no educational system may be better than the face-to-face communication with a more experienced and knowledgeable person. Especially, in the moments when the major teaching tool in his hands is the red pencil.

I would like to thank the members of my PhD committee Peter Apers, Colin Atkinson, Jean Bezivin, Anneke Kleppe, and Marten van Sinderen that honored me and accepted to participate in the evaluation of my thesis.

My colleagues from the Software Engineering Group created a friendly and cheerful working atmosphere that I enjoyed a lot.

Special thanks go to Lodewijk Bergmans who participated in the weekly research meetings and gave a lot of useful feedback.

My office mate Maurice Glandrup helped me to handle many problems often caused by my ignorance in the Dutch language.

I thank my friends in Holland, in Bulgaria, and across Europe for the fun that we had together and the support that they gave me in difficult moments. Unfortunately, some of them are far away but the current communication technologies made our contacts easier. As a result, I became a valuable customer of T-Mobile Nederlands.

Many of the ideas in my thesis were born in the evenings and were inspired by some great human inventions. To name some (in a chronological order of my encounter with them): Belgium beers, wines from the Mediterranean countries, and the Highland pure malt (still unexplored enough).

I have to admit that I have been visited couple of times by the muse of Computer Science and she gave me a lot of motivation and inspiration. Yes, this muse exists, although she is not mentioned in the Greek mythology. For me, she is a concrete character and I think she is a kind of personal muse. I am looking forward seeing her soon.

Благодарен съм на родителите си за това, че ме възпитаха в дух на несъмнени ценности. Вярвам, че решението ми да се занимавам с научна работа до голяма степен е вдъхновено от тях. Благодаря на баща си за безрезервната и опрощаваща подкрепа, която ми даваше през последните близо 5 години.

Abstract

Model Driven Engineering (MDE) is an emerging approach for software development. This thesis focuses on one of the main research topics in MDE: transforming models. This topic may be studied from process and artifact perspectives. The artifact perspective involves transformation definitions and models. Transformation definitions written in transformation languages are software assets that may be considered as important as models of systems. Transformation definitions may be subject of design, implementation, reuse and evolution. In addition, they are affected by changes in their environment.

An important quality factor of software is *adaptability*, which indicates the capabilities of software to be modified for a changing environment and as a response to changes in software requirements. Adaptability of model transformations is required in several cases motivated by various changes that may occur.

This thesis addresses three problems related to the adaptability property of model transformations: identification and comparison of alternative transformations, definition of transformation languages capable of expressing transformations among models written in different languages, and language support for reusable and adaptable transformations.

We claim that the identification of alternative transformations should be included as a step in an MDE process. This is motivated by the observation that multiple ways are usually available to transform a given source model to a target model. The resulting target models may be functionally equivalent but different in their quality properties such as adaptability and performance. Software engineers must be able to identify transformations that produce models with the required quality properties. For this purpose a formal technique is defined for describing the space of alternative transformations for a given source model. The technique provides operations for reduction of and selection from transformation spaces on the basis of the desired quality properties of the resulting target model.

The thesis presents a hybrid transformation language named MISTRAL capable of defining transformations between models expressed in different modeling languages. The transformation language is separated from the instantiation and generalization mechanisms, which are represented in the modeling space in which the transformation language operates. Transformation definitions are specified on the basis of *intensions*. The concept of intension is a generalization of the concepts of meta-model and domain model expressed in a modeling language. The transformation language MISTRAL is capable of working with more than one *instanceOf* relation and more than one model level in a single transformation definition. This overcomes a major drawback in current transformation languages that are often coupled with particular modeling languages. A prototype shows the feasibility of this approach.

Transformation definitions should be reusable and adaptable artifacts. The thesis studies requirements for a transformation language to provide adequate support for reusability and adaptability of transformation definitions. An evaluation of a set of representative languages against requirements is given. A light-weight approach is proposed for extending transformation languages with new features.

The techniques proposed in the thesis are applied in a case study on XML (eXtensible Markup Language) processing based on model transformations. Compared to current techniques, this approach improves the extensibility of XML applications.

Samenvatting

Model Driven Engineering (MDE) is een nieuwe aanpak voor het ontwikkelen van software. Dit proefschrift is gericht op een van de belangrijke onderzoeksonderwerpen in MDE, namelijk het transformeren van modellen. Dit onderwerp kan worden bestudeerd vanuit het perspectief van het proces en vanuit het product. Bij het productperspectief gaat het om de transformatiedefinities en de modellen. Transformatiedefinities - geschreven in transformatietalen - zijn softwareproducten die zeker zo belangrijk zijn als de modellen van systemen. Transformatiedefinities zelf moeten worden ontworpen en geïmplementeerd, ze moeten hergebruikt kunnen worden en ook kunnen veranderen in de tijd. Bovendien worden deze definities mede bepaald door veranderingen in de context waarin ze gebruikt worden.

Een belangrijke kwaliteitsfactor voor software is de aanpasbaarheid. Deze factor geeft aan in welke mate software kan worden aangepast ten behoeve van veranderingen in de context en veranderde gebruikerseisen aan de software. Ook is de aanpasbaarheid van modeltransformaties in een aantal situaties vereist zoals blijkt in een aantal scenario's van mogelijke veranderingen.

Dit proefschrift behandelt drie problemen die samenhangen met de aanpasbaarheid van modeltransformaties: het identificeren en vergelijken van alternatieve transformaties, de definitie van transformatietalen waarin transformaties kunnen worden uitgedrukt voor modellen in verschillende talen en ten derde de ondersteuning in deze talen van herbruikbare en aanpasbare transformaties.

Wij stellen dat de identificatie van alternatieve transformaties een noodzakelijke stap is in een op MDE-gebaseerd softwareontwikkelproces. Dit is op grond van de observatie dat er gewoonlijk meerdere transformaties mogelijk zijn van een bronmodel naar een doelmodel. De resulterend doelmodellen kunnen weliswaar functioneel gelijkwaardig zijn maar zij kunnen verschillen in hun kwaliteitseigenschappen zoals aanpasbaarheid en prestaties. Softwareontwikkelaars dienen in staat te zijn transformaties te kiezen die leiden tot modellen met de gewenste kwaliteitseigenschappen. Om dit te bereiken wordt een formele techniek gedefinieerd voor het beschrijven van de ruimte met alternatieve transformaties voor een gegeven bronmodel. Deze techniek voorziet in bewerkingen van de ruimte, waarbij de ruimte verkleind kan worden en in deze ruimte transformaties gekozen kunnen worden die leiden tot doelmodellen met de gewenste kwaliteitseigenschappen.

Het proefschrift presenteert een hybride transformatietaal genaamd MISTRAL waarmee het mogelijk is transformaties te definiëren voor modellen die zijn uitgedrukt in verschillende modelleertalen. De taal is gescheiden van de mechanismen voor instantiatie en generalisatie die worden gerepresenteerd in de ruimte waarin de transformatietaal wordt gebruikt. De transformatiedefinities zijn gebaseerd op intenties. Het begrip intentie is een generalisatie van enerzijds het begrip metamodel en anderzijds het begrip domeinmodel zoals uitgedrukt in een modelleertaal.

Met deze transformatietaal MISTRAL is het mogelijk binnen een transformatiedefinitie te werken met meer dan één instantierelatie en meer dan één modelniveau. Hiermee wordt een belangrijk nadeel voorkomen van de gangbare transformatietalen die meestal zijn gekoppeld aan specifieke modelleringstalen. Een prototype toont aan dat deze benadering ook praktisch uitvoerbaar is.

Transformatiedefinities dienen herbruikbare en aanpasbare softwareproducten te zijn. Dit proefschrift beschrijft de eisen waaraan een transformatietaal dient te voldoen opdat herbruikbaarheid en aanpasbaarheid in voldoende mate worden ondersteund. Een aantal representatieve transformatietalen zijn op basis van deze eisen geëvalueerd. Een eenvoudige aanpak wordt voorgesteld waarmee transformatietalen kunnen worden uitgebreid met nieuwe eigenschappen.

De technieken die in dit proefschrift worden beschreven zijn toegepast in een casus over applicaties voor het verwerken van XML-documenten (eXtensible Markup Language) met behulp van modeltransformaties. Vergeleken met andere gangbare technieken levert deze nieuwe benadering een betere uitbreidbaarheid van de XML-applicaties.

1

Introduction

1.1 Introduction

Today's software systems are complex artifacts. This is mainly due to the inherent complexity of the domains in which software solutions are used. This makes the software development process a difficult task. Moreover, the complexity of the domains is only one factor contributing to the difficulties observed in software development processes. Other difficulties are related to methodological, organizational, economical, cultural and other issues.

Competition in the software industry leads to a steady stream of new technologies. During the last decade we witnessed the acceptance of several new programming languages (e.g. C++, Java, C#), middleware technologies (e.g. CORBA [73], Web Services [110]), and data representation standards (e.g. SGML and XML [103]). This constant shift from one technology to another seems unavoidable and puts an emphasis on adequate coping with technology changes in software development.

Current software development practices are in most cases code-centric relying on a set of implementation technologies. Changes in the technologies may require porting of software artifacts to the new technologies. It is difficult to maintain stable and reusable software assets due to constant technology changes.

To cope with this problem an approach to software development has been proposed by Object Management Group (OMG) named *Model Driven Architecture* (MDA) [72]. The core idea of MDA is to use modeling and models as main activity and artifacts in software development. Keeping software assets in the form of models makes these assets more resilient to changes in the implementation technologies. The notion of *Model Driven Engineering* (MDE) [52] builds upon the idea of MDA and adds to it the notion of software development process.

In an MDE software process a system is developed by refining models starting from higher and moving to lower levels of abstraction until system code is generated. This refinement is implemented as transformations over models.

MDE covers a wide spectrum of research areas some of them are already well established and some are newly emerged. Further efforts are required to bring them into a coherent approach based on open standards and supported by mature tools and techniques.

1.2 Problem Statement

In this thesis we focus on one of the activities in MDE: transforming models. It has a process dimension being an effort happening in time, and an artifact dimension since it uses software artifacts such as transformation definitions and models.

OMG issued a Request for Proposals that addresses the need for a standard language for model transformations [76]. Transformation definitions written in transformation languages are software assets that may be considered as important as models of systems. Transformation definitions may be subject of design, implementation, reuse and evolution. Moreover, they are affected by the changes in the environment they “live” in.

An important quality of software is adaptability. This quality property indicates the capabilities of the software to be modified for a different environment and as a response to changes in software requirements. Adaptability of model transformations is required in several cases motivated by various changes that can occur.

A model transformation is a process of converting one model to another model. A model may be transformed to multiple models that are functionally equivalent but still differing in the quality properties they possess. For example, one model may be more extensible while another model may be optimized for time performance. Software engineers must be able to identify transformations that produce models with the desired quality properties. If the software engineer would like to compare functionally equivalent alternative models from various quality perspectives then alternative transformations must be identified. Generation and evaluation of alternative transformations must be explicitly addressed in an MDE software development process.

Furthermore, models are expressed in various modeling languages. MDE is proposed as a solution that copes with changes in technologies. We may assume that there will be no perfect modeling language fulfilling all expectations in a timeless manner. Therefore, model transformation languages should be able to express transformation definitions on models expressed in various modeling languages. To effectively apply MDE, a number of questions must be answered, however. What are the characteristics of modeling languages that affect model transformation languages? How can transformation languages be separated from and be parameterized with respect to variations in these characteristics of modeling languages?

It is expected that transformation definitions will be subject of reuse, adaptation and composition in the same manner as software artifacts such as classes and libraries are reused, adapted and composed. Representing a transformation definition as composition of simpler units may help in performing these tasks. Explicit representation of such composition provides finer control over the components affected by changes and therefore improves adaptability of transformations. Therefore, it is important to study factors that drive the decomposition of transformation definitions. In addition, transformation languages

should provide adequate constructs for expressing decomposed entities and proper compositional operators to integrate them into a single definition. Finally, since languages need to compromise many quality issues depending on the application context, they can be extended with additional constructs for modularity and composition.

1.3 Approach

To cope with the problem of identification and selection of alternative transformations we study reasons for the emergence of such alternatives. We propose an explicit activity of alternative space analysis in an MDE software development process. A formal technique is proposed to explicitly model a set of alternative transformations called *transformation space*. This technique allows specification of certain quality properties of the models. The quality properties are used as selection criteria among the alternatives.

To make a transformation language capable of working with models expressed in different modeling languages we study common operations in current transformation languages and how they interact with modeling language features. Three language features are crucial in that interaction: instantiation mechanism, generalization relation and the type system of a language. We propose a hybrid transformation language that abstracts from these features.

To identify ways for decomposition of transformation definitions we study how a decomposition is affected by decomposition of the models involved in a transformation definition. Furthermore, we study how changes in models affect transformation definitions for these models. We are interested in identifying language constructs able to express required decompositions and to handle required changes. This is done by studying a set of scenarios for model transformations found in research and practice. For every scenario we derive a set of requirements that a transformation language must meet. These requirements are used for evaluation of a set of current transformation languages. A "lightweight" approach for extending a transformation language with new features is proposed.

Finally, to demonstrate the applicability of the techniques proposed in the thesis we perform a non-trivial case study of model transformation based processing of XML documents.

1.4 Contributions

This thesis provides the following contributions:

1. *A formal technique for definition of transformation spaces that supports analysis of alternative transformations for given source models*

Chapter 3 motivates the need for inclusion of an activity in an MDE software development process that considers identification and comparison of alternative transformations for a given model. A formal technique is defined for modeling the space of alternative transformations. The technique provides operations for reduction of and selection from a transformation space on the base of desired quality properties of the resulting model. This

technique provides the software engineer with a means for explicit description and reasoning about alternative transformations so that the software engineer is able to identify the transformations that produce a result with certain quality properties. Chapter 3 shows a transformation space that depicts alternative models based on their extensibility properties.

2. *A hybrid transformation language capable of expressing transformation definitions between models expressed in different modeling languages*

Chapter 4 presents a hybrid transformation language capable of expressing transformation definitions between source and target models created in different modeling languages following a variety of transformation scenarios. The basic characteristic of the language is the separation of some transformation operations from the instantiation mechanism defined for languages used to express models. The instantiation mechanism for a modeling language is represented in the environment in which the transformation language operates and may be integrated with the transformation engine. The transformation language is capable of working with more than one *instanceOf* relation and more than one model level in a single transformation definition. This overcomes a major drawback in current transformation languages that are often dependent on particular modeling language.

3. *A set of requirements for transformation language constructs that allows definition of reusable and adaptable transformation definitions*

Chapter 5 studies two quality properties of transformation definitions: reusability and adaptability. It shows scenarios that illustrate situations where reusability and adaptability of transformation definitions are required. Chapter 5 contributes to the formulation of requirements that a transformation language must fulfill to provide an adequate support for reusability and adaptability of model transformations. An evaluation of some representative languages against these requirements is given. Finally, the chapter outlines a “lightweight” approach for extending a transformation language with new constructs.

4. *A model transformation based approach for XML processing*

Chapter 6 presents an approach for processing XML documents. Current techniques for XML processing are considered in the broader context of Model Driven Engineering. We assume that the syntax of an XML language is defined in a schema and is interpreted via an application-specific model. Interpretation is driven by a transformation definition from the language schema to the application model. This approach overcomes problems experienced in today’s XML applications, such as lack of proper extensibility mechanisms in case of syntax changes.

1.5 Outline of the Thesis

The map of the thesis with chapters and relations among them is shown in Figure 1.1.

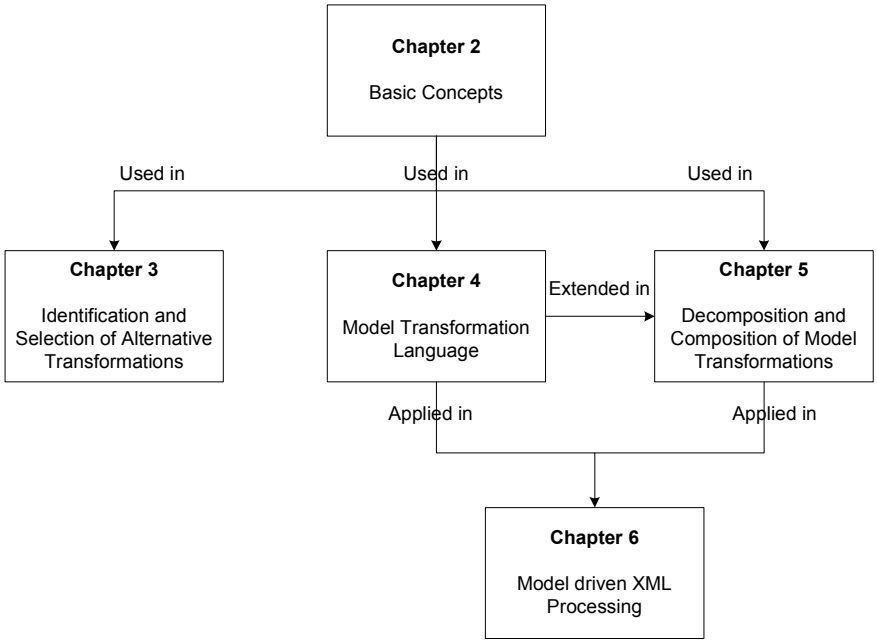


Figure 1.1 Thesis map

This thesis consists of the following chapters:

Chapter 2 describes the concepts used in the thesis. It introduces the notion of Model Driven Architecture and Model Driven Engineering as they are described in literature. Furthermore, the concepts of model, meta-model and model transformation are discussed in detail in the context of MDE. The chapter gives a global overview on software development as a problem solving process and interprets a set of common problems in terms of MDE concepts. These problems are further treated in chapters 3, 4, and 5.

Chapter 3 describes the activity of transformation space analysis, which is based on a formal technique that models a set of alternative transformations for a source model. The approach described in the chapter is applied in an example for managing transformations of domain models in UML into XML-schema's. This chapter is based on work published in [56] and [57].

Chapter 4 proposes a modeling space for organizing models and modeling languages. Models are based on a uniform representation of model elements no matter at which model level they are defined. In the modeling space different instantiation and generalization mechanisms are represented as a set of functions. The chapter presents a transformation language that operates in the proposed modeling space. The language is separated from instantiation and generalization mechanisms specific for a given modeling language. This chapter is an enhancement of results published in [59] and [60].

Chapter 5 discusses problems related to composition and decomposition of transformation definitions. The ability to decompose a transformation definition into simpler units improves adaptability and reusability of the definitions. A number of requirements for language constructs that support composition and decomposition of transformation defini-

tions are derived and a set of languages are evaluated against these requirements. This chapter also describes an extension of the language presented in Chapter 4.

Chapter 6 demonstrates the applicability of the techniques described in Chapter 4 and Chapter 5 in the context of XML processing. The chapter provides an evaluation of current techniques for XML processing and identifies a set of problems. It proposes an approach for XML processing based on declarative specification and execution of model transformations from XML language syntactical structures to application structures. This chapter is based on work published in [58] and [61].

Chapter 7 gives conclusions and an evaluation of the contributions in this thesis, and describes directions for future work.

2

Basic Concepts

2.1 Introduction

This chapter gives an overview of basic concepts used in this thesis. Various definitions of these concepts may be found in literature. We aim at selecting a consistent set of definitions that support the understanding of the subsequent chapters.

This chapter introduces the notions of *Model Driven Architecture (MDA)* and *Model Driven Engineering (MDE)* as they are described in literature. Furthermore, the concepts of *model*, *meta-model* and *model transformation* are discussed in detail in the context of MDE. The chapter gives a global overview of software development processes and interprets a set of common problems in terms of the MDE concepts.

The structure of the chapter is as follows. Section 2.2 describes the notion of Model Driven Engineering as an enhancement of Model Driven Architecture. Section 2.3 discusses definitions of *model* and *modeling activity*. Section 2.4 is devoted to the concept of *meta-model*. Section 2.5 presents an organization of models across levels that form a meta-modeling architecture. Section 2.6 defines the concept of *model transformation* in the context of MDE. Section 2.7 provides a description of scenarios for model driven development. Scenarios are based on a problem solving and evolution perspective on software development and demonstrate the presence of alternatives, composition/decomposition, reuse, and evolution in MDE. Section 2.8 discusses the need for adaptability property of model transformations. Finally, Section 2.9 gives conclusions.

2.2 Model Driven Engineering

The concept of Model Driven Engineering (MDE) emerged as a generalization of the Model Driven Architecture (MDA) approach for software development. In this section we first introduce the basic concepts in MDA and then explain MDE as it was initially introduced by Kent [52].

2.2.1 Model Driven Architecture

MDA is a framework for software development adopted by Object Management Group (OMG) in 2001 and is described in a series of OMG documents. The document MDA Guide [72] provides an overview and definitions of concepts used in MDA.

The goal of MDA is to provide a solution to the problem of continual emergence of software technologies that forces companies to port their software systems every time a new 'hot' technology appears. As an example we can consider the evolution of middleware technologies. CORBA [73] is a widely accepted standard for middleware and many companies used it to implement their systems. We observe that Web Services [110] partially replaces CORBA as middleware technology. Many existing systems may require porting to this new technology although their functionality may remain the same. When such a new technology appears it will cause the same problem of porting of existing systems. The constant changes in the technologies create a problem with *portability* which may require significant efforts. MDA aims at solving the portability problem.

Other problems that are expected to be solved by MDA are described by Kleppe et al. [53]. The *productivity* problem in current software development practices is caused by the fact that software development processes are driven by low-level design and coding. The maintenance and understanding of code is a difficult and error-prone process in case of large software systems.

The *interoperability* problem is introduced by the fact that large systems are not monolithic but modular. Different modules are built upon technologies suitable for the problem at hand. Therefore, software systems consist of components implemented in various technologies that need to interoperate.

To cope with the portability problem MDA combines two principles that have been used in computer science and other engineering disciplines. The first principle is to use modeling and models to develop software systems. This principle is well-established in engineering disciplines. For example, a building in civil engineering is first specified in a set of diagrams, calculations are performed based on the properties of the materials and finally the actual building process can start. Sometimes scale models of the building may be created. There is a clear separation between the phase of modeling and study of models, and the phase of realizing the models into a building.

The second principle is the separation of the specification of a system from the details how the system is implemented via concrete technologies. The abstract specification of the system becomes the main asset in software development. Many implementations using concrete technologies may be derived from the abstract specification. Two goals are achieved in that way: portability and reusability.

OMG defines MDA as an approach to system development based on models. It is said to be *model-driven* because it provides means for using models to direct the course of understanding, design, construction, deployment, maintenance and modification [72].

MDA approach classifies models into two classes: Platform Independent Models (PIMs) and Platform Specific Models (PSMs). These classes contain models at different levels of abstraction. The MDA guide [72] gives the following definition of PIM: "Platform Independent Model is a view of a system from a platform-independent viewpoint". PSM is defined as: "Platform Specific Model is a view of a system from platform-specific viewpoint". These definitions rely on the concept of *platform*. The MDA guide defines platform as follows: "A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented".

The development of a system according to the MDA approach starts by building a PIM of that system. Then the PIM is transformed to one or more PSMs. PSMs use constructs provided by the chosen platforms. Finally, the PSMs are transformed to code.

The basic operation applied on models in MDA-based software development is *model transformation*. Model transformation is the process of converting one model to another model of the same system. MDA aims at automating model transformations as much as possible. Transformations are executed by tools based on specifications of transformations.

Figure 2.1 shows the MDA transformation pattern as defined in the MDA guide. The transformation process accepts a PIM as an input and generates a PSM as an output. The pattern is generic since it does not specify how exactly the transformation is specified. Usually additional information about the platform is required to execute the transformation.

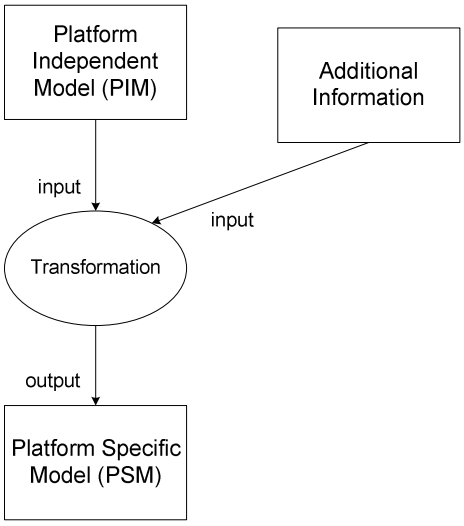


Figure 2.1 The MDA transformation pattern

MDA also aims at integrating the existing technologies standardized by OMG. Models are expressed in Unified Modeling Language (UML) [77] and UML profiles. If another modeling language is used it should be defined using the meta-modeling language of the Meta Object Facility (MOF) [75]. MDA uses XML Metadata Interchange (XMI) [79] for serialization of MOF models in XML format. OMG is in the process of standardizing a

language for specification of transformations over MOF models as a response to the Query/Views/Transformations Request for Proposals [76].

Some of the concepts in MDA are vaguely defined and lack solid scientific foundation. Bezivin [19] emphasizes the need for a deeper understanding and agreement on the meaning of concepts such as *system*, *model* and *meta-model* and the relations among them. Atkinson and Kuhne [12] analyze the modeling framework provided by OMG and give requirements that are not initially posed. We discuss these concepts in the subsequent sections.

2.2.2 Model Driven Engineering

MDA introduces a set of basic concepts such as *model*, *meta-model*, *modeling language* and *transformation* and proposes a classification of models as PIMs and PSMs. MDA, however, lacks the notion of software development process. Model Driven Engineering (MDE) is an enhancement of MDA in this direction.

Kent [52] defines MDE on the base of MDA by adding the notion of software development process and modeling space for organizing models. The modeling space is organized over several dimensions. Kent also defines requirements for tools needed to perform operations on models in MDE.

MDA defines only one classification dimension for models based on the dichotomy of PIM and PSM. This is a dimension that indicates the level of model abstraction. PIMs can be considered at a higher level of abstraction than PSMs. Therefore, one dimension in the modeling space reflects the degree of *Abstractness* or *Concreteness* of models.

Another dimension comes from the distinction of models on the base of the subject area they belong to. Different users of a system may have different views over the system focusing on a subset of the system features. These views are reflected in different models of the same system. Subject areas generally correspond to concerns as defined in Aspect Oriented Software Development [39]. Concerns of interest are enumerated in another dimension in the modeling space. This dimension is nominal. No order is assumed among the concerns. Examples of concerns are concurrency control, security, distribution, and error handling.

A third dimension is concerned with organizational issues such as versioning and authorship over models. The number of dimensions is not limited and depends on the specific needs in a development project.

Apparently, the proposed modeling space is capable of locating a model on the base of more criteria than the simple PIM/PSM dichotomy. Thus it may be considered as more suitable for software development.

Two views on possible MDE processes are presented in [17] and [7] where required concepts, methods and tools are defined. Favre [37][38] proposes a vision on MDE where MDA is just one possible instance of MDE implemented in the set of technologies defined by OMG (MOF, UML, XMI). The concepts of model, meta-model and transformations are found in other technologies as well [55]. According to Favre, their meaning should be analyzed in a broader perspective and MDE should be able to accommodate various technological domains in a uniform way. Clearly, such a vision departs from the set of OMG standards and makes a step toward a more open approach.

Sections 2.2.1 and 2.2.2 presented an overview of Model Driven Architecture and Model Driven Engineering as approaches for software development. They rely on a set of

common concepts such as model, system, modeling language, meta-model, and transformation. The remaining part of this chapter discusses these concepts.

2.3 The Notion of Model in Software Engineering

In this section we analyze the concept of model and modeling activity from various perspectives. After a short description of the concept of model in general, we focus on definitions used in computer science and, in particular, in software engineering. We propose our own definition of model that will be used throughout the thesis.

2.3.1 Definitions of Model and Modeling

We start with definitions found in the literature that consider models and modeling in general. Furthermore, we give definitions specialized for concrete scientific areas.

The origin of the word *model* can be traced to the Latin *modulus*, which means a small measure. The Merriam-Webster online dictionary gives 13 meanings of *model*. The first few of them are: a miniature representation of something; an example for imitation and emulation; a description or analogy used to help visualize something (as an atom) that cannot be directly observed.

The next two definitions are given in the context of philosophy of science. Apostel [11] defines model as: “any subject using a system A that is neither directly nor indirectly interacting with a system B to obtain information about the system B, is using A as a *model* for B”. Van Gigch [44] gives a description of the modeling activity: “the object of study in *modeling* is the real world, where the observer-researcher attempts to detect patterns of recurring relationships which can be represented systematically. To *model* is to represent this pattern or relationships in a manner which can lead itself to a formal study”.

In the FRISCO report, a framework for information systems [34], the following definition is given: “a *model* is a purposely abstracted, clear, precise and unambiguous conception”. This definition stresses on the conceptual nature of models. To be materialized and shared among people the models are denoted by using a language. The report continues with the definition “a *model denotation* is a precise and unambiguous representation of a *model*, in some appropriate formal or semi-formal *language*”.

In other definitions the model denotation is regarded as the *model*. Starfield et al. [96] give the following definition: “a *model* is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details”.

Different sciences may have a specialized view over the concept of model. In many engineering disciplines mathematical models are employed. These models are expressed in a mathematical language. The following definition of *mathematical model* and *modeling* is given in [66]: “mathematical modeling is the description of an experimentally verifiable phenomenon by means of the mathematical language. The phenomenon to be described is called *the system*, and the mathematics used, together with its interpretation in the context of the system is called *the mathematical model*”.

Disciplines such as mechanical and civil engineering and architecture often use scale models that are material objects copies of the real world objects (e.g. vehicles, buildings,

bridges, etc.). Furthermore, these objects may be specified in diagrams used to drive the production of their details. All these examples are models expressed in a certain way.

Computer science employs models in different phases of software development. In information systems development an approach based on conceptual modeling may be used. According to Boman et al. [21] a model is "a simple and familiar structure or mechanism that can be used to interpret some part of reality".

MDA and MDE rely on modeling and model as basic concepts. However, there is no commonly agreed definition of model yet. Various definitions have been given in literature. Recent publications indicate a need for a clear and precise understanding of modeling concepts in order to put MDE on a sound base. For this purpose several researchers working in the area of Model Driven Engineering have introduced the following definitions.

Seidewitz [93] defines *model* as: "a set of statements about a system under study". The MDA guide [72] defines: "a model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language". Kleppe et al. [53] define *model* as follows: "A model is a description of a system written in a well-defined language". Bezivin and Gerbe [16] define: "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".

Next section analyses definitions and makes assumptions suitable to the context of this thesis.

2.3.2 Analysis of Definitions

Analysis of the definitions of what model and modeling are reveals a set of commonalities and a set of variations among them.

Model as a Relative Concept

The most obvious commonality in the definitions is that they define model as a relative concept. Model always requires a model's domain, that is, the part of the real world being modeled. Therefore, an entity should not be regarded as a model, if its domain can not be identified. Of course, in some cases the domain of the model may be neglected and not considered. Also, the notion of model should be understood as a role. An entity can be a model with respect to its domain and the same entity can by itself be a domain of another model.

The domain of the model is called in various ways in different definitions: a system, a phenomenon, an object system, a Universe of Discourse (UoD), a target system, a system under study (SUS), etc. We adopt the term *'object system'* or just *'system'* to refer to that part of the reality for which a model is built.

Models are Abstractions

Another important characteristic of models is that they are abstractions of their object systems. This means that some characteristics of the object system have been represented in the model and others have been discarded. In that way, the model is a simplification of the reality, used instead of this reality in the process of studying it.

Furthermore, a model should serve a given purpose. This is explicitly mentioned in some of the previously given definitions. Having a purpose is crucial for the process of abstracting from the reality. Reality exposes potentially infinite number of features. The purpose of the model is the guiding principle in the selection from these features.

Van Gigch [44] describes the dilemma between generality and specificity in models predetermined by the abstract nature of models. A general model contains fewer details than a specific one. A general model describes a larger set of phenomenon while a specific model is scoped on a smaller set of phenomenon and in general provides more information.

Relations between a Model and its Object System

The importance of the relation between a model and its object system is emphasized in several definitions. This relation is still a source of discussion and its nature is not well-understood yet in the more general philosophical context. A model is used for indirect study of the reality (the object system). This indirectness may be caused by different obstacles. The object system may be inaccessible or its direct study is too expensive or even the object system may not exist yet and the model plays the role of a specification of the object system. Regardless of the obstacles, the model must be a valid representation of the object system. The knowledge acquired from the model must hold for the object system. Often, this knowledge is not exact but it is an approximation of the reality satisfying an acceptable degree of inaccuracy.

This very important aspect of the relation between a model and the object system is explicitly addressed in the definitions given by Apostel [11], Bezivin and Gerbe [16] and Boman et al. [21].

Furthermore, the knowledge acquired from the model is initially expressed in terms of model elements. This knowledge must be interpreted and converted to knowledge in terms of the object system. For example, in a mathematical model, the variables are abstracted from their real life meaning and the obtained values must be interpreted in terms of the object system. Depending on the phenomenon under study, the variable values may correspond to a temperature, velocity, time, etc.

The relation between a model and an object system is bi-directional and two separate relations may be considered as Figure 2.2 shows. This figure is called the *DDI account* (DDI – Denotation, Demonstration, Interpretation) introduced by Hughes [47].

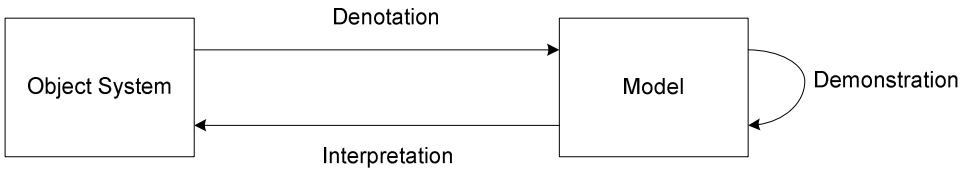


Figure 2.2 Relationships between an object system and model [47]

The object system is *denoted* (represented) in a model. This denotation must preserve some characteristics of the object system to allow acquiring knowledge about it through the model. The model is used to obtain claims about the model elements. This process is known as *demonstration*. It happens only in the context of the model. Finally, the obtained results are mapped to the object system. This mapping is called *interpretation*. The knowledge obtained from the model must be verifiable against the object system. If the

results obtained from the model do not meet the empirical evidence obtained from the reality then the model is invalid with respect to the object system.

Literature sources usually depict only one relation between a model and its object system. Various names for the relation are used: *ModelOf, RepresentationOf, RepresentedIn, ModeledBy,* etc. In the remaining part of the thesis we will use a relation directed from a model to its object system named *ModelOf*. The meaning of this relation is illustrated by the DDI account. As we saw the meaning has two aspects: representation (denotation) which captures some of the characteristics of the object system in its model, and interpretation which indicates the possibility to trace the knowledge obtained from the model back to the object system. From that point of view *ModelOf* relation accumulates two other relations: *Denotation* and *Interpretation*.

Models as Systems

Apart from similarities the definitions also demonstrate differences. The most important variation point in the definitions is what kind of entity a model is.

In many definitions a model is regarded as a system. Therefore models can be classified according to some classification of systems as *conceptual*, *concrete* (or physical) and *symbolic*. Many definitions are general enough to allow the three types of models. Other definitions focus on a particular type of model.

The definition in the FRISCO report [34] explicitly defines model as a conceptual entity and its model denotation as a symbolic entity. In contrary, the definition in [96] stresses that models are representations of concepts. The definition of mathematical model from [66] implies that model is a symbolic entity expressed in the language of mathematics.

In general, a symbolic model implies also the presence of a conceptual counterpart in the modeler's mind and an entity in the real world that the symbol refers to. This a consequence of the fact that symbolic models can be viewed as signs. Therefore, they are subjects of semiotics. A central role in semiotics plays the meaning triangle of Ogden and Richards. This triangle gives the relation between a symbol, the object denoted by that symbol and the conception required to connect the symbol and the object. The meaning triangle is shown in Figure 2.3.

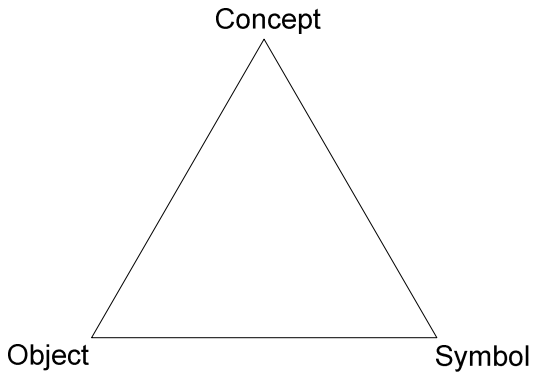


Figure 2.3 The meaning triangle

Computer science is largely concerned with manipulation of symbolic systems. Scale models that are tangible entities with dimensions in space are not often used in computer

science. Also, a conceptual system that resides only in a human mind is not useful because that system is not represented and shared among people. Representation of a conceptual system means that it is transformed to a physical or symbolic system. Therefore, we may assume that models we deal with in computer science are symbolic systems. Usually, a symbolic system is expressed in a language. In case of a model we say that the language used to express the model is a *modeling language*. The notion of modeling language is very important in MDE. We will see in the next parts of this chapter that it plays a central role in the definition of another important concept, namely the *meta-model*.

2.3.3 Choosing a Definition

On the base of the discussion in section 2.3.1 and section 2.3.2 we choose for this thesis the following definition of model:

> *A model represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system.*

This definition is a combination of some of the definitions given so far and makes the assumption that models are symbolic systems expressed in a language.

2.3.4 Modeling in Computer Science

In this section we address two issues: the role of a model that a software system can play with respect to other systems and the role of modeling in software development.

Software Systems as Models

A running software system may contain a model of a certain part of reality. Consider an information system that contains a database with information about students in a university and the related software that operates on that database. Can we consider the system as a model? If the answer is yes then what is the object system of that model? Further questions are easily derived from Figure 2.2: how the object system is represented in the model and how the knowledge the model provides is related to the object system.

We can consider the information system as a model. The object system of this model contains the real people that are current (and possibly former) students of the university. Some information about these people is represented in the information system reflecting their relation to the university. Furthermore, the information system is used to obtain knowledge about its object system. Instead of asking each student personally about its academic state we can obtain this information from the system. Apparently this satisfies the general definitions of model given in the previous section.

The obtained information is of a symbolic nature but the users of the information system are aware of its meaning. They can *interpret* the information in terms of real students, concrete university faculties, and real events that have occurred such as exams, graduations, etc.

This is one of the obvious examples when a software system is a model of a part of reality. Other examples are software systems that perform process simulations, and systems for computer aided-design.

Modeling Software Systems

Software systems may be object systems of models. Indeed, in different areas of computer science modeling of software systems play an important role.

We assumed in the previous discussion that models are symbolic representations of object systems. Representations may be informal such as a drawing on a whiteboard or a sketch or may be based on a formal modeling language. In computer science the latter case is of primary interest since models are supposed to be defined in a precise and unambiguous manner.

One such a formal modeling language is the language employed in mathematics. A mathematical model uses mathematical notation and is usually comprised of variables, parameters and a set of equations. This formal representation of a model allows the application of the available mathematical knowledge which ultimately produces additional knowledge from the model.

In a similar way in computer science we are interested in expressing models in precise modeling languages. Such models may be manipulated by automated and semi-automated tools ultimately resulting in complete software systems.

2.4 Meta-model and Meta-modeling

In this section we analyze the concepts of *meta-model* and *meta-modeling* in the context of computer science. Similarly to section 2.3 we choose a definition that will be used in the thesis. We analyze the relation among models and meta-models. In section 2.4.2 we present a more general relation defined among *intensional* and *extensional* entities. The relation between a model and a meta-model is interpreted in the broader context of intensional and extensional entities.

Sometimes a model is considered as an instance of a meta-model. Section 2.4.3 presents a comparison between *ModelOf* relation and *instanceOf* relation in the context of meta-modeling. These relations often coincide between given model and a meta-model and tend to be considered as identical.

The concepts of *meta-model*, *intensional* and *extensional* entities, and *instanceOf* relation are used further in section 2.5 where the notions of model levels and meta-modeling architecture are introduced.

2.4.1 Definitions of Meta-model and Meta-modeling

As the name suggests meta-modeling is a modeling activity. Similarly, the product of the meta-modeling, called *meta-model*, is a model.

If an entity is a model we have to be able to clearly identify its object system. In literature two possibilities are found for the object system of a meta-model: a modeling process and a modeling language.

Meta-modeling as Modeling of Processes

Van Gigch defines meta-modeling as modeling of modeling processes [44]. In other words, meta-modeling studies how we model. Hence, a meta-model is a model of the modeling process. In a similar line Brinkkemper [25] defines meta-modeling as the process of conceptualization of a modeling technique. A meta-model is a conceptual model of a modeling technique.

Meta-modeling as Modeling of Languages

The second possibility for the object system of a meta-model is a modeling language. In this view on meta-modeling the relation to a given modeling language is essential. This view is adopted in MDE. We give some of the definitions found in the related literature.

Seidewitz [93] defines: "a meta-model is a model of models expressed in a given modeling language". Therefore, a meta-model describes what can be expressed by models in that language. The FRISCO report defines: "meta-model is a model of the conceptual foundation of a language, consisting of a set of basic concepts, and a set of rules determining the set of possible models denotable in that language" [34].

The MDA guide defines meta-model as "model of models". Apparently this definition misses the relation to a modeling language that expresses the models under study.

It seems that the two approaches to meta-modeling use different object systems. In certain cases the first approach to meta-modeling may imply the second one. This depends on the understanding of the terms modeling process and modeling technique. If the modeling process explicitly includes the products (models) and modeling languages then the first approach is more general.

As stated before our primary research focus is on models expressed in a given language. The process of producing these models (the modeling activity) is not in the scope of this thesis. Therefore we adopt the second possibility for the object system of a meta-model: modeling language. In this thesis we accept the following definition of the term meta-model:

Meta-model is a model of modeling language.

2.4.2 Intensional and Extensional Entities

The concepts of *intensional* and *extensional* meaning are well known in linguistics and logic. The concepts of intension and extension come from the study of the semantics of languages. A word in a language may have two meanings. The extensional meaning is a set of all the entities in the real world that the word refers to. This set is called *extension* or *denotation*. For example, the extensional meaning of the word *student* is the set of all the students in the universe of discourse (for example, the University of Twente). The intensional meaning gives a description of the properties of the word, and how it is related to other words. Intensional meaning (or just *intension*) is also known as *connotation*. For example, the intensional meaning of *student* can be *'someone who studies at a university'*. Note that the intensional meaning does not mention entities in the universe of discourse. There is a relation between the extensional and the intensional meaning of a word. The entities in the extension must possess the properties defined in the intension of the word.

The distinction intensional/extensional can be applied on the elements of a given model. Consider a model of a domain where the model contains representations of entities from the domain and also representations of the categories found in the domain. One example of such a model is a database used in an information system. Assume that the domain under study covers the students at a given university. The database is a model that represents students studying at the university. Along with the representation of individual students the information system also contains *database schema* which is a set of relations. The schema defines the structure of the information kept for the individuals. The database schema represents the categories (known also as *classes* or *types*) found in the domain. In our example it would define a relation for student's personal information. A category does not represent a concrete individual from the domain. Instead, it is a set of individuals sharing common properties. The category is derived from a certain conceptualization of the domain. The extension of the category *Student* is a set of students. The intension is a description of the common properties of students.

Individual students and the class *Student* have their representations in the model. The part of the model that represents individuals is often referred to as *extensional* part. The part of the model that contains the representation of the domain categories is regarded as *intensional* part. In some literature sources, mainly coming from the area of conceptual modeling and information systems both parts are considered belonging to a single model known as *base model* [21][34]. Intensional part is called *schema* and extensional part is called *information base*.

The distinction between intensional and extensional parts is related to the adopted modeling language. In programming languages the intensional and extensional parts are separated. The intensional part is usually a program written in a language and contains definitions of classes. The extensional part is a run-time state of the execution of that program. In class-based object-oriented languages the extensional part cannot exist without an intensional part. No object can be created if there is no class for it.

In contrast, some languages allow the existence of an extension without intensional part. Examples are the XML language [103] and Resource Description Framework (RDF) [14]. In XML, a document may exist without a schema that constrains the structure of the document. More than one schema may be defined for a document and multiple schema languages may be used for this purpose. Similarly, RDF allows description of type-less data. Optionally, RDF data may be typed by an RDF schema.

This overview shows that the separation of intensional and extensional parts into separate models is language dependent. If both are present there is a *conformance* relation between these two models. The extensional part satisfies the constraints defined in the intensional part. The definition of the conformance relation is according to the semantics of the language. Furthermore, the intensional part can be considered as a model for the extensional part: it defines the common properties of the elements in the extensional part.

In this thesis we use the term *intension* to denote a model that models other models. An intension is associated to exactly one *extension* which is a set of models modeled by the intension. The following two definitions are assumed:

Intension is a model that models other models.

Extension is a set of models modeled by an intension.

A meta-model of a language is an intension. The set of models expressible in the language forms the extension. However, not all intensions are meta-models. There exist models that are intensions and do not model a modeling language.

2.4.3 Meta-models, Models, and *InstanceOf* Relation

In general, there is a *ModelOf* relation between a meta-model and its object system: a modeling language. In many cases, the *ModelOf* relation is not considered and is replaced by *instanceOf* relation between a model and a meta-model. In this section we analyze the nature of these relations in the context of meta-modeling. Although they coincide between the same pair of entities (a model and a meta-model) they are different in nature. *instanceOf* relation may help in the interpretation of the knowledge obtained from a meta-model (see Figure 2.2). We adopt the view that *instanceOf* relation is defined in the context of a given language that defines the semantics of the relation. We make the conclusion that *instanceOf* relation is not always present between a model and a meta-model.

In this section we first analyze the *ModelOf* relation among models and meta-models. Then we show examples of *instanceOf* relation among models and meta-models. The two relations are compared on the base of the examples.

The *ModelOf* Relation between a Model and a Meta-model

In section 2.4.1 we have adopted the definition that meta-model is a model of a modeling language. In the general theory of languages used in computer science, a language is a set of sentences over a set of symbols (an alphabet). These symbols may be letters as in natural languages but may also be icons or other diagrammatic symbols found in visual languages.

Languages of practical interest usually have infinite sets of sentences. It is impossible to generate all sentences in such a language. In computer science we use set of rules that can be used to check if a sentence belongs to a language. One example of such a set of rules is the grammar of a language specified in Extended Backus-Naur Form (EBNF). We consider the language as defined by its grammar. The grammar rules capture characteristics that all sentences in the language have. Therefore, we can regard the grammar of a language as a model of that language. In case of a modeling language any model of the language is a *meta-model*. A given modeling language may have more than one meta-model.

Strictly speaking the object system of a meta-model is the set of models expressed in a modeling language. However, in practice we are interested in obtaining knowledge about the members of the set instead about the set as a whole. A meta-model of the language defines constraints that every model in that language must satisfy. Therefore, our focus is on the relationship between concrete models written in the language and the meta-model. This relationship is called *conformantTo* in [19]. Detailed treatment of the nature of this relation is given by Favre [36]. In this article, *conformantTo* is defined as a composition of two relations: *elementOf* denoting the membership of a model to a language and *representationOf* denoting the relation between an object system and its model.

For simplicity we focus on the relation between members of a language and its meta-model. The *ModelOf* relation will be depicted between a meta-model and a model instead between a meta-model and the set of models (the language).

The *instanceOf* Relation between a Model and a Meta-model

The *conformantTo* relation explained above is often replaced by *instanceOf* relation. Many OMG documents and many articles on the subject state that a model is an instance of a meta-model. The following discussion elaborates on that issue.

Consider Figure 2.4 which shows a Java program as text and its grammar specified in EBNF notation. The grammar is a meta-model of the Java language. This is indicated by the solid arrow from the grammar to the program labeled 'ModelOf'. It must be possible to interpret the knowledge obtained from the grammar in terms of the textual form of the Java program. For instance, there may be a rule that contains a non-terminal corresponding to the syntactical category for Java method definition. This non-terminal should be traced to phrases in the Java program that represent method definitions. How is this interpretation supported? Here the notion of *instanceOf* relation comes in usage.

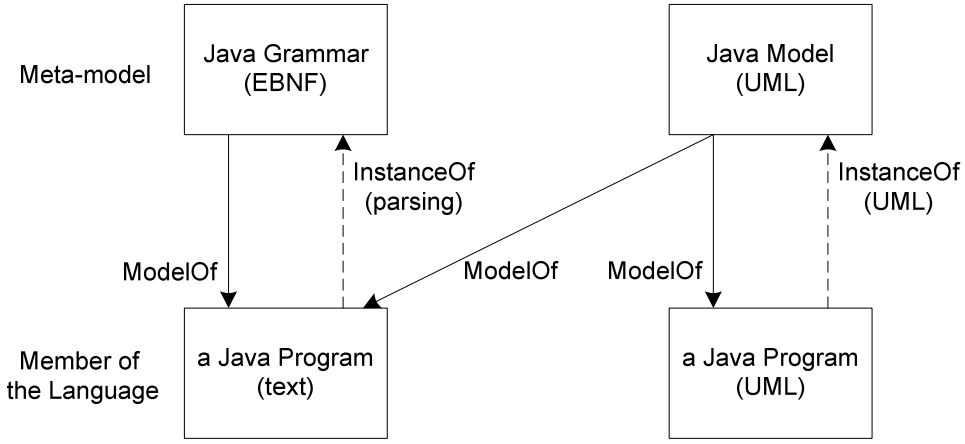


Figure 2.4 Example of meta-models, models and instanceOf relations

The general theory of formal languages provides parsing algorithms that build a derivation tree for a given program and a given grammar. If such a tree is built we say that the program is an instance of the grammar. From the derivation tree we can find the correspondence between the non-terminals in the grammar and the phrases in the program. In other words, on the basis of the derivation tree we can interpret knowledge in the grammar in terms of the program.

At first glance the two relations *ModelOf* and *instanceOf* are the same. Indeed, they coincide between the same entities but are different in nature.

Strictly speaking *instanceOf* relation denotes a membership of an entity to a class where the class is a set. The class is different from the definition of that class. The definition of the class is the intension of the class. The class is the extension of the class definition. From that point of view saying that a model is an instance of a meta-model is not correct since the meta-model is not a set. This is also stated in [19] although the paper does not elaborate much on the subject. However, in most programming and modeling languages *instanceOf* relation is established between a member of the class and the class definition (intension). If an entity is asked for its class the result is the definition of the class and not the class (the set).

An important difference between the two relations is observed when the language-dependent nature of *instanceOf* is considered. Assume that we define another meta-model of Java language expressed in UML (Figure 2.4). The UML model may contain a class

called *Method*. The knowledge we obtain is that there are a set of methods in every Java program that have a certain structure. We must be able to identify methods in the program text and to recognize their structure according to the definition of class *Method*. That is the consequence of the *ModelOf* relation that exists between the Java meta-model and a Java program. However, we cannot consider the Java program as an instance of the UML model in the same way as we did it for the Java grammar. An instance of the UML model is defined according to the semantics of UML and is a set of objects. This instance is a representation of the Java program and is a different entity. The UML model of Java is also a model of the Java program represented in UML. In addition, there is an *instanceOf* relation between these entities governed by the UML semantics. Similarly to the relation between the textual program and its grammar, this *instanceOf* relation helps us to interpret the knowledge from the UML model in terms of the Java program represented in UML. These two *instanceOf* relations are different. The first one is defined for the parsing process. The second one relies on the UML semantics. There is no direct language-specific *instanceOf* relation between the textual Java program and its UML model. However, the latter is a model of the former, although we cannot trace the knowledge from the model to the object system via an *instanceOf* relation.

The purpose of this discussion is to distinguish between the *ModelOf* relation and the *instanceOf* relation. In general, we may associate a predicate to a class and all the entities that are members of the class must satisfy the predicate. From that broader perspective, we can imagine also an *instanceOf* relation between the textual form of a Java program and the UML model of the Java language. It is a matter of proper definition of the predicate and its evaluation over the text of the program. When *instanceOf* is used in the context of a concrete language then we know how the predicate is evaluated. For example, we have an algorithm to check if a program conforms to a grammar. Similarly, we have tools that check if an UML object diagram conforms to a class diagram. In these cases using the *instanceOf* relation instead of the *ModelOf* relation gives us more information about the interpretation of the meta-model.

In summary, *instanceOf* relation exists between a class and its members and supports the interpretation of the knowledge obtained from the class definition in terms of class members. In that case, we also have a *ModelOf* relation between the class definition and class members.

In this thesis we will consider *instanceOf* relation strictly in the context of a particular language that defines the meaning of that relation. Therefore, we do not put an *instanceOf* relation between the Java program in textual form and the Java model in UML. However, a *ModelOf* relation may exist between these entities.

The subtle difference between *instanceOf* and *ModelOf* relations is also emphasized by Seidewitz [93]. In this paper, the *instanceOf* relation is defined according to the semantics of the modeling language. The *ModelOf* relation between the model and its object system is called *interpretation*. The interpretation is a mapping between the model of the language and the elements of the language.

2.5 Model Levels

The meta-modeling activity may be applied repeatedly to build a hierarchy of models that spans multiple *levels*. This level organization is also referred to as meta-modeling stack,

meta-modeling framework or meta-modeling architecture. We adopt the term *meta-modeling architecture* but the other two terms may be used as synonyms.

In section 2.5.1 we consider meta-modeling architectures in general. We focus on a meta-modeling architecture with three levels since it is often used in various technologies. We apply the previously defined concepts of intension, extension, and *instanceOf* relation in the context of meta-modeling architectures. Section 2.5.2 gives examples of meta-modeling architectures.

2.5.1 Meta-modeling Architecture

Figure 2.5 shows examples of meta-modeling architectures. At the bottom level of the stack we have models expressed in various modeling languages. This level is called *model level*.

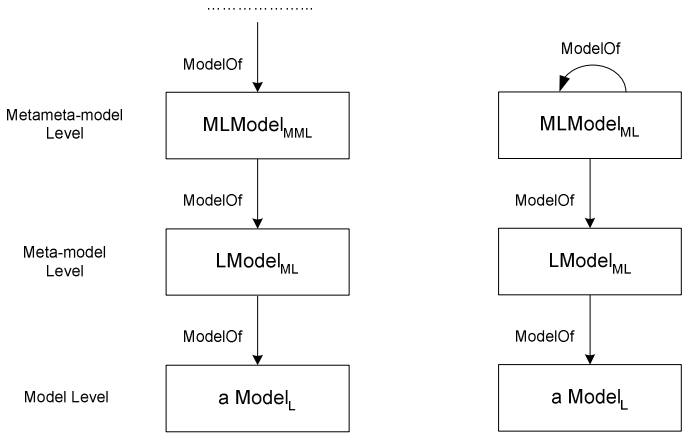


Figure 2.5 Meta-modeling architectures

An example model in this level is $Model_L$ written in a modeling language called *L*. We follow a notation in which the sub-script part of model name indicates the language in which the model is expressed. We can build a model of *L* (that is, a meta-model) $LModel_{ML}$ expressed in another language called *Meta-language* (*ML*). Models of the languages used in the model level form the second level in the stack. It is called *meta-model level*. There is a *ModelOf* relation between a meta-model of a language and models expressed in that language. We can apply the same approach to the models at the meta-model level. The models of the languages that express meta-models form a third level called *metameta-model level*. The left hand-side of Figure 2.5 shows a model of *ML* called $MLModel_{MML}$ expressed in a third language called *Metameta-language* (*MML*).

This approach may be applied infinitely as the left-hand side of the figure suggests. However, in the practice usually only three levels are used. We can simply skip the model of *MML* assuming that it is given outside the meta-modeling stack. Another way is to express the model *MLModel* in the *ML* language itself. This is shown in the right-hand side of Figure 2.5. In this way the top level contains a self-reflective model. It is expressed in the language that is modeled by that model. The intuition behind this is the following. At the meta-model level we have models of modeling languages expressed in *ML*. However, *ML* is a modeling language by itself and therefore it should be possible to apply *ML* on

itself to express its model. In the next section we will give examples of such self-reflective models.

Intensional and Extensional Parts in Meta-modeling Architecture

In section 2.4.2 we introduced the distinction between intensional and extensional parts in models. As we noted a model may be split into two parts containing intensional and extensional entities. How does this distinction affect the level organization?

Figure 2.6 shows $Model_L$ split into two parts: intensional and extensional. As we already explained in section 2.4.2 the intensional part may be considered as a model of the extensional part. Furthermore, a model of a language is an intension with respect to the set of models expressible in that language. $LModel_{ML}$ is intension of $Model_L$. $Model_L$ is a member of the extension defined by $LModel_{ML}$. In the same way $MLModel_{ML}$ is intension of $LModel_{ML}$. Since $MLModel_{ML}$ is modeled by itself it is an intension of itself.

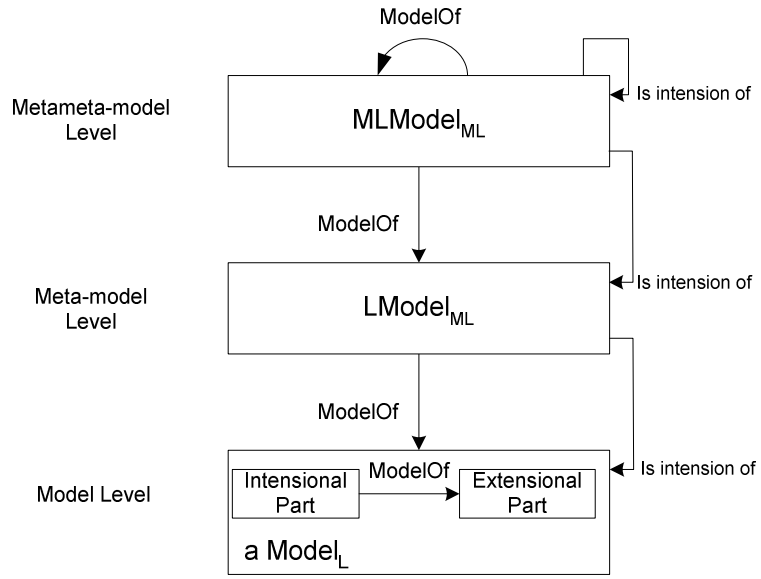


Figure 2.6 Model levels and intensional/extensional dichotomy

In Figure 2.6 we observe that the intensional/extensional dichotomy can be applied between levels and within a given level. This dichotomy is relative. On one hand, $LModel_{ML}$ is an intension of $Model_L$. On the other hand, $LModel_{ML}$ is a member of the extension of $MLModel_{ML}$. The same entity may play both roles of intension and a member of an extension.

A question that arises is why the extensional part in $Model_L$ is not placed at a separate model level. Recall that a level is introduced when a model of a modeling language is built. From that point of view the intensional part in the base model does not model a language. The relation between the intensional and extensional parts in the base model is not meta-model/model relation. These parts have to remain at the same level. To clarify this motivation, consider a base model written in Java. The intensional part is a set of Java classes and their extension is a set of Java objects at runtime. We do not consider the set of Java classes as a definition of a new language nor are the objects considered as an expression in that language. Instead, these parts are belonging to Java language. Java classes

are written in Java language syntax and Java objects behave according to the semantics of Java language.

If we distinguish between intensional and extensional parts in models then these parts should have corresponding modeling parts in the meta-models. Meta-models may therefore be split into two parts as well. The first part contains the model of intensions and the second part contains the model of extensions. Figure 2.7 illustrates this separation for the model of language *L*. It contains two sub-models: *IntensionModel* and *ExtensionModel*.

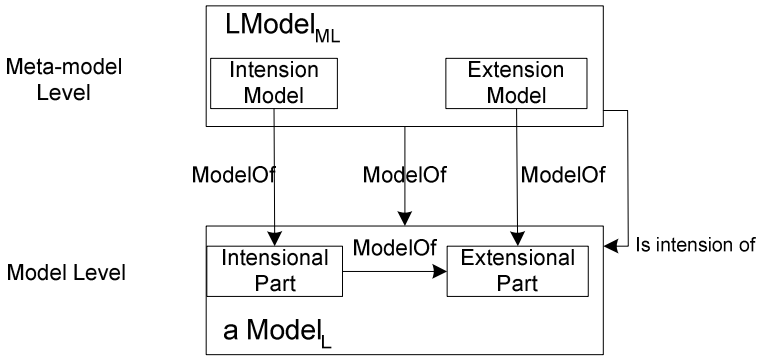


Figure 2.7 Meta-model split into intension model and extension model

Model Levels related by *instanceOf* Relation

As we noted in section 2.4.3 *ModelOf* relation is often replaced by *instanceOf* relation although there is a difference between these relations. In section 2.4.3 we assumed that *instanceOf* should be used in the context of a particular language. In the remaining part of this section we study the structure of a meta-modeling architecture where model levels are related by *instanceOf* relation. We focus on the three-level architecture with self-reflective metameta-model in Figure 2.6. Initially, we focus on the two model levels shown in Figure 2.7. Figure 2.8 shows how these model levels are related by *instanceOf* relation.

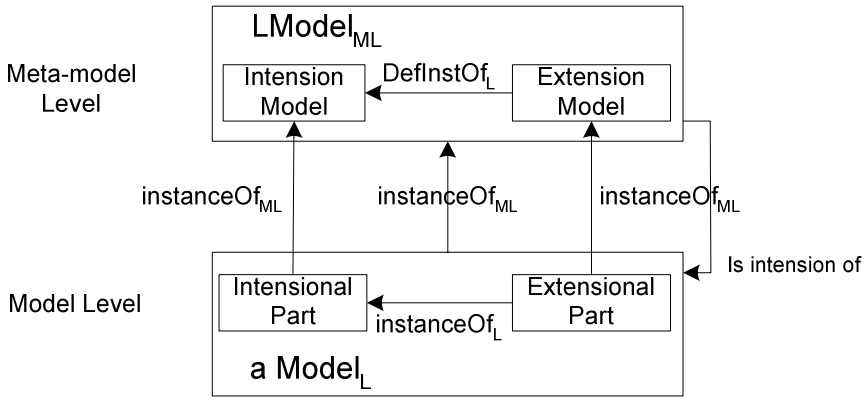


Figure 2.8 InstanceOf relation between model levels

For every occurrence of instantiation relation we have to be able to identify the language that defines it. In the figure the defining language is indicated in sub-script. Extensional part in $Model_L$ is instance of the intensional part according to the instantiation mechanism defined for *L*. Furthermore, $Model_L$ is instance of $LModel_{ML}$ according to the

instantiation mechanism defined for *ML*. The relation between *IntensionModel* and *ExtensionModel* in $LModel_{ML}$ is named $DefInstOf_L$. This is the defining construct of the instantiation mechanism of *L*. Figure 2.8 represents a pattern that may be applied between any two consecutive model levels. We apply this pattern in order to add the meta-metamodel level to the meta-modeling stack. We split $MLModel_{ML}$ into two parts as we did for $LModel_{ML}$.

The result of the application of the pattern is illustrated in Figure 2.9.

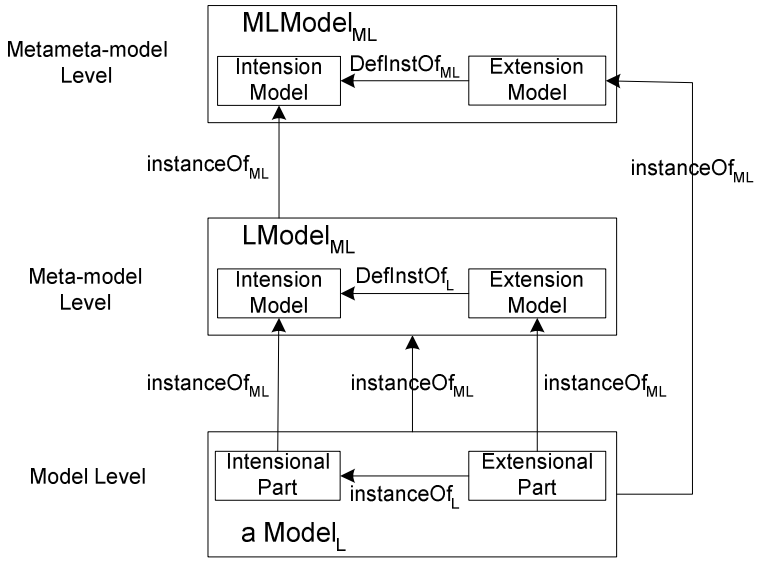


Figure 2.9 Three model levels related with instantiation relations

$LModel_{ML}$ is intension for $Model_L$. Therefore, $LModel_{ML}$ must be instance of the intension model defined in $MLModel_{ML}$ The instantiation relation is in the context of the language that defines $MLModel_{ML}$, i.e. *ML*. On the other hand, $Model_L$ is a member of an extension and therefore must be instance of the extension model defined in $MLModel_{ML}$. However, this is not the full picture yet. We can apply the pattern to $MLModel_{ML}$ to illustrate the self-reflective nature of this model.

Figure 2.10a illustrates the fact that $MLModel_{ML}$ is instance of itself.

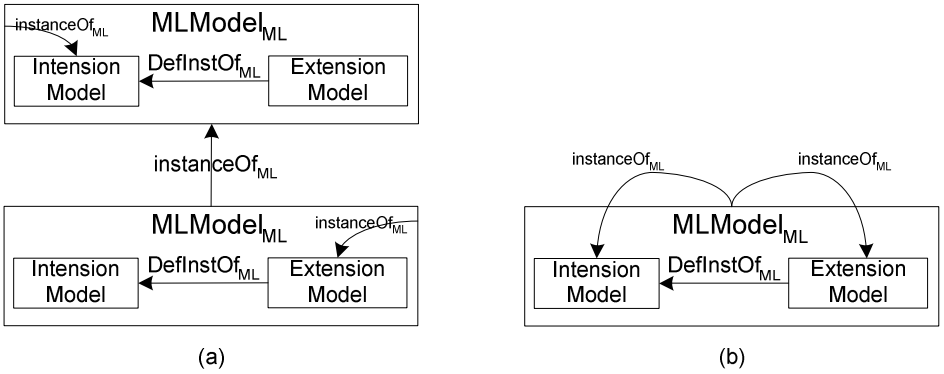


Figure 2.10 The self-reflective nature of the top model level

There are two implications from this fact. The model is an intension of itself. Therefore, it must be an instance of the *IntensionModel* defined for it. Moreover, the model is also an extension of itself. Therefore, it must be an instance of *ExtensionModel*. Figure 2.10b summarizes these observations.

Now we can unite Figure 2.9 and Figure 2.10 into Figure 2.11 shown below.

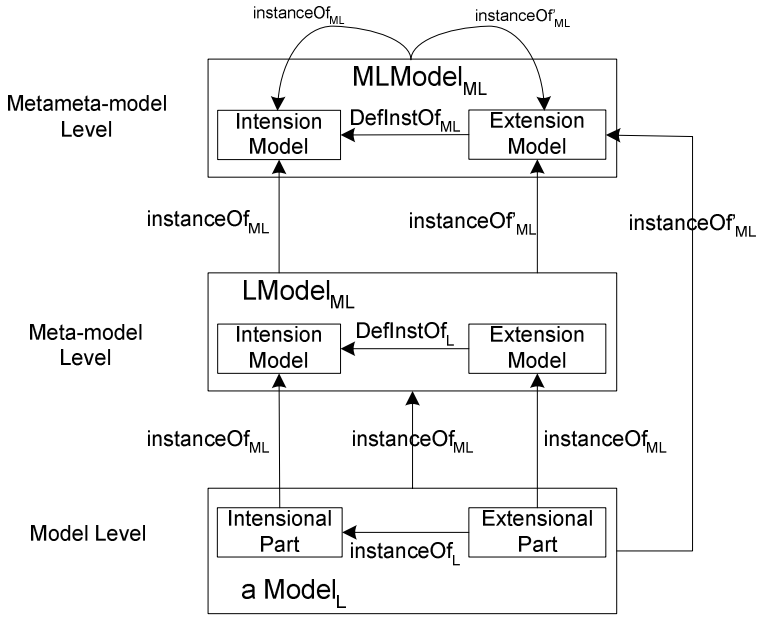


Figure 2.11 Three level self-reflective meta-modeling architecture

Figure 2.11 adds an additional instantiation relation between $LModel_{ML}$ and the *ExtensionModel* defined in $MLModel_{ML}$. Indeed, $LModel_{ML}$ is a member of the extension defined by $MLModel_{ML}$ and therefore must be an instance of *ExtensionModel*.

Some important observations can be made on the base of Figure 2.11:

- all the elements in the meta-modeling architecture share a common structure defined by *ExtensionModel* of $MLModel_{ML}$. This allows creation of tools that manage models in a uniform way. These tools may be model repositories, visual modeling environments, transformation engines, etc. Such a tool would implement the model of extension of the meta-modeling language *ML*. All the elements in the models at any level will be instances of this model. This is the reason that this instantiation relation is denoted as $instanceOf'_{ML}$ in contrast to $instanceOf_{ML}$. It would be implicit in the tools in the sense that the model is hard-wired;
- *instanceOf* relations for a particular modeling language such as *L* is represented in this architecture as an ordinary model element and its meaning is not explicit. For example, $instanceOf_L$ relation in the model level is just a model element that conforms to $LModel_{ML}$. If a distinction must be made between an arbitrary model element and an element that indicates instantiation relation then language *ML* must provide special constructs for that;
- a model may be an instance of multiple models. Instantiation relations are language-specific. Instantiation relations are relations between a model and an intension. Intensions are also models. The relation between an intension and the models members of its extension is more general than the relation between a model and its meta-model. It

can be noticed that the model/meta-model relation exists only between model levels. Intension/extension relation exists between levels and within levels;

The approach for defining meta-levels explained in this section is based on two basic concepts. The first one is the relation between models expressed in a language and a meta-model of that language. This relation is used to identify model levels. The second concept is the distinction between intensional and extensional entities. These concepts can be found in literature that addresses meta-modeling architectures.

In [43] the difference between intensional and extensional entities is identified and a meta-modeling framework is presented where a meta-model is explicitly separated into parts with intensional and extensional meta-entities (these parts correspond to *IntensionModel* and *ExtensionModel* in our terminology). The language used to define meta-models is Object-Z [32]. It is not self-reflective but still provides a common framework for representing models at various levels in a uniform way. The distinction between intensional and extensional parts is also described in the FRISCO report [34]. However, it is only indicated in the base model. Meta-models are not split into parts that model intensional and extensional entities.

Atkinson and Kuhne [13] recognize two types of *instanceOf* relations in a meta-modeling architecture: linguistic and ontological. Linguistic instantiation occurs between model levels. Ontological instantiation occurs in a given level between entities expressed in a given language. For example, in Figure 2.11 $instanceOf_{ML}$ that exists between the model level and the meta-model level is an example of linguistic instantiation. $instanceOf_{L}$ which is located in the model level is an example of ontological instantiation. The authors motivate the need for multiple levels of ontological instantiation within a single model level. Their approach differs from the approaches that focus only on multiple levels of linguistic instantiation and neglect or consider only one level of ontological instantiation. In fact, there is no big difference between the two types of instantiations: both are examples of the relation between an entity and the definition of the class the entity belongs to.

A definition of modeling languages based on separate definitions of the syntax and the semantics of the language is proposed by Precise UML group (pUML) [86]. Their approach is described by Alvarez et al. [9]. The abstract syntax of a modeling language is the model of the intensional entities. Semantics of a language is represented as a model that defines the structure of the extensions. Alvarez et al. also identify the presence of more than one *instanceOf* relation. Their meta-modeling framework is similar to the one presented in Figure 2.11. The difference is that intensional and extensional parts of their base model are located in different levels.

It should be stated that the outlined meta-modeling architecture in Figure 2.11 is somehow idealized. First, there is no commonly agreed approach for defining modeling languages. Second, not all the models shown in the figures are explicitly presented in the modeling stack. In many cases only models of intensional parts (e.g. abstract syntax definition) are defined. Third, some models may be reused across levels.

A number of meta-modeling architectures have been defined for various purposes. In the next section we give examples of concrete meta-modeling architectures and compare them with the framework described in this section.

2.5.2 Examples of Meta-modeling Architectures

This section presents three examples of technologies that rely on meta-modeling architectures: Meta Object Facility (MOF), Resource Description Framework Schema (RDFS), and Extensible Markup Language (XML) and XML-Schema.

Meta Object Facility (MOF)

MOF is a meta-modeling architecture proposed by OMG [75]. MOF is recommended as main technology for definition of modeling languages for MDA. Many other OMG specifications rely on MOF. MOF defines four model levels depicted in Figure 2.12. The meaning of the levels evolved over time. We present the view adopted in the UML 2.0 Infrastructure Library [78].

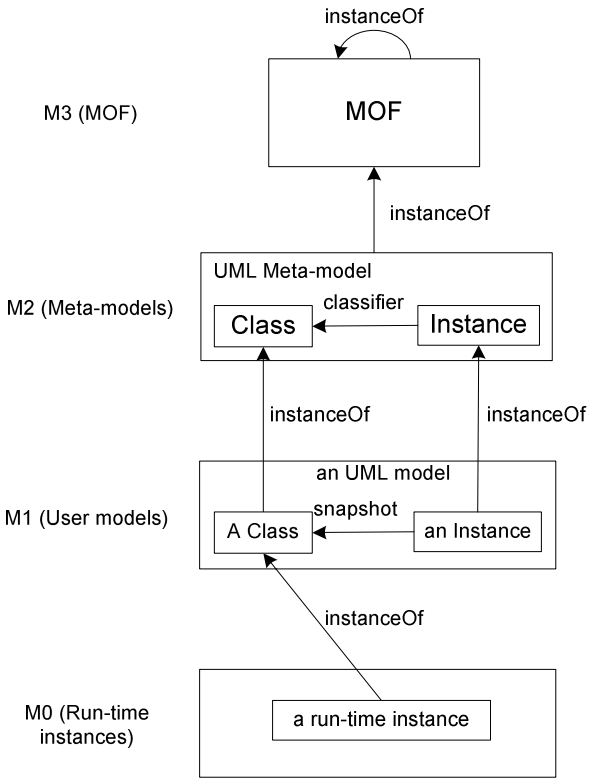


Figure 2.12 Levels in Meta Object Facility (MOF)

The bottom level *M0* contains run-time instances. These instances are considered as part of the real world that is modeled and therefore *M0* should not be regarded as a model level. *M0* instances are modeled by user models located at level *M1*. Figure 2.12 gives an example of an UML model located at level *M1*. Both the classes and their instances are united in a single model level. Instances at *M1* (e.g. *an Instance*) are called *snapshots*. They represent elements at *M0*.

Models at *M1* are expressed in modeling languages defined by means of meta-models located at level *M2* (e.g. *UML Meta-model*). Finally, the top level contains a model of the models at level *M2* and is known as meta-metamodel, *MOF Model* or just *MOF*. It is said

that MOF provides a language for defining modeling languages. MOF is modeled in itself, that is, it is self-reflective.

One of the requirements for MOF is to provide reflective APIs over models and facilities for model interchange. However, this is possible only for levels *M1*, *M2* and *M3*. MOF semantics known also as MOF abstract mapping [75] defines the structure of elements at *M1* as being *objects* with *slots* that hold values and connected to other objects through *links*. Since the model at level *M3* is reflective levels *M3* and *M2* conforms to the same structure that elements at *M1* have. Therefore, all the elements at the top three levels share common structure and may be treated in a uniform way.

It should be noted that the main purpose of the MOF model is to define modeling languages at level *M2*. Therefore, MOF is a domain-specific language whose domain consists of modeling languages. For historical reasons, however, MOF is more like a general purpose modeling language defined as a sub-set of UML. It does not provide a clear way for definition of languages. Current practices show that only the abstract syntax of languages is modeled. To become a language for defining other languages MOF should provide first-class constructs for definition of well-known language elements such as types, instances, instantiation relations, etc.

Resource Description Framework (RDF)

RDF [14] is a data model defined by World Wide Web Consortium (W3C) for the purpose of representation and exchange of meta-data on the web. The RDF data model defines three constructs: *statement*, *resource* and *property*. RDF data created with these constructs form directed labeled graphs where resources are the nodes of the graph and properties are labels at the edges.

RDF data may be typed and untyped. Type information can be added by defining RDF schemas. RDF Schema (RDFS) is the language proposed by W3C for definition of RDF schemas [24]. RDF data, RDF schemas and the RDF Schema language form a hierarchy of levels shown in Figure 2.13.

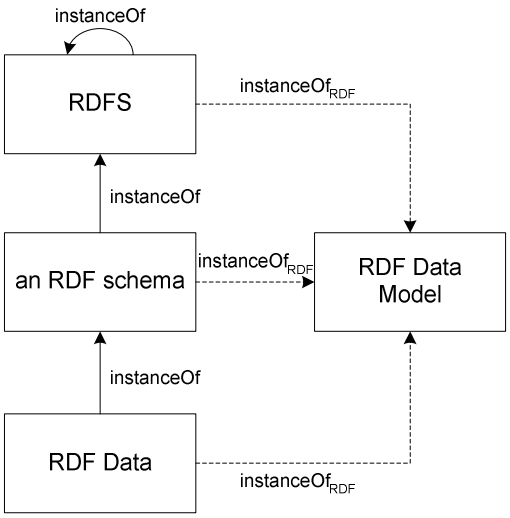


Figure 2.13 Levels in RDF

RDF data may conform to an RDF schema that defines classes and properties of classes. RDF schemas are instances of RDFS which defines a language for schemas. It is

an instance of itself. Apart from the relation between an RDF graph and its schema all the levels conform to the generic RDF data model. Therefore, the whole space may be treated in a uniform way on the base of the RDF data model. It plays the role of extension model on every level. RDFS and RDF schemas play the role of intensions.

Extensible Markup Language (XML)

XML is a technology for data representation and exchange defined by W3C [103]. It is a framework for definition of the syntax of markup languages. A level organization can be observed in the XML domain. It is shown in Figure 2.14.

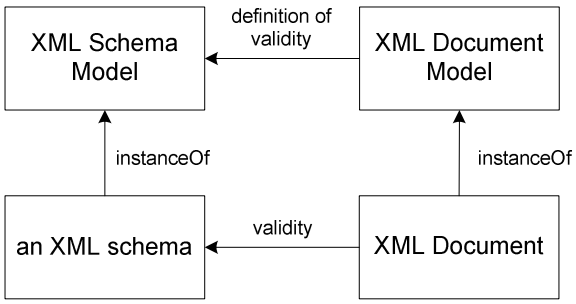


Figure 2.14 Levels in the XML domain

XML specification defines rules for creating XML documents. The rules are expressed in a grammar, however, in practice XML Document Model is most commonly used. XML documents are instances of the XML Document Model. Furthermore, an additional set of constraints may be imposed on XML documents. They are known as validity constraints and are expressed in an XML schema. XML schemas are written in the XML Schema language [106] and therefore, are instances of XML Schema Model. An XML schema is an intension for a set of XML documents. On the other hand, an XML document must always conform to the XML Document Model. In the XML domain an XML document may be an instance of two intensions at the same time. Both are important and can be used together. For example, XPath 2.0 [117] allows selection of XML nodes in an XML document on the base of types defined in the XML Document Model and in an XML schema.

2.6 Model Transformations

Section 2.2.1 introduced the basic concepts in MDA. In MDA, model transformations are sequentially applied over models until the system's code is generated. In this section we describe model transformations and transformation languages in more detail.

2.6.1 Definitions

Here we repeat the definition of model transformation given in the MDA guide: "*model transformation* is the process of converting one model to another model of the same system" [72].

Furthermore the MDA guide defines the concept of *mapping* as follows: "An MDA mapping provides specifications for transformation of a PIM into a PSM for a particular platform". Several types of mappings are identified: model type mappings, meta-model mappings, model instance mappings, combined mappings. A mapping is specified in a *mapping language*.

Kleppe et al. [53] define: "a *transformation* is the automatic generation of a target model from a source model, according to a *transformation definition*". Transformation definition is defined as follows: "a *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language".

The second definition given by Kleppe et al. explicitly mentions the presence of a specification that guides the transformation process. This specification is called *mapping* in the MDA guide [72]. Furthermore, the MDA guide definition of *mapping* narrows the scope of this concept to transformations between PIM and PSM. Since we are interested in the broader context of MDE where PIM/PSM is only one dimension in the modeling space we accept the second set of definitions given by Kleppe et al. The term *transformation definition* explicitly refers to the fact that models are expressed in a given language. This fits to the assumption we made before that our focus is on symbolic models expressed in modeling languages.

Below we give the definition of model transformation that will be used in this thesis:

> *A model transformation is a process of automatic generation of a target model from a source model, according to a transformation definition, which is expressed in a model transformation language.*

Currently, the MDE community has not a common agreement on the meaning of the concepts of *transformation* and *transformation definition*. An initial work to put the understanding of these concepts on mathematical base can be found in [3], [37] and [45].

Many languages can be used to specify transformation definitions. For example, a general purpose language may be used to write a program that transforms one model to another. The trend in the MDE community, however, is towards usage of domain-specific languages for writing transformation definitions. This is exemplified by the Query/Views/Transformations (QVT) Request for Proposals issued by OMG [76] and the number of model transformation languages proposed [8][30][51][84][89][102].

An important requirement put forward by OMG is that transformation languages used in MDA are defined as MOF meta-models. A consequence is that transformation definitions written in such a language become models at level M1 in the MOF meta-modeling stack and they may be treated as any other model.

We can refine the MDA pattern shown in Figure 2.1 to the more general pattern shown in Figure 2.15.

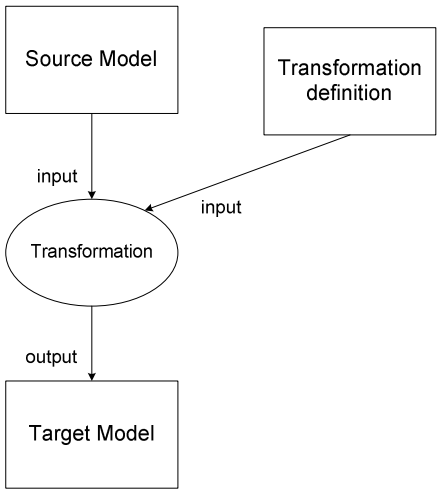


Figure 2.15 Generalized transformation pattern

A transformation definition is usually capable of transforming a set of source models. It is rarely the case when a transformation definition is created for exactly one source model. A typical case is when a transformation definition is designed to transform models written in a given language.

To achieve such a generality the transformation definition is created on the base of some intensional knowledge about the source and target models. For example, a transformation that transforms models expressed in a source language to models expressed in a target language uses the meta-entities defined in the meta-models of the languages. This type of transformation definitions is called *meta-model mappings* in the MDA guide [72].

Section 2.5 showed that a given model may be related to more than one intension. Intensions may be meta-models and models of a domain. Generally, it should be possible to use information from more than one intension in a transformation definition. An example is the XML technology discussed in section 2.5.2

A further refinement of the transformation pattern that shows the information used to specify a transformation definition is shown in Figure 2.16.

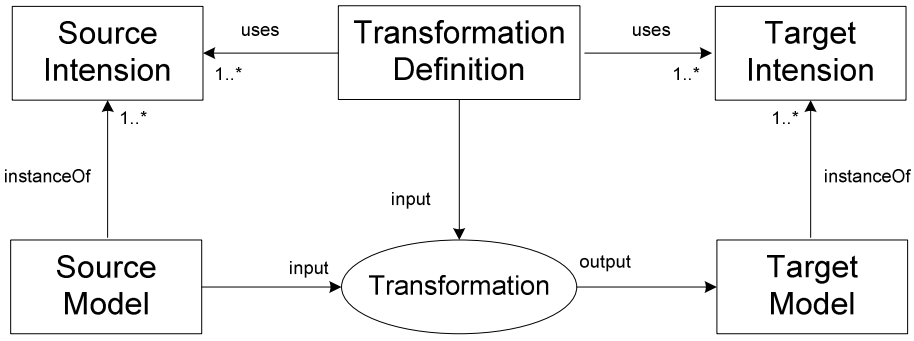


Figure 2.16 Transformation pattern based on intensional knowledge

The figure shows that the source and target models may be instances of more than one intension (denoted as *Source Intension* and *Target Intension* respectively), each with their own *instanceOf* relation. Transformation definition uses constructs from the intensions to

specify rules. Other information that could be used are property values of the model elements in the source model.

2.6.2 Model Transformation Languages

In this section we describe some classifications of transformation languages based on various criteria defined by Czarnecki and Helsen [27] and Gardner et al. [42]. For more extensive discussion on that subject the reader is referred to the full articles.

Declarative and Imperative Transformation Languages

A transformation language is *declarative* if transformation definitions written in that language specify relationships between the elements in the source and target models, without dealing with execution order. Relationships may be specified in terms of functions or inference rules. Transformation engine applies an algorithm over the relationships to produce a result.

In contrast, an *imperative* transformation language specifies an explicit sequence of steps to be executed in order to produce the result.

An intermediate category may also be introduced known as *hybrid* transformation languages. These languages have a mix of declarative and imperative constructs. Usually, a transformation definition written in a hybrid language contains a set of rules that specify relationships between elements. Rules may have imperative bodies that specify the steps to be taken to produce the required result.

Transformation Directionality

Some languages allow specification of transformation definitions that can be applied only in one direction: from source to target model. These transformations are known as *unidirectional* transformations.

Other languages (usually declarative ones) allow definitions that may be executed in both directions. These transformations are known as *bidirectional* transformations. This capability is useful when two models must be synchronized. Assume that a target model is derived from a source model by using a bidirectional transformation and later some changes are made in the target model. The transformation may be executed in the opposite direction to introduce the corresponding changes in the source model as well.

It should be noted that the directionality of transformation definitions does not only depend on the transformation language. Some transformation definitions may transform multiple source elements to a single target element thus making the inverse operation nondeterministic.

If a transformation language does not support definitions of bidirectional transformations then two separate definitions may be used: one for each direction.

Input and Output Cardinality in Transformation Definitions

A typical transformation scenario takes one model as input and produces one model as output (*1-to-1* transformation). Generally, there are three other cases: *1-to-N*, *N-to-1* and *M-to-N*.

In many cases multiple models are produced from a single source model (*1-to-N* transformation). For example, a single model may be used to generate Java code and an XML

schema used for data exchange. Model composition in which several models are integrated into a single one is an example of *N-to-1* transformation. In the general case support for *M-to-N* transformations ensures availability of the other three cases.

2.7 Scenarios for Model Driven Engineering

This section presents a problem solving and evolution oriented view on software development. It describes at a very global level common scenarios observed during software development. Scenarios are based on handling alternative software solutions, decomposition and composition, reuse, and evolution. Every software development process has to cope with these scenarios. They are also manifested in MDE and have to be interpreted in terms of the main concepts of *models* and *model transformations*.

Interpretation of the scenarios in the context of MDE gives at a global level a motivation for the problems defined in the introductory chapter of the thesis (Chapter 1, section 1.2). Scenarios are further discussed in section 2.8 from the perspective of adaptability of model transformations.

Section 2.7.1 presents a problem solving perspective on software development. Section 2.7.2 discusses evolution of software systems. Sections 2.7.3 and 2.7.4 interpret the common scenarios in the context of MDE.

2.7.1 Problem Solving Perspective on Software Development

Software development can be regarded as a problem solving process in which a set of *requirements* are mapped to an executable solution called *software system* **[6]**. The problem for which this solution is derived resides in a given business domain and the software system is intended to *live* within that domain. The business problem drives the software system requirements. The software system must fulfill the requirements and therefore solves the initial business problem.

Requirements are generally classified into *functional* and *non-functional* requirements [95]. Functional requirements describe the services of the system and its behavior over some input. Non-functional requirements describe properties of the system and constraints over the system. A particular subset of non-functional requirements may indicate certain qualities that the system must posses such as reliability, extensibility, adaptability, performance, etc. Such requirements are known as *quality requirements*. Figure 2.17 shows an UML class diagram that represents the relation between requirements and software systems.

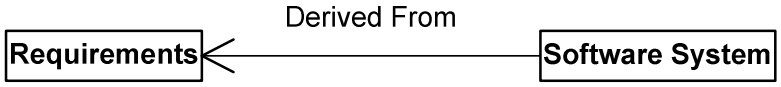


Figure 2.17 Relation between software system requirements and a software system

The process of derivation of a solution from a set of requirements is not trivial. It has been a subject of various studies and many methods have been proposed to drive this process. Regardless of the method that is followed we can observe a set of common scenarios found in the process of software development. They allow us to refine the relation

between the requirements and the solution shown in Figure 2.17. Figure 2.18 displays two scenarios that may occur during a process of deriving a solution from requirements. They usually occur multiple times and are combined with each other during development of complex software systems.

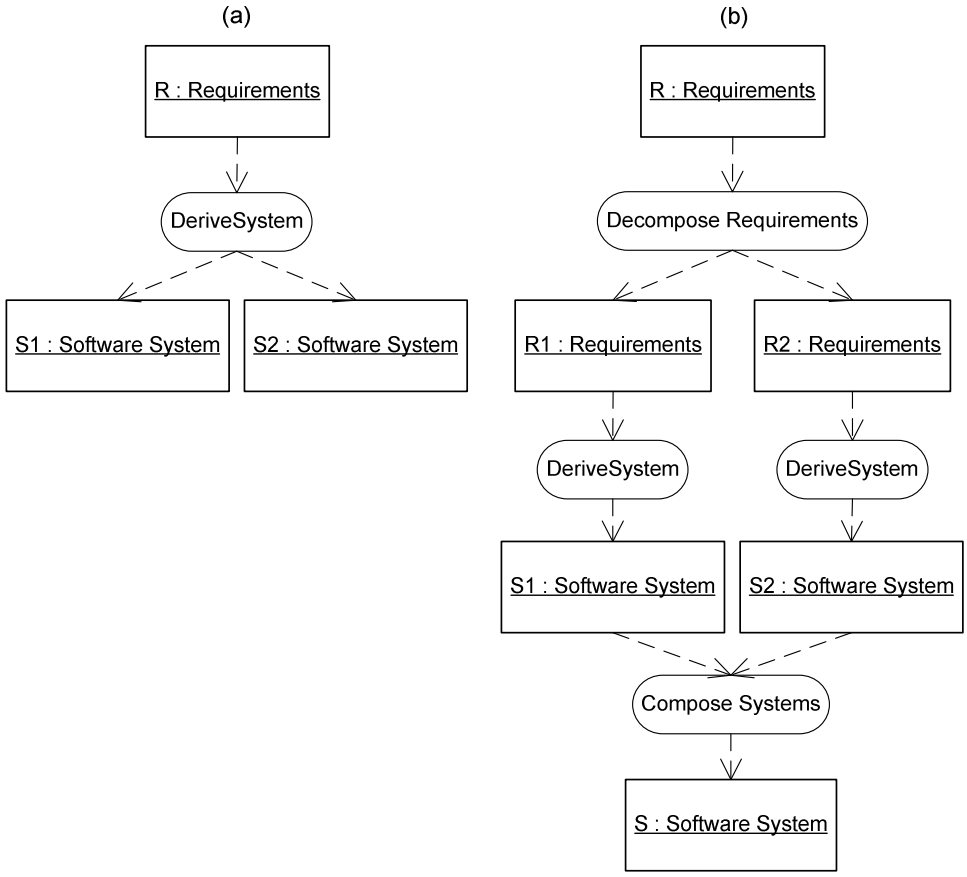


Figure 2.18 Refined relations between software system requirements and software systems: alternative systems for a set of requirements (a), and decomposition and composition of requirements and systems (b)

The first scenario shows the presence of alternative solutions for a set of requirements. The second scenario illustrates decomposition and composition of requirements and solutions.

Alternatives

Figure 2.18a shows the first scenario that illustrates the presence of alternatives in software development. It is possible to have more than one software system (e.g. *S1* and *S2*) for a given set of requirements *R*. Alternative systems may be equivalent with respect to the exposed functionality but different with respect to their quality properties. For example, one alternative may expose better time performance while another may be more extensible. Choosing one of the alternatives is a matter of prioritizing different quality factors.

Decomposition/Composition

Figure 2.18b shows the second scenario that illustrates the possibility to decompose requirements and to compose the corresponding solutions. Decomposition is among the main strategies to deal with complexity. A complex software system is usually derived from a large set of requirements arisen from multiple stakeholders. Requirements may be decomposed into sub-sets that are addressed individually. In Figure 2.18b the initial set of requirements *R* is decomposed into two sets *R1* and *R2*. Systems *S1* and *S2* are derived from *R1* and *R2* respectively. *S1* and *S2* are composed together to form a complete solution *S* that satisfies the initial set of requirements *R*.

Reuse

A software solution may be reused in more than one software system. Reuse of software solutions is one of the main means for faster and cheaper software development.

2.7.2 Evolution of Software Systems

Software systems usually change during their life time. Changes may be driven by different factors. The following list enumerates some possible factors for evolution:

- new requirements arise driven by the dynamics of the business domain they reside in. These requirements may lead to addition of new functionality to the system;
- new technologies are introduced that require porting of some parts of the system for the new technologies;
- parts of the system are changed to correct errors or to improve quality (e.g. time performance, adaptability, etc.);
- some requirements become obsolete and the corresponding parts of the system are removed;

These factors can be considered in the context of software maintenance. In [85] a number of software maintenance categories are described. The first two examples of evolution in the list above are examples of *adaptive maintenance*. Adaptive maintenance is a modification of a software product after its delivery as a response to changes in the environment. Correction of errors in a system is an example of *corrective maintenance*. Improving the quality of a system is an example of *perfective maintenance*.

In this section we consider three scenarios of evolution: additive evolution, replacement of a system component, and subtractive evolution. They are described in the following three sections.

Additive Evolution

This scenario is illustrated in Figure 2.19.

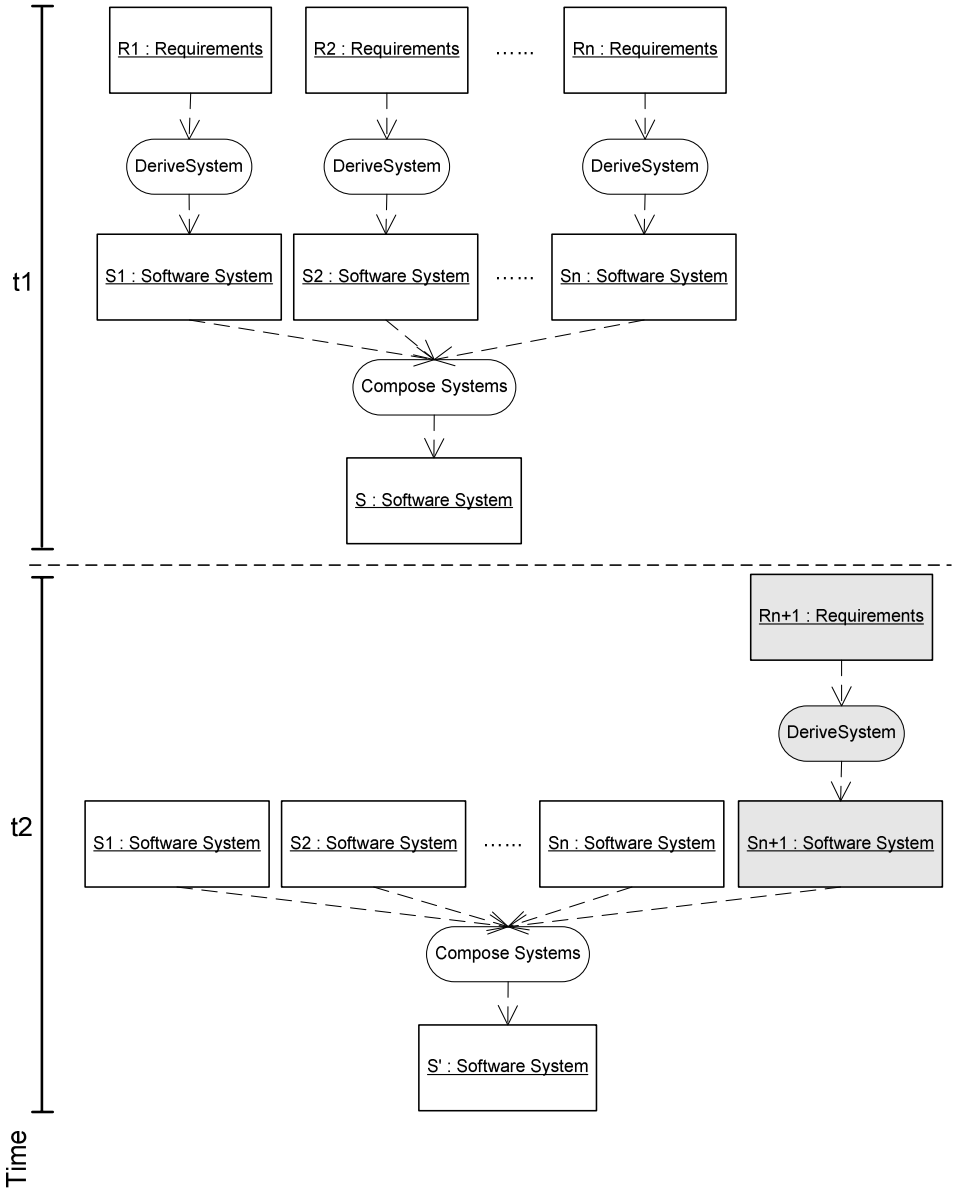


Figure 2.19 Evolution driven by an addition of a requirement: moment t1 shows a base system, moment t2 shows the base system with added requirement Rn+1 and added solution Sn+1

Assume that a system has been derived from *N* requirements *R1,…, Rn* and composed of the corresponding solutions *S1,…, Sn*. This is the initial state of the system at certain moment *t1*. Later on, at moment *t2* a new requirement *Rn+1* is added and a solution *Sn+1* is derived from it. In the figure the new requirement and solution are shown in gray color. The new solution *Sn+1* must be composed with the existing system components. We refer to this scenario as *additive evolution*.

Replacement of a System Component

This scenario is illustrated in Figure 2.20.

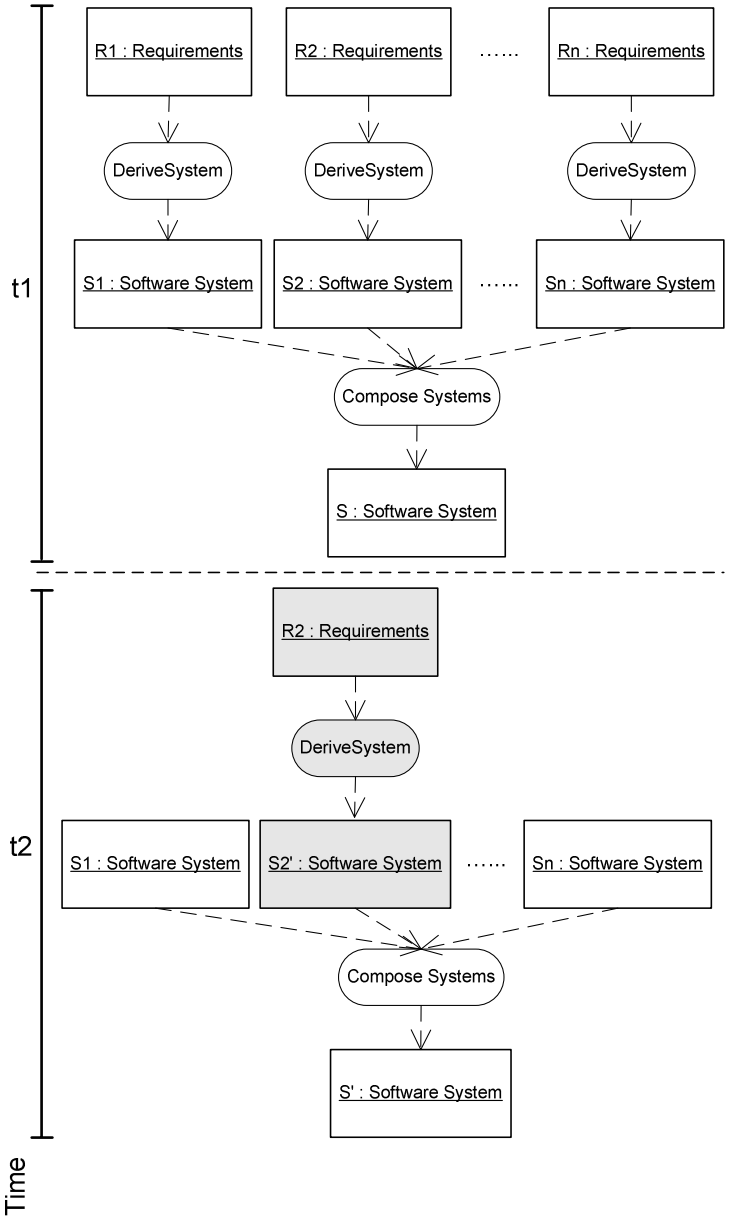


Figure 2.20 Replacement of a system component: moment t1 shows a base system, moment t2 shows the base system where S2 is replaced by S2'

In the scenario a new solution is derived for a given requirement (*R2* in the figure). Software component *S2'* is derived from *R2* and it must replace the initially used component *S2*. This scenario may be of one of the three categories mentioned above. *S2'* may improve the quality of the system (perfective maintenance), may provide a corrected version of *S2* (corrective maintenance), and may provide an implementation in a different technology (adaptive maintenance).

Subtractive Evolution

The last scenario is shown in Figure 2.21. In response to changing requirements, requirement *R2* is removed and the corresponding solution *S2* also must be removed from the system.

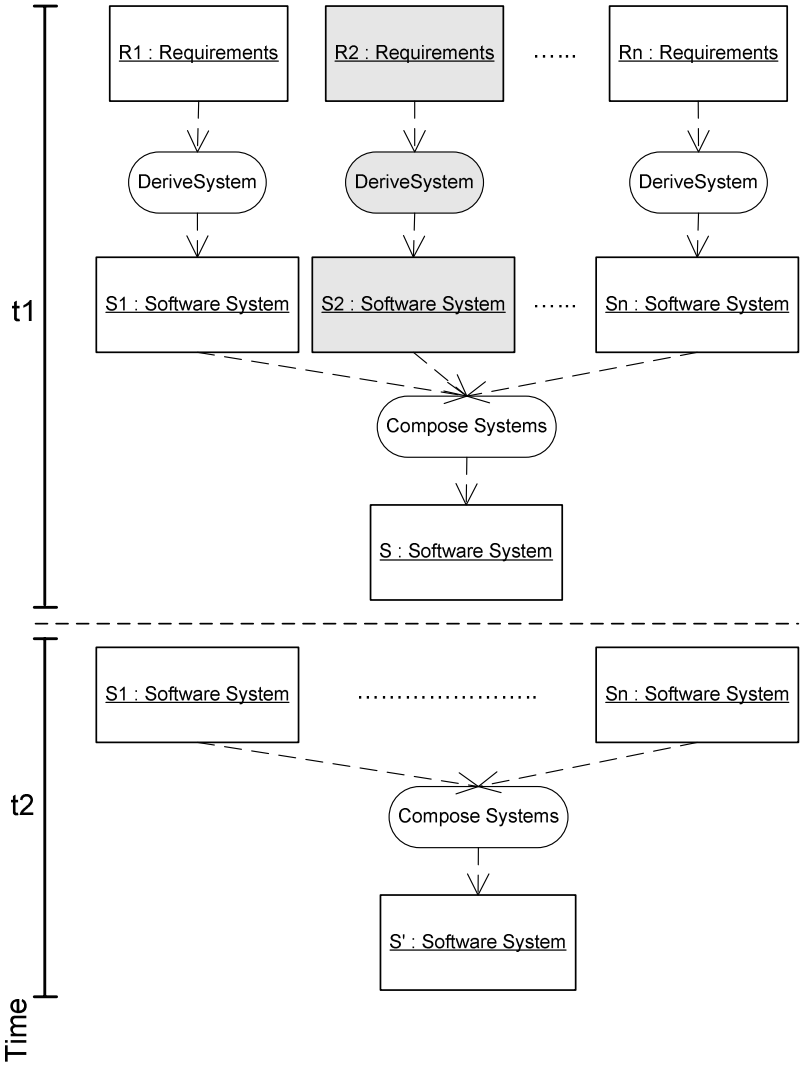


Figure 2.21 Subtractive evolution: moment t1 shows a base system, moment t2 shows the base system where requirement R2 and the corresponding solution S2 are removed.

The lifecycle of a software system including its development and maintenance is in the general case a sequence of steps that apply the scenarios in sections 2.7.1 and 2.7.2. Different software development methods cope with the scenarios in different ways. Every scenario has a specific form in the context of different methods. In the remaining part of the chapter we describe how the scenarios are applied in the context of Model Driven Engineering.

2.7.3 Alternatives and Decomposition/Composition in Model Driven Engineering

The first part of this chapter introduced basic concepts in MDE. Being an approach for software development MDE has to handle the scenarios observed in software development processes explained in the previous two sections. In this section we explain how the scenarios are interpreted in terms of the MDE concepts. A basic step in MDE development is the application of the transformation pattern where a transformation definition is executed on a source model to obtain a target model. In the following sections we study how the scenarios for alternatives and decomposition/composition in software development are applied by using the transformation pattern.

Alternatives

In terms of the transformation pattern, the software solution is the target model from which the system will be ultimately built. Alternative solutions may arise in two ways as Figure 2.22 shows.

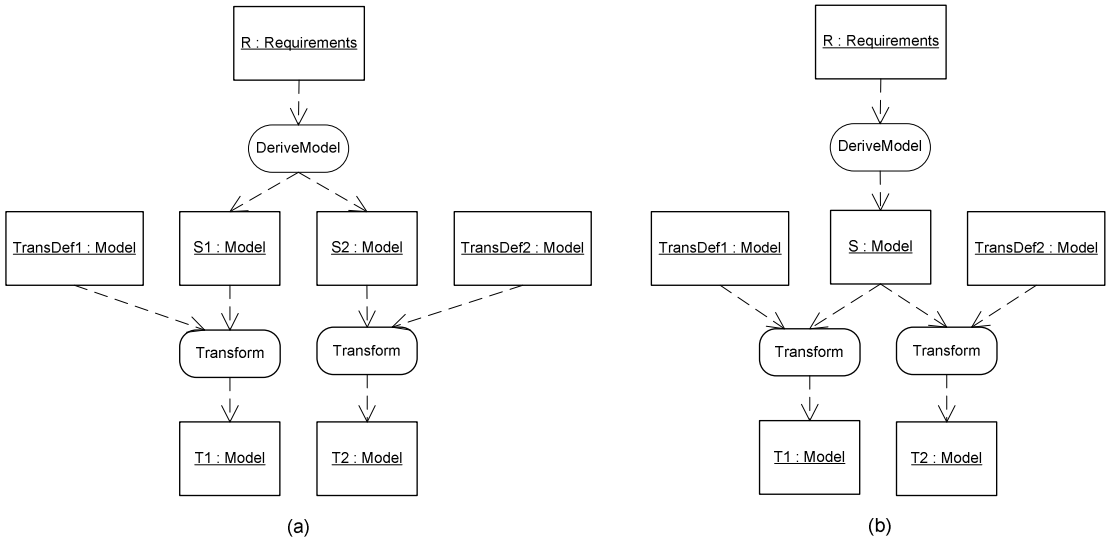


Figure 2.22 Alternatives in the transformation pattern: alternatives derived from alternative source models (a), and alternatives derived via alternative transformations (b)

In Figure 2.22a two alternative source models (*S1* and *S2*) are created for a requirement. After applying transformations on the source models we obtain two alternative target models (*T1* and *T2*). Transformation definitions used to execute the transformations are different in the general case but may also be identical. Figure 2.22b shows another possibility for generating alternative solutions. A single source model *S* may be transformed in multiple ways by using different transformation definitions. The figure shows a scenario with two transformation definitions *TransDef1* and *TransDef2*. Target models *T1* and *T2* may differ in their quality properties. In this scenario it is important to identify the transformation definitions that produce the result with the desired quality.

Since the transformation pattern may be applied multiple times in a sequence the scenarios in Figure 2.22 may occur multiple times as well. Sequential application may produce an arbitrary number of alternative solutions.

Decomposition and Composition

In the transformation pattern two components may be affected by a decomposition of requirements: the source model and the transformation definition. They may be decomposed independently from each other and also the decomposition of the source model may influence the decomposition of transformation definition. The resulting model may be decomposed as an effect of the transformation. The case when only the source model is decomposed is shown in Figure 2.23.

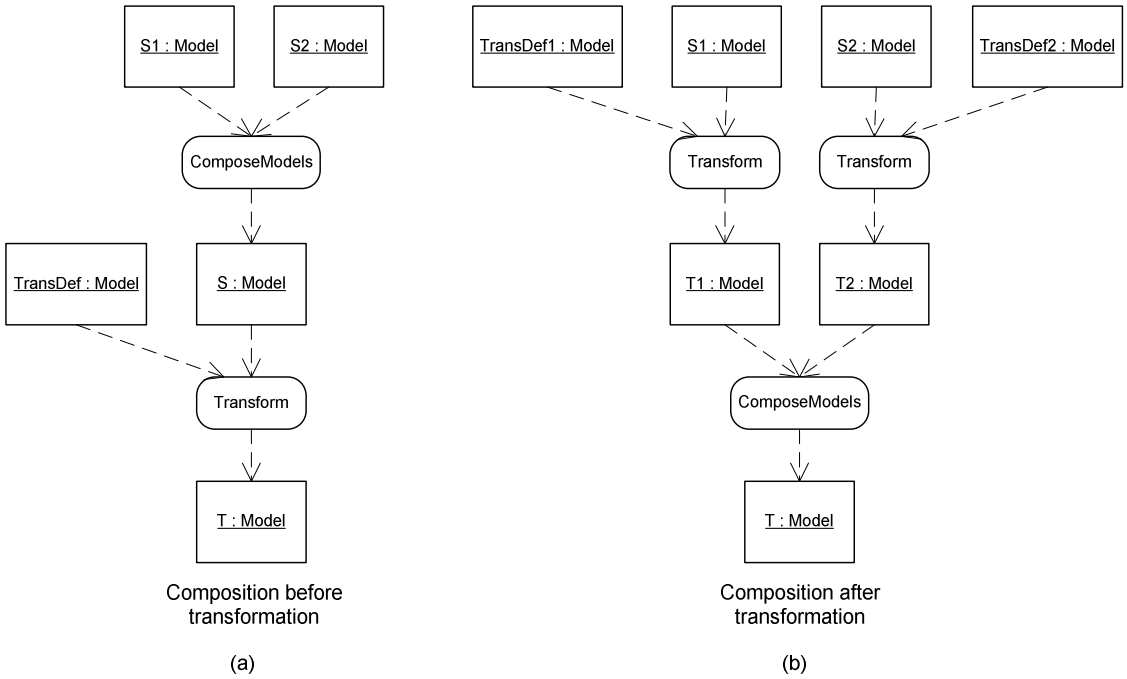


Figure 2.23 Decomposition and composition of the source model in the transformation pattern: composition of source models before transformation (a) and composition of target models after transformation of source models (b)

Assume that an initial set of requirements is decomposed into two sub-sets and two source models *S1* and *S2* are derived. There are two possibilities:

- the two source models are first composed and then the result is transformed to the target model (Figure 2.23a);
- the two source models are first transformed to new models which in turn are composed to form the result target model (Figure 2.23b);

It should be noted that the composition of the models may be implemented as a transformation that takes two models as input.

Transformation definitions in the transformation pattern may also be a subject of decomposition and composition. There are two cases in this scenario, depicted in Figure 2.24.

The first case is when a transformation definition is decomposed in two definitions *TransDef1* and *TransDef2* (Figure 2.24a). These definitions are composed to form a complete transformation definition.

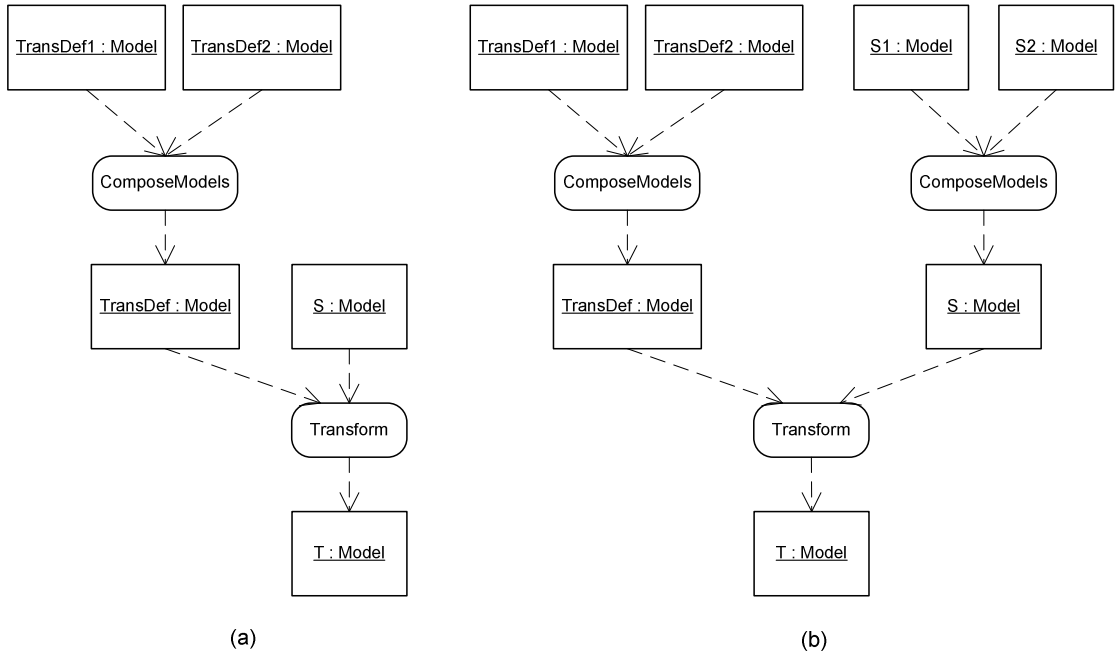


Figure 2.24 Decomposition and composition of transformations in the transformation pattern: decomposed transformation definition (a) and decomposed transformation definition and source model (b)

The second case is when decomposition in the source model drives the decomposition in the transformation definitions. Figure 2.24b illustrates the case. The source model *S* is composed of two models *S1* and *S2*. Two transformation definitions *TransDef1* and *TransDef2* are defined for *S1* and *S2* respectively. They are composed together in the definition *TransDef* applied on model *S*.

2.7.4 Evolution Scenarios in MDE

In this section we analyze additive and subtractive evolution in terms of the application of transformation pattern.

Additive Evolution

We describe how addition of new functionality affects the transformation pattern. We identify four cases. The first two are shown as a sequence of three evolution steps in Figure 2.25. At moment *t1* a target model *T* has been derived from source model *S* through a transformation. The first case of adding new functionality is based on extension of the source model. At moment *t2* the source model *S* is composed with new model *S1* and the resulting model *S'* is transformed to a new target model *T'*. The same transformation definition *TransDef* is used.

The second case is based on extending the target model. At moment *t3* a new model *T1* is composed with *T'* to form a new model *T''*. In this second case the source model *S'* is not affected. A possible consequence is a need for synchronization between *T''* and *S'*. We do not study this possibility here.

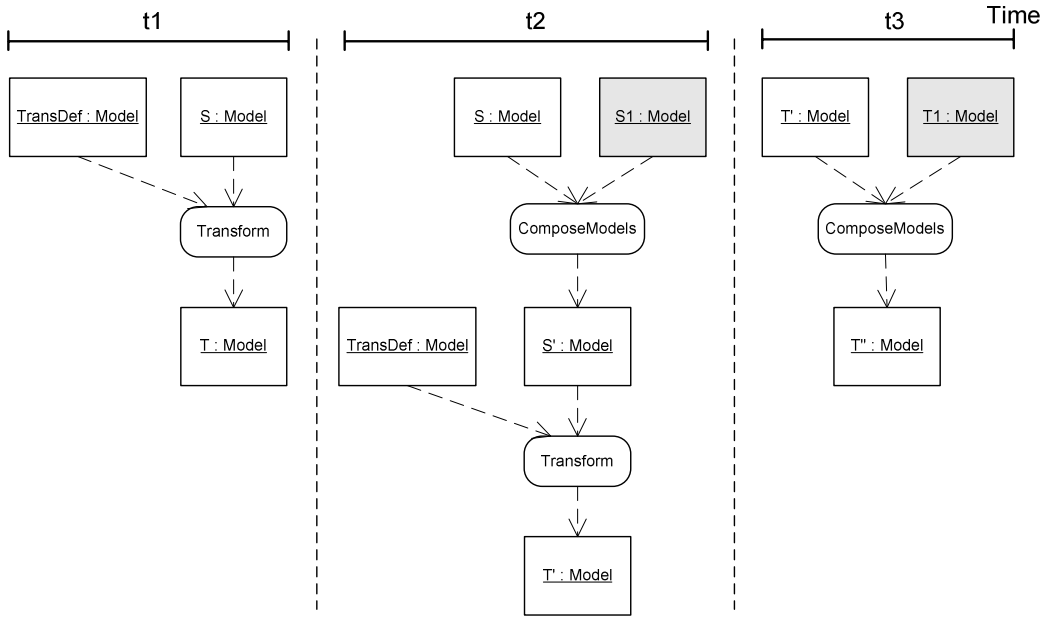


Figure 2.25 Implementation of additive changes in the transformation pattern by adding new models: moment t1 shows the initial state, moment t2 shows an addition of a source model S1, moment t3 shows an addition of a target model T1.

Figure 2.25 adds new functionality that takes the form of a new model and do not change the transformation definition that has been applied. Addition of new functionality may also be achieved by changing the transformation definition as Figure 2.26 shows.

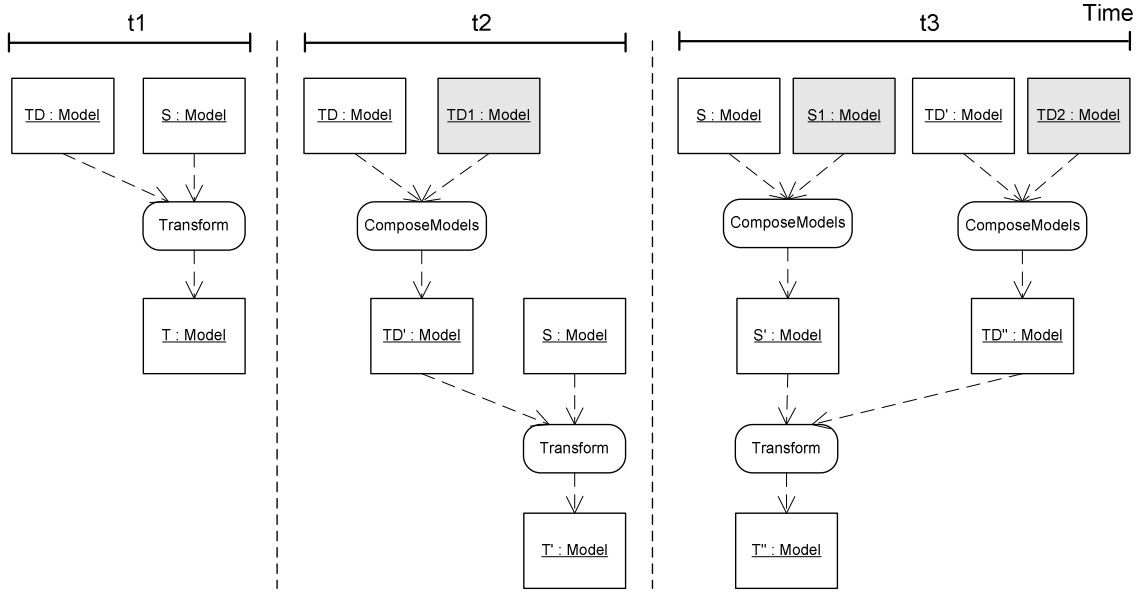


Figure 2.26 Implementing additive changes by changing transformation definition: moment t1 shows the initial state, moment t2 shows an addition of transformation definition TD1, and moment t3 shows additions of new source model S1 and transformation definition TD2

Again two cases are possible. In the first case, at moment *t1* model *T* is derived from model *S* by executing transformation definition *TD*. Later on, at moment *t2* new functionality is introduced by extending the transformation definition *TD* with new transformation definition *TD1*. The result is a new definition *TD'* that is executed on *S* to obtain a new target model *T'*.

The second case introduces additions by extending both the source model and the transformation definition. At moment *t3* model *S* is extended with model *S1*. Model *S1* contains elements that have to be transformed by transformation definition *TD2*. *TD2* is composed with *TD'* to form a new definition *TD''*. The resulting target model *T''* contains additional functionality compared to model *T'*.

Subtractive Evolution

This scenario is opposite to the scenario for additive evolution. It is based on removing components from a software system. Similarly to the previous section we describe the impact of the subtractive evolution on the application of the transformation pattern.

Since the subtractive evolution is opposite to the additive evolution the possible cases are inverse variants of the cases already considered for additive evolution. For example, Figure 2.26 can be inversed in the way illustrated in Figure 2.27. Gray boxes indicate components to be removed.

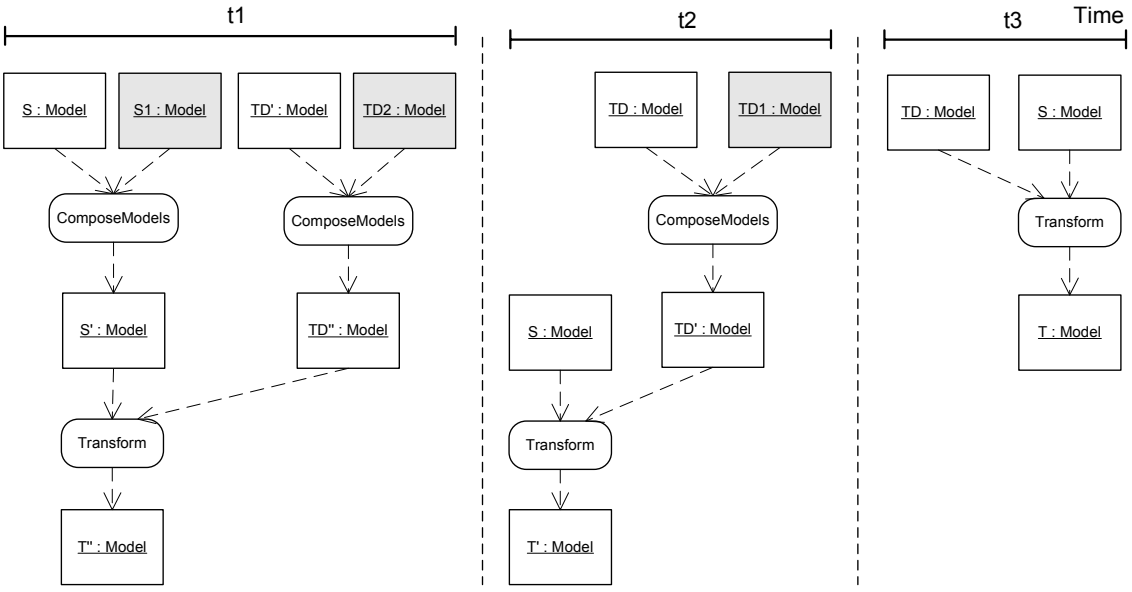


Figure 2.27 An example of distractive evolution and its effect on the transformation pattern: moment t1 shows the initial state, moment t2 shows a state after removal of source S1 and transformation definition TD2, moment t3 shows a state after removal of transformation definition TD1

The cases shown in Figure 2.25 may be inversed in a similar way.

2.8 Adaptability of Model Transformations

In sections 2.7.3 and 2.7.4 we focused on the interpretation of software development scenarios in the scope of MDE. Many scenarios were driven by changes that may occur due to the factors listed in section 2.7.2. The ability of a software system to cope with changes is an important characteristic that determines the *adaptability* of the system.

Adaptability is a quality property of software products. Software product quality is treated in ISO 9126 [48] which is an international standard. It defines six quality characteristics that are further refined into sub-characteristics. Adaptability is defined as a sub-characteristic of *portability*.

Portability is a capability of software to be transferred from one environment to another. The definition of adaptability according to ISO 9126 is the following:

> *Adaptability - the capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered.*

We interpret the term ‘environment’ in a broad sense. Elements of environments may be various implementation technologies and given business domains. Introduction of new software requirements driven by the dynamics in the business domain may be considered as a change in environments in which the software acts.

ISO 9126 defines adaptability in the context of software products. Adaptability can be considered in other contexts as well. In [98] adaptability is discussed in the context of object-oriented development. Adaptability can be defined at various levels: analysis and design level, program level, compilation and run-time level. Adaptability may be considered also in the scope of software development processes.

In a similar way, in MDE based development the need for adaptation may occur at different levels. The development process may be adapted to the needs of a specific organization. We, therefore, need adaptability property at process level. Furthermore, artifacts such as models, languages and transformations may be adapted as a response to new requirements and introduction of new environments. This is an example of adaptability at the artifact level. Finally, the tools used in MDE also may be adapted. Therefore, adaptability is a property that can be applied in a larger scope and not only on a software product.

In this thesis we study the need for adaptability in the context of model transformations. Changes that may occur are analyzed on the base of the transformation pattern. An enhanced version of the pattern is shown in Figure 2.28.

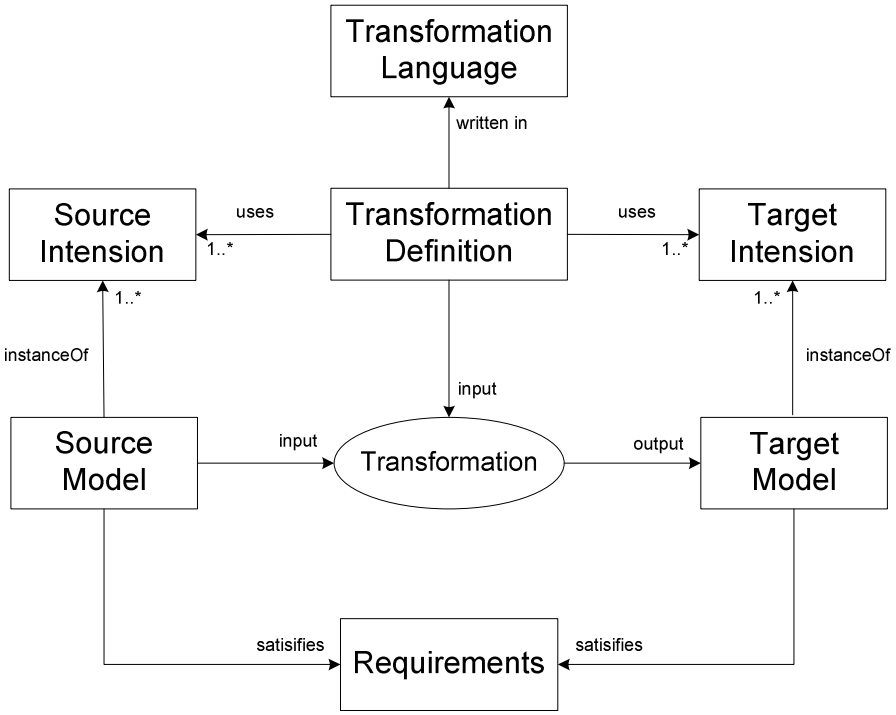


Figure 2.28 Enhanced version of the transformation pattern in Figure 2.16 showing requirements

Figure 2.28 shows explicitly the transformation language used to express transformation definitions and the requirements that are satisfied by the source and target models.

The following changes may occur that motivate the need for adaptability of components shown in the pattern.

Changes in Requirements

Changes in requirements may lead to changes in the source model and in the target model. Changes in the models, as we saw may be additive, subtractive or replacement of some model elements. Scenarios illustrated in Figure 2.26 and Figure 2.27 show that these changes may lead to changes in the transformation definition as well. The ability to adapt a transformation definition depends mainly on two factors. The first one is how the definition is specified and the second one is related to the features provided by the transformation language to adapt existing definitions.

Furthermore, changes in the requirements may put a need for an entirely new transformation definition that produces a new target model that satisfies the new requirements. This is the case of alternative transformations for a given source model considered in section 2.7.3.

Changes in Source Models and Target Models

Changes in source models have been discussed in the context of the evolution scenarios. As we saw they potentially lead to changes in the transformation definition. For instance, a transformation definition may select certain elements in the source model on the base of their property values. If new property values are introduced that must be considered in the transformation definition this leads to changes in the transformation definition. An exam-

ple of such a scenario is given in Chapter 6 where the source model is an XML document. The processing of some XML elements depends on the values of certain attributes. Although the source intension (XML schema) remains unchanged the transformation definition must be changed if new attribute values are introduced.

Applying changes in the models may be facilitated by a proper decomposition in the models. Again, this has impact on the decomposition of the transformation definition as section 2.7.3 showed.

Changes in target models may require synchronization with the corresponding source models. This puts certain requirements on the transformation language such as a capability of expressing bidirectional transformation definitions.

Changes in Intensions

Changes in intensions may be interpreted in two ways: change of a modeling language and change of the domain model. The first change has impact on the transformation language. It should be capable of transforming models expressed in different modeling languages.

The second change affects the transformation definition. As we already mentioned, changes in the transformation definition are related to the way how the definition is designed and decomposed and to the easiness to implement the changes. The latter depends on the transformation language features.

Changes in Transformation Definitions

Transformation definitions are models. Changes may be additive, subtractive and replacement of elements. Some changes may be supported by the transformation language. Other changes may not be implemented within the given transformation language. A proper decomposition of transformation definitions may facilitate the application of changes.

Changes in Transformation Languages and Transformation Engines

Changes in transformation languages may lead to changes in transformation definitions and in transformation engines. It is also possible that a new transformation language and engine are used instead of the previously used. Transformation languages may differ in their expressive power. We may envisage examples where a given transformation language is not capable to express transformation definitions that produce models according to the given requirements.

Changes in transformation engines usually do not affect other components in the pattern. A new engine, however, may expose better quality characteristics such as improved performance.

2.9 Conclusions

In this chapter we introduced the basic concepts in Model Driven Engineering. MDE is a new approach for software development and its conceptual foundation is in the beginning of formation. Our approach for defining a consistent set of concepts is based on analysis of the definitions found in literature and selection of those definitions that suits the objec-

tives of the thesis. The chapter presented discussion on the notion of model, meta-model, and model transformation.

Model transformation is a main operation in MDE. It is governed by a transformation pattern that specifies the input, output and the additional information required for performing model transformations. Generally, MDE is based on a sequence of applications of the transformation pattern.

In software development we observe a set of common scenarios related to alternatives, decomposition and composition, reuse and evolution. In case of MDE these scenarios are implemented on the base of the transformation pattern. Analysis of the scenarios helps in identifying problems that must be tackled in an MDE based software development. An important property that appeared in many scenarios is the adaptability of model transformations. Next sections summarize problems related to the usage of transformations and the need for adaptability.

Identification of Alternative Transformation Definitions

Alternative solutions may be derived from a single model by means of alternative transformation definitions. Alternative solutions may differ in the quality properties they possess. Therefore, a proper transformation definition must be identified that produces a solution with the required quality properties. Software engineers should be supported in the process of identification, comparison and selection of alternative transformation definitions for a given model or a set of models. This problem is addressed in Chapter 3.

Definition of Transformations on the base of multiple Intensions

As Figure 2.16 suggests a transformation definition may be based on multiple intensions for the source and target models. In the general case, every intension is used together with a language specific instantiation mechanism. A domain-specific transformation language should be able to cope with models written in different languages and conforming to multiple intensions. This problem is addressed in Chapter 4.

Composition and Decomposition of Transformation Definitions

Section 2.7.3 presented scenarios in which decomposition and composition of transformation definitions were required. The first important issue related to this requirement is to identify the sources of decomposition in transformation definitions. Some of the scenarios suggest that the decomposition of the source model may drive the decomposition of the transformation definition.

Moreover, the analysis of the impact of various changes in the components in the transformation pattern revealed that transformation definitions often follow those changes. It is generally known that a proper decomposition of a software artifact may help in coping with changes.

The second issue is related to the language support provided by a transformation language. A transformation language must supply proper modularity constructs capable of expressing the required decomposition. Furthermore, a set of compositional operators is also required to achieve the composition over transformation definition modules. It is important to study the required modularity constructs and required compositional operators in transformation languages. These issues are addressed in Chapter 5.

3

Identification and Selection of Alternative Transformations

This chapter motivates the need for inclusion of an activity in an MDE software development process that considers identification and selection of alternative transformations for a given model. A formal technique is defined for modeling the space of alternative transformations. The technique provides operations for reduction of and selection from a transformation space on the base of desired quality properties of the resulting model. This technique provides the software engineer with a means for explicit description and reasoning about alternative transformations. He is able to identify the transformations that produce a result with certain quality properties. The chapter shows a transformation space based on the extensibility quality property.

3.1 Introduction[1]

In an MDE software development process, models are repeatedly transformed to other models to finally achieve a set of models containing enough details to implement the system. As we discussed in Chapter 2, generally, there are multiple ways to transform one model into another model. Alternative target models may have the same functional behavior but may differ in their quality properties. The selection of a particular model should then be determined on the base of quality requirements.

[1] This chapter is an adapted version of [56] and [57]

According to the transformation pattern given in Chapter 2, a transformation is executed on the base of a transformation definition. Transformation definition contains transformation rules that relate constructs in the source model to the constructs in the target model. Generally, a single construct in the source model may be transformed into multiple alternative constructs in the target model.

Figure 3.1 is based on the MOF meta-modeling architecture [75]. The source and target models are situated in the level M1 according to the MOF terminology and their source and target meta-models are in the level M2. The source model M_A is an instance of the source meta-model MM_A. Assume that M_A contains three elements *a1*, *a2*, and *a3* shown as rectangles with relations among them. M_A has to be transformed to a model that is an instance of the target meta-model MM_B. The two meta-models can be used to determine the possible transformations rules. MM_A contains two constructs: *A1* and *A2*. MM_B contains four constructs: *B1*, *B2*, *B3*, and *B4*.

Generally, for each construct of the source meta-model there are multiple constructs of the target meta-model to which it can be mapped. Assume that the construct *A1* in the source meta-model can be alternatively mapped to the three constructs *B1*, *B2*, and *B3* in the target meta-model. This introduces alternative mappings for each instance of *A1* in the source model M_A. Figure 3.1 shows one possible mapping in which *a1* is mapped to an instance of *B2* and *a2* is also mapped to an instance of *B2*. Other combinations are generally possible and this results in multiple target models.

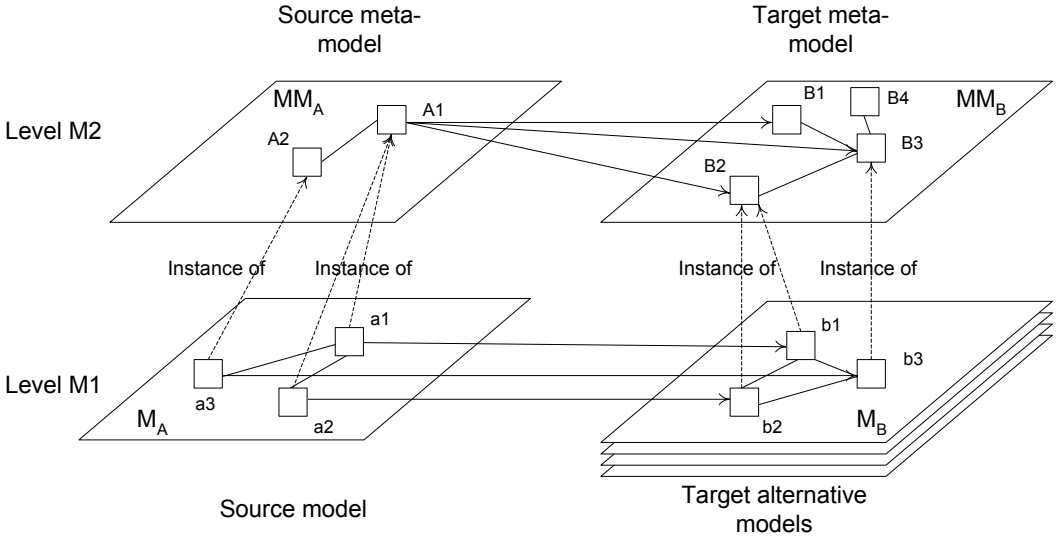


Figure 3.1 Example of alternative transformations for a given source model based on meta-model transformation pattern

The resulting target models may differ from each other in the quality properties they possess. For example, assume that the target model is implemented using an object-oriented language. Generally, there are various implementation alternatives. One implementation may inline code and therefore display a better performance than the implementation, which clearly separates code into distinct run time objects. The latter implementation, however, may offer more adaptability. Software engineers have to compare and choose among the alternatives based on the quality requirements. Different quality requirements should lead to functionally equivalent implementations that differ in their

quality properties. For a concrete problem the software engineer must be able to identify the transformations that lead to a model with the desired quality properties.

Unfortunately, the current transformation languages and techniques do not provide means to identify alternative transformations and to compare them based on specific quality characteristics of the resulting models.

This chapter describes an approach for identification of alternative model transformations that should be included as an activity in an MDE-based software development process. We propose a technique to explicitly model a set of alternative transformations for a source model based on construction of *transformation space*. This technique also allows specification of quality properties of the models. The quality properties are used as selection criteria among the alternatives. The approach proposed in the chapter is applied in a case study used to identify transformations of UML models into XML schemas.

The structure of the chapter is as follows. Section 3.2 presents the case study and explains the problems addressed in the chapter. Section 3.3 describes the place of the approach in an MDE-based software development process. Section 3.4 defines the notion of transformation space. Section 3.5 applies the approach to identify and select transformations for the case study in Section 3.2. Section 3.6 discusses the applicability of these techniques in the context of model transformations in MDE. Section 3.7 gives an overview of the related work and Section 3.8 concludes the chapter.

3.2 Problem Statement

This section gives an example of transforming UML class models to XML schemas. For this example some transformations are identified following Figure 3.1. We formulate two problems experienced in identification and selection of alternative transformations in the context of the example.

3.2.1 Transformations from a UML Class Model to XML Schemas

Figure 3.2 shows the presence of alternative model transformations using an example which transforms UML class models to XML schemas.

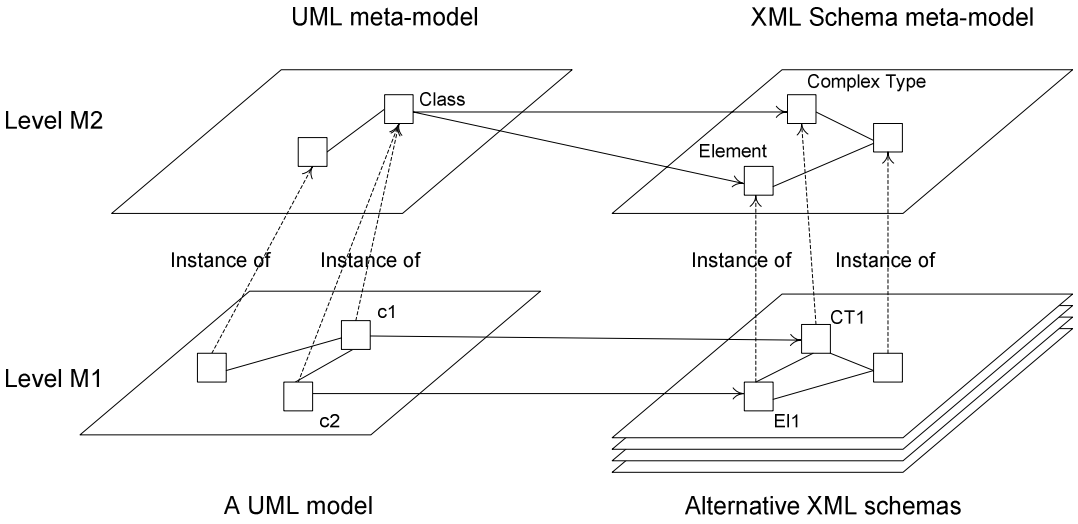


Figure 3.2 Example of alternative transformations to XML schemas for a given UML class model with classes c1 and c2

Figure 3.2 is a concretization of the general picture in Figure 3.1. The UML class model is a source model and the XML schema is the desired target model. In Figure 3.2, the source model is an UML class model and its meta-model is the UML meta-model [77]. The target meta-model is an XML Schema meta-model that can be derived from the XML Schema specification [106]. *Class* construct defined in the UML meta-model may be mapped, for example, to *Element* declaration or *Complex Type* definition construct in the XML Schema meta-model. At level M1 there are two possibilities for mapping each class in the source model: either to an element declaration or to a complex type. These possibilities can be combined in alternative mappings that produce alternative XML schemas.

As a case study for this transformation problem we consider a system that supports teachers in preparing examinations. Examinations take the form of questionnaires answered by students. Figure 3.3 shows an UML model of the questionnaires.

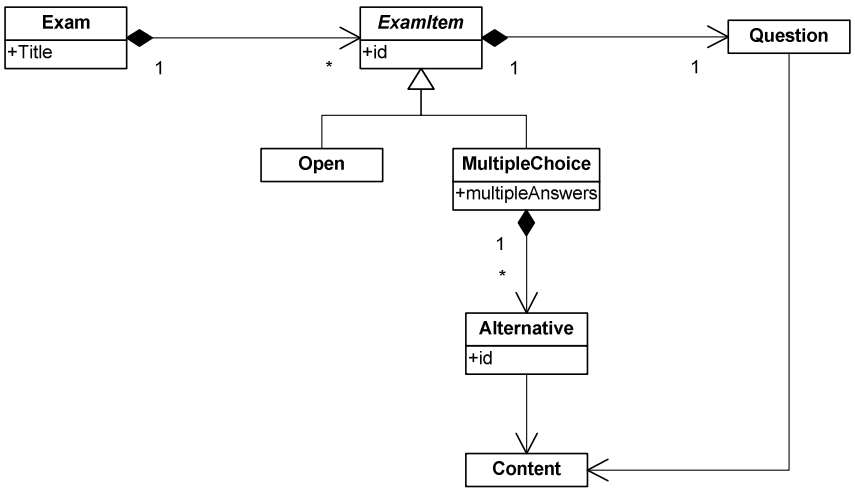


Figure 3.3 Example source model: UML class model of examination questionnaires

Class *Exam* is used to represent examinations and contains zero or more exam items. Each exam item has exactly one question. There are two types of exam items: open and multiple-choice. In the open type, the student may give any answer. The multiple-choice type requires selection from a list of alternative answers. Class *Content* is used to describe the format of the questions and answers and can be expressed in any combination of text, images, audio, etc.

This model can be implemented using different techniques. As an example, we assume that the examination documents are stored as XML documents and validated by an XML schema. The schema is derived by transforming the model shown in Figure 3.3 to a target model that is an instance of the XML Schema meta-model. The actual schema expressed in XML syntax is derived from the target model.

In our example, since new exam item types are expected as specializations of class *ExamItem*, the schema should be able to change in the future to incorporate these new exam items. Therefore, we aim at extensible schemas that remain intact even after new items are introduced.

Section 3.2.2 describes the problems encountered in transforming UML models to XML schemas.

3.2.2 Problems in Transforming UML Class Model into XML Schema

In order to transform the UML model in Figure 3.3 to an XML schema we need to map every construct in the model, i.e. classes, attributes and relationships to an appropriate construct available in the XML Schema meta-model. There is no standard XML Schema meta-model expressed in MOF but some proposals already exist [79]. Also, the abstract schema data model from [106] may be used.

We can identify in an ad-hoc way some transformations that produce alternative XML schemas. Fragments from three schemas are shown below. For simplicity they only show how classes *ExamItem*, *Open* and *MultipleChoice* are transformed.

In the first alternative schema (Figure 3.4) classes *ExamItem*, *Open* and *MultipleChoice* are transformed to complex types and elements of these types (the complex types for *Open* and *MultipleChoice* are not shown).

```
<element name='examItem'>
  <complexType>
     <sequence>
       <choice>
          <element name='open' type='…'/>
          <element name='mutipleChoice' type='…'/>
       </choice>
       <element name='question' type='…'/>
     </sequence>
  </complexType>
</element>
```

Figure 3.4 First alternative XML schema

The generalization relation is mapped to containment based on the choice content model. The problem with this schema is that it is not extensible. Additions of new exam item types require additions of new elements. The new elements must be included in the content model of the complex type of *examItem* element. This type, however, is based on the *choice* content model that does not allow additive changes through the operations in

XML Schema. Therefore, this type must be entirely regenerated every time a change is made.

In the second alternative schema (Figure 3.5) *ExamItem* class is mapped to an element and a complex type. The two specialized classes *Open* and *MultipleChoice* are mapped to complex types derived by extension from the complex type of exam item.

```
<element name='examItem' type='examItemType'/>

<complexType name='examItemType'>
    ……
</complexType>

<complexType name='openType'>
  <complexContent>
      <extension base='examItemType'>
       ……
      </extension>
  </complexContent>
</complexType>

<complexType name='multipleChoiceType'>
  <complexContent>
      <extension base='examItemType'>
       ……
      </extension>
  </complexContent>
</complexType>
```

Figure 3.5 Second alternative XML schema

This schema is extensible and can handle additions of new exam item types. If such an addition occurs then one complex type will be added as an extension of e*xamItemType*. XML documents will use the predefined attribute *xsi:type* to indicate the complex type that must be used to validate the content of the exam item. In this way the content model of *Exam* remains unchanged and all the changes in the schema are additive.

The third alternative schema (Figure 3.6) declares complex types and elements for every exam item type.

```
<element name='examItem'
         type='examItemType'
         abstract='true'/>

<complexType name='examItemType'>
    ……
</complexType>

<complexType name='openType'>
  <complexContent>
      <extension base='examItemType'>
       ……
      </extension>
  </complexContent>
</complexType>

<complexType name='multipleChoiceType'>
  <complexContent>
      <extension base='examItemType'>
       ……
      </extension>
  </complexContent>
</complexType>
```

```
<element name='open'
         type='openType'
         substitutionGroup='examItem'/>

<element name='multipleChoice'
         type='multipleChoiceType'
         substitutionGroup='examItem'/>
```

Figure 3.6 Third alternative XML schema

Elements are organized in a substitution group. This schema is also extensible but now the extensibility is achieved by the substitution group mechanism. New complex type and element will be added in case of addition of a new exam item type. The new element will be related via a substitution with the element for exam items. All the changes are additive and the initial set of schema components remain unchanged.

These alternatives show that even for a simple source model there are several possible target models. The first problem that we would like to address in this chapter is the difficulty in selecting among the alternatives. It is rarely the case that every alternative provides the desired solution. Usually some quality requirements must be fulfilled, in our example, the schemas must be extensible. Software engineers are often faced with such problems but usually the process of comparison among the alternatives is more implicit than explicit.

The second problem that we would like to address is that in current techniques there is no support for identification of alternative transformations. The transformation to the required schema may not be always trivial and obvious and therefore a systematic approach is required.

3.3 Alternatives Identification in a MDE based Development Process

At the moment there is no standard widely accepted MDE-based development process. The previous section motivated the need for an explicit activity of identification and selection of alternative transformations for a given source model. This activity can be applied at various stages of software development. We use the Software Process Engineering Metamodel (SPEM) [81] as a reference model to describe the characteristics of such an activity of identification and selection of alternative transformations.

SPEM is a metamodel proposed by OMG used to describe concrete software development processes. A given MDE based development process (or a part of it) may be considered as an instance of SPEM. We will identify the meta-concepts from SPEM that describe alternative transformations analysis.

According to SPEM a software development process is defined as a collaboration between abstract active entities called *process roles* that perform operations called *activities* on concrete, tangible entities called *work products*. Activities consist of a number of *steps*. A *discipline* partitions activities within a process according to a common theme. For example, in the Unified Process [49] nine disciplines are described: Business Modeling, Requirement Management, Analysis and Design, Implementation, Test, Deployment, Project Management, Configuration and Change Management, and Environment.

In terms of SPEM, the MDE work products are models and transformation definitions. Models may be further classified according to the level they occupy in a meta-modeling

architecture (meta-metamodel, metamodel, and model). Source code is also a work product and can be considered as a model. Transformation definitions are also models. It is beyond the scope of this chapter to give a full classification and description of the work products in MDE.

Development in MDE is generally organized around application of the transformation pattern. The SPEM activities are therefore obtaining the source models, obtaining the transformation definitions, and obtaining the target model. These activities can be grouped into disciplines around the MDE levels.

The problem with identification of alternative transformations may be captured in an activity according to the SPEM terminology. This activity is refined into several steps. The technique that we apply to identify alternative transformations is based on the notion of a *transformation space*. Therefore, transformation space is a work product for the identification of alternative transformations activity. The output of this activity can be used as one of the inputs for another activity that produces the actual transformation definition.

The notion of transformation space and the steps in the activity are explained in the next section.

3.4 The Notion of Transformation Space

In this section we define the concept of *transformation space* and the operations defined for transformation spaces. Transformation space is the main work product in the activity of identification and selection of transformation alternatives for a given source model. We present an activity diagram that describes the actions in this activity.

3.4.1 Definition of Transformation Space

A transformation space is built with concepts similar to the concepts found in Design Algebra [5]. In the context of Design Algebra a design space is a set of alternatives for a given design problem.

A *transformation space* is a multidimensional space spanned over a number of independent *dimensions*. Each dimension is associated with a number of *coordinates* that form a *coordinate set*. Every *point* in the space represents a transformation that produces a model that is an instance of the target meta-model.

To illustrate the concept of transformation space we use the example given in Figure 3.1. Assume we have a source model that contains model elements *a1, a2, a2,…, an*. We also assume that the model is succinct, that is, all the model elements are necessary and orthogonal as much as possible.

Every model element in the source model defines one *dimension* in the transformation space. A model element that defines a dimension is always an instance of a construct from the source meta-model. The *coordinate set* for that dimension is determined on the base of the construct from the source meta-model. For this construct a set of constructs from the target meta-model is determined. The construct from the source meta-model may be mapped to every construct in this set. This set is used to form the coordinate set for the dimension.

Consider the element *a1* in the source model in Figure 3.1. We define a dimension named *a1* for that element. Element *a1* is an instance of construct *A1* in the source meta-model. As Figure 3.1 suggests *A1* may be mapped to constructs *B1*, *B2*, and *B3* in the target meta-model. Therefore, we define the coordinate set {*B1*, *B2*, *B3*} for the dimension *a1*. The other two model elements *a2* and *a3* also introduce dimensions in the space. Since *a2* is also an instance of *A1* we have the same coordinate set for the dimension *a2*. Assume that construct *A2* in the source meta-model can be mapped to one of constructs *B2*, *B3*, and *B4*. Therefore, the coordinate set for the dimension *a3* is {*B2*, *B3*, *B4*}. The defined transformation space for the source model in Figure 3.1 is shown in Figure 3.7. The space is indicated with gray color.

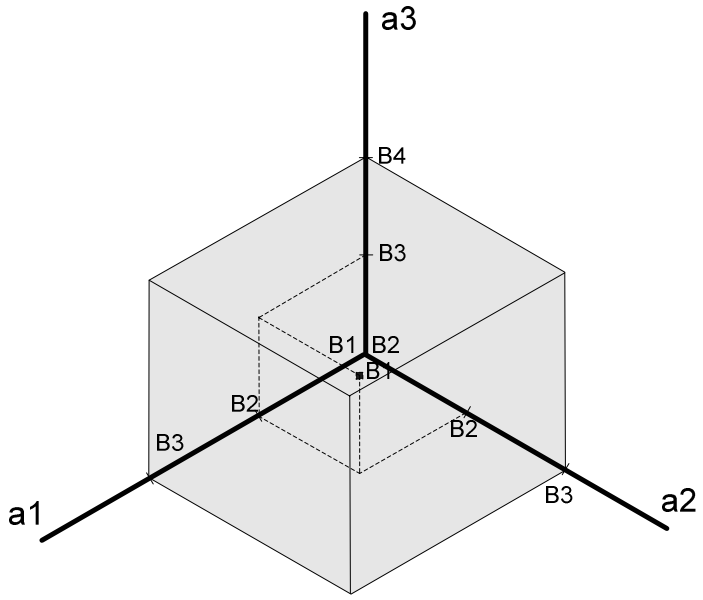


Figure 3.7 Transformation space for the source model in Figure 3.1. There are 3 dimensions: a1, a2, and a3. Dimension a1 has coordinate set {B1, B2, B3}, dimension a2 has coordinate set {B1, B2, B3}, and dimension a3 has coordinate set {B2, B3, B4}

A point in a transformation space *S* is formally represented as a tuple with components for every dimension and the coordinate in that dimension:

```
(d1.cd1, d2.cd2,…, dn.cdn)                                    (1)
```

where *di* is the name of the dimension and *cdi* is the coordinate of the point on that dimension.

Points in a transformation space are interpreted as alternative transformations of the source model. For every source construct the target construct is an instance of the coordinate of a point over the dimension corresponding to that source construct. Figure 3.7 shows a point represented as small black rectangle with coordinates `(a1.B2, a2.B2, a3.B3)`. This point corresponds to the target model shown in Figure 3.1. Indeed, elements *b1* and *b2* are instances of *B2*. *b3* is an instance of *B3*.

We define two functions that will be used in the rest of the chapter: `dimensions(S)` and `coordinateSet(dimension, S)`. The first function returns the set of dimensions

for a given space *S* and the second returns the coordinate set for a given *dimension* in a given space *S*. For our example space *S* we have:

```
dimensions(S)={a1, a2, a3}
coordinateSet(a1, S)={B1, B2, B3}
coordinateSet(a2, S)={B1, B2, B3}
coordinateSet(a1, S)={B2, B3, B4}
```

The number of alternatives for a space *S* with *n* dimensions is calculated with the formula:

```
numAlternatives(S)=numCoordinates(D1) *…* numCoordinates(Dn)
                                                              (2)
```

Here *numAlternatives* and *numCoordinates* are functions defined over a space and dimensions from that space respectively. *Di* denotes a dimension in the space *S*. *num Alternatives* returns the number of alternatives in a space and *numCoordinates* returns the number of coordinates for a given dimension. The number of alternatives for our example space is:

```
numAlternatives(S)=numCoordinates(a1)
                   *numCoordinates(a2)
                   *numCoordinates(a2)=3*3*3=27
```

Since a transformation space may be too large some operations are defined to reduce the space. Operations are for *selection* and *exclusion* from transformation spaces. They are explained in section 3.5.2 on the base of an example.

The required quality characteristics of the target model are represented in a quality model. The concepts from the quality model are merged with the source model elements and they indicate some characteristics that the target model must possess. Operation *merge* that merges transformation spaces is defined for this purpose. It is explained in section 3.5.2.

3.4.2 Activity for Identification and Selection of Alternative Transformations on the base of Transformation Spaces

In this section we describe the process of constructing and utilizing transformation spaces for a given source model. As we discussed in section 3.3 this process takes the form of an activity which is a part of an MDE-based development process. This activity takes a source model, its meta-model and the meta-model of target models as input and generates a transformation space for the source model. Furthermore, transformation definitions may be specified on the base of this transformation space. Figure 3.8 shows an activity diagram that describes the process of using transformation spaces for identification and selection of alternative transformations for a given source model.

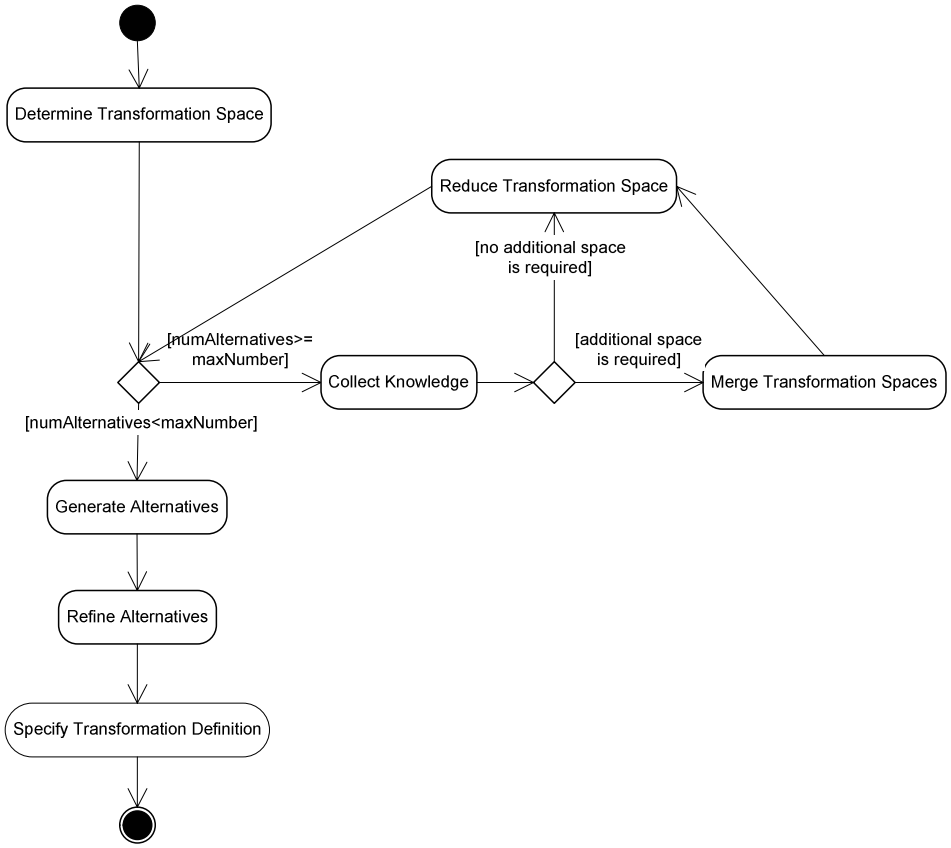


Figure 3.8 Activity diagram of the process for constructing and utilizing transformation spaces

The following actions are defined:

Determine Transformation Space

This action determines a transformation space based on a given source model, its meta-model and the meta-model of target models. A set of dimensions and coordinate sets for each dimension are identified. This action does not generate the total space of alternatives. The space provides a framework for reasoning about alternatives. An example how this action is performed is given in section 3.5.1.

Generate Alternatives

This action may follow the previous action if the number of the alternatives in the constructed space is less than a given number *maxNumber*. As a result all the alternatives in the space are generated and used in *Refine Alternatives* action. However, the number of alternatives in a newly created space is usually large. Therefore, the space has to be reduced sufficiently before generating the alternatives.

Reduce Transformation Space

This action is performed in case of large transformation spaces. Operations for *selection* and *exclusion* are defined to reduce spaces. They are explained in section 3.5.2.

Collect Knowledge

This action is performed before the action of reducing a transformation space. Reduction of a space requires knowledge for specifying selection criteria. Sometimes, the criteria may be based on properties expressed in a new transformation space. Usually this space expresses the quality properties of the target model.

Merge Transformation Spaces

In this action, a new space that may be generated in action *Collect Knowledge* is merged with an existing space. Merging is supported by operation *merge*. An example of performing this action is given in section 3.5.3.

Refine Alternatives

Once the space is sufficiently reduced the alternatives are generated by *Generate Alternatives* action. Alternatives do not have to represent a complete transformation definition and some additional tuning is required. The result of this action contains enough information for the specification of a transformation definition in a given language. This is described in section 3.5.4.

3.5 Constructing and Utilizing Transformation Spaces

In this section we apply the activity diagram in Figure 3.8 to select alternative transformations for the case study of examination documents described in section 3.2.1.

3.5.1 Determining Transformation Space

We will describe the process of constructing transformation space $S_{ExamSchema}$ for the example model in Figure 3.3. For brevity we decide not to consider transformation alternatives for the attributes of the classes. Also, for the sake of brevity only classes *Exam*, *ExamItem*, *Open* and *MultipleChoice* are considered together with the relations among them. This simplification does not affect the illustration of the basic ideas behind the formalism we are describing.

For each class and relation we define one dimension:

```
dimensions(S_ExamSchema)={Exam, ExamItem, Open, MultipleChoice,
                          Exam_ExamItem, ExamItem_Open,
                          ExamItem_MultiChoice}                    (3)
```

In (3) *Exam*, *ExamItem*, *Open*, and *MultipleChoice* are dimensions derived from the classes with the same name and *Exam_ExamItem*, *ExamItem_Open*, and *ExamItem_MultiChoice* are dimensions derived from the relations that connect the classes used to form the name of the dimension.

In our example the classes in the source model are instances of *Class* construct defined in the UML meta-model and the relations are instances of *Generalization* and *Association* constructs respectively. Coordinate sets for the dimensions are determined on the base of the target constructs defined in the target meta-model, in that example - the XML Schema meta-model.

Despite that there is no standard meta-model for XML Schema a set of constructs may be identified on the base of the W3C Schema specification [106]. We will use an XML Schema meta-model defined as a set of components and a set of relations among them:

```
XMLSchemaMetaModel = (C, R)                                    (4)
```

where *C* is a set of components and *R* is a set of relations. The set *C* has the following elements:

```
C = {CT, ST, E, A, AG, MG}                                     (5)
```

The elements of *C* correspond to the XML Schema abstract data model components Complex Type Definition (*CT*), Simple Type Definition (*ST*), Element Declaration (*E*), Attribute Declaration (*A*), Attribute Group (*AG*) and Model Group Definition (*MG*). Complex Type and Element Declaration have a Boolean property *abstract*. It indicates whether the type or element is abstract. The other components defined in the XML Schema abstract data model such as particles, wildcards, and identity constraints are not included here.

The set of relations *R* has the following elements:

```
R = {Der, Subst, Cont, Ref}                                    (6)
```

All relations are binary. Derivation (*Der*) corresponds either to extension or restriction mechanisms over schema types as defined in the specification. Substitution (*Subst*) relates two elements and corresponds to the element substitution mechanism. Containment (*Cont*) denotes participation of one component in the content model of the other. Element-subelement relation and the ownership of elements on their attributes are examples of containment. Reference (*Ref*) relation is based on the usage of elements or attributes of types ID and IDREF(S) in the content model of the related components.

Apart from the components and relations a set of constraints may be defined that restricts the allowed relations between two components. Constraints are shown in Table 3.1. The columns and rows represent components. Table cells contain the allowed relations between a pair of components.

From\To	**CT**	**ST**	**E**	**A**	**MG**	**AG**
CT	Der, Cont, Ref	Der, Cont	Der, Cont, Ref	Der, Cont	Cont, Ref	Cont, Ref
ST		Der	Der	Der		
E	Der, Cont, Ref	Der, Cont	Der, Cont, Ref, Subst	Der, Cont	Cont, Ref	Cont, Ref
A		Der	Der	Der		
MG	Cont, Ref	Cont	Cont, Ref		Cont, Ref	Ref
AG	Ref	Cont	Ref	Cont	Ref	Cont, Ref

Table 3.1 Constraints over possible relations among XML Schema components

Now we will identify the coordinate sets for the dimensions defined in (3). Assume that the software engineer decides that *Class* construct can be mapped to the components defined in (5). Some constraints that are derived from the nature of the technologies (UML and XML Schema) are applicable. For example, simple type (*ST*) can be excluded because it represents simple values. Classes are usually structures and it is not feasible to map

them to simple values. The same reason leads to exclusion of attribute declaration (*A*) as a possible target. The remaining four components form a coordinate set that can be attached to the dimensions associated with the classes in the source model:

$$\begin{aligned}
&\texttt{coordinateSet(Exam, } S_{\texttt{ExamSchema}}\texttt{)=\{CT, E, MG, AG\}}\\
&\texttt{coordinateSet(ExamItem, } S_{\texttt{ExamSchema}}\texttt{)=\{CT, E, MG, AG\}}\\
&\texttt{coordinateSet(Open, } S_{\texttt{ExamSchema}}\texttt{)=\{CT, E, MG, AG\}}\\
&\texttt{coordinateSet(MultipleChoice, } S_{\texttt{ExamSchema}}\texttt{)=\{CT, E, MG, AG\}} \qquad (7)
\end{aligned}$$

Assume now that the software engineer assigns the set *R* of XML Schema relations as coordinate set for the dimensions derived from the relations in the source model. Also, XLink standard [28] uses elements to encode relations and therefore element declaration is also considered as a possible coordinate. The following three coordinate sets are defined for the dimensions corresponding to the relations in the source model:

$$\begin{aligned}
&\texttt{coordinateSet(Exam_ExamItem, } S_{\texttt{ExamSchema}}\texttt{)=\{Der, Subst, Cont,}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{Ref, E\}}\\
&\texttt{coordinateSet(ExamItem_Open, } S_{\texttt{ExamSchema}}\texttt{)=\{Der, Subst, Cont,}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{Ref, E\}}\\
&\texttt{coordinateSet(ExamItem_MultiChoice,} S_{\texttt{ExamSchema}}\texttt{)=\{Der,Subst,Cont,}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\texttt{Ref,E\}}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (8)
\end{aligned}$$

Each point in the transformation space $S_{ExamSchema}$ represents a transformation from the UML model in Figure 3.3 to an XML schema. An additional requirement is that the transformation must produce a schema that satisfies the constraints from Table 3.1.

Figure 3.9 shows a graphical representation of a part of the space $S_{ExamSchema}$ where only the dimensions for classes *Open*, *ExamItem* and the relation between them *ExamItem_Open* are shown together with the coordinates defined in (7) and (8).

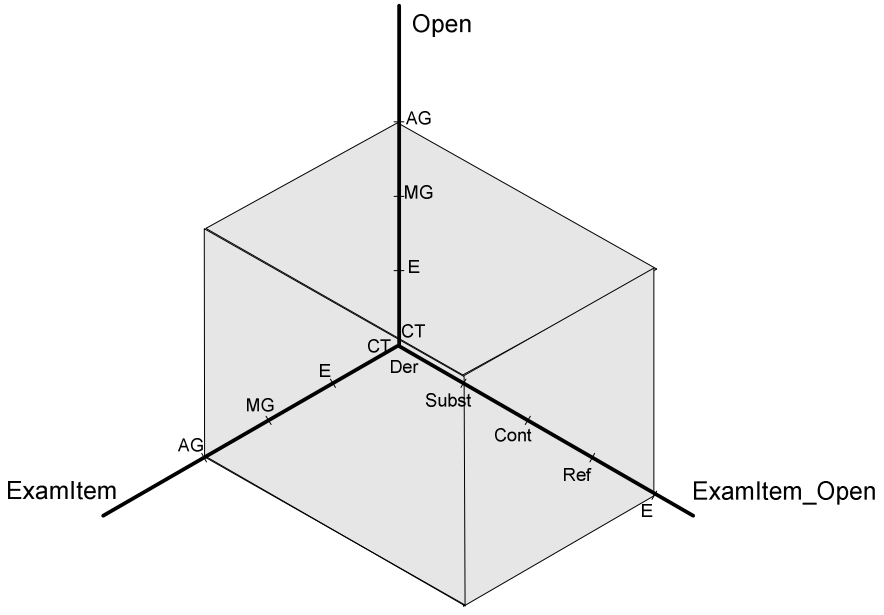


Figure 3.9 Transformation space with three dimensions corresponding to classes ExamItem and Open and the generalization relation between them

3.5.2 Reducing Transformation Spaces

It is possible to generate all the alternatives in a transformation space and to compare them. However, the number of alternatives is usually large. The application of the formula (2) to the transformation space $S_{ExamSchema}$ gives 4 * 4 * 4 * 4 * 5 * 5 * 5= 32000 theoretically possible alternatives which is a large number even for that simple source model of 7 elements. Therefore it is unfeasible to generate the whole space of alternatives. Instead, the software engineer may reduce the space either by selecting or by excluding alternatives from the transformation space.

Two operations for *selection* and *exclusion* from a space are defined:

```
Select from S where <condition>                                    (9)

Exclude from S where <condition>                                   (10)
```

Operation *Select* selects from a given space *S* only the points that satisfy the *condition* whereas the operation *Exclude* excludes from the space *S* the points that satisfy the condition.

In our example the software engineer may decide that the classes in the source model are mapped either to an element declaration (*E*) or to a complex type definition (*CT*). This can be specified as a selection from the transformation space $S_{ExamSchema}$:

```
S_ReducedExamSchema1 = Select from S_ExamSchema where
                    <(ExamItem.E or ExamItem.CT) and
                    (Exam.E or Exam.CT) and
                    (Open.E or Open.CT) and
                    (MultipleChoice.E or MultipleChoice.CT)>   (11)
```

The space may be further reduced by excluding some alternatives for the relations. Assume that the software engineer decides to exclude the alternatives *Ref* and *E* for the dimensions that represent relations in the source model:

```
S_ReducedExamSchema2 = Exclude from S_ReducedExamSchema1 where
                    <(ExamItem_Open.Ref or ExamItem_Open.E)
                      and
                     (ExamItem_MultiChoice.Ref or
                      ExamItem_MultiChoice.E)
                      and
                     (Exam_ExamItem.Ref or Exam_ExamItem.E)>   (12)
```

After these operations the transformation space $S_{ReducedExamSchema2}$ contains 2 * 2 * 2 * 2 * 3 * 3 * 3 = 432 alternatives.

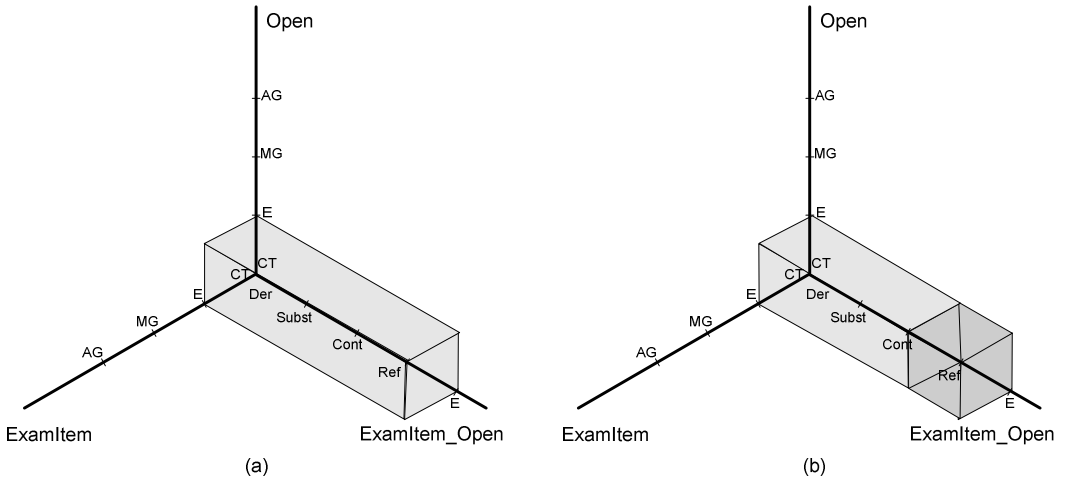


Figure 3.10 Transformation spaces after selection (a) and exclusion (b) operations

Figure 3.10a shows the space depicted in Figure 3.9 after the selection operation and Figure 3.10b shows the same space after the exclusion operation. Dark shaded area in Figure 3.10b shows the part that is excluded.

3.5.3 Reducing Transformation Space on the base of Quality Properties

Transformation space may be further reduced by considering the quality requirements that the target model must fulfill. In our example we aim at extensible schemas that preserve the content model of some schema components if new components are added as a result of extension of the source model. This indicates some quality properties that must be satisfied by the constructs in the target model.

Quality properties are derived from a quality model. In our example that model is a model of extensibility. This model classifies constructs as either *extensible* or *inextensible*. This classification is used to form a coordinate set with two coordinates: {Exts, InExts}. Here *Exts* indicates that a construct is extensible and *InExts* indicates that there is no explicit requirement for extensibility. A construct is extensible if the addition of certain components to the model is possible and does not cause changes in the construct.

We define a new transformation space $S_{ExtensibleExam}$ with the same set of dimensions as $S_{ReducedExamSchema2}$ and coordinate set {Exts, InExts} for each dimension. Assume that the software engineer decides to classify classes *ExamItem* and *Exam* as extensible and the rest of the constructs as inextensible. This means that if new exam item type is added as a specialization of *ExamItem* the content model of the corresponding target XML schema component remains unchanged and still can be reused by the specialization. Class *Exam* is considered extensible because the addition of a new exam item type must not affect the content model of the schema component to which *Exam* is mapped.

Based on this decision a selection operation is used to reduce the space $S_{ExtensibleExam}$:

$S_{ReducedExtensibleExam}$ = Select from $S_{ExtensibleExam}$ where
<ExamItem.Exts and Exam.Exts and
Open.InExts and MultipleChoice.InExts and
ExamItem Open.InExts and
ExamItem_MultiChoice.InExts and
Exam_ExamItem.InExts> (13)

The space $S_{ReducedExtensibleExam}$ identifies extensible constructs from the source model. Now we aim at target models that guarantee the extensibility property of the resulting constructs. Extensibility properties are integrated into the transformation space $S_{ReducedExamSchema2}$ using operation *merge* defined over two transformation spaces.

Operation *merge* has two transformation spaces *S1* and *S2* as arguments and returns a new space:

$$S_{merged}=merge(S1, S2) \quad (14)$$

where dimensions(Smerged)=dimensions(S1)=dimensions(S2).

The coordinate set of a dimension *d* in S_{merged} is a Cartesian product defined in the following way:

$$coordinateSet(d, S_{merged})=coordinateSet(d, S1) \times coordinateSet(d, S2)$$

The points from S_{merged} are defined on the base of the points from *S1* and *S2*:

$$S_{merged}=\{(d1.(p1', p1''),\ldots,dn.(pn', pn'')) \mid di \in dimensions(S_{merged}), pi' \in coordinateSet(di, S1), pi'' \in coordinateSet(di, S2), (d1.p1',\ldots, dn.pn') \in S1, (d1.p1'',\ldots, dn.pn'') \in S2 \text{ for } i=1,\ldots,n\}$$

Operation *merge* may be applied on the spaces $S_{ReducedExamSchema2}$ and $S_{ReducedExtensibleExam}$ to create a transformation space for extensible exam schemas $S_{ExtensibleExamSchema}$:

$$S_{ExtensibleExamSchema} = merge(S_{ReducedExamSchema2}, S_{ReducedExtensibleExam}) \quad (15)$$

Now the software engineer has the quality properties explicitly represented and can use selection criteria based on them. In the process of space reduction various knowledge sources may be used, for instance, heuristic rules. In our example, we identify the problem of extensible schemas as the problem of containers with variable content described in the XML Schema Best Practices [118]. The solution proposed there may be summarized in the following heuristic rule:

IF extensibility is required for a container class that
   aggregates other classes, which are specializations
   of a common general class,
THEN define an element (E) for the container class,
   define an element (E) for each specialized class,
   define an abstract element ($E_{abstract=true}$) for the
   general class,
   define substitution (Subst) between the general
   class element and each specialized element
   OR
   define an element (E) for the container class,
   define an element (E) for the general class,
   define complex type (CT) for each specialized class,

```
define an extension (Ext) between the type of the
general class and the type of each specialized
class
```

From this rule the software engineer may construct a condition used in a selection operation over $S_{ExtensibleExamSchema}$. The resulting space contains two alternatives shown in Table 3.2:

N	Exam	ExamItem	Open	MultipleChoice	ExamItem_Open	ExamItem_MultiChoice	Exam_ExamItem
1	E	E	E	E	Subst	Subst	Cont
2	E	E	CT	CT	Der	Der	Cont

Table 3.2 Two alternatives for extensible exam schemas

It can be noticed that the first alternative corresponds to the schema shown in Figure 3.6 and the second alternative corresponds to the schema shown in Figure 3.5. Both alternatives are extensible with respect to the expected changes.

3.5.4 Refinement

This is the last step of the process for construction and reduction of a transformation space. Once the space is sufficiently reduced the software engineer may generate the alternatives explicitly. However, these alternatives are not complete transformation definitions yet. Some additional details are required before the transformation definition is specified and executed. For instance, the element declaration components always require a type to be defined. This type may be named or anonymous. Substitution group relation between element declarations implies derivation between the corresponding element types. In addition, in the example presented here it was decided not to include the attributes of the classes in the transformation space. Certain decisions must be taken how the attributes are mapped.

Clearly, the software engineer must decide on some minor alternatives and to tune some additional properties. Once all the required details are provided the transformation definition may be specified in a given transformation language.

3.6 Discussion

We have presented techniques for construction and reduction of transformation spaces for a given source model. In this section we evaluate the approach with respect to the complexity of the transformation spaces and the applicability of that technique in the context of model transformations as defined in MDE.

Complexity of the Transformation Space

Although transformation spaces tend to be rather large even for simple models, they are purely conceptual. The software engineer does not need to generate the alternative transformations from a transformation space unless a number of reduction steps are applied and the size of the space is reduced sufficiently. The structure of a transformation space

specified by dimensions and coordinate sets provides a framework to reason about the alternatives in general instead of per alternative individually.

There is an analogy between the concepts of transformation space and relational schema used in relational databases. In case of a relational schema, for example, the user does not have to deal with the concrete data set of the corresponding database, which is usually dynamic. Instead, the user specifies queries based on the structure of the relational schema. Similarly, the concept of transformation space provides means to specify the conditions that can be used to reduce and select alternatives from a transformation space.

Apart from selection and exclusion, other techniques may be applied to reduce the complexity of a space. After the necessary reduction steps, the model may be decomposed into parts that are isolated from each other. In this case, alternatives in isolated parts do not influence each other. In the example presented here, we can assume that the content model of *Alternative* and *Question* is somehow encapsulated within the schema component. Then, the part of the source model comprised of classes *Question*, *Alternative*, and *Content* and the relations among them may be treated separately from the rest of the model. We are currently investigating the applicability of this idea and the required operational support for this.

Applicability of the Technique in the Context of MDE

The example shown in this chapter can be considered as a transformation from a platform independent model (UML) to a platform specific model (XML schema). The source model was rich enough and the mapping of the constructs was nearly one-to-one. The constraints in the target technology were leading in the selection of the alternatives. It appeared that this approach is applicable in this type of transformation.

Another transformation scenario is based on refinement of models where the target model contains more information compared to the source model. In this case the identification of the coordinate sets simply on the base of the target meta-model may not be sufficient. Some constructs in the source model may be expanded through patterns. Therefore, these patterns must be used as possible coordinates in the dimensions. In fact, the example showed such a situation. The reference relation is not a single component in the XML Schema. It is a pattern that involves pair of attributes/elements. As we saw the technique is able to cope with such a simple pattern. It is not clear, however, how more complex patterns can be incorporated in the technique.

3.7 Related Work

The work presented here is an adaptation of a formalism called Design Algebra [5][99] used for identification of design alternatives for a given design problem. From the perspective of model transformations Design Algebra supports the construction of a transformation space for models specified as abstract software solutions. The target meta-model contains the constructs found in the traditional object-oriented languages. In this chapter this technique is generalized to support transformations between arbitrary models in the context of MDE where the source and target models conform to given source and target meta-models that provide the information for construction of transformation spaces.

The problem of deriving XML schemas from UML class models described in the example is addressed in [20][26][33]. The authors identify the presence of multiple target

schemas that differ in their quality characteristics. In [20], generated schemas must ensure minimum data redundancy and maximum connectivity in the documents. This is achieved by a proper construction of the document hierarchy. Models are transformed by an algorithm based on 12 heuristic rules. The method described in [33] also aims at producing schemas that ensure minimum data redundancy in the documents. The authors give a formal definition for the minimum data redundancy property named canonical normal form for XML documents (XNF).

In these papers the problem of identification of the alternative transformations is not addressed. Instead, algorithms are defined that produce results with certain quality properties. We consider these contributions as complimentary to our work. The knowledge they provide can be incorporated and applied during the reduction process.

Our technique is also complimentary to existing transformation languages and can precede the specification and execution of transformations.

3.8 Conclusions

Transformation between models is a key operation in the MDE vision for software development. Generally one may adopt different but functionally equivalent target models for the same source model. Functionally equivalent target models, however, may differ from each other in the quality properties they possess. For example, one target model may be more extensible than the other target models. Since software engineers generally have to fulfill both functional and quality requirements, they should be able to identify and compare the quality properties of the functionally equivalent alternative target models for the same source model.

In this chapter we proposed a technique for identification of a set of transformations which form a transformation space for a given source model. This technique requires a source model, its meta-model and the meta-model of the target as input and generates a transformation space defined by the alternative target models as output. Reduction of transformation spaces is supported by the selection and exclusion operations. These operations can be parameterized by the quality attributes and be supported by heuristic rules. By this way, the software engineer is able to select the desired transformations from the transformation space.

We proposed that the analysis of alternative transformations for a source model should be included in a MDE based development process as an activity that operates on transformation spaces. From the point of view of the SPEM terminology transformation spaces are considered as work products in an MDE process.

The case study in this chapter showed the applicability of the approach in case of deriving implementation specific models from rich source models. Applicability of the approach in case of refinement of models needs further investigation. Currently, we are working on a prototype for a tool that supports software engineers in performing the activity of alternative transformations analysis.

4

A Model Transformation Language

This chapter presents a hybrid transformation language capable of expressing transformation definitions between source and target models created in different modeling languages following a variety of transformation scenarios. The basic characteristic of the language is the separation of transformation operations from the instantiation and generalization mechanisms defined for languages used to express models. The instantiation mechanism for a modeling language is represented in the environment in which the transformation language operates and is integrated with the transformation engine. The transformation language is capable of working with more than one instanceOf relation and more than one model level in a single transformation definition. This overcomes a major drawback in current transformation languages that are often dependent on particular modeling languages.

4.1 Introduction[1]

In Chapter 2 we observed that transformation languages used in MDE should be capable of working with models expressed in multiple modeling languages. Furthermore, a given model may be an instance of more than one intension. In this chapter we study in greater detail transformation scenarios that occur in the context of MDE. Since MOF is proposed by OMG as a standard for organizing models in MDE we consider scenarios in the context of MOF. Our goal is to provide examples of transformations on models written in

[1] This chapter is based on [59] and [60]

multiple modeling languages and to evaluate how MOF and the proposed transformation languages for MDE cope with the examples.

We analyze the problems that prevent the usage of a single transformation language for all the examples. Two problems are identified. The first problem is the inability of the MOF architecture to define explicitly the mechanisms of model instantiation for a given modeling language. The second problem is that currently proposed transformation languages are coupled with the instantiation mechanism of models they operate upon.

Apart from instantiation mechanisms, generalization relations also have an impact on transformation languages concerning selection of source model elements and the substitutability among values. Different modeling languages may have similar but different semantics of the generalization relation. The same coupling is observed between transformation languages and a given generalization relation.

To cope with these problems we propose a modeling space based on a uniform representation of all model elements no matter at which model level they are defined. The modeling space provides a framework for organizing models and modeling languages. This framework does not depend on MOF. MOF can be incorporated in the framework as a particular meta-modeling architecture. We do not introduce changes to any MOF-related standard. MOF, UML and other languages may be imported in the modeling space. In this modeling space the instantiation and generalization mechanisms are represented explicitly. We present a model transformation language called MISTRAL, which is separated from the instantiation and generalization mechanisms specific for a given modeling language. This transformation language is integrated with the proposed modeling space and is used to specify some of the features of modeling languages. If a transformation is defined between two models the transformation engine is configured with the definitions of the corresponding instantiation and generalization relations. Thus, the transformation language is decoupled from these relations and is able to express transformations between models at arbitrary level.

The chapter is organized as follows. Section 4.2 presents transformation scenarios observed in the MOF architecture and explains how they are handled by the current transformation technologies. Section 4.3 gives detailed description of the problems we want to tackle. Section 4.4 describes the modeling space and how models and languages are represented in this space. Section 4.5 presents transformation language MISTRAL, the algorithm for transformation execution, and the design of a prototype transformation engine. Section 4.6 shows an example specification of instantiation mechanism for the relational model. Section 4.7 evaluates the transformation language and Section 4.8 gives conclusions.

4.2 Motivation

A number of model transformation scenarios can be identified in current OMG standards and other publications [76][74][79][90][50]. Figure 4.1 shows four examples of transformation scenarios in the context of the MOF meta-modeling architecture.

Figure 4.1a shows a scenario with a transformation defined for two MOF meta-models (UML and Java meta-models). This is the context of the Query/Views/Transformation (QVT) Request for Proposals issued by OMG [76]. In this context transformations are specified between models at level M2 and executed on models at level M1. Figure 4.1b

shows similar scenario shifted one level down. It involves transformation execution on data at level M0. The diagram shows a transformation specified between a concrete DTD (Document Type Definition) and a concrete relational schema. The execution of the transformation converts an XML document to a relational database. This scenario is common in data warehousing and is addressed in the Common Warehouse Metamodel (CWM) specification [74]. Other scenarios similar to this may be provided: migration from a relational database to object-relational database that requires data transformations, data transformations driven by database schema evolution, etc. The common characteristic of all these scenarios is that the transformations are executed on data at level M0.

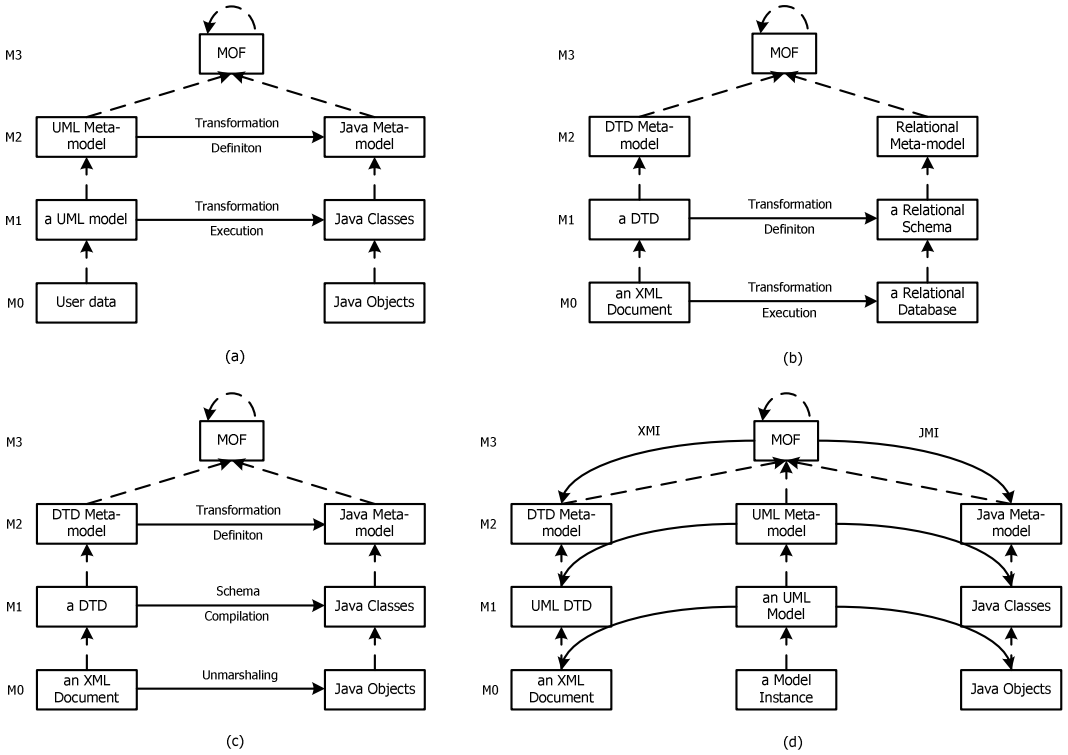


Figure 4.1 Examples of model transformation scenarios (a. intra-level transformation defined at M2 and executed at M1; b. intra-level transformation defined at M1 and executed at M0; c. intra-level transformation defined at M2 and executed at M1 and M0; d. inter-level transformation)

Figure 4.1c shows the Data Binding approach for XML processing [90] from the perspective of the MOF architecture. In this scenario a transformation is specified at level M2 and is executed at the two lower levels M1 and M0. The execution at level M1 is known as *schema compilation*. The correspondences derived between the constructs in the models at level M1 serve as a specification of the transformation executed at level M0 known as *unmarshaling*. In current data binding tools transformation rules applied during the unmarshaling are usually fixed and cannot express the correspondence between an arbitrary schema and an arbitrary set of application classes. In this respect XML processing can benefit from the ability of model transformation languages to express complex transformations [58].

The three scenarios shown so far may be regarded as intra-level transformations. Source and target models reside at the same model level. Another scenario is shown in Figure 4.1d. We refer to this scenario as inter-level transformation. Figure 4.1d shows two standard mappings for MOF: XML Metadata Interchange (XMI) [79] and Java Metadata Interchange (JMI) [50]. Both map the MOF Model at level M3 to a meta-model at level M2 (in the example these are DTD meta-model and Java meta-model respectively). These transformations are executed on models at level M2 (for example, the UML meta-model) and the result is a model at level M1. Furthermore, a UML model at level M1 may be transformed to an XML document or to a set of Java objects residing at level M0.

How do the current model transformation techniques support these scenarios? The QVT initiative aims at defining a standard transformation language for the first scenario. QVT languages are used for specification of transformation definitions on MOF models (at level M2). The execution of transformations is done on models at level M1.

CWM addresses the second scenario for a number of commonly used data sources: object-based, relational, record-based, multidimensional, and XML-based.

The third scenario is supported by proprietary data binding tools that do not consider transformations in the context of the MOF architecture.

The mappings in the fourth scenario are described in a semi-formal notation using grammars, templates and textual descriptions.

Although the transformation approaches taken in QVT and CWM share a number of similar concepts it is not possible to use a single transformation language for the first two scenarios. In the scenario in Figure 4.1c the result of a transformation at level M1 is used to derive a new transformation definition executed at level M0. Current model transformation languages do not address the problem of automatic derivation and execution of a transformation over more than two consecutive levels. Finally, both QVT and CWM do not consider inter-level transformations.

One would expect that the described scenarios could be addressed in a uniform way, that is, the organization of the MOF architecture would allow a transformation language to operate on models at any level. The analysis of the current transformation techniques reveals that the reality is different. In the next section we analyze the main problems preventing the usage of a single transformation language for all the scenarios.

4.3 The Instantiation and Generalization Problems

Transformation languages proposed for the QVT RFP are based on the instantiation mechanism used to create MOF meta-models and models (at level M2 and M1 respectively). Transformations select instances of MOF classes in a source model at level M1 and produce instances in a target model at the same level. Definition of a transformation language that works on every model at level M1 is possible because all model elements conform to the MOF semantics. MOF semantics defines which constructs at level M2 may be instantiated (instances of MOF *Classifier* that are not abstract) at level M1 and the structure of these instances (having *identity*, *slots* and *links*). MOF specification defines the meaning of the generalization relation: how features from a super-class are inherited in a sub-class and the rules for type substitutability based on the class hierarchy. Since the constructs at levels M3, M2 and M1 has a common structure and the models share the

same instantiation and generalization mechanism it is possible to define a transformation language that works on any model at level M1.

The MOF specification, however, does not specify the structure of the instances at level M0 and how they are related to their meta-constructs at level M1. The *instanceOf* relation between a construct at level M1 and its instances at level M0 may differ from the *instanceOf* relation between a construct at level M2 and its instances at level M1. This observation is made in [15] where it is argued that the actual number of levels is 3 instead of the widely accepted view of 4 levels. In fact, as we observed in Chapter 2, a model at level M2 defines a new language (e.g. UML, CWM, and Java) and this language brings its own definitions of the instantiation and generalization relations. These definitions are outside of the scope of the MOF specification. If a transformation is defined between two models at level M1 then the transformation language has to recognize which model elements are instantiatable and how the instances are populated with data. The lack of standard way to indicate the instantiatable constructs and instantiation mechanisms for model elements at level M1 prevents the execution of QVT transformation definitions at level M0. QVT languages consider models they operate upon as models expressed in MOF and therefore QVT languages are coupled with the MOF language.

How does the CWM deals with a variety of data sources such as XML, relational, and record-based? CWM takes the concepts of classes and instances defined in UML and reuses them in the CWM meta-model. A given meta-model that would be separately defined at level M2 is defined as a specialization of the CWM meta-model. Constructs that specialize *Class* construct can be instantiated and their instances conform to constructs that specialize *Object*. The problem here is the inability to handle models instances of meta-models at level M2 if the meta-models are not defined as specializations of the CWM meta-model.

If we aim at a transformation language that handles models expressed in different modeling languages then this language should be separated from any language-specific instantiation mechanism. For this purpose, it is required to study how transformation languages interact with the features of modeling languages. Identification of variabilities and commonalities in this interaction will allow transformation languages to be parameterized at the points of variance.

Furthermore, if a transformation language is capable of transforming models residing at arbitrary level, then it will require a common representation of the model elements and a uniform way of treating the different *instanceOf* and generalization relations. The discussion above showed that the MOF architecture does not provide these mechanisms. MOF misses a systematic approach for definition of modeling languages. Common language elements such as classes, types, generalization relations, etc. are not represented as first-class entities in MOF. In other words, MOF does not completely satisfy its purpose: to be modeling language for definition of other modeling languages. MOF should play the role of a domain specific modeling language whose domain is the domain of modeling languages. For historical reasons MOF emerged as a subset of UML and is more like general-purpose modeling language.

4.4 Approach

The approach for solving the problems explained in the previous section is based on two concepts. First, we propose a modeling space conforming to a simple generic model. The space hosts models and their modeling languages[1]. The space is independent of any concrete meta-modeling architecture and allows existing architectures to be incorporated in it. Second, we define a model transformation language named MISTRAL (Multiple Intension TRAnsformation Language) that operates on models in the modeling space. The language is defined after an analysis of common operations supported in existing transformation languages.

We consider five operations that occur in model transformations: *instantiation* of an element from an intensional construct, *selection* of source elements on the base of their intensional constructs, *querying* of features of model elements for their values, *setting* values to the features of model elements, and *deletion* of model elements.

We show how these operations are affected by the instantiation and generalization mechanisms. The specifics of the mechanisms are encapsulated in a set of functions used by the transformation engine to execute these five operations. The transformation engine is configured with the implementations of the functions before executing a transformation. In this way we achieve decoupling between the transformation language and the instantiation and generalization mechanisms of modeling languages.

4.4.1 Structure of the Modeling Space

Figure 4.2 shows an UML model of the modeling space.

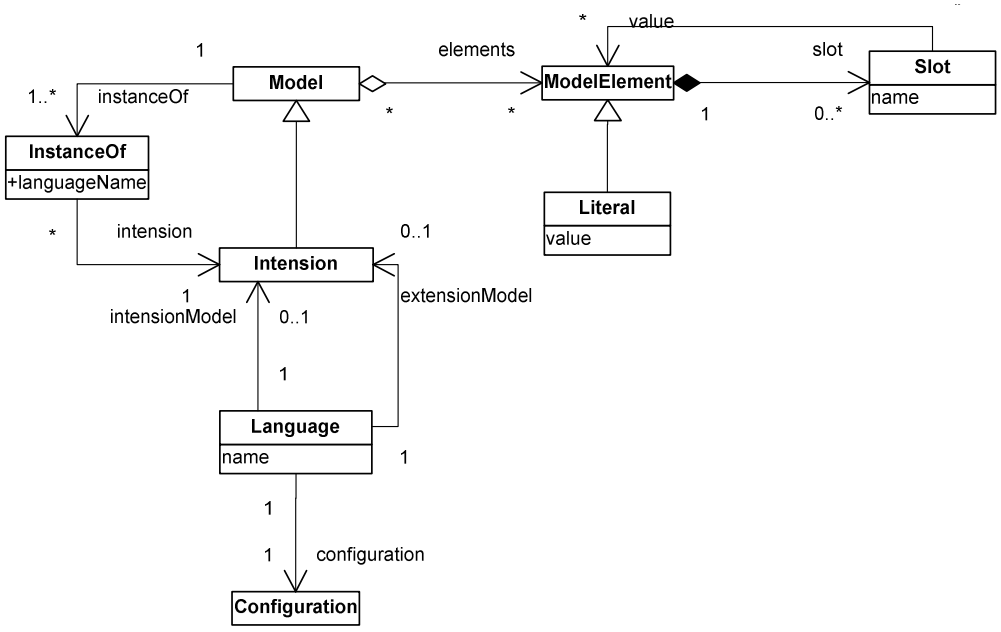


Figure 4.2 Generic model of the modeling space

[1] The modeling space defined in this chapter is different from the concept of modeling space introduced by Kent [52] and described in Chapter 2, Section 2.2.2.

The modeling space contains *models* and modeling *languages*. Some models are *intensions*. The concept of *intension* was introduced in Chapter 2, section 2.4.2. We briefly repeat the definition of intension: it is a model of other models. A meta-model is always an intension. A domain model is an intension of its instances. The concept of intension unites the concepts of meta-model and a domain model (which in general is not always a meta-model) in a single concept.

A model is always an instance of at least one intension. The concept of instantiation relation is represented as class *InstanceOf*. Instantiation relations are always defined in the context of a given modeling language indicated by the attribute *languageName*.

Modeling languages are represented by class *Language*. In Chapter 2 (Figure 2.7) we established that meta-models may be split into two parts: one defining the intensions that can be expressed in the language (e.g. UML models, Java classes) and one defining the members of extensions (e.g. UML objects, Java objects). As we saw in Chapter 2 this distinction is not always explicit and in many cases either of these parts is not described. Therefore, in Figure 4.2 we associate languages with two intensions: one for defining intensions (indicated by the role *intensionModel*) and one for defining members of extensions (indicated by the role *extensionModel*). These intensions are parts of the meta-model of a language. At least one of these intensions must be present for a given language. Languages are associated with *configurations*. A configuration provides information related to the language features required by the transformation engine. The meaning of configurations will be explained in details in section 4.4.5.

Every model is a collection of *model elements*. Every model element has an identity and a number of named *slots*. Simple values (strings, numbers, etc.) are instances of *Literal*, which is a specialization of *ModelElement*. The concept of slot used here is similar to the concepts with the same name defined in MOF and UML but we do not require that slots are instantiated from attributes. In our modeling space slots are used to connect model elements or to hold literal values.

The model in Figure 4.2 is represented in UML notation only for the purpose of readability. It can be described in some other notation such as Resource Description Framework (RDF) or can be formalized by using mathematical concepts such as sets and relations. It should be noted that the notion of meta-model and meta-level are not presented in this model. The two intensions associated to every language are the parts of the language meta-model. The modeling space is organized around the more general relation between a model and its intension. As we saw in Chapter 2 the relation between a model and its meta-model is an example of a relation between an intension and a member of its extension.

The model in Figure 4.2 will be used to implement the transformation engine for our transformation language. It expresses the view of the engine over models.

To give an example how languages and models are represented in the modeling space we use MOF meta-modeling architecture as example. In the next two sections two examples are given. Section 4.4.2 shows how the MOF Model is represented in the modeling space. The MOF Model does not have construct defining the *instanceOf* relation and the instantiation information is implicit. In our framework we make this explicit through a slot. The example in section 4.4.3 illustrates a language that defines *instanceOf* relation explicitly.

4.4.2 Example: Representation of the MOF Model

To incorporate the MOF architecture in the modeling space we introduce a language called *MOF* associated with an intension that is a representation of the MOF Model. MOF Model is instance of itself according to the instantiation mechanism defined for the MOF language. Figure 4.3 shows a simplified MOF model based on [75]. Primitive data types and the multiplicity of attributes and association ends are omitted for simplicity. We assume that all the associations are unidirectional. This model is already rich enough to illustrate the approach.

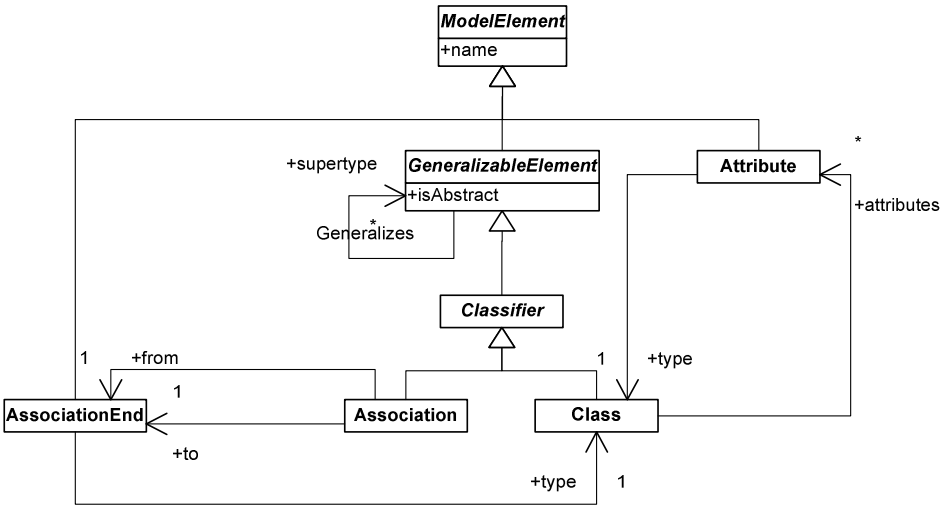


Figure 4.3 Simplified MOF model

The MOF model can be represented by the concepts of *ModelElement*, *Slot* and *Literal* in Figure 4.2. The following rule is used being close to the MOF instantiation mechanism: all instances of MOF *Class* construct and also their instances are represented as model elements. Instances of attributes and associations are represented as slots that connect the model elements. For simplicity, instances of associations are also represented as slots. The concept of *Link* is not used. An object diagram for a part of the representation of the MOF abstract class *ModelElement* is shown in Figure 4.4.

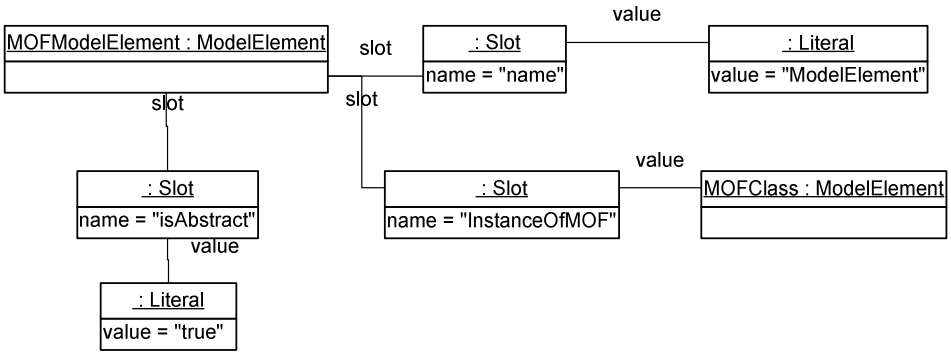


Figure 4.4 Object diagram representing MOF abstract class ModelElement as instance of the generic model in Figure 4.2

Class *ModelElement* from the MOF Model (named “MOFModelElement” in the object diagram) is an instance of class *ModelElement* from Figure 4.2. Class *Class* from the MOF Model is also an instance of class *ModelElement* named “MOFClass” in the object diagram. Instances of attributes *isAbstract*, and *name* are represented as slots. The slot named “InstanceOfMOF” indicates the instantiation relation to the MOF *Class* construct.

In this chapter a more concise notation will be used instead of object diagrams for showing models. Figure 4.5 shows the model in Figure 4.3 represented as a graph of model elements.

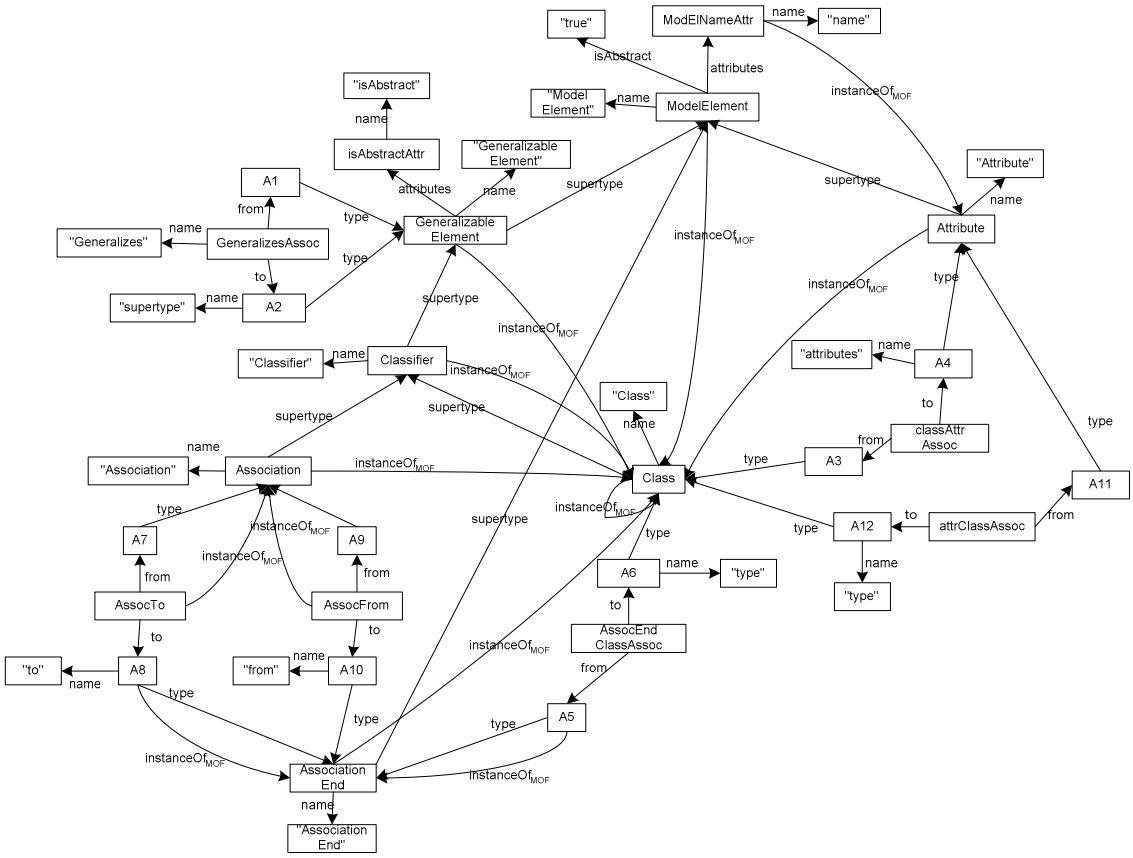


Figure 4.5 Representation of the MOF model (Figure 4.3) as a graph of generic model elements

Model elements are shown as rectangles that contain an identifier of the element. Usually this identifier is the value of the slot ‘name’ and here it is used only for illustrative purposes. Literals are also shown as rectangles containing the literal value enclosed by quotes. Slots are represented as arrows labeled with the name of the slot. Arrows are pointing from the slot holder to the slot values. If the value is a collection then multiple arrows with the same label are drawn. The slots that represent *instanceOf* relation are shown as arrows labeled ‘$\text{instanceOf}_{\text{MOF}}$’ from an element to its meta-construct. To improve readability we do not show the full representation of the MOF Model. Simple types and their relations to the literal elements and attributes are skipped. Slots ‘isAbstract’ and ‘$\text{instanceOf}_{\text{MOF}}$’ are also skipped for most elements.

In the MOF model there is no construct that defines instantiation relation. In our framework this relation is explicitly represented by a slot. From now on we will refer to that relation as $instanceOf_{MOF}$.

The MOF *instanceOf* Relation

We give a simplified definition of the constraints of the MOF *instanceOf* relation. These constraints are specified informally but any formal specification can be given based, for instance, on OCL or on first-order predicate logic. We do not specify the multiplicity constraints over the slot values.

The following notation is used. If *me* is a model element then *me.slots* returns the set of all slots of *me* excluding the *instanceOf* slot. For a particular slot with name 'n', the expression *me.n* evaluates to the value of the slot. Slot values are treated as sets. If *s* is a slot then *s.value* and *s.name* return the value and the name of the slot respectively.

Let *mI* and *mT* are model elements from the MOF Model and $mI.instanceOf_{MOF}=mT$. The following holds:

- $mT.instanceOf_{MOF}=Class$ and *mT.isAbstract=false.*

  Only non-abstract classes may be instantiated to model elements.

- For each $a \in mT.attributes$ there exist exactly one slot $s \in mI.slots$ such that *s.name=a.name* and for every $v \in s.value$ we have type substitutability (as defined in MOF) of $v.instanceOf_{MOF}$ with *a.type.*

The meaning is that for every attribute defined in the class there is exactly one slot in the instance with name equal to the name of the attribute.

- For each model element *assoc* for which $assoc.instanceOf_{MOF}=Association$ and *assoc.from.type=mT* there exist exactly one slot $s \in mI.slots$ such that *s.name=assoc.to.name* and for every $v \in s.value$ we have type substitutability (as defined in MOF) of $v.instanceOf_{MOF}$ with *assoc.to.type.*

The meaning is that for every outgoing association of the class there is exactly one slot in the instance with name equal to the name of the target association end. For simplicity we have chosen to create a slot only for one direction of the association.

It is possible to check that the graph in Figure 4.5 satisfies these constraints. The constraints also show how to identify the names and types of the slots of model elements.

4.4.3 Example: Representation for the Relational Model

As second example we represent the relational model as a meta-model expressed in MOF. Some approaches [13][15] reduce the number of levels in the MOF architecture to 3 by defining a given meta-model and the model of the M0 instances at the same level M2. Therefore the models and their instances are situated in level M1 and instantiated with the MOF mechanism. The reduction of the levels, however, does not remove the presence of the second *instanceOf* relation defined for the concrete modeling language. Atkinson and Kuhne [13] identify the existence of these distinct *instanceOf* relations and distinguish between *linguistic* and *ontological* instantiations. In our examples $instanceOf_{MOF}$ relation

is the linguistic instantiation whereas the instantiation relation defined for a given modeling language is the ontological instantiation.

By defining the relational model we introduce a new language in our modeling space. Its meta-model contains two sub-models that model the intensional and extensional parts respectively. The meta-model is expressed in the MOF language (Figure 4.6).

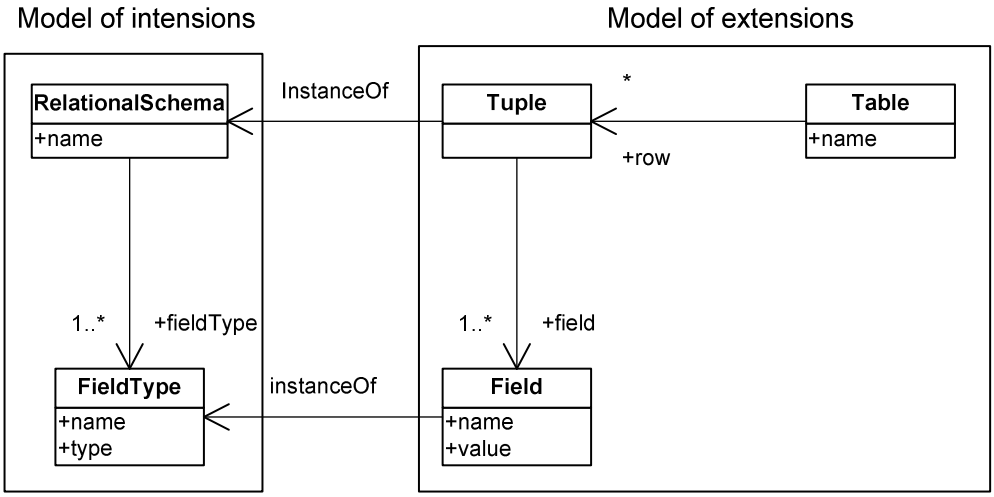


Figure 4.6 Relational schema model and its instance model

Figure 4.6 contains both the model of relational schemas (classes *RelationalSchema*, and *FieldType*) and the model of their instances (classes *Table*, *Tuple* and *Field*). The *instanceOf* relation is represented as an ordinary MOF association.

This model is an intension from the point of view of the MOF language. Therefore, the model may be instantiated through the $instanceOf_{MOF}$ mechanism and one example instance is shown in Figure 4.7. The data (the model elements *aTuple*, *f1*, and *f2*) that would reside in level M0 are now at level M1 and may be queried according to the elements *Tuple* and *Field*. The concrete schema *aSchema* and its field types *ft1* and *ft2* form an intension from the point of view of the relational model.

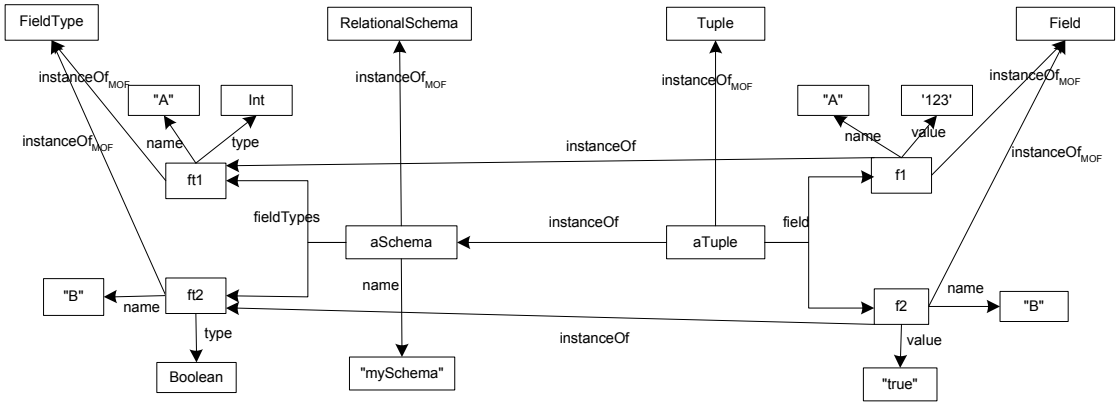


Figure 4.7 A particular relational schema and relational data represented as generic model elements

The tuple *aTuple* therefore conforms to two intensions via two different instantiation relations. It is possible to query this tuple via these two intensions. To access the value of field *B* one has to write the expression *aTuple.field→select(name="B").value*[1] that returns 'true'. Another way is to use the relational schema of the tuple (represented by *aSchema* model element) that defines the fields *A* and *B* and to write the expression *aTuple.B* that reflects the ontological instantiation relation between *aTuple* and *aSchema*. This querying is not possible because *aTuple* does not have slot with name "B" created by the linguistic $instanceOf_{MOF}$. Therefore, some navigation over the graph in Figure 4.7 should be specified to access the values of slots implied by the ontological instantiation. The ontological *instanceOf* relation is represented by the slots named 'InstanceOf' in Figure 4.7.

This example illustrates that both instantiation mechanisms and both intensions should be available. A single model element may conform to more than one intension through different *instanceOf* relations and these intensions may be used to query the slots of the model elements.

The intension defined as a relational schema usually represents an application domain. For example, it says that tuples in this domain have fields named *A* and *B*. The intension defined in the language meta-model provides a reflective view over tuples. It specifies that *all* tuples have fields and every field have a name and a value. Our framework allows explicit representation of more than one *instanceOf* relation and more than one intension for a given model element.

We summarize the examples by showing the state of the modeling space based on the languages and models defined until now. Figure 4.8 shows an object diagram with instances of classes defined in Figure 4.2.

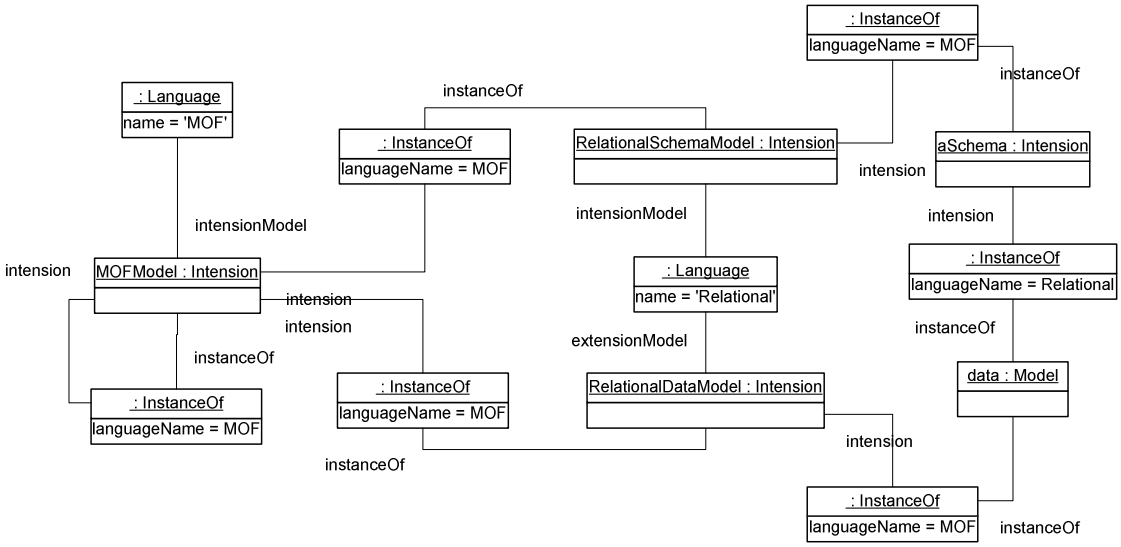


Figure 4.8 An example state of the modeling space

Two languages are shown in the figure named *MOF* and *Relational*. Intensions that constitute the definitions of the languages were already presented in this section. The model named *data* is an instance of the two intensions *RelationalDataModel* and *aSchema* via two different instantiation relations.

[1] OCL notation [80] is used to specify the expression.

4.4.4 Operations in Model Transformations

In this section we analyze five operations observed in model transformations: *selection* of model elements on the base of their intensional constructs, *instantiation* of model elements from an intensional construct, *accessing* the slot value of an element, *setting* a slot value, and *deletion* of a model element from a model. This list is not exhaustive. For instance, invocation of an operation of a model element is not considered. This chapter focuses only on the five operations mentioned above. Every operation is affected by the instantiation and generalization mechanisms specific for a modeling language. For every operation some functions are identified and used to perform the operation. Different languages provide different implementations of these functions.

Selection of model elements on the base of their intensional construct

In our framework *instanceOf* relations are explicitly represented. They are represented either by a slot that does not have a defining construct as in the example of the MOF model or by a slot instantiated from a construct in a meta-model as in the example of the relational model. Moreover, because multiple *instanceOf* relations are possible for a model element we may select on the base of more than one intensional construct. This helps in dealing with both linguistic and ontological instantiations.

With every modeling language we associate a function called *meta* defined over model elements in the models expressed in the language. The function returns an element from which its argument is instantiated:

```
meta(me: ModelElement): ModelElement
```

An implementation of that function in the context of *MOF* language would return the value of the slot $instanceOf_{MOF.}$ The implementation in the context of *Relational* language would return the value of the slot *instanceOf* (see Figure 4.7). It should be noted that the function is defined only for model elements. We assume that the information about the intensional constructs used to create slots can be derived from the instantiation mechanism.

Apart from selection on the base of the intensional construct another form of selection is possible. In many cases not only the instances of a given construct are selected but also the instances of the specializations of the construct are selected. This selection uses the generalization-specialization hierarchy in the intension being used. Sometimes there are more than one hierarchy (e.g. derivation by restriction and extension in XML Schema, extension among classes and extension among interfaces in Java). To model this situation and to enable the transformation language to deal with different generalization hierarchies we associate every language with a set of named transitive relations representing its generalization relations. Every relation is associated with a function that for a given element in an intension returns all the specialized elements (direct and indirect):

```
getSpecializedConstructs(me: ModelElement) : Set of ModelElement
```

Instantiation of model elements from an intensional construct

Instantiation mechanism is modeled as a function that takes a construct from an intension and produces a model element with empty slots:

```
instantiate(meta-construct: ModelElement) : ModelElement
```

The implementation of that function is influenced by the generalization mechanism defined for a given language since this mechanism defines the inheritance of features from more general constructs.

Accessing slot values of an element

The slots of a model element are derived from its intensional construct based on the instantiation mechanism. Slots derived by one instantiation mechanism may differ from the slots derived by other mechanism as in the example of the relational model. In this example, element *aTuple* in Figure 4.7 may be instantiated by two instantiation mechanisms. If the mechanism of *MOF* language is used then the slots of the element are named *field* and *instanceOf*. If the mechanism of *Relational* language is used then slots are named *A* and *B*. The model element, however, has a single representation. This representation is generated by the instantiation mechanism of *MOF* language. Slots are physically represented in the graph. Slots *A* and *B* derived by the second instantiation are virtual. They are result of a certain interpretation of the graph shown in Figure 4.7.

This observation raises two questions: first, what is the instantiation mechanism used to generate the representation of a model element and second, if the slots implied by an instantiation are not directly presented how their values are obtained.

The answer of the first question is that one instantiation mechanism is always chosen as default. In our example the models of languages are intensions expressed in *MOF* language. Their extension members are instantiated according to *MOF* language. We choose $instanceOf_{MOF}$ as default mechanism (or linguistic instantiation in terms of [13]).

To answer the second question we define a translation mechanism that obtains the slot values of slots derived by a given instantiation mechanism different from the default mechanism. This translation is implemented as a function that takes a model element and the name of a slot and returns the value of the slot, which is a set of model elements (recall that a slot value may be multi-value). This function is named *getSlotValue* and its signature is shown below:

```
getSlotValue(me: ModelElement, slotName: String) : Set of ModelElement
```

Setting values of a slot

This operation is reverse to the operation of accessing values of slots. The same two cases are presented here. If the slot exists in the representation of the element the value is set directly on the slot. If the slot is derived by an instantiation different from the default instantiation a translation mechanism is required. Setting the slot value is treated as an in-place transformation over the model element whose slot will be set.

Generalization mechanism affects this operation in respect to the compatibility of the type of the value being set and the expected type of the value. The rules for type substitutability must be known when the transformation engine performs type checking of the value. Apart from the type checking, the transformation engine has to perform check for multiplicity constraints.

Four functions are proposed to perform this operation. The first function sets the value of the slot by taking into account how the slot is represented. The signature of the function is shown below.

```
setSlotValue(me: ModelElement, slotName: String, slotValue: Set of ModelElement)
```

The function *setSlotValue* takes three arguments: the model element *me* whose slot value is set, the name of the slot *slotName* and the value *slotValue* represented as a set of model elements.

The second function is named *getSlotType* and returns the type of the slot with name *slotName* defined in *meta-construct*:

```
getSlotType(slotName: String, meta-construct: ModelElement) : ModelElement
```

The third function is named *getSlotMultiplicity*:

```
getSlotMultiplicity (slotName: String, meta-construct: ModelElement) : List
```

The function accepts the slot name and the defining meta-construct as arguments and returns a list of two integers that correspond to the lower and upper bounds respectively.

The fourth function checks if two model elements represent compatible types:

```
isCompatible(expectedType: ModelElement, actualtype: ModelElement): Boolean
```

The checking is performed according to the type rules for a given language.

Deletion of Model Elements

From the point of view of the model in Figure 4.2 deletion of a model element is simple. The element is removed from the modeling space and all the slots associated with it are also removed. If the model element has been a value of some slot of another model element then the model element is not a value of the slot anymore.

This simple approach is modeling language independent. However, its application may lead to a number of graphs of model elements that are not connected to other elements and are not used. For instance, if we delete the model element *aTuple* in Figure 4.7 then its fields *f1* and *f2* will remain in the model. The semantics of *Relational* language, however, governs that fields cannot exist independently from the containing tuple. Therefore, elements *f1* and *f2* have to also be deleted. The same is valid for the literal elements that contain the name and the value of the fields.

One approach to handle the problem is to provide a garbage collection in the transformation engine in a way similar to the Java virtual machine. Second approach is to consider deletion of model elements as a language dependent operation and to model it as a function similarly to the previously described operations. In the example, the deletion of a tuple should delete also the fields of that tuple. We take the second approach. This, however, puts a certain burden on the modeling language designer to specify deletion operation for the meta-constructs in the language.

We propose function *delete* that takes a model element and removes it from a given model.

```
delete(me: ModelElement)
```

This function may navigate through the model and delete other model elements before deleting its argument *me*.

Model Element Identifiers

In the transformation language we use identifiers to refer to model elements. For example, we need to specify model elements that are types of variables. In general, different modeling languages use different naming schemes to refer to the model elements. In UML the identifier of an element is based on the package hierarchy. In XML Schema it is possible

to have components with the same name but with different types. Therefore, the distinction is made on the base of the type of the component.

We use a generic syntax for identifiers that decomposes identifiers into parts delimited by double colon '::' similarly to the naming scheme employed by OCL. The algorithm for identifier resolution is language dependent. It is implemented in the function *lookup*:

```
lookup(identifier : String, model: Model) : ModelElement
```

An example identifier for an UML model could be *package1::package2::Class1*. In case of an XML schema an identifier would look like *complexType(MyType)::element(e)*. This identifier is resolved to the element declaration component with name 'e' defined in the scope of the top-level complex type with name 'MyType'.

4.4.5 The Structure of Language Configuration

Previous section shows that the five operations in model transformations are highly dependent on the instantiation and generalization mechanisms for a particular modeling language. In a more general perspective they reflect different aspects of the semantics of the modeling language. Unfortunately, there is no widely accepted way to specify the semantics of modeling languages in the context of MDE.

We model the information required by the transformation engine to perform these five operations as a set of functions. Every modeling language provides its own implementation of these functions. We call the set of these functions a *configuration*. The structure of configurations is shown in Figure 4.9.

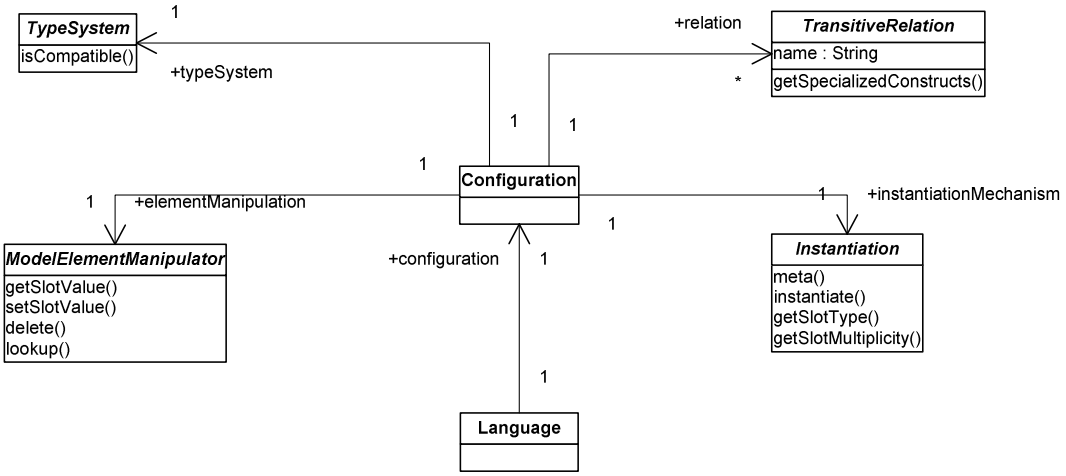


Figure 4.9 The structure of modeling language configurations

The functions presented in the previous section are represented as operations of abstract classes. There are four abstract classes associated with a configuration: *Instantiation*, *TypeSystem*, *ModelElementManipulator*, and *TransitiveRelation*.

Every modeling language has to provide concrete sub-classes in its configuration that implement the operations. Operations in a configuration may be implemented in any language and may be linked to the transformation engine as an external library. For illustrative purposes we will show how these operations can be implemented as transformation rules written in our transformation language. During the execution of a transformation the

engine will invoke these rules. Examples of the implementation of the configurations of *MOF* language and *Relational* language are given in Section 4.6.

4.5 Model Transformation Language MISTRAL

In this section we describe a model transformation language capable of specifying transformations between models written in various modeling languages. The language is a generalization of the one applied to XML processing described in [58]. Language features are presented by examples. The grammar of the language concrete syntax is in Appendix A. The abstract syntax is defined as a MOF meta-model and is given in Appendix B. Parts of the abstract syntax are used throughout this section.

4.5.1 Overview of the Language

Figure 4.10 shows the basic concepts in the transformation environment in which the transformation language is used.

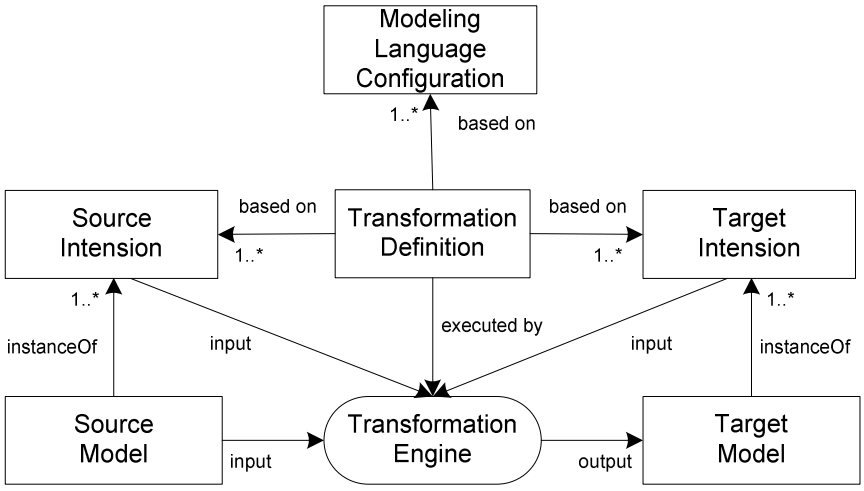


Figure 4.10 Overall structure of the transformation environment

The transformation engine transforms a source model to a target model by executing a transformation definition. Transformation definitions are based on the intensions of the source and the target models and use the configurations of the languages used to express the models. Source and target models must be associated to at least one intension. The *instanceOf* relations must be declared in the context of a given modeling language. The intensions are passed as input to the transformation engine. Language definitions and their configurations are also passed to the transformation engine (this is not shown in the diagram to avoid cluttering).

The transformation engine can only create instances of the constructs *ModelElement*, *Literal* and *Slot* shown in Figure 4.2. However, this is a transparent process from the point of view of a writer of transformation definitions. Transformation definitions are specified by using the constructs in the source and target intensions. Based on the configurations of

the languages the transformation engine performs the operations analyzed in the previous section that ultimately result in creating model elements and slots in the target model.

A transformation definition contains *declarations*, set of *rules* and a sequence of *transformation steps*. There are two types of rules: *model element rules* and *slot rules*. Model element rules select elements in the source model and execute actions. Actions create elements in the target model, update elements and delete elements. Slot rules are used to relate the elements by setting their slot values. Both types of rules have rule source that selects elements in the source model.

The elements of transformation definitions are explained in the remaining part of this section.

4.5.2 The Structure of Transformation Definitions

Transformation definitions form units called *modules*. Module is the packaging mechanism in the transformation language. A module may reuse another module by including its content. The including module may override rules defined in the included module.

Transformation modules have a name, a declaration part and a set of transformation rules organized in steps. A model of the structure of transformation modules is shown in Figure 4.11.

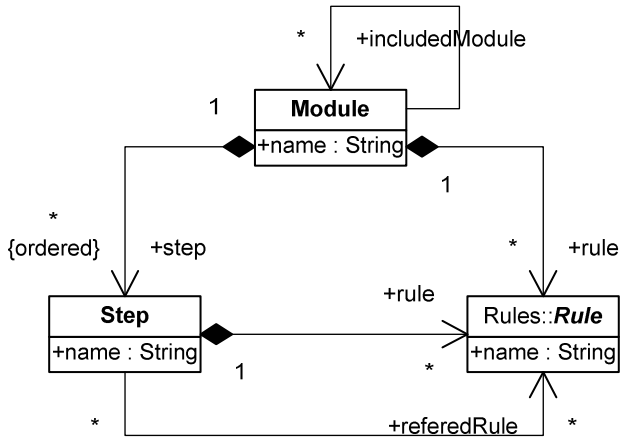


Figure 4.11 The structure of transformation modules

The overall structure of transformation definitions is given in the following example:

```
transformation transformationName
  include ….list of transformation definition names...
  languages ….list of language names…...
  …….declarations of models……..…..
  …….definitions of rules……..…..
  helperRules
  ……definitions of rules……..
  Step step_1 {…..rules in the step……}
  Step step_2{….. rules in the step……}
  ……
  Step step_n{….. rules in the step……}
```

The *include* clause enumerates names of transformation modules whose rules become a part of the including transformation. The including transformation module may define rules with the same names as the names of the included rules. In this case, the included rules are overridden.

The *languages* clause indicates the declaration part of a transformation module.

The *helperRules* clause defines rules that are invoked by other rules. The transformation engine does not consider helper rules while determining the execution order of rules.

The *step* clauses partition transformation rules into sets of rules that are executed sequentially.

Module Declaration Part

A simple example transformation module that contains only a declaration part is shown below. Keywords are in bold.

```
1.  transformation myFirstTransformation
2.    languages MOF
3.    UML instanceOf(MOF) MOF
4.    source mySourceModel instanceOf(MOF) UML default
5.    target  myTargetModel instanceOf(MOF) UML  default
```

Every transformation definition has a name (line 1). The declaration part of the transformation definition (lines 2-5) declares languages and models available to the transformation engine. In the example transformation, only one language is declared: MOF (line 2). Furthermore, three models are declared (lines 3-5). Every model is an instance of an intension according to the instantiation mechanism defined for a modeling language.

The first model is the meta-model of UML (line 3). It is an instance of MOF according to the MOF instantiation mechanism. The instantiation mechanism is specified in braces after the keyword *instanceOf* (e.g. instanceOf(MOF)).

Two more models are declared (lines 4 and 5). They are expressed in UML, however, the instantiation mechanism of MOF is still used for them. The first model *mySourceModel* (line 4) is declared as a *source* model. This means that it is the model that will be transformed to the target model. The target model is named *myTargetModel* and is declared as *target* (line 5). This model does not exist before the execution of the transformation. The MOF language will be used as default language in performing the model transformation operations. Recall that in Section 4.4.4 we concluded that these operations are language dependent. The keyword *default* is placed in the end of the declarations to indicate the usage of MOF as default language.

This first transformation definition declares models instances of only one intension. In a second example we show a transformation definition that declares models instances of more than one intension. The second definition is based on the example models in Section 4.4.

```
1.  transformation mySecondTransformation
2.    languages MOF, Relational
3.    aSchema instanceOf(MOF) RelationalSchemaModel
4.    source data instanceOf(MOF) RelationalDataModel
5.    data instanceOf(Relational) aSchema default
```

In this transformation definition the model named *data* (line 4) is an instance of two different intensions. The instantiation mechanism defined for the relational model will be used as a default mechanism.

Transformation Steps

Transformation rules may be distributed over a sequence of *transformation steps*. Steps are executed in the order of their specification. A step is executed if all the rules in it are executed. Within a step, however, there is no predefined order of execution among the transformation rules. How the execution order of rules is determined is described in section 4.5.8. If a transformation definition does not specify any steps then rules belong to one default step.

The syntax of steps is the following:

Step *stepName* { (*ModelElementRule* | *ruleName*)+ }

Every step has a name and a set of associated rules. Rules may be defined inline in the step or may be referred to by name. In the latter case the rule must already be defined in the transformation definition. Duplication of rule names in steps is not allowed.

Every rule has a source part that selects elements in the source model. The structure of source parts is explained in the next section.

4.5.3 Rule Source

Rule source specifies the characteristics of the elements in the source model that will be selected by a transformation rule. Rule source is an expression that is evaluated to a set of tuples containing elements from the source model. The syntax of rule source expressions is shown below in an EBNF-like notation. Non-terminals are in italic.

source [*Component* +, (**condition** {*BooleanExpressionInOCL*})?]

A rule source enumerates at least one component. An optional condition may be imposed upon the components. The components of a rule source are two kinds: a model element identifier that uniquely identifies an element in the source model and a variable that can be bound to more than one source model element. Each variable has a type. The type can be a model element from one of the intensions of the source model or one of the primitive and collection types available in Object Constraints Language (OCL) [80]. If the type is a model element then the variable matches the instances of this model element in the source model. Variables can be initialized by an OCL expression. The condition of a rule source is a Boolean expression written in OCL.

The result of the evaluation of a rule source over a source model is a set of tuples formed by the Cartesian product of the matches for each component. Tuples that do not satisfy the condition are excluded.

An example of a rule source is given below:

source [c: Class, a: Attribute=c.attributes, **condition**{c.isAbstract=false}]

This expression contains variables *c* and *a* of types *Class* and *Attribute* respectively. During the evaluation of the expression the variables will be bound to instances of *Class* and *Attribute* respectively. The variable *a* is initialized with an OCL expression using the variable *c*. The condition constrains the set of model elements that will be bound to *c* to

those classes that are not abstract. The evaluation of this source expression on a source model results in a set of tuples. Every tuple has two components corresponding to the two variables. If a variable is initialized with an expression then the expression is evaluated over the bindings of the other variables used in the expression. In our example the value of the variable *a* is derived from the value of the variable *c*. Variable *a* will be bound to every one of the attributes of *c*.

Sometimes the source model is instance of more than one intension. Usually the default instantiation relation is assumed when the type of a variable is specified and the colon ":" notation is used in that case. The default instantiation relation is declared in the declaration part of the transformation definition (see section 4.5.2). If another instantiation relation has to be used then its name must be explicitly given. The following example illustrates the syntax that is used:

source [aTuple **instanceOf**(Relational) aSchema]

In that example if the ":" notation was used that would be a shortcut for *instanceOf(MOF)* syntax.

Variables in a rule source may be of one of the collection types available in OCL. For example, if we want to select all the attributes of a class and to refer to them as a single collection (e.g. as a set) the following expression is defined:

source [c: Class, a: Set=c.attributes, **condition** {c.isAbstract=false}]

So far we described the selection of model elements that are instances of exactly one model element. It is also possible to select model elements instances of the specializations of a given type. In general, more than one specialization relation may exist for a given language. The names of the relations must be defined in the configuration of the language (see Figure 4.9 in section 4.4.5). These names may be used to specify a selection. Assume that in a given language there is a relation called 'Generalization'. A source expression that selects instances of a given class *aClass* and also instances of all its subclasses looks like this:

source [c: aClass **follow** Generalization]

Here the keyword *follow* is used in combination with the name of the relation. The expression '*aClass* **follow** *Generalization'* results in a set. The transformation engine will use the function *getSpecializedConstructs* associated to the relation to obtain all the related classes to the given *aClass*.

The type of a variable used in a rule source may be determined by evaluation of an expression. The expression must result in a set of model elements from the intension of the source model. In the following example we illustrate the possibility to specify an expression that determines the type instead of using an identifier of the type.

Assume that a transformation rule has to be specified that selects all elements in a source XML document instances of top level element declarations. The XML schema of input documents (*anXMLSchema*) and the XML Schema model are available to the transformation engine. A rule source expression that selects the required elements looks like this:

```
1. transformation aTransformation
2.   languages MOF, XMLSchema
3.   anXMLSchema instanceOf(MOF) XMLSchemaModel
4.   source anXMLDocument instanceOf(XMLSchema) anXMLSchema default
……..
5.   source [ element : ElDeclaration,
6.            ElDeclaration instanceOf(MOF) ElementDeclaration,
7.            condition {ElDeclaration.topLevel=true} ]
```

In this example the type of the variable *element* (line 5) is not a concrete model element like in the previous examples. Instead, it is a variable (*ElDeclaration*) whose range is over *anXMLSchema* model. *ElDeclaration* must be an instance of *ElementDeclaration* element defined by *XMLSchema* language (line 6). We are interested only in those element declarations that are top level (line 7).

Evaluation of Rule Source Expression

This section gives an algorithm for evaluation of rule source expressions. The algorithm is presented below in a pseudo code notation. Keywords are in bold. Informal actions are in italic.

We distinguish between initialized (line 2) and uninitialized (line 3) components in a source expression. Variables whose type is another variable are considered as initialized component. For every uninitialized variable the algorithm determines the range of values (lines 6-17). The Cartesian product of ranges is formed (line 19). For every tuple in that product the expressions of initialized components are evaluated (lines 21-34). It is possible to have a circular dependency between variables. For instance, variable *a* may be initialized with an expression that refers to variable *b*, which in turn is initialized with another expression that refers to *a*. In that case the expressions cannot be evaluated and an exception is raised (lines 25-27). Tuples with variable bindings are checked if they satisfy the condition of the source expression (lines 31-33). Only tuples that satisfy the condition are kept in the resulting set of tuples.

```
1.  EvaluateSourceExpression (String sourceExpression) : Set of Tuple {
2.     Set  initializedComponents = all the initialized components in sourceExpression
3.     Set  uninitializedComponents   = all the uninitialized components in
                                        sourceExpression
4.     array rangesForUninitializedComponents =new array
5.     Set of Tuple result = new set
6.     ForEach component in uninitializedComponents {
7.        Set componentRange
8.        If (component is a variable) {
9.           ModelElement  variableType = the type of  component
10.          componentRange = all model elements from the source model of type
11.                           variableType
12.       }
13.       If (component is a model element) {
14.          add component to componentRange
15.       }
16.      add componentRange to rangesForUninitializedComponents
17.   }
18.   Set of Tuple   bindingsForUninitializedComponents
19.   bindingsForUninitializedComponents = make Cartesian product of the elements of
20.                                        rangesForUninitializedComponents
21.   ForEach tuple in bindingsForUninitializedComponents {
```

```
22.        ForEach variable in initializedComponents {
23.            String expression  = the initialization expression of variable
24.            exprResult = evaluate expression
25.            If (there is a circular dependency among variables) {
26.              Throw Exception
27.            }
28.           Else {add exprResult to tuple}
29.        }
30.        String condition = the condition of sourceExpression
31.        If (tuple satisfies condition) {
32.          add tuple to result
33.        }
34.     }
35.     Return result
36.  }
```

4.5.4 Model Element Rules

Model element rules can create model elements in the target model, modify existing model elements in the source and the target models, and delete model elements.

Creation of model elements can be done in two ways: by instantiating the constructs *ModelElement, Literal* and *Slot* in the model in Figure 4.2 and by instantiating model elements in a target intension. In our modeling space only the constructs in Figure 4.2 can be directly instantiated by the transformation engine. This instantiation is provided as a native operation in the transformation engine. When a model element from an intension has to be instantiated this is done by invoking the function *instantiate* that implements the instantiation for a given language (see section 4.4.4). The implementation of the function relies on instantiations of constructs *ModelElement, Literal* and *Slot*. Modification of model elements is performed by changing their slot values and by deleting model elements from a model.

The structure of model element rules is shown in Figure 4.12.

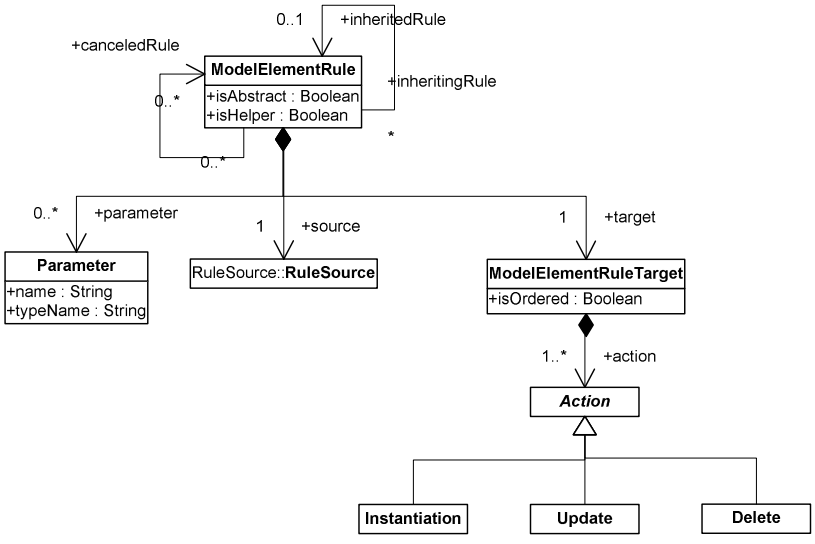


Figure 4.12 The structure of model element rules

The syntax of model element rules is the following:

```
ruleName (abstract)? ModelElementRule InputParameters? {
  RuleSource
  target [Action +]
  SlotRule*
}
```

Every model element rule has a name, a source, a target, an optional list of input parameters and is associated to a number of slot rules. Model element rules use the rule source expression to match elements from the source model. For every match the model element rule executes actions specified in the target of the rule. Actions can instantiate new elements, update elements, and delete elements.

Model element rules may be defined as abstract. If a rule is abstract it cannot be executed directly. It is intended to provide its components for reuse by other rules through inheritance.

An example of a model element rule is given below:

```
MOFClassInstantiation ModelElementRule {
  source [ c: Class, condition {c.isAbstract=false} ]
  target [ModelElement {slots=instSlot},
          instSlot : Slot {name='instanceOfMOF', value=c}]
}
```

In this example the name of the rule is specified (*MOFClassInstantiation*) followed by the keyword that indicates that this is a model element rule. The body of the rule contains source and target parts. The structure of the rule target is explained in the next section.

Rule Target

The target of a model element rule contains a list of actions. Currently three types of actions are supported: instantiation, update, and delete. Actions may be ordered. The three action types are explained in the remaining part of this section.

Instantiation

Every instantiation specifies an element in the target intension or an OCL type that will be instantiated. The element created by an instantiation may be assigned with an identifier and later referred to through this identifier. Instantiations are associated with two slot lists containing the names of the slots of the instantiated element. The first slot list contains the so called *constructor slots*. Their values must be available before the execution of the instantiation. The second slot list contains the rest of the slots whose values are set up after the execution of the instantiation. Slot values are determined from optional expressions specified in the slot lists and an optional set of slot rules.

Instantiations may be initialized with an expression. In that case there is no creation of a new model element and slot lists are not needed.

Our example rule specifies two instantiation actions based on *ModelElement* and *Slot* constructs respectively. The second instantiation is assigned with the identifier *instSlot*. Both instantiations specify only non-constructor slots that are set up after the execution of the instantiation. All slots are assigned with expressions. Expressions may refer to variables defined in the source of the rule (e.g. *value=c*) and to identifiers assigned to the other instantiations in the same rule (e.g. *slots=instSlot*).

When an instantiation is executed the new model element is created by using the default instantiation mechanism specified in the declaration part of the transformation definition. If another mechanism has to be used then it is explicitly specified following the same syntax used in the rule source. For example, if a class *A* must be instantiated according to the instantiation mechanism defined in a language *L* then this is specified in the syntax: a **instanceOf**(L) A

Update

The second type of action is the update action. It modifies the slot values of model elements selected by the rule source. An example of a model element rule that uses update action is given below.

```
setSlotValue ModelElementRule inputParameters [slotName:String, newValue:Set] {
  source [context: ModelElement, slot: Slot=context.slots, condition {slot.name=slotName}]
  target [update slot {value=newValue}]
}
```

This is a generic rule used to modify slot values of model elements for a given slot name and a new slot value. The name and the new value are declared as input parameters (*slotName* and *newValue*) and their values are passed before the execution of the rule. An update action is used to modify the value of the selected slot (*slot* variable in the rule source). Update actions are executed on model elements selected in the rule source or on model elements already created in the target model. A slot list is specified for the update action. The values of the slots enumerated in the list are modified after the execution of the update. The new values are obtained through the same mechanism used in the instantiation action: through an initialization expression or a set of slot rules.

The default interpretation of the update action is as replacement of the current value of the slot with the new value. Finer control on the update is achieved by using keywords before the slot name in the slot list. The keywords specify the type of update. Supported updates are *replace*, *insert*, *append*, and *delete*. The slot value is treated as a sequence of model elements. Updates may be applied on a particular element in the slot value or on the value as a whole. The keyword *replace* denotes change of the value of the slot. This is the default update type. The keyword *insert* denotes insertion of new values in the beginning of the sequence or at the position specified by an index. The keyword *append* denotes insertion after the last element of the sequence. The keyword *delete* denotes deletion of the slot value or deletion of a particular element in the value specified by an index.

The previous rule may be changed to a new version that appends the new value to the sequence of model elements. The target of this new rule is shown below.

```
target [update slot {append value=newValue} ]
```

Delete

The third type of action is deletion of model elements. The effect of deletion is removing the model element from a model. It is accompanied by removing the connections that this element has to other model elements via slots. Elements that will be deleted are selected by the rule source. Delete action may specify a list of cleanup actions that are executed before the deletion. These actions are deletions of specified set of elements or invocation of a transformation rule.

The example below shows a rule that selects classes and deletes them. Before deleting a class its attributes are also deleted.

```
classDeletion ModelElementRule{
  source[c: Class]
  target [delete c { delete c.attributes} ]
}
```

Rule Inheritance

The transformation language supports single inheritance among model element rules. An inheriting rule (child rule) inherits from the inherited rule (parent rule) its source, target and the associated slot rules. The inheriting rule may define its own source, target and slot rules and may override the corresponding inherited source and target components and slot rules. Rule components are union of the source components, the actions specified in the rule target, and the slot rules defined for the model element rule. We distinguish between derived components of rules and defined components of rules. Defined components are specified in the rule definition. Derived components include the inherited components of the rule after applying the algorithm for overriding the inherited components. The algorithm for determining the derived components of model element rules is as follows:

1) Determining the derived rule source
   a) The set of derived source components is a union of the derived source components in the parent rule and the defined source components in the child rule. If there is a duplication of names of source variables the variables defined in the child rule overrides the derived variables with the same name in the parent rule. Expressions specified in the child rule may refer to variables defined in the parent rule;
   b) The derived condition of the child rule source is a logical conjunction between the derived condition of the parent rule and the defined condition of the child rule;
2) Determining the derived rule target
   a) The set of derived actions of the child rule is a union of the derived actions of the parent rule and the defined actions of the child rule. If there is a duplication of names of the identifiers assigned to the instantiations then the instantiations defined in the child rule overrides the derived instantiations with the same name in the parent rule. Overriding an instantiation does not cause overriding of the slot rules associated with that instantiation. They are inherited in the child rule. Update and delete actions cannot be overridden;
   b) The set of derived slot rules of the child rule is a union of the derived slot rules of the parent rule and the defined slot rules of the child rule. Slot rules defined in the child rule override the derived slot rules of the parent rule with the same name. It is not necessary to override an instantiation in order to override some of its slot rules;

Rule Priority Hierarchy

In general, multiple model element rules may be applied on a single source element or a tuple of source elements. In some cases finer control over the application of rules is required. A rule may forbid the application of other rules if all are applicable on a given

model element. Consider the following example. A general rule selects all the instances of a class and a more specific rule operates differently for a subset of the instances. The more specific rule can not be used in conjunction with the general rule. In this case the more specific rule has a higher priority and cancels the application of the more general rule.

The transformation language provides a mechanism for building a priority hierarchy among the rules. This hierarchy is specified by the software engineer. A rule may declare precedence over other rules and to cancel their application on the model elements selected by the canceling rule. The following example shows a rule named *rule1* that cancels the application of two other rules named *rule2* and *rule3*:

```
rule1 ModelElementRule cancels rule2, rule3 {
……..
}
```

The cancellation mechanism is transitive. If a rule *A* cancels a rule *B* then *A* cancels all the rules cancelled by the rule *B*.

Priority and inheritance hierarchies are independent of each other. However, in many cases inheritance and cancellation among rules occur together.

4.5.5 Slot Rules

Slot rules specify how to obtain the slot values of the elements created or modified by model element rules. Slot rules are always associated with a given model element rule. Slot rules may be specified either in the body of model element rules or separately.

Figure 4.13 shows the overall structure of slot rules.

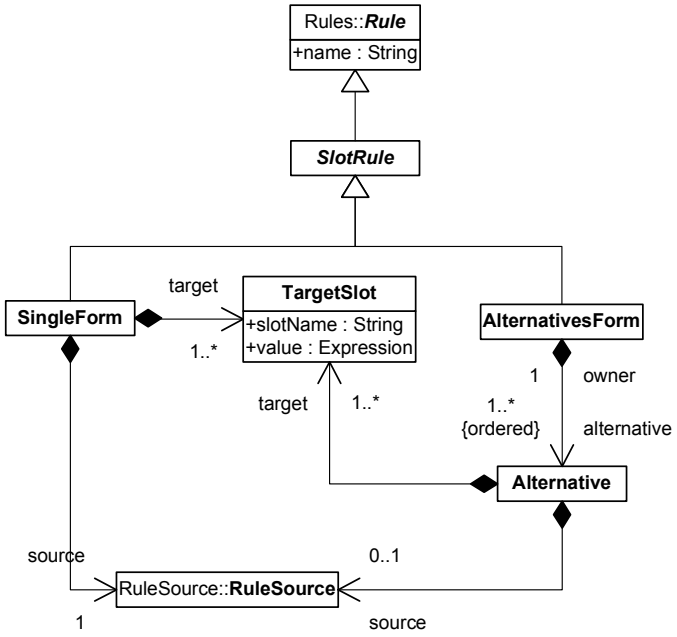


Figure 4.13 Structure of slot rules

The syntax of slot rules is defined by the following rule:

ruleName RuleSource **target**[(*slotName=Expression*)+]

Every slot rule has a name, a source and a target. Rule source specifies the elements in the source model that will be used to obtain the values of the slots. Rule target enumerates the slots to be set up with a value. A given slot may have more than one slot rule for the calculation of the value. Slot rules are executed in the context of the owning model element rule. This context provides the current bindings of the source components in the owning rule.

There are two forms of slot rules: single form and form with alternatives.

Single Form

Slot rules in single form have only one source expression. The source expression is evaluated in the context of a particular matching for the owner model element rule. At the time of execution of the slot rule the variables in the owner rule are already bound to values. The source of the slot rule may contain references to the source components in the owner rule. It is often the case that the values of source components in the slot rule are determined relatively to the source components in the owner rule. The values of the slots are determined by evaluating expressions over the source components.

An example of a slot rule in single form defined in the body of the owner rule is given below:

```
1.  MOFClassInstantiation ModelElementRule {
2.    source [c: Class, condition {c.isAbstract=false} ]
3.    target [ModelElement {slots} ]

4.    SlotRules {
5.      attributeSlots
6.          source [a:Attribute=c.attributes]
7.          target [slots]
8.    }
9.  }
```

In that example the value of the slot named *slots* in the *ModelElement* instantiation (line 3) is calculated by the slot rule named *attributeSlots* (lines 5-7). This is indicated by including the slot name in the target of the slot rule (line 7). The source of the rule (line 6) declares a variable of type *Attribute* whose value is determined by an expression that refers to the variable *c* defined in the source of the owner rule *MOFClassInstantiation*. Since that expression is evaluated after the binding of *c* it is possible to determine the value of *a*.

Form with Alternatives

Slot rules in the form with alternatives specify multiple alternative sources. They are evaluated in the order of their appearance and the first source expression that results in a non-empty set of tuples is used to determine the values of the slots in the target. The rest of the alternatives are not evaluated. It is also possible to specify an alternative without a source in the end of the list with alternatives. It is used if none of the preceding alternatives is applied.

The next example shows a slot rule in the form with alternatives. The example is an extended version of the previous example and takes into account the slots defined in the su-

per-class of the class being instantiated. We assume that collisions in slot names are not allowed.

```
1.  MOFClassInstantiation ModelElementRule {
2.    source [c:Class, condition{c.isAbstract=false}]
3.    target [ModelElement {slots}]
4.    SlotRules {
5.      attributeSlots
6.        source [a:Attribute=c.attributes]
7.        target [slots]
8.      slotsFromSuperClass
9.        target [slots]
10.       alt { source [parent : Class=c.superclass]  target [slots]  }
11.       alt { target [slots=Set[]] }
12.   }
13. }
```

In this example slots of the model element created in line 3 are obtained through two slot rules: *attributeSlots* and *slotsFromSuperClass*. The second one is in the form with alternatives. The first alternative (line 10) selects the super class of the class and obtains its slots[1]. If the class does not have super class then the source expression results in an empty set. Therefore, the second alternative is used (line 11). It does not have a source and specifies an empty set of values for the slot in the initialization expression.

Calculating Slot Values

Slot values are determined by the algorithm given below.

```
1.  CalculateSlotValue (ModelElementRule ownerRule,
2.                      String slotName,
3.                      Collection slotRules) : Collection {
4.    Tuple context =the current bindings for ownerRule source components
5.    Collection result
6.    If (slot with slotName has initialization expression in ownerRule) {
7.       result=evaluate expression in context
8.      Return result
9.    }
10.   Else{
11.      ForEach rule in slotRules {
12.         Set of Tuple slotRuleBindings
13.         slotRuleBindings=EvaluateSourceExpression(rule.sourceExpression)
14.         Collection ruleSlotValue =new empty Collection
15.         ForEach binding in slotRuleBindings  {
16.            resultForBinding=evaluate rule.slotInitializationExpression on binding
17.            ruleSlotValue.add(resultForBinding)
18.         }
19.         result=union(result, ruleSlotValue)
20.      }
21.      If (result does not match multiplicity and type constraints of the slot) {
22.         Throw Exception
23.      }
24.      Return result
25.   }
26. }
```

[1] The details how the slots of the super class are obtained are not shown. They will be clarified further when the example is completed.

The algorithm applies the slot rules (input parameter *slotRules*) available for a given slot indicated by the input parameter *slotName* in the context of the current match of the owner model element rule (input parameter *ownerRule*). If the slot has initialization expression then the value is the result of the evaluation of the expression (lines 6-9). Otherwise results obtained from each slot rule are united in a collection (lines 11-20). This collection is checked for the type and multiplicity constraints of the slot (lines 21-23).

4.5.6 Linking Source Elements to Target Elements

During the execution of a given model element rule the transformation engine may establish a link between the selected source elements and the target elements instantiated by the rule. The target elements may be used as slot values of elements instantiated by other rules. Target model elements may be obtained by navigating through the source model and querying the source elements for the linked elements in the target model. The transformation language provides two constructs for that purpose. These constructs are explained in the next two sections.

Linking Source Elements to Target Elements via *link-to* Construct

The *link-to* construct is used in the source of a model element rule to instruct the transformation engine to establish a link between a source element and the instantiations in the target. An example is shown below:

```
MOFClassInstantiation ModelElementRule {
  source [c: Class link-to {instance} , condition {c.isAbstract=false}]
  target [ instance: ModelElement {slots} ]
}
```

In this example every instance of *Class* bound to the variable *c* will be linked to the corresponding target model element identified by *instance*.

It is possible to list more than one target element in a single *link-to* construct. Sometimes the rule target contains instantiations unassigned with identifiers. It is possible to establish a link to them by using the reserved identifier *default*. The previous example is shown below illustrating that possibility. The target instantiation is not assigned with an identifier.

```
MOFClassInstantiation ModelElementRule {
  source [c: Class link-to{default}, condition {c.isAbstract=false}]
  target [ModelElement {slots} ]
}
```

Querying Source Elements for Linked Target Elements

Model elements in the source model may be queried for linked target elements. The transformation language provides the built-in function *target* for that purpose. The function is usually used in the slot rules. An example usage of function *target* is shown below:

```
1.  MOFClassInstantiation ModelElementRule {
2.    source [c:Class, condition{c.isAbstract=false} ]
3.    target [ModelElement {slots} ]

4.    SlotRules {
5.      attributeSlots
6.        source [a:Attribute=c.attributes]
7.        target [slots=target(a)]
8.    }
9.  }

10. MOFAttributeToSlot ModelElementRule{
11.   source [a:Attribute link-to{default} ]
12.   target   [Slot {name=a.name}]
13. }
```

The rule *MOFAttributeToSlot* (lines 10-13) generates a slot for every instance of *Attribute*. A link is established between every attribute and the slot created for it by the construct *link-to*. When a model element is instantiated from a class the slots of this model element are obtained by querying the attributes of the class. *MOFClassInstantiation* rule implements this. Slots are determined by the slot rule *attributeSlots* (lines 5-7). For every attribute the corresponding slot is obtained by the function *target* with the attribute as an argument: *target (a)* (line 7).

The function *target* may take several forms:

- *target(sourceElement)*: returns all the default elements linked to the source element;
- *target(sourceElement, list_of_identifiers)*: returns the set of elements linked to the source element and assigned with the identifiers enumerated in the list. A name collision may occur if an identifier is used in more than one rule. To specify the exact rule a prefixed notation is used (e.g. *aRuleName.aVariableName*);
- *targetByRule(sourceElement, ruleName)*: returns all the elements linked to the source element and instantiated by the rule named *ruleName*;

4.5.7 Invoking Rules

By default, model element rules are executed exactly once on every match of their source. The resulting elements may be linked to the source elements and may be reused by other model elements. There are cases, however, when a model element rule is executed multiple times per single source element. In these cases new target elements are created for every execution.

To illustrate this possibility we slightly modify the previous example. In the example only one slot is created per attribute (rule *MOFAttributeToSlot*, lines 10-13). Thus, slots are reused by more than one model element instantiated from a given class. To create new slots every time when a model element is instantiated from a class we explicitly invoke *MOFAttributeToSlot* rule in the expression *slots = MOFAttributeToSlot(a)* (line 7). This invocation is in the initialization expression in the target of the slot rule *attributeSlots* (lines 5-7):

```
1.  MOFClassInstantiation ModelElementRule {
2.    source [c:Class, condition{c.isAbstract=false}]
3.    target [ModelElement {slots}]

4.    SlotRules {
5.      attributeSlots
6.        source [a:Attribute=c.attributes]
7.        target [slots=MOFAttributeToSlot(a) ]
8.    }
9.  }

10. MOFAttributeToSlot ModelElementRule{
11.   source [a:Attribute]
12.   target   [Slot {name=a.name}]
13. }
```

When a model element rule is invoked it receives concrete source elements as input and returns a tuple with results. These results can be accessed by using the dot ('.') notation and the identifiers of the elements instantiated by the invoked rule. If no identifier is specified then the default element is returned.

There is another way for rule invocation. Instead of invoking a specified rule over the provided input the transformation engine may be instructed to search for a rule whose source matches the input data and to execute it. If more than one rule is applicable then an error is generated. The language construct introduced for this purpose is named *applyRules*. This construct is similar to the *apply-templates* instruction available in Extensible Stylesheet Language for Transformations (XSLT) [105]. The usage of the construct is exemplified below.

Assume that we want to check if two classes are connected by a generalization relation. We implement the checking as a transformation that operates on two classes and returns *true* or *false*. The transformation contains three model element rules:

```
1.  noSuperClass ModelElementRule {
2.    source [c1: Class, c2: Class, condition{c1.supertype->isEmpty()}]
3.    target [result: Boolean=false]
4.  }

5.  directRelationOrIdentity ModelElementRule {
6.    source [c1: Class, c2: Class, condition{(c1=c2) or (c1.supertype=c2)}]
7.    target [result: Boolean=true]
8.  }

9.  other ModelElementRule {
10.   source[c1:Class, c2: Class, condition{(c1.supertype->notEmpty()) and
11.                                        (c1.supertype<>c2)}]
12.   target[result: Boolean=applyRules(c1.supertype, c2).result] }
```

The *applyRules* construct (line 12) is used in the third model element rule called *other*. Depending on which rule source is satisfied one of the three rules will be executed. This example can also be implemented as a single recursive rule by using the *if-then-else* construct of OCL.

4.5.8 Execution of Transformations

Transformation engine supports two execution scenarios. In the first scenario the engine executes all non-helper model element rules on a given source model. A model element rule is considered executed if its target is executed for every match of its source. In the second scenario the execution starts from a specified model element in the source model. The transformation engine finds the applicable rules for this element and executes them. These rules, in turn, may cause the execution of other rules. In this scenario some rules may remain unexecuted.

Transformations may be executed either through an interpretation or a compilation. If the engine is implemented as an interpreter then it is able to execute any transformation definition on any input model. If a transformation definition is compiled then the compiled program is applicable on a particular set of input models.

Transformation rules are declarative and generally there is no predefined execution order upon them. Transformation developer may impose partial execution order via two constructs: transformation steps and rule invocation.

Execution order is determined from the dependencies among the rules. These dependencies are detected at runtime and a dependency graph is built. The execution order is derived after a topological sort on this graph. Generally, more than one execution order is possible. Transformation engine chooses an order in a non-deterministic way. The dependency graph may contain cycles and therefore, the transformation can not be executed. The next section explains the dependencies among rules and the determination of the execution order.

Choosing an Execution Order of Transformation Rules

The execution of a model element rule is a sequence of executions of the actions in the target of the rule. Dependencies among the instantiations may occur introduced by the slot rules. A dependency occurs because instantiations require the values of their constructor slots to be available before the execution of the instantiation. If the values of the constructor slots are results of other instantiations then these instantiations must be executed earlier. If no constructor slots were specified then all the instantiations would be executed in an arbitrary order followed by execution of the slot rules.

Slot rules may also introduce dependencies. Assume that a slot rule named *slotRule1* requires the value of a slot of some model element. This value is determined by a second slot rule called *slotRule2*. *slotRule2* must be executed before *slotRule1*.

In summary, during the execution we have to deal with two types of orders: order among instantiations and order among execution of slot rules. Both orders obey dependencies. The dependencies are captured in a dependency graph constructed for a concrete input model. To construct the graph all the source elements selected by model element rules are associated with the instantiations defined in the rules. Instantiations are the nodes in the dependency graph. If a given instantiation requires the result from another instantiation as a value for the constructor slots then an edge is created from that instantiation to the instantiation that creates the required value. If a slot rule requires value provided by another slot rule then another edge is created to capture the dependency.

An example of a dependency graph is shown in Figure 4.14.

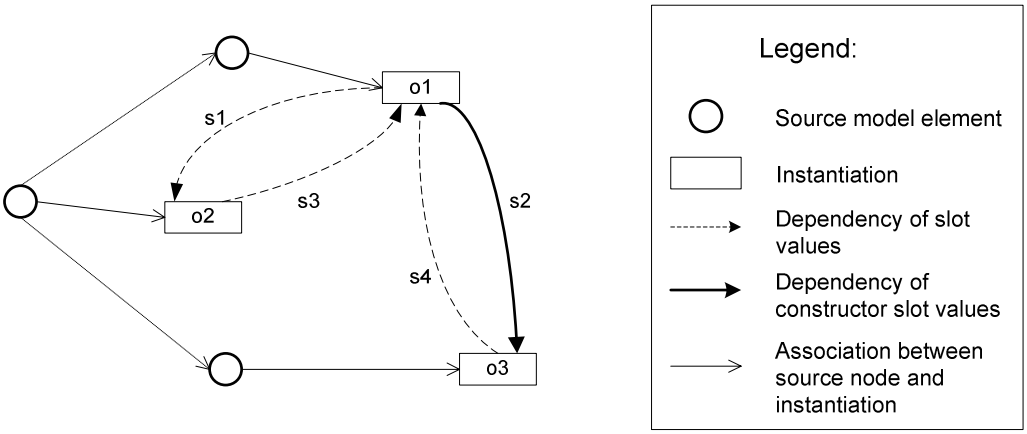


Figure 4.14 Dependency graph

In this figure there are three source model elements represented as circles. Each model element is associated with instantiations depicted as rectangles labeled *o1*, *o2* and *o3* respectively. The solid labeled arrow from *o1* to *o3* indicates that the result of *o3* will be used as a value of the constructor slot *s2*. This means that *o3* must be executed before *o1*. Dashed arrows indicate non-constructor slots. One possible execution order is *o3*, *o2*, *o1* and then setting the non-constructor slots *s1* and *s3*. Another execution order is *o2*, *o3*, *o1*.

If the dependency graph contains a cycle over the constructor slots edges then the transformation cannot be executed for the given input model.

An example of such a situation is given in Figure 4.15 where the edges *s2*, *s3* and *s4* form a cycle. In that case there is a circular dependency among *o1*, *o2*, and *o3* and they cannot be executed.

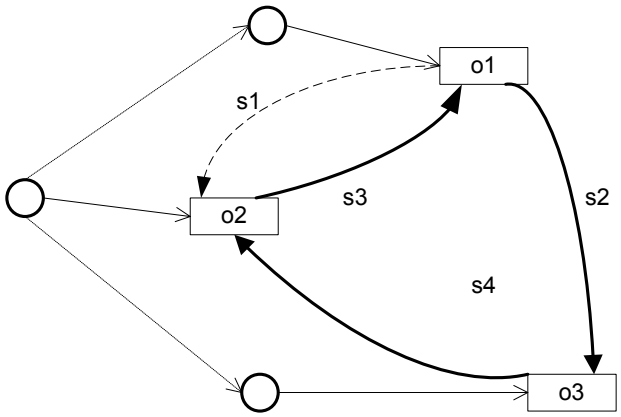


Figure 4.15 Dependency graph with cycle

Non-deterministic Execution

The previous section explained that multiple execution orders may exist among transformation rules. The transformation engine chooses an order in a non-deterministic way. Transformation developers have only a partial control on this choice. A natural question is if different execution orders produce the same result.

If model element rules in a transformation definition contain only instantiation actions then the result is independent of the execution order. Instantiations do not change the

source model elements and the already created elements in the target model. In that sense, instantiations are side-effect free. The same is valid for slot rules. Their effect is limited to slots of a given target model element.

To prove that the result of transformation execution is independent of the execution order if only instantiations are present we consider two instantiations *i1* and *i2* that must be executed in this order. Recall that every execution order preserves dependencies among instantiations and therefore *i1* will always be executed before *i2*. The rest of the instantiation actions may occur in any order between *i1* and *i2* without affecting them. The same considerations are valid for the execution of slot rules.

If, however, model element rules contain *update* and *delete* actions then different execution orders may produce different results. Both update and delete actions may change the source and target models. Therefore they are not side-effect free. In this case a transformation definition must be carefully organized to achieve the desired result. Software engineers should use the available constructs for explicit control of transformation execution. These constructs are transformation steps and rule invocation.

Another important issue is the termination of transformation executions. In the general case, when transformation rules contain update and delete actions and recursive rules the termination of executions is not guaranteed.

Working with Elements outside of the Modeling Space

Some models may consist of elements that are located outside of the modeling space. Usually, they are run time elements created and maintained by tools used in a particular technology. In the context of the MOF architecture these may be elements at level M0. Some examples are database records kept in a database management system and Java objects 'living' in a virtual machine. The manipulation of these elements may require communication with their native environment.

We propose three approaches for working with elements outside of the modeling space:

- import and export of the elements in the modeling space;
- creation of proxy model elements that interact with an external environment;
- integration of the modeling space and transformation engine with a native environment;

The first approach is feasible when the number of model elements is not huge. For example, an XML document is parsed and its internal representation as a hierarchy of model elements is built. Later on, this hierarchy will be serialized back to a textual XML document. All the transformation operations are performed on the internal representation.

The second approach is recommendable if the amount of data is huge and the import leads to memory consumption. An example is working with a large database that must be transformed to another data format. In that case it is better to work with the native environment (the database management system) through the native language (e.g. SQL). This can be achieved by providing a suitable implementation of the language configuration operations for accessing slot values, setting slot values and instantiations. Configuration operations isolate the technical details of communication with the native environment.

The feasibility of this approach is still unclear since we do not have enough experience with its application.

We applied the third approach in a case study of transforming XML documents to Java objects [91]. This case study is presented in Chapter 6. The transformation engine was

implemented in Java. XML documents were accessed via Java APIs (DOM [104] and SAX [92]). In the implementation, the transformation engine is invoked by the Java program that uses the result of the transformation (see also Section 4.5.9). The source and target models and the engine are in the environment of the same Java virtual machine.

Execution of Model Element Rules

In this section we present an algorithm for execution of model element rules that detects dependencies explained in the previous sections. The algorithm is implemented in a number of functions written in pseudo code notation. We assume that model element rules contain only instantiation actions.

Function *ExecuteRules* takes a model and a set of model element rules as input and executes the rules over the input model. A model element rule is considered executed if all rule instantiations are executed on every match of the rule source expression.

Function *ExecuteRules* is based on several steps. The first step is to evaluate the source expression for each rule over the source model and to establish associations between source model elements and instantiations of rules that have selected those elements. This step is implemented in the function *AssignInstantiationsToSourceModel* invoked in line 2. At the second step, every rule is executed (lines 3-11). The execution of a given model element rule iterates over the source bindings of the rule source components and for every binding executes the instantiation actions of the rule (lines 5-10). Execution of instantiations is implemented in the function *ExecuteInstantiation* invoked in line 8.

```
1.  ExecuteRules (Model sourceModel, Set modelElementRules) {
2.    AssignInstantiationsToSourceModel(sourceModel, modelElementRules)
3.    ForEach rule in modelElementRules {
4.      Set of Tuple sourceBindings = obtain the set of bindings for rule
5.      ForEach bindingTuple in sourceBindings {
6.        Set of Instantiation instantiations=obtain the instantiations for bindingTuple and rule
7.        ForEach (instantiation in instantiations){
8.          ExecuteInstantiation(instantiation, bindingTuple)
9.        }
10.     }
11.   }
12. }
```

Function *AssignInstantiationsToSourceModel* evaluates the source expressions of a set of model element rules. Every rule is associated with the tuples that match its source expression (line 6). Furthermore, for a given rule, every element selected by the rule source expression is associated with the rule and the instantiation actions of that rule (lines 8-12). After the execution of this function the dependency graph shown in Figure 4.14 is partially built. The dependencies among instantiations are not explicitly established. They are detected during the execution of function *ExecuteInstantiation*.

```
1.  AssignInstantiationsToSourceModel(Model sourceModel,
2.                                    Set modelElementRules){
3.    ForEach rule in modelElementRules {
4.      String sourceExpression=obtain the source expression of rule
5.      Set of Tuple sourceBindings=EvaluateSourceExpression (sourceExpression)
6.      Associate rule with sourceBindings
7.      Set of Instantiation instantiations=obtain instantiations of rule
8.      ForEach tuple in sourceBindings {
```

```
9.            ForEach component in tuple {
10.              Associate component with instantiations for rule
11.           }
12.        }
13.     }
14.  }
```

Every instantiation associated to a given source element may be in one of four states explained in the following list:

- *not processed*: the instantiation is not executed. This is the initial state;
- *in processing*: the instantiation is in a process of obtaining the values of the constructor slots (if present). Creation of a target element from that instantiation is not done yet;
- *processed*: instantiation is executed but the slot values of the resulting target element are not assigned with values;
- *completed*: instantiation is executed and the slots of the created target element are assigned with values;

Function *ExecuteInstantiation* first checks if the instantiation passed as input is in state of *processing* (lines 3-5). If it is, this means that there is a second attempt to execute the instantiation before completing the first attempt. This indicates a circular dependency and therefore the instantiation cannot be executed. An exception is thrown (line 4).

If the instantiation is not processed then the algorithm checks the presence of constructor slots (line 7). If there are such slots the instantiation accepts the state *in-processing* (line 8). At the next step, the values of the constructor slots are obtained and the instantiation is executed (lines 9-13). After that the instantiation accepts the state *processed* (line 14).

If there are no constructor slots then the instantiation is executed and marked as *processed* (lines 16-19). Finally, the remaining slots are assigned with values by invoking the function *AssignSlots* (line 20).

```
1.   ExecuteInstantiation( Instantiation instantiation,
2.                          Tuple bindingTuple){
3.      If (instantiation is in processing state){
4.           Throw CircularityDependencyException
5.      }
6.      If (instantiation is not processed) {
7.         If (instantiation has constructor slots) {
8.            Mark instantiation as in processing state
9.            Collection constructorSlotsValues
10.           constructorSlotsValues= ProcessConstructorSlots(sourceModel,
11.                                                           instantiation,
12.                                                           bindingTuple)
13.           Instantiate the type of instantiation with constructorSlotsValues
14.           Mark instantiation as processed
15.        }
16.        Else {
17.           Instantiate the type of instantiation
18.           Mark instantiation as processed
19.        }
20.        AssignSlots(instantiation, bindingTuple)
21.        Mark instantiation as completed
22.     }
23.  }
```

Function *ProcessConstructorSlots* calculates the values of constructor slots. The function first ensures that all the required instantiations are executed. In this way dependencies among instantiations are detected and followed. This is done by invoking the function *ProcessSlot* (line 7). Slot values are calculated by the function *CalculateSlotValue* described in section 4.5.5 (line 10).

```
1.  ProcessConstructorSlots(Model sourceModel,
2.                          Instantiation instantiation,
3.                          Tuple bindingTuple) : Collection {

4.     Set constructorSlots=obtain constructor slots of instantiation
5.     Collection constructorSlotsValues=new empty Collection
6.     ForEach slot in constructorSlots {
7.       ProcessSlot(instantiation, slot, bindingTuple)
8.       ModelElementRule ownerRule=obtain the owner rule for instantiation
9.       Collection slotRules=obtain the slot rules for slot
10.      Collection value=CalculateSlotValue(ownerRule, slot.name, slotRules)
11.      constructorSlotsValues.add(value)
12.    }
13.    Return constructorSlotsValues
14. }
```

Function *ProcessSlot* determines the required instantiations for calculating the value of a given slot. This is done by analyzing the initialization expressions in slot rules. The function executes required instantiations by invoking the previously defined function *ExecuteInstantiation* (line 10) in case an instantiation is not processed yet.

```
1.  ProcessSlot(Instantiation instantiation, Slot slot, Tuple bindingTuple) {

2.    Set of SlotRule slotRules=obtain the slot rules for slot
3.    Set requiredInstantiations=new empty Set
4.    ForEach slotRule in slotRules {
5.       Set instantiations=obtain the required instantiations for slotRule
6.       requiredInstantiations=union(requiredInstantiations, instantiations)
7.   }
8.   ForEach instantiation in requiredInstantiations {
9.      If (instantiation not processed) {
10.         ExecuteInstantiation(instantiation, bindingTuple)
11.     }
12.     If (instantiation is in processing state){
13.        Throw CircularityDependencyException
14.     }
15.  }
16. }
```

Function *AssignSlots* assigns values of slots of a given instantiation in the context of a given *bindingTuple* containing values of the source components. It is similar to the function *ProcessConstructorSlots*.

```
1.  AssignSlots(Instantiation instantiation,
2.                Tuple bindingTuple) {

3.    Set slots=obtain slots of instantiation
4.    ForEach slot in slots {
5.       ProcessSlot(instantiation, slot, bindingTuple)
6.       ModelElementRule ownerRule=obtain the owner rule for instantiation
```

```
7.          Collection slotRules=obtain the slot rules for slot
8.          Collection value=CalculateSlotValue(ownerRule, slot.name, slotRules)
9.          Assign slot value for slot in instantiation
10.     }
11.  }
```

Interaction of the transformation engine with the configuration of the languages of the source and target models is demonstrated in two places in the functions:

- Function *ExecuteInstantiation*, lines 13 and 17 where a model element from the target intension is instantiated. This instantiation is done by executing the operation *instantiate* defined in class *Instantiation* (see Figure 4.9);
- Function *AssignSlots*, line 9. Assigning a slot value is done by the operation *setSlotValue*. Before invoking this operation type and multiplicity checking are performed by using the operations *isCompatible*, *getSlotType*, and *getSlotMultiplicity*. The details how this is done are not shown;

Operations in language configuration are also used when source expressions are evaluated and navigation expressions over the source model are evaluated.

4.5.9 Transformation Engine Prototype

This section describes a prototype of transformation engine developed for a previous version of the transformation language presented here. This previous version is designed for XML processing. A description of the approach for XML processing based on model transformations can be found in [58]. Figure 4.16 shows a summary of this approach.

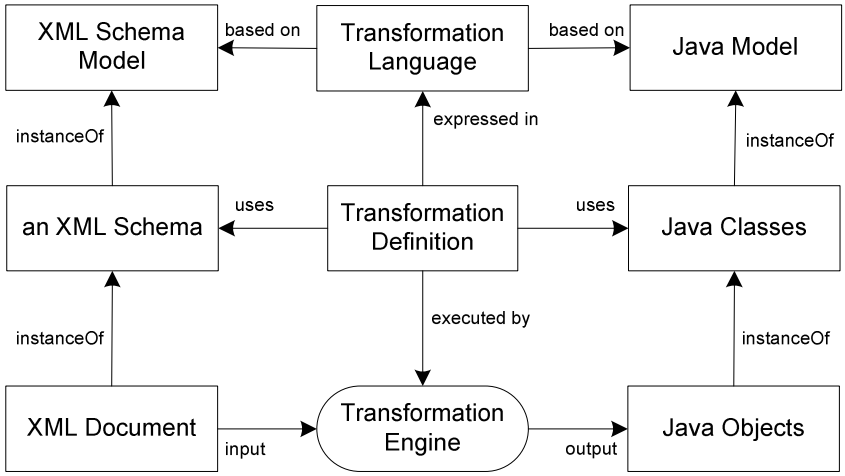


Figure 4.16 Transformation approach and language for XML processing

The transformation language indicated in the figure is designed for transforming XML documents to Java objects. The language is defined on the basis of the concepts in XML Schema model and Java model. A transformation definition expressed in that language uses an XML schema and a set of Java classes. Transformation engine transforms XML documents instances of the schema to a set of Java objects instances of the Java classes.

This language is coupled with XML Schema model and Java model. This is the main difference with the language MISTRAL described in this chapter. MISTRAL is an evolu-

tion of the language indicated in Figure 4.16 and is decoupled from a particular modeling language used to define source and target models. The two languages employ the same constructs presented in this chapter: model element rules and slot rules. The execution semantics is almost the same: it relies on topological sort over dependency graphs.

The implementation of the transformation engine for the language applied for XML processing served as a proof of concept for the algorithm of rule execution explained earlier in this section. We give a short description of the architecture of the transformation engine. More detailed description of the architecture and implementation of the engine accompanied by several case studies are given in [91].

Figure 4.17 shows the components in the architecture of the transformation engine prototype.

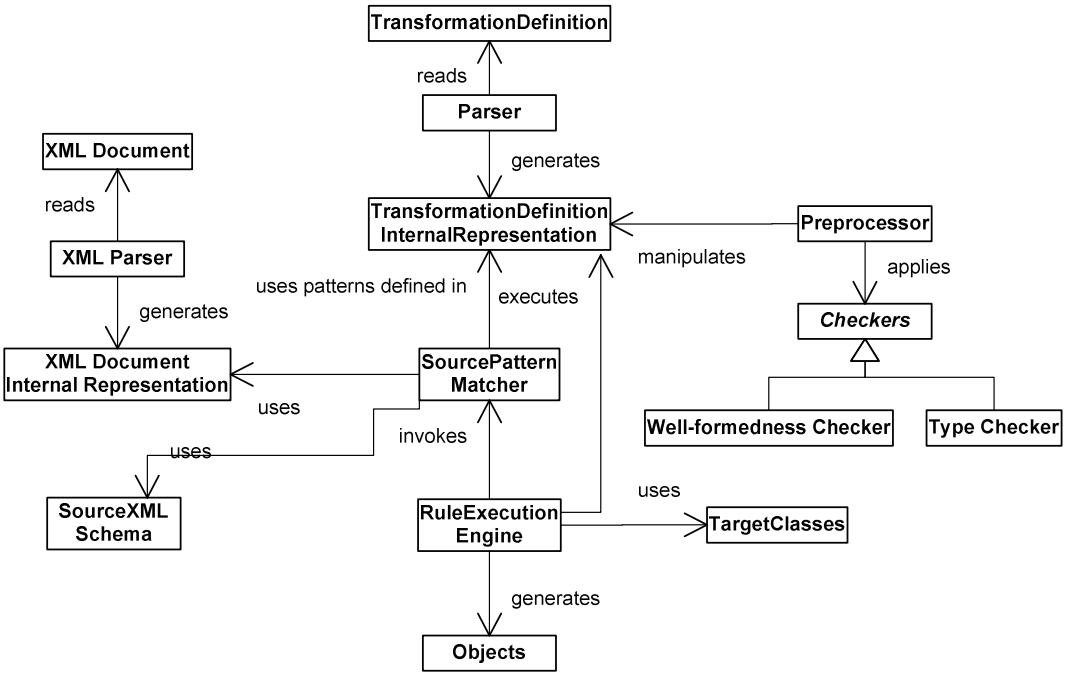


Figure 4.17 Architecture of the prototype of transformation engine

An input *XML Document* is parsed by *XML Parser*, which generates *XML Document Internal Representation* used by other components in the engine. *TransformationDefinition* is parsed by *Parser*, which generates *Transformation Definition Internal Representation*. This representation is checked by *Preprocessor*. *Preprocessor* applies *Well-formedness Checker* and *Type Checker* to perform the checking. Transformation is executed by *Rule Execution Engine* that invokes S*ource Pattern Matcher* to extract nodes from *XML Document Internal Representation*. Pattern matcher uses information from *Source XML Schema*. *Rule Execution Engine* uses *Target Classes* to generate the output *Objects*.

4.6 Example: Defining the Configurations of MOF and Relational Model

In this section we present an example implementation of some of the functions identified in Section 4.4.4. For illustrative purposes functions are implemented as transformation rules written in the transformation language.

4.6.1 Configuration of MOF Language

Configuration of *MOF* language is defined as a set of transformation rules. The source intension is the MOF Model in Figure 4.3 and the target intension contains the constructs *ModelElement, Slot* and *Literal* defined in the modeling space. Only three functions are implemented here: *instantiate*, *getSpecializedConstructs*, and *isCompatible*. The remaining functions are trivial and are skipped.

The following rules implement the instantiation mechanism in the MOF language.

```
1.  instantiate ModelElementRule{
2.    source [c:Class, condition {c.isAbstract=false}]
3.    target [instance: ModelElement {slots=slotRulesValue->union(instSlot)},
4.              instSlot :Slot {name='instanceOfMOF', value=c} ]

5.    SlotRules {
6.      attributeSlots
7.      source [a:Attribute=target (c, derivedAttributes) ]
8.      target   [slots=MOFAttributeToSlot(a)]

9.      associationSlots
10.     source  [assoc:Association=target(c, derivedAssociations)]
11.     target    [slots=MOFAssociationToSlot(assoc)]
12.   }
13. }
```

```
1.  DerivedConstructsForClass ModelElementRule{
2.    source [c: Class link-to{derivedAttributes, derivedAssociations}]
3.    target [derivedAttributes: Set{elements},
4.              derivedAssociations: Set{elements}]

5.    SlotRules {
6.      ownAttributes
7.      source [a: Attributes=c.attributes]
8.      target [derivedAttributes.elements=a]

9.      attributesFromParent
10.     target [derivedAttributes.elements]
11.      alt { source [parent:Class=c.supertype]
12.            target [derivedAttributes.elements=target(parent, derivedAttributes)]  }
13.      alt { target [derivedAttributes.elements=Set[] ] }

14.     ownAssociations
15.      source [assoc:Association, condition{assoc.from.type=c}]
16.      target [derivedAssociations.elements=assoc]

17.     associationsFromParent
```

```
18.     target [derivedAssociations.elements]
19.      alt { source [parent:Class=c.supertype]
20.            target [derivedAssociations.elements=target(parent, derivedAssociations)] }
21.      alt { target [derivedAssociations.elements=Set[] ] }
22.   }
23. }
```

```
MOFAttributeToSlot ModelElementRule{
  source [a:Attribute]
  target [Slot {name=a.name}]
}
```

```
MOFAssociationToSlot ModelElementRule{
  source [assoc:Association]
  target [Slot {name=assoc.to.name}]
}
```

The *instantiate* model element rule creates an instance of a non-abstract class and determines its slots. Slots are obtained from the derived sets of attributes and outgoing associations of the class. These derived sets contain the attributes and outgoing associations defined in the class and also the inherited ones from its parent. Derived sets are created by the rule *DerivedConstructsForClass*. We assume that collisions among the names of the attributes and association ends are not allowed. Slots are created by the rules *MOFAttributeToSlot* and *MOFAssociationToSlot* respectively.

The following rule creates a set of all the classes that directly or indirectly inherit from a given class.

```
1.  getSpecializedConstructs ModelElementRule{
2.    source [c: Class]
3.    target [result: Set{elements=slotRulesValue->union(c)}]

4.    SlotRules {
5.      Elements
6.      source [s: Class, condition{s.supertype=c}]
7.      target [result.elements=getSpecializedConstructs(s)]
8.    }
9.  }
```

This rule is an example of a recursive rule (see line 7) that for a given class determines all direct specializations and makes a union of their specialized constructs.

The following rule checks if two classes are related directly or indirectly by a generalization relation. The target element is of Boolean type.

```
1.  isCompatible ModelElementRule{
2.    source[expectedType: Class, actualType: Class]
3.    target [result: Boolean=(if expectedType=actualType then true
4.                             else if actualType.supertype->isEmpty() then false
5.                                  else isCompatible(expectedType, actualType.supertype))]
6.  }
```

4.6.2 Configuration of Relational Model

In this section four rules are defined for the configuration of the relational model. Rules use the models in Figure 4.6 as source and target intensions.

The instantiation mechanism of the relational model is defined below.

```
1.  instantiate ModelElementRule{
2.    source [s:RelationalSchema]
3.    target [Tuple {field, instanceOf =s}]

4.    SlotRules{
5.      Fields
6.      source [f:FieldType=s.fieldTypes]
7.      target [field=FieldTypeInstantiation(f)]
8.    }
9.  }

10. FieldTypeInstantiation ModelElementRule{
11.    source [ft:FieldType]
12.    target [Field{name=ft.name, instanceOf=ft}]
13.  }
```

In this rule the target constructs are not the generic classes *ModelElement* and *Slot* as in the definition of the instantiation for the MOF Model. Instead, *Tuple* (line 3) and *Field* (line 12) are used and these classes are instantiated through the MOF instantiation rule defined in the previous section. The transformation engine will use the rule to build the tuples and fields.

The following rule implements the function *meta* that determines the intensional construct for a given tuple or field. Note that it does not distinguish between instances of *Tuple* and *Field* since the generic class *ModelElement* is used. For a given source model element *me* the rule finds among its slots the slot named "instanceOf" and returns its value as a result.

```
meta ModelElementRule (
  source [me: ModelElement, slot=me.slots, condition{slot.name='instanceOf'}]
  target [meta-construct: ModelElement=slot.value]
}
```

We also specify a transformation rule used for slot value access.

```
getSlotValue ModelElementRule inputParameters [slotName: String]{
  source [context: Tuple, f:Field=context.field, condition {f.name=slotName}]
  target [result: Set=[f.value]]
}
```

This rule will be executed on an instance of *Tuple* class supplied by the transformation engine and bound to the variable *context*. It will navigate over the fields and will select a field with name equal to the input parameter *slotName*. The value will be obtained via the expression *f.value* and the result will be assigned to the target which is of type *Set*.

To set slot values we use the following rule, which modifies model elements instances of *Tuple*:

```
setSlotValue ModelElementRule {
  inputParameters [slotName:String, newValue:Set] {
  source [context:Tuple, f:Field=context.field, condition {f.name=slotName}]
  target [update f {value=newValue}]
}
```

The rule takes two input parameters: the slot name and the value. The source of the rule (the variable *context*) is determined by the transformation engine. The field *f* with name equal to the parameter *slotName* is found. Then the slot of *f* with name 'value' will be set.

4.7 Evaluation of the Model Transformation Language

In this section we evaluate the proposed transformation language MISTRAL and the modeling space from different perspectives. The transformation language is evaluated according to classifications found in the literature. We use the classifications given by Czarnecki and Helsen [27] and Gardner et al. [42] (Section 4.7.1). Furthermore, we consider the language in the context of the requirements of QVT RFP [76] (Section 4.7.3). The adaptability property of the language is evaluated in Section 4.7.4. Finally, a comparison of the modeling space with other approaches for meta-modeling is given.

4.7.1 Classification of Model Transformation Languages

Czarnecki and Helsen [27] present a domain analysis of existing model transformation approaches. The results are summarized in a feature model that presents the commonalities and variabilities in the domain. Here we briefly describe the main categories of classification and their variation areas. Variations indicate the design choices made in existing model transformation approaches. The categories are presented in the subsequent sections.

Transformation Rules

Transformation rules are the basic constructs in transformation definitions. They have left-hand side and right-hand side, which may or may not be syntactically separated. The areas of variation found in transformation rules are:

- *Directionality*: rules may be executed in one or two directions;
- *Rule parameterization*: rules may receive additional input via parameters;
- *Intermediate structures*: some approaches allow intermediate model structures;

Source-Target Relationship

This classification category captures the relation between source and target models. The following variations are found:

- Source and target models are different;
- Source and target is the same model: this allows updates in the source model (*in-place update*). Another update category is *destructive update* in the source and target;

Rule Application Strategy

Generally, a rule may match more than one element/tuple in the source model. Therefore a strategy for the rule application on the matches is required. Strategies may be:

- *Deterministic*: this strategy is based on specified algorithm that governs the order of application of rules over matches. It ensures that the same order is selected every time when a transformation is executed on the same source model;
- *Non-deterministic*: in this strategy the order of application of rules may be different for different executions of the same transformation on the same source model;
- *Interactive*: the user specifies the strategy;

Rule Scheduling

Rule scheduling mechanism is responsible for the order in which the rules are applied. It may vary in four areas:

- *Form*: concerns the way the order is expressed. The form may be *implicit* and *explicit*. Implicit form of scheduling relies on implicit relations among the rules. Explicit form of scheduling uses dedicated constructs to control the order. Explicit scheduling may be *internal* and *external*. Internal scheduling uses control flow structures within rules and explicit rule invocation. External scheduling uses scheduling logic separated from the transformation rules;
- *Rule selection*: rule selection may rely on explicit *condition* on the source elements. Since many rules may be applicable on a single source element there may be a need of *rule conflict resolution* (e.g. via rule priority);
- *Rule iteration*: iteration may be based on recursion, looping, fixpoint iteration, and combination of them;
- *Phasing*: a transformation definition is separated into phases usually executed sequentially. Each phase uses certain set of rules;

Rule Organization

Rule organization concerns relations among transformation rules. Three variation areas are related to this category:

- *Modularity mechanisms*: these are mechanisms for grouping of rules into packaging constructs that can be reused;
- *Reuse mechanisms*: allow rules to reuse existing rules in new rule definitions. Inheritance among rules is often used as a reuse mechanism;
- *Organizational structure*: rules may be organized according to the source language, target language, or in other independent way;

Traceability Links

Traceability links record correspondences between source and target elements established during transformation execution. Generally, two approaches are followed for maintaining traceability links:

- *User-based*: the user is responsible to create links as ordinary model elements;
- *Dedicated support*: transformation language and transformation engine provide support for maintaining links. This support may be *automatic* and *manual;*

Directionality

Directionality of transformations was explained in Chapter 2. Some languages allow specification of transformation definitions that can be applied only in one direction: from source to target model. These transformations are known as *unidirectional* transformations. Other languages (usually declarative ones) allow definitions that may be executed in both directions. These transformations are known as *bidirectional* transformations.

4.7.2 Application of the Classification on the Transformation Language

This section classifies transformation language MISTRAL according to the categories explained in Section 4.7.1. Results are summarized in Table 4.1 shown below.

Category (based on [27], see section 4.7.1)		**Classification of language MISTRAL** (based on section 4.5)
Transformation Rules	Structure	Syntactically separated left-hand side and right-hand side parts with variables and patterns.
	Directionality	Unidirectional rules from source to target elements
	Rule parameters	Supported
	Intermediate Structures	Not supported
Source-Target Relationship		Separated source and target models with possibility for in-place update only on source models and destructive update of the source and target models.
Rule Application Strategy	Deterministic	Not supported
	Non-deterministic	Rule application on all the matches of the rule source in a non-deterministic order.
	Interactive	Not supported
Rule Scheduling	Form	Mix of implicit and explicit form of scheduling. Explicit form is supported through rule invocation (internal explicit form) and transformation steps (external explicit form)
	Rule selection	Based on a rule source condition and a cancellation mechanism among rules for conflict resolution.
	Rule iteration	Recursion is supported.
	Phasing	Supported through transformation steps.
Rule Organization	Modularity mechanisms	Transformation modules
	Reuse mechanisms	Rule inheritance and module inclusion combined with rule overriding.
	Organizational Structure	Source-driven
Traceability Links		Dedicated manual support via *link-to* construct. Storage of links is handled by the transformation engine. Links are associated with the source elements.
Directionality of transformations		Unidirectional (from source model to target model)

Table 4.1 Transformation language features according to the classification of transformation approaches

4.7.3 Transformation Language in the Context of MOF 2.0 QVT RFP

QVT RFP [76] addresses the need for standard language for transformation definitions in MDA. It states a set of mandatory and a set of optional requirements for QVT languages. In this section we provide a summary of these requirements.

QVT Requirements

Mandatory Requirements:

- *Query language*: proposals should define a language for querying models;
- *Transformation language*: proposals should define a language for expressing transformation definitions. Transformation definitions are executed over MOF models, i.e. models that are instances of MOF meta-models;
- *Abstract syntax definition*: QVT languages should define their abstract syntax as a MOF meta-model;
- *View language*: QVT languages should enable creation of views on models;
- *Declarative language*: proposals should define declarative transformation language;

Optional Requirements:

- *Bidirectional transformation definitions*: proposals may support transformation definitions executable in two directions;
- *Traceability*: proposals may support generation of traceability information;
- *Reuse mechanisms*: QVT languages may support mechanisms for reuse and extension of generic transformation definitions;
- *Transactional transformations*: proposals may support execution of parts of transformations as a transaction;
- *Update of existing models*: proposals may support execution of transformations where the source and the target model are the same;

Some requirements raise issues already discussed in Section 4.7.1. These are the optional requirements concerning directionality of transformation definitions, traceability and reuse mechanisms, and the update of existing models.

Evaluation of the Transformation Language against the QVT Requirements

As was stated in the introduction the main requirement for our transformation language is to support transformations among models written in multiple languages. The language is not intended to satisfy the QVT requirements. Instead, it overcomes the problem of coupling of QVT languages with one modeling language: MOF. Furthermore, our transformation language aims at supporting all the scenarios shown in Figure 4.1. In contrast, QVT languages support only one of these scenarios: transformations on MOF models.

Despite that QVT languages and our transformation language are developed for different requirements it is possible to evaluate the transformation language against the QVT requirements. The evaluation is presented in Table 4.2.

Requirement (based on [76])	**Support by language MISTRAL** (based on section 4.5)
Mandatory Requirements	
Query language	Source expressions are used to query source models. OCL is used as navigation language (see section 4.5.3).
Transformation language working on MOF models	The language is capable of expressing transformation definitions on MOF models. It also supports transformation scenarios not addressed in QVT.
Abstract syntax definition	Available (see Appendix B)
View language	Not available
Declarative transformation language	The transformation language is hybrid. Declarative and imperative constructs are available.
Optional requirements	
Bidirectional transformations	Only unidirectional transformations are supported.
Traceability support	Available (see section 4.5.6)
Reuse and extension mechanisms	The language provides rule inheritance and reuse of transformation modules (see section 4.5.4)
Transactional transformations	Not available
Update of models	Available (via *update* and *delete* actions)

Table 4.2 Support of the QVT requirements by the transformation language

4.7.4 Adaptability of the Transformation Language

In Chapter 2 we formulated a requirement that a transformation language should be adaptable with respect to the used modeling languages. A consequence is that a transformation language should be able to express transformations on models written in different languages. Figure 4.1 showed examples of such transformation scenarios. Section 4.3 discussed the main problem that may hinder the adaptability of transformation languages. This is the problem of coupling of transformation languages with given modeling language, and in particular with its instantiation and generalization mechanisms. Furthermore, in Section 4.4.4 we analyzed the interaction between common operations in model transformations and modeling language features.

In this section we evaluate the adaptability property of the transformation language presented in this chapter. We explain the capability of the transformation language MISTRAL and the transformation engine to work with an open set of modeling languages.

This capability is mainly based on the interaction between the transformation engine and the modeling space hosting models and modeling languages. From the point of view of the transformation engine all models have a uniform structure. Models consist of instances of *ModelElement*, *Slot* and *Literal* (see Figure 4.2). The transformation engine does not assume any specific interpretation of these constructs. It is only capable of exe-

cuting operations over model elements: creating and deleting model elements and slots, adding and removing values of slots.

Language-specific interpretations of generic model elements are provided as computations over the model elements. These computations are implemented as operations members of language configurations. For example, a model element is considered as a class definition if there is a function that creates instances of the model element according to a language-specific instantiation mechanism.

During the execution of a transformation the transformation engine handles all the models as generic structures. It performs the operations explained in Section 4.4.4 by invoking the operations of the language configuration for the given model. In that sense, the transformation engine can be seen as a simple virtual machine for manipulations of model elements. More complex operations may be built upon the basic operations of the machine. These more complex operations are further grouped into language configurations and bring the notion of language in the transformation engine.

As we saw the modeling space allows arbitrary number of languages to be defined. Every language is associated to a configuration. Addition of a new language requires implementation of the operations in the language configuration. Language configurations are available to the transformation engine through a fixed interface.

The adaptability of the transformation language MISTRAL with respect to various modeling languages, therefore, depends on the possibility to capture the modeling language features in a configuration according to the proposed model in Figure 4.9. We showed example configurations of two languages in Section 4.6. Another example is given in Chapter 6.

4.7.5 Comparison of the Modeling Space with other Modeling Approaches

Meta-modeling architectures based on a common representation of the elements in different model levels can be found in various domains of computer science. RDF Schema [24] defines a three level architecture where all constructs are represented as triples according to the RDF data model [14].

Bowers and Delcambre [23] propose an approach for meta-modeling that has three levels and 5 types of instantiation mechanisms called conformance relationships. Different languages (called data schemas) may be modeled in this framework such as XML, Topic maps, relational model, etc. The framework uses a transformation language based on logical formulas similar to Prolog rules. Transformations between any levels are possible. In our approach we propose a domain specific transformation language. Another difference is that in our approach we rely on the notion of a modeling language that provides not only instantiation mechanism but also other operations not considered in [23].

Varro and Pataricza [101] propose a multilevel meta-modeling framework where instantiation and generalization are treated in a uniform way. In our approach instantiation and generalization are separated and multiple generalization/specialization relations may be defined per language.

Alvarez et al. [9] propose an approach for definition of modeling languages in the context of MOF. They treat the instantiation mechanism as a function that can be applied on a model at any level. This function resembles the MOF instantiation mechanism and is reused also in the UML meta-model. However, the authors do not study how other instantiation mechanisms would be defined in the framework.

4.8 Conclusions

We presented a model transformation language capable of expressing transformation definitions on models written in various modeling languages. In contrast to the current transformation languages for MDE the proposed language is decoupled from a given modeling language. This is possible within the proposed modeling space in which the transformation language acts. The modeling space hosts models and modeling languages. Models are instances of at least one intension. Modeling languages are associated with a configuration that defines operations used by the transformation engine. Different *instanceOf* and generalization relations may be defined within that space.

We studied the interaction between transformation languages and modeling languages. Five common operations found in transformations were discussed. The analysis shows that these operations are influenced by the instantiation and generalization relations defined for a given modeling language. The primary design goal for our transformation language is decoupling between the language and the specific instantiation and generalization mechanisms. The latter are implemented as functions and linked to the transformation engine. We showed examples of how these functions can be implemented in the transformation language.

It should be noted that the modeling space does not introduce any changes to the existing standards. The MOF model and other existing meta-models may be represented in the space. The space allows an explicit definition of the missing important constructs such as *instanceOf* relations. Our approach illustrates the need for a systematic definition of modeling languages within the MOF architecture and one particular example how transformation technology can benefit from that.

We described a prototype for a previous version of the language that implements an execution semantics close to the semantics of the language presented in this chapter.

Future research activities concern further development of the modeling space and the transformation language.

Model elements in the modeling space may be enhanced with operations. This will introduce the possibility for operation invocation in transformations. Transformation language has to be adapted accordingly. However, such a feature could be dangerous since it is generally not side-effect free. Another direction for research is studying the representation of various UML profiles within the modeling space. An interesting issue for investigation is the scenario in Figure 4.1c. It can be generalized to the problem of language translation where a transformation is specified between two MOF meta-models (regarded as language definitions) and executed on the lower model levels.

The transformation language may be enhanced with other control structures apart from transformation steps. We are particularly interested in studying fix-point iterations. An open issue is how to handle situations when a transformation rule changes the source model. This may lead to application of rules on the newly added elements. This case is well known in higher-order attribute grammars and the solutions in that area may eventually be applied. Furthermore, the transformation language may be enhanced with usage of intermediate model structures created during transformation executions. Currently, only elements in the source model may be selected by rules. Selection of elements in the intermediate structures seems an interesting possibility. Finally, performing non-trivial case studies (for instance, specifying XMI and JMI as transformation definitions) will provide a valuable insight on the applicability of the language.

5

Decomposition and Composition of Model Transformations

This chapter studies the required language support for decomposition and composition of transformation definitions. It shows scenarios where reusability and adaptability of transformation definitions are required. These scenarios are implemented by representing transformation definitions as a composition of units. In order to implement the scenarios, a transformation language must provide proper language constructs. The chapter contributes to the formulation of requirements that a transformation language must fulfill to provide an adequate support for reusability and adaptability of model transformations. An evaluation of some representative languages against these requirements is given. Finally, the chapter outlines a light-weight approach for extending a transformation language with new constructs.

5.1 Introduction

In Chapter 2 we identified scenarios in MDE processes that requires decomposition and composition of transformation definitions. Scenarios revealed that transformation definitions may be subjects of reuse, adaptation and composition in the same manner as the traditional software artifacts such as classes and libraries are reused, adapted and composed. Representation a transformation definition as a composition of simpler units helps in performing these tasks. The ability to decompose a transformation into simpler units reduces the complexity in the design of a transformation. Composing a transformation from existing components may make the development faster. Explicit representation of a composi-

tion provides a finer control over the components affected by changes and therefore improves the adaptability of transformations.

The experience in the area of programming languages shows that the problem of composition and decomposition of software artifacts is far from trivial. Finding the proper modular constructs and compositional operators in languages is a field of intensive research during the last few years as demonstrated by the Aspect Oriented Software Development (AOSD) initiative [39].

In the area of model transformation languages it is important first to study requirements for language constructs that would allow definition of transformations based on the principles of separation and composition of concerns. The QVT RFP [76] mentions among its optional requirements the need of mechanisms for reuse and extension of generic transformations. All the QVT proposals provide constructs that are similar to the inheritance mechanisms found in object-oriented languages. However, to the best of our knowledge there is no a detailed study on the need for decomposition and composition of transformations. Such a study will help to identify requirements for constructs in transformation languages. The availability of such requirements would also allow the evaluation of the current transformation languages and eventually proposing new techniques.

This chapter investigates the motivation for and the nature of decomposition and composition of transformation definitions. A transformation is defined on the base of source and target intensions. They are usually decomposed in a certain way and it is naturally to expect that the decomposition of the models affects the decomposition of transformation definitions. However, is that the only source for decomposition of transformation definitions? Is there some transformation functionality not related to the intensions that should be separated and reused? Also, today's software artifacts are not fixed, they evolve over time. Therefore, it is important to study how transformation definitions are affected if one or more models change. We would like to cope with these changes by applying constructs already defined in the transformation language. And finally, since a language hardly provides a perfect set of constructs for modularity and composition what to do to enhance the language in a disciplined manner if needed?

The approach we take to answer these research questions starts by exploring several transformation scenarios inspired by different examples found in publications and practice. We study how the decomposition of models affects the decomposition of transformation definitions and how the evolution of models affects the identified transformation units. For every scenario a set of transformation rules is identified. This is done in a manner independent of a particular transformation language. Identified transformation rules are conceptual and they specify relations between the elements in the source and target models. Furthermore, current transformation languages are evaluated with respect to their ability to cope with the transformation scenarios.

This chapter is organized as follows. Section 5.2 gives the motivation for this work in greater details and outlines the problems related to decomposition and composition of transformations. Section 5.3 describes our approach based on several transformation scenarios. Section 5.4 presents the transformation scenarios. Section 5.5 derives requirements for transformation language constructs on the base of the scenarios. Section 5.6 contains an evaluation of a set of transformation languages from the perspective of their applicability to the scenarios. Section 5.7 discusses techniques for extending transformation languages whenever this is needed due to language limitations. Section 5.8 gives conclusions.

5.2 Problem Statement

We focus on three general cases that motivate the transformation decomposition and composition. The first case is managing complexity when complex transformations have to be defined. The second case involves evolution in the source and the target models. The third case is based on the need to reuse existing transformation definitions into a new definition. From the perspective of transformation languages these cases pose requirements for proper language support.

5.2.1 Decomposition of Transformations for Managing Complexity

A transformation definition may be a complex artifact especially if the source and the target intensions are also complex. The way to deal with this complexity is to identify simpler units that are parts of the definition.

Transformation languages organize a transformation as a collection of rules where each rule connects elements from the source intension with elements from the target intension. Source and target intensions are by themselves decomposed into units (e.g. packages, classes) and this decomposition seems to be a suitable source for the decomposition of the transformation. One possible approach is to start with the source intension and to define how every unit in this model is related to units in the target intension. This approach is known as source-driven transformation. Second possibility is to take the units from the target intension and to determine how they are built from the units in the source intension. This approach is often regarded as target-driven transformation. It is known, however, that a given model may often be decomposed in more than one way and the specification of a model usually reflects only one particular way of decomposition. This default decomposition can be used to decompose a transformation. For instance, transformation rules may be organized following the part-whole hierarchy in the target intension. In case of the UML meta-model as an intension a package contains classes and other packages, classes contain attributes and methods, and so on. An interesting question is how the rest of the possible decompositions affect the transformation rules. Should other ways of decomposition be ignored or carefully considered? Moreover, the transformation designer is usually faced with at least two models involved in a transformation: the source and target intensions. They may be decomposed into ways without obvious matching of each other. Picking up one particular decomposition and neglecting the others may possibly lead to some anomalies in the transformation modularity known in the area of programming languages as the *tyranny of dominant decomposition* [82][97]. Can we observe this phenomenon also in a transformation specification? If so, what are the required language constructs to break the tyranny? These are questions that the transformation designer should take into account when identifying the transformation rules.

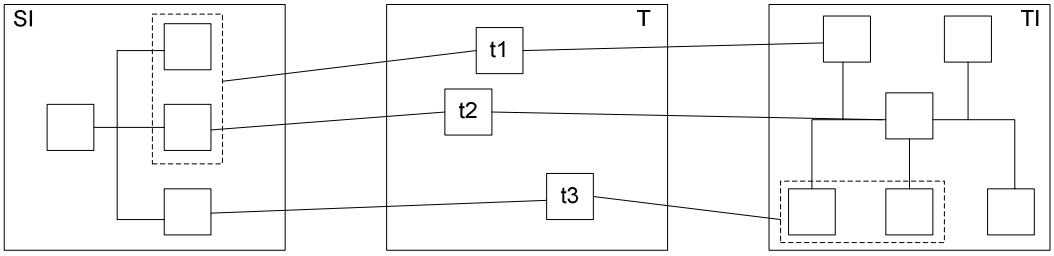


Figure 5.1 Possible correspondences between elements in the source and target intensions SI and TI used in transformation definition T

Figure 5.1 shows part of the problems rose above. The source and target intensions *SI* and *TI* are decomposed into structures that do not directly correspond to each other. Transformation *T* is defined on the base of these models. *T* consists of rules *t1*, *t2*, and *t3* shown as rectangles inside the transformation. It is possible that a single construct in one of the models corresponds to multiple constructs in the other model. Also, single construct may be processed by many transformation rules.

Apart from the decomposition into rules influenced by the correspondence between the source and target intensions some other functionality of a transformation may also be considered as unit that could be modularized. Transformation definitions should also specify how slot values of the target model are obtained. This functionality may also be a subject of separate consideration and therefore it should be specified separately.

5.2.2 Evolution of Transformations caused by Evolution of Models

Models evolve over time and the transformations based on them follow this evolution. This is illustrated in Figure 5.2.

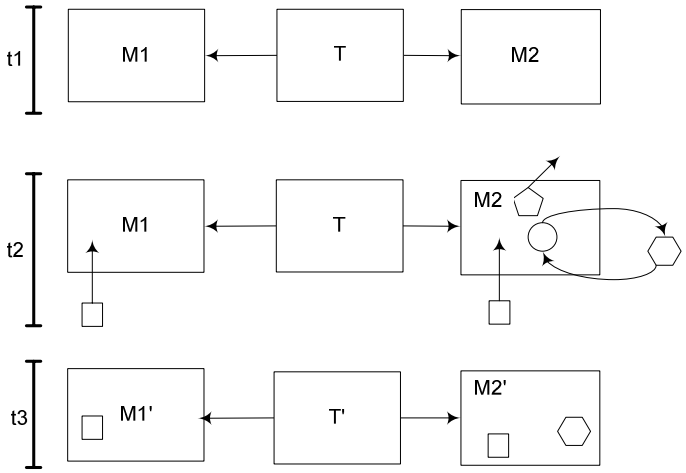


Figure 5.2 Evolution of transformation definitions caused by evolution of models. Moment t1 shows the initial state, moment t2 shows changes that occur, and moment t3 shows the state after changes.

Figure 5.2 shows three sequential moments in evolution of models and a transformation definition based on them. At the initial moment *t1* a transformation definition *T* is based

on intensions *M1* and *M2*. At moment *t2* both models undergo some evolution that includes addition, deletion and replacement of some of their model elements. At moment *t3* the models and the transformation definition are already changed. New models are denoted as *M1'* and *M2'* respectively. We would like to modify the transformation *T* to accommodate to the changes in the models. It is expected that parts of the transformation *T* that are not affected by the changes in *M1* and *M2* will remain the same and will be reused in the transformation *T'* defined for *M1'* and *M2'*. The requirement for this evolution scenario is to reuse as much as possible the transformation *T* and to adapt *T* by using available transformation language constructs.

The easiness of derivation of *T'* from *T* at a large extent depends on the chosen decomposition of the transformation *T* and on the available language constructs to do the adaptation.

5.2.3 Composition of Transformation Definitions based on Composition of Models

The case of this composition is shown in Figure 5.3**.** It follows three evolutionary stages.

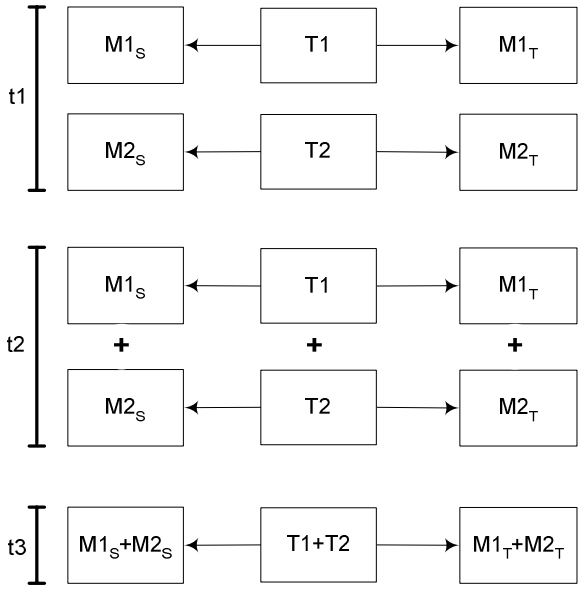


Figure 5.3 Composition of transformation definitions based on composition of models. At moment t1 two transformation definitions T1 and T2 are given, at moment t2 they are composed, and moment t3 shows the result of composition.

At the initial moment *t1* two transformation definitions are shown: *T1* based on the models $M1_S$ and $M1_T$ and *T2* based on the models $M2_S$ and $M2_T$. Later on, at moment *t2* the source and target intensions are composed to produce new models. A new transformation definition is to be developed that uses the composed models. This transformation definition is a composition of the definitions *T1* and *T2*. The moment *t3* shows the result of the composition of models and transformation definitions.

This case is similar to the previous case since the model composition can be considered as a case of model evolution: new components are added to the source and target inten-

sions. The difference is that we now have two transformation definitions that must be reused and composed with each other. This case puts the focus on the compositional operators available in the transformation language. Model composition may be simple and limited to a modular addition of new model elements without affecting the rest of the model. However, it is possible to have a complex composition where many model elements are affected in a complex way by the newly added ones. We may expect the same complexity to be brought to the transformation composition too.

The described three cases show that the need for modular constructs and compositional operators in transformation languages is mainly motivated by the possible evolution and reuse of the transformation definitions. The three cases usually coincide in real situations. For example, a decomposition of a transformation definition is identified by having in mind an expected evolution or reuse within a more complex transformation.

5.3 Approach

In this chapter we analyze different ways to decompose transformation definitions into units in order to manage the complexity and to achieve adaptability and reusability of definitions. We are interested in identifying language constructs able to express the required decompositions and compositions of transformation units.

Our approach is based on studying four transformation scenarios that show examples of decomposition, composition and adaptation in model transformations. The scenarios demonstrate at least one of the three general cases explained in the previous section.

The first three scenarios are concerned with the ways how a transformation definition may be decomposed and what are the required modular constructs to express the decomposition. The first scenario shows how multiple decompositions found in the source and the target intensions interact with each other within a single transformation definition. The second scenario illustrates how different ways to decompose one of the intensions (source or target) participating in the transformation affects the order of execution of transformation rules. The third scenario shows an example where certain units in a transformation definition are not derived from the source and target intensions but are related to a generic transformation functionality independent from the involved models. The fourth scenario emphasizes the effect of evolution of models on transformation definitions.

Scenarios are described in details in Section 5.4. Apart from these scenarios, Chapter 6 will present an example of composition of transformation definitions according to the case shown in Figure 5.3.

For every scenario we identify some decomposition of transformation definitions into rules. We use a simple notation based on constructs commonly found in current model transformation languages. We assume that rules have left-hand side and right-hand side that relate elements in the source and target models. Slot values of model elements are calculated by functions. We rely on a mechanism for traceability between source and target elements. Concrete examples with explanation of the notation are given in Section 5.4. Transformation rules specified in that notation are used to analyze the changes caused by evolution in the models and the composition of transformations. We do not assume any particular mechanism for implementing the changes. Instead, we are interested in evaluating how transformation languages express the identified rules and how the languages handle the required changes in the rules. Different languages provide different mechanisms

such as inheritance, aggregation and redefinition. The evaluation of these mechanisms is given in Section 5.6.

5.4 Scenarios for Decomposition and Composition of Transformation Definitions

This section describes four transformation scenarios. Each scenario presents a combination of the general cases explained in the problem statement.

5.4.1 Scenario 1: Decomposition of Models in Multiple Dimensions

This scenario studies how different decompositions found in the source and target intensions are used to identify the rules in a transformation definition. The anticipated evolution in the models guides the identification of the transformation rules. The scenario illustrates that more than one possible decomposition in the models have to be considered. If certain decompositions are neglected then the resulting transformation definition may expose anomalies such as tangling and scattering of transformation functionality. These anomalies reduce the adaptability of the transformation definition.

This scenario uses a simple system called The Examination Assistant that supports teachers in performing computer-based examinations. The system presents exam questionnaires to students in several modes: exam, self-test, and tutorial mode. Modes vary in the level of control the student has over navigating and answering the questions. Questionnaires are stored as XML documents. Documents are interpreted by the Examination Assistant and the exam items are shown to students. The interpretation is implemented as a transformation executed on XML documents to produce a set of application objects. Figure 5.4 shows the XML schema of examination documents shown as an UML diagram where different model elements are decorated with stereotypes to indicate the corresponding XML schema constructs.

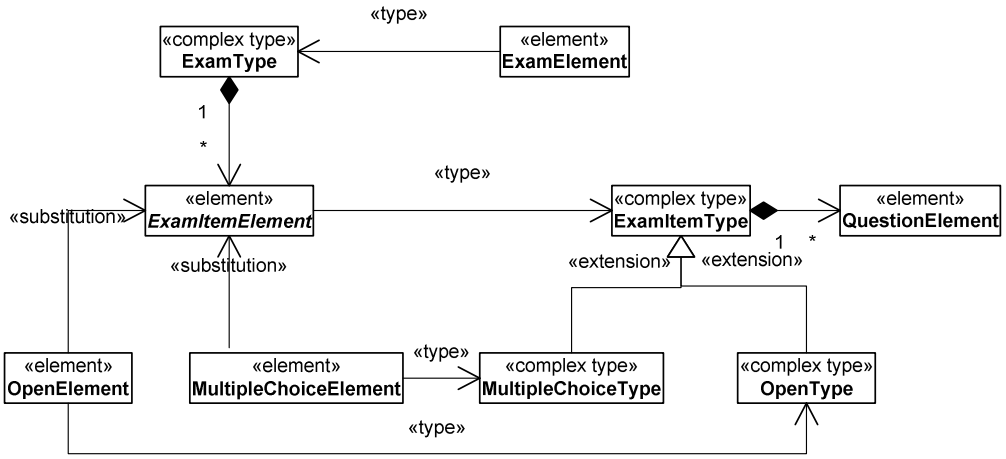


Figure 5.4 The XML schema for examination documents

In this scenario we only focus on the design of the user interface of the Examination Assistant. The design is based on the *Model-View-Controller* (MVC) design pattern [41]. Figure 5.5 and Figure 5.6 illustrate the classes that form the Model part of the application and the View and Controller parts respectively.

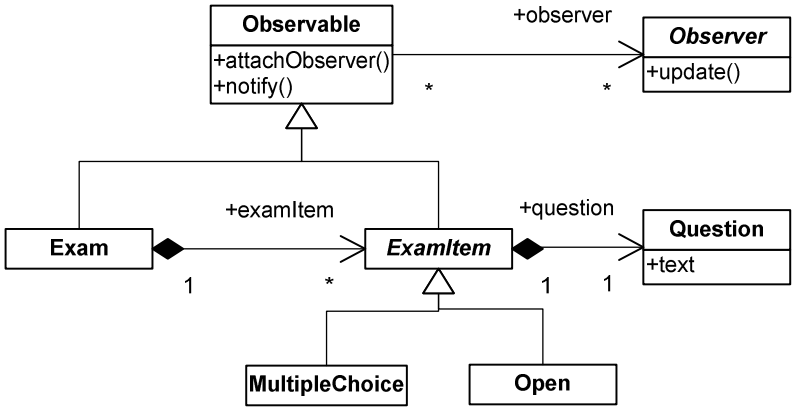


Figure 5.5 Model classes of Examination Assistant

Figure 5.5 shows the classes of the Model part of the application. There are classes for the questionnaire (*Exam*) and classes for the exam items (*ExamItem*, *MultipleChoice* and *Open*) that represent different exam item types. Every class is a specialization of class *Observable* according to the *Observer-Observable* design pattern [41]. This class provides an interface to the views and the controllers to register at the model classes.

Figure 5.6 shows the class hierarchy of the views and the controllers used in the application. There are two abstract classes *Controller* and *View* that specialize class *Observer*. For every exam item type there are a concrete view and a concrete controller. New views and controllers may be added by specializing classes *ExamItemController* and *ExamItemView*.

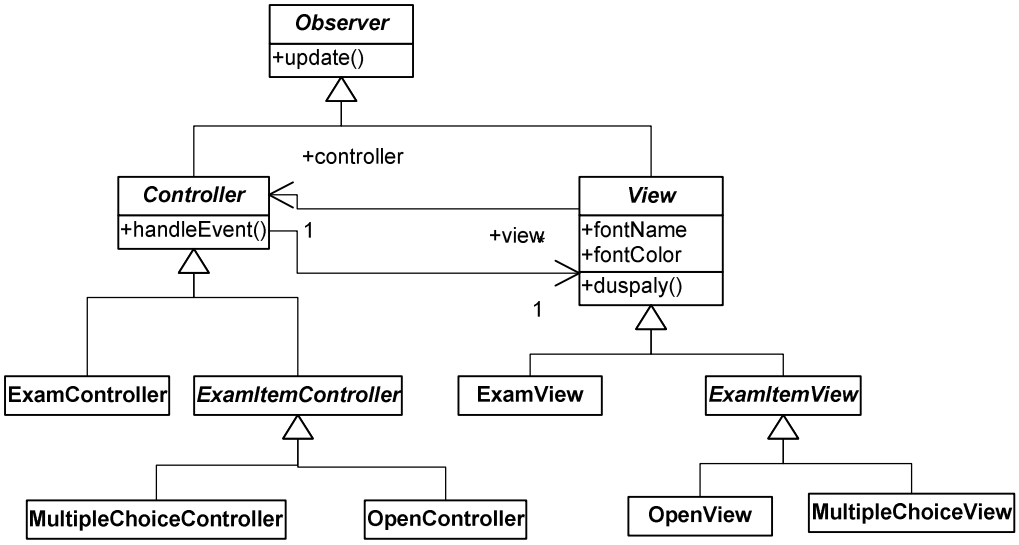


Figure 5.6 View and controller classes of Examination Assistant

An essential part of the Examination Assistant is the processing of XML documents that encode questionnaires. We focus on this processing since it uses a transformation definition to transform XML documents to a set of application objects. The source intension of this transformation definition is the XML schema shown in Figure 5.4 and the target intension is the application model parts of which are in Figure 5.5 and Figure 5.6.

How do we identify the rules in the transformation definition? Both the source and the target intensions may be used as a starting point. They are decomposed into model elements and a particular decomposition may be followed to decompose the transformation into rules.

The source XML schema is decomposed into elements and complex types. The target application model is decomposed into classes and seems more complex than the schema having in mind the number of the classes in it. Moreover, for every exam item type three classes from the application model have to be instantiated: one for the model, one for the view and one for the controller. This leads to a possible solution based on the decomposition of the source schema into exam item types. For every exam item element we instantiate the classes for the model, the view, and the controller. This approach is source-driven. It leads to rules shown below:

Source-driven transformation:

openQuestionRule (**source**[OpenElement],
target[Open, OpenView, OpenController])

multipleChoiceRule (**source**[MultipleChoiceElement],
target[MultipleChoice, MultipleChoiceView, MultipleChoiceController])

This transformation definition contains two rules: for open question (*openQuestionRule*) and multiple choice (*multipleChoiceRule*) exam item types respectively. The rest of the rules are skipped for simplicity. Every rule has a source and target. The source is a tuple that contains model elements from the XML schema. The target is another tuple that contains model elements from the Examination Assistant application model. The interpretation of these rules is that for every match of the source tuple over an input model, the model elements in the target tuple are instantiated. This basic interpretation is sufficient for our discussion and captures also the essential part of the semantics of the rules found in most transformation languages. We neglect many other details related to the rule execution. Our consideration is focused only on the decomposition of the transformation definition into rules.

The problem with this decomposition is that it involves *tangling* and *scattering* of some of the important concerns found in the target application model. Four concerns may be identified in the target model. The first one is the type of the exam item. The rest three are related to the model part, view part and controller part of the application. These concerns, for example, may lead to a decomposition of the application classes into three sets containing the model, the view and the controller classes respectively.

Transformation rules are decomposed along the first concern: the exam item types. Every rule refers to the classes for the model, view and controller in its target (tangling). Constructs related, for example, to the instantiation of the view classes are scattered across multiple rules. The same is valid for model and controller-related classes.

The scattering and tangling lead to problems if the target application model evolves. Assume that the current implementation of the controller classes performs the self-test

mode of the examination in which the student is able to navigate through and to answer to the exam questions. As a possible evolution of the Examination Assistant a new mode is introduced called *tutorial* mode in which students only navigate through the exam items without entering information. During the navigation the system provides explanation about the different user interface elements. To implement this mode at least new controller classes have to be introduced that implement this mode of interaction. The old controller classes that implement the self-test mode will be replaced.

The change of the exam mode leads to a series of changes in all the rules. This reduces the maintainability of the transformation definition and makes it more error-prone. Furthermore, the system may be again switched back to the self-test mode and then the self-test mode controller classes will be included in the transformation. Therefore, these two parts of the transformation definition should be separated and reused. In the current specification, however, the controller-related transformation functionality is not separately specified. This hinders the reusability of the transformation rules.

In summary, the anticipated changes of the examination mode put the requirement for two quality properties of the transformation definition: adaptability and reusability. In the current decomposition, however, this requirement is not met.

The reason is that in the current version of the transformation definition we focus only on one dimension of decomposition in both the target and the source intensions. The transformation is organized around the decomposition of the source schema into components for different exam item types. The same decomposition is possible within the target application model. This ensures evolvable transformation definition along that dimension. If a new exam item type is introduced then a new transformation rule is added. However, we neglect the decompositions along other concerns in the target model. The target model may be decomposed along multiple dimensions that form a multidimensional space according to the definitions given in [82] and [97].

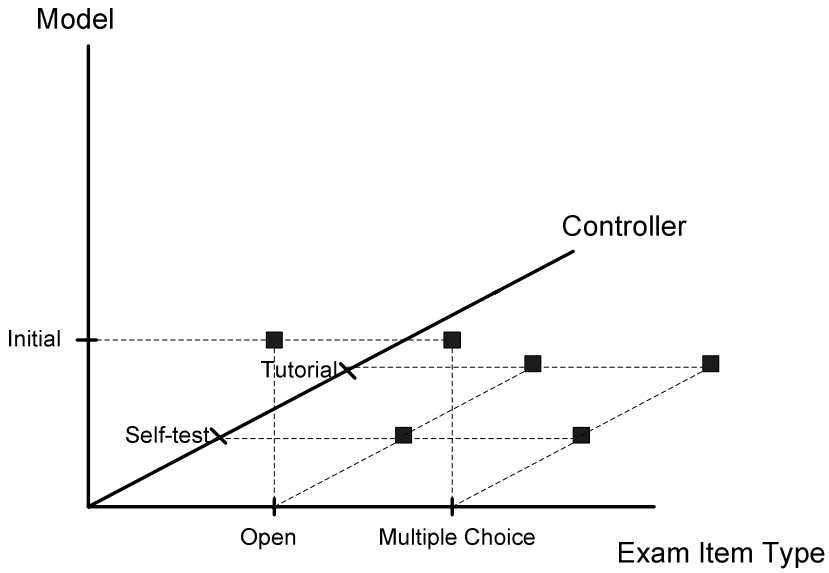


Figure 5.7 Decomposition of the target model along multiple dimensions

Figure 5.7 shows three dimensions of decomposition: *Exam Item Type*, *Controller* and *Model*. The fourth dimension that corresponds to the View concern is not shown. Every

dimension of decomposition represents an important concern in the target model. Each dimension has a set of coordinates that represent some alternatives in the dimension. For example, *Controller* dimension has two alternatives: *Self-test* and *Tutorial*. The points in the space are classes. In Figure 5.7 points are shown as small solid rectangles. The whole application model is constructed by choosing classes from the space.

In the current version of the transformation definition the targets of the rules are based on only one dimension of decomposition in the target model: *Exam Item Type*. However, classes from that dimension also pertain to other two dimensions: *Model* and *Controller*. If the target model evolves along these two dimensions all the transformation rules must be changed.

It is better to isolate each dimension in the target from the other dimensions. The new version of the transformation definition is the following:

Model-related part:

openQuestionModel (**source**[OpenElement],
target[Open])

multipleChoiceModel (**source**[MultipleChoiceElement],
target[MultipleChoice])

View-related part:

openQuestionView (**source**[OpenElement],
target[OpenView])

multipleChoiceView (**source**[MultipleChoiceElement],
target[MultipleChoiceView])

Controller-related part:

openQuestionController (**source**[OpenElement],
target[OpenController])

multipleChoiceController (**source**[MultipleChoiceElement],
target[MultipleChoiceController])

This decomposition eliminates the tangling and scattering observed in the previous version. The source of the rules is decomposed along the exam item type dimension. For a given type the target of rules is decomposed along the other three dimensions: Model, View and Controller.

The benefits of the second version of the transformation definition are the following. First, it allows evolution along all the dimensions found in the target model. If a change occurs over one of the dimensions the corresponding rules will be replaced with new rules. The rest of the rules remain unaffected. For example, if a new mode of interaction is introduced only the rules in the controller-related part will be replaced with new rules. Second, the rules for different alternatives in the dimensions are reusable and can be composed with rules related to other dimensions. Obviously, the second version of the transformation possesses the adaptability and reusability properties.

This scenario shows that following only one dimension for decomposition may lead to a transformation definition that exposes anomalies such as tangling and scattering. The

software engineer should carefully consider all the important dimensions for decomposition in both the source and the target intensions. Depicting the models over a multidimensional space may help in understanding the structure of the models. The inclusion of elements from dimensions that evolve independently in a single rule deteriorates the quality of the transformation definition.

Until now we explored the impact of the decomposition in the intensions on the transformation definition. In the remaining part of this scenario we will focus on another aspect of transformations: setting the slot values.

Assume we want to impose a uniform layout style of the exam items by using the same font and color in all the views. This is set by the attributes *fontName* and *fontColor* that all the view classes inherit from the abstract *View* class. These attributes must have the same values in all the exam item types. The part of the transformation that sets up the values is shown below. The attribute names and their values are shown as a list of comma separated assignments surrounded by curly braces.

View-related part:

```
openQuestionView (source[OpenElement],
                  target[OpenView {fontName='Times', fontColor='Red'}])

multipleChoiceView (source[MultipleChoiceElement],
                    target[MultipleChoiceView {fontName='Times', fontColor='Red'}])
```

Apparently the values are repeated for every exam item type. This is another example of scattering in the transformation definition. Similarly to the previous example we have a reduced quality of the transformation definition. Assume that the presentation style of the exam items is changed and the uniformity of layout is still required. This leads to identical changes in many transformation rules. Furthermore, if we want to switch between different layout styles it is suitable to separate the values of the attributes in different modules that can be reused. The functionality of attribute value calculation is also an example of *crosscutting* in the transformation definition.

The difference with the previous example is that the crosscutting in the transformation definition is not caused by the decomposition of the application model. The attributes *fontName* and *fontColor* are located in a single place in the application model (class *View*). The crosscutting appears in the transformation definition where the concrete values are set for many views. Of course, the example is very simple and the problem can be easily solved by providing the required values as a set of external parameters shared by many transformation rules. However, in more complex cases the logic for calculating attribute values may become a crosscutting concern.

This second example shows the need for modularization of the calculation of the attribute values for some model elements. Below we show the fragment of the view related part that sets up the values:

```
{fontName='Times', fontColor='Red'}
```

This fragment must be integrated with a number of transformation rules. In Section 5.6 we discuss the support found in model transformation languages to accomplish this integration.

5.4.2 Scenario 2: Multiple Hierarchies in Models

In the first scenario we studied the impact of the decompositions in the intensions on the decomposition of transformation definitions. In the second scenario we focus on another issue that influences the decomposition of transformation definitions. It is related to the structure of models created by transformations. A model is a graph of model elements and the elements are created in a certain sequence. Model element slots are assigned with values also in a sequence. Transformation designer must take into account these sequences to encode them into the transformation definition. Sometimes more than one sequence of model element instantiations may be identified. These sequences often influence each other. This requires a careful planning of order of rule execution.

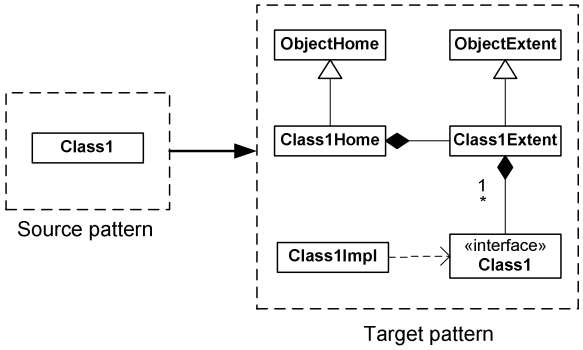
This scenario illustrates a case in which two hierarchies found in the resulting model influence each other. The scenario is derived from a case study provided by Thales Naval Nederland and is based on a real problem used to evaluate the applicability of MDA and model transformations. The full description of the case study is given in [22]. Here we present a simplified version.

In the case study a platform independent model that constitutes a part of a threat evaluation system is specified in UML and is transformed into a platform specific model. The platform specific model is also specified in UML and contains information about the data distribution at runtime. In this scenario the source and target intensions are the same: the UML meta-model. In the previous scenario we dealt with two distinct intensions and the correspondence between them was the starting point for transformation rules identification. Here, this is less important and other factors are more influential.

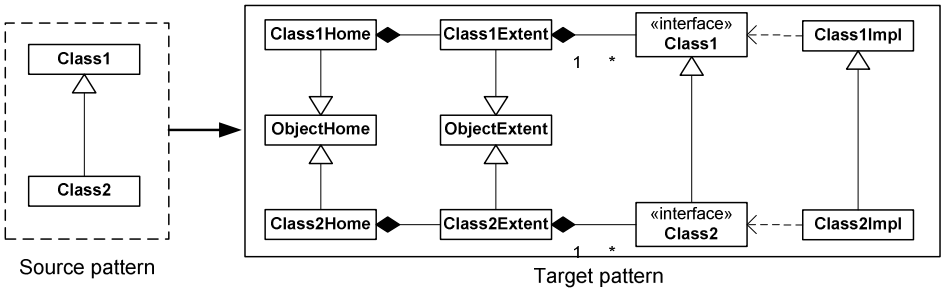
Figure 5.8 shows two transformation rules and one example of a pair of input/output models. The first transformation rule shows how classes in source models are transformed. For every class in source models (e.g. *Class1*) the pattern on the right-hand side of the arrow is instantiated in the target model. The second transformation rule shows how generalization between classes is transformed. The target pattern uses constructs already defined in the pattern in the first rule and adds two generalization relations: between interfaces *Class1* and *Class2*, and between classes *Class1Impl* and *Class2Impl*.

The bottom part of Figure 5.8 shows an example source model that contains three classes related by generalization. The target model is created after applying the transformation rules.

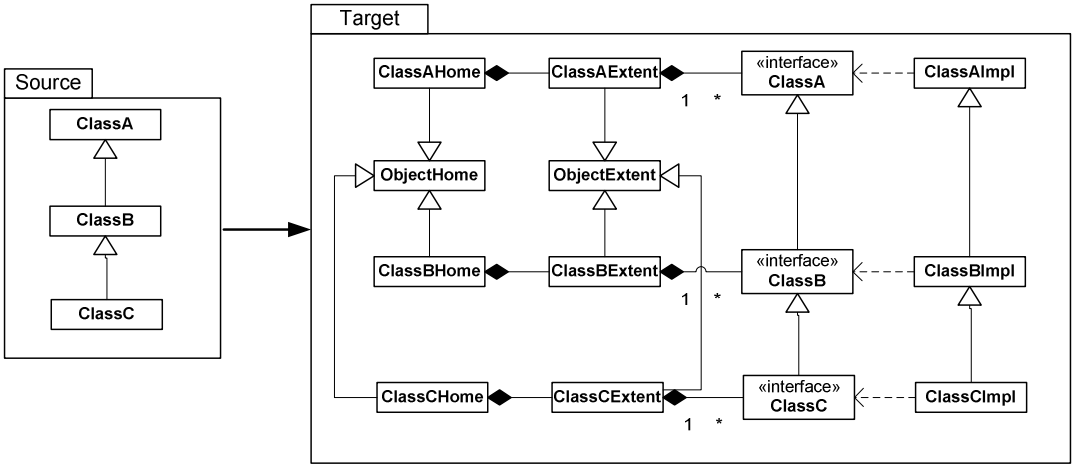
Apart from the execution of the rules shown in Figure 5.8 the transformation also has to generate the classes *ObjectHome* and *ObjectExtent* and the package that holds the target model. Every UML model is a hierarchy of nested model elements. At the top level we have a package (Package *Target* in Figure 5.8) that contains other model elements including classes and possibly other packages. Classes, in turn, contain operations, attributes, etc. These containment relations form a tree. Therefore, transformation definitions may be organized around rules that build this tree. Classes in the target model are derived by the transformation rule applied on classes in the source model.



Rule for transformation of classes



Rule for transformation of generalization



Example of model transformation

Figure 5.8 An example of two transformation rules and a pair of source and target models

There is, however, another hierarchy of model elements alongside the containment hierarchy. It is based on the generalization relationships among the classes. This hierarchy may be incongruent with the containment hierarchy. The two hierarchies may influence the execution order of transformation rules. In our example, the application of the transformation rule on classes is not as straightforward as it seems. *ClassB* must be processed after *ClassA* in order to handle the generalization relation. Another option is first to create

all the classes in the output model and then to process the generalization relations among them. Therefore, the execution order of the rules over concrete classes is important. This brings another source of decomposition of transformation definitions into rules. Following only the part-whole decomposition in the target model is also not enough since there are dependencies among elements at the same level of nesting and dependencies across the levels. There are at least two distinct hierarchies that must be taken into account: aggregation and generalization hierarchies. This leads to decomposition of rules that must be executed in a certain order. The rules may traverse the source model multiple times. The first set of rules may be organized around the containment hierarchy; the second set of rules may be organized around the generalization hierarchy and so on.

This scenario shows the importance of the dependencies among model elements in a single model. This is an issue that must be considered explicitly during the design of a transformation definition.

It should be noted that the type of the transformation language is very important in that case. An imperative or hybrid language requires an explicit order of rules execution. A declarative language may leave the determination of the execution order to the transformation engine. We postpone this discussion till Section 5.6 where different languages are evaluated.

5.4.3 Scenario 3: Trace Information

This scenario illustrates the need for separating part of the transformation logic that is independent of the source and target models and only relies on knowledge about the structure of transformation rules.

Assume that we want to keep trace information about the correspondence between the source and target elements established during transformation executions. For every source element we want to identify the elements in the target model that are created from the source element and the rule that is used. This information can be used for traceability in order to identify the required changes if one of the models is changed. Trace information forms a model whose model elements are created every time a rule is executed over a source element. Models with trace information conform to the meta-model shown in Figure 5.9.

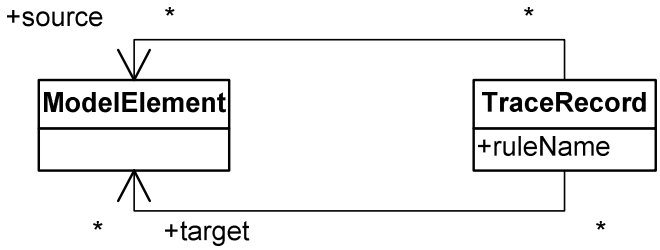


Figure 5.9 The meta-model for trace information

Whenever a transformation rule is executed an instance of *TraceRecord* class is created and the name of the rule becomes value of the attribute *ruleName*. This instance also relates to the source and target elements of the rule.

Some transformation engines and languages may provide this service in their built-in capabilities. However, we want to be independent of transformation languages and transformation engines. We choose to create a custom model for storing the trace information.

The generation of the trace information can be done by enhancing all the rules in a transformation definition with an instantiation instruction that creates an instance of *TraceRecord* and sets up its attribute values. For example, if we enhance some of the rules in Scenario 1 they will look like the following (newly added constructs are underlined):

Model-related part:

```
openQuestionModel (source[OpenElement],
                   target[Open,
                          TraceRecord{ruleName="openQuestionModel",
                                      source=OpenElement,
                                      target=Open}])

multipleChoiceModel (source[MultipleChoiceElement],
                     target[MultipleChoice,
                            TraceRecord{ruleName="multipleChoiceModel",
                                        source=MultipleChoiceElement,
                                        target=MultipleChoice}])
```

Apparently the logic for generating the trace information is scattered across many rules. If this logic is changed this would require changes in all the rules. For instance, a time stamp may be added to indicate the sequence of execution of the rules. Therefore, the logic for trace records should be separated in a single unit that is maintained separately and may be composed with other rules. Also, this unit is a piece of generic functionality independent of a particular transformation definition and can be reused in an arbitrary transformation.

How do we specify this functionality in a separate unit that can be reused? Most transformation languages provide transformation rule as a basic construct. A rule relates source and target elements. However, in the case of generation of tracing information no concrete source for a rule can be identified. Instead, the rule has parts that should be included in the target of every rule in a given transformation definition. The transformation logic is in fact a component that is used in the context of rules and depends on this context. It cannot be executed in a stand-alone manner. Therefore, it may be a problem that a given transformation language does not provide a proper construct for expressing the logic for trace records.

Furthermore, each trace record contains information that is derived from the context rule: the rule name and the source and target model elements. In order to specify the trace generation logic in a generic way we have to abstract from the rule-specific details and to reason about the structure of the transformation rules in general. The generic specification of the trace generation will access the name of the enclosing rule and its source and target elements. In other words, we need introspection capabilities in the transformation language. This may be another problem since the chosen transformation language may lack this capability.

Below we give a sketch how the generation of trace records would be specified as a part of the target in rules. The specification is a template that contains three parameters: *$ruleName*, *$ruleSource* and *$ruleTarget.* This template has to be instantiated for every

rule by including it in the rule target. The parameters have to be replaced with concrete values (e.g. rule name).

Generation of trace record:

target[TraceRecord {ruleName=$ruleName, source=$ruleSource, target=$ruleTarget}]

This functionality is generic and crosscuts transformation rules in a given transformation definition. It should be specified separately to support future changes and for reuse. Other similar examples are generating logging and debugging information about the rules.

5.4.4 Scenario 4: Additive Evolution

In this scenario we study how the additive evolution in one of the intensions affects the transformation definition. The scenario is expired by the example about geometrical shapes presented in [94].

Assume we have an XML schema that defines a simple XML language for describing geometrical shapes. The UML representation of the schema is given in Figure 5.10. The stereotypes refer to XML Schema constructs. The schema defines elements and types for some well-known shapes such as rectangles and ovals and operators over these basic shapes. For simplicity, our schema defines only one shape: *Rectangle*, and two operators: *Intersection* and *Union*. The operators may be applied on shapes and on other operators.

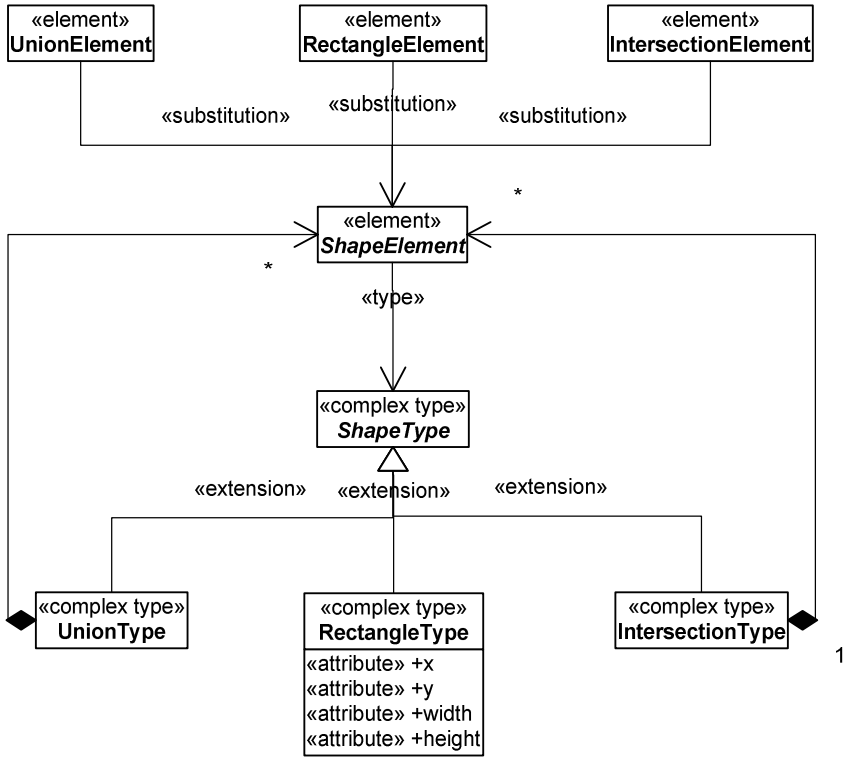


Figure 5.10 UML representation of the XML schema for geometrical shapes.

XML documents conforming to the schema are processed by an application that builds an internal representation of the shapes for the purpose of visualization. The processing is based on a transformation definition that uses the XML schema as a source intension and an application model as a target intension. The application model resembles the source schema. An UML diagram of the application model is shown in Figure 5.11.

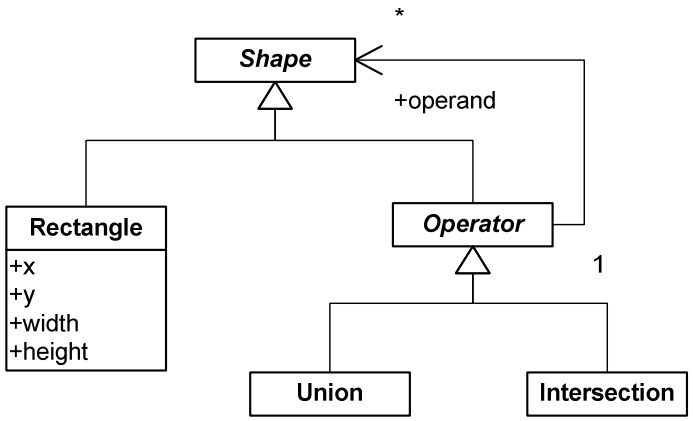


Figure 5.11 The application model of geometrical shapes.

The model is based on the Composite design pattern [41]. All the shapes and operators are represented as classes that specialize the abstract class *Shape*. Operators *Union* and *Intersection* specialize *Operator* class and may have zero or more other shapes as operands.

The transformation definition is decomposed into rules following the types of shapes. The elements for rectangle, union and intersection are transformed to instances of the corresponding classes. A sketch of the transformation definition is given below. We do not show the details how the shapes and the operators are connected together to form a composite hierarchy.

```
Shape transformation:

rectangleRule (source[RectangleElement],
               target  [Rectangle{x=RectangleElement.x,
                                  y=RectangleElement.y,
                                  width=RectangleElement.width,
                                  height=RectangleElement.height}])

unionRule (source[UnionElement],
           target  [Union])

intersectionRule(source[IntersectionElement],
                 target [Intersection])
```

Assume now that the XML schema is extended with a new operator that scales the shapes with a given factor. The scale operator is applied on exactly one shape. The extended schema is shown in Figure 5.12. The new operator *Scale* is represented as an element.

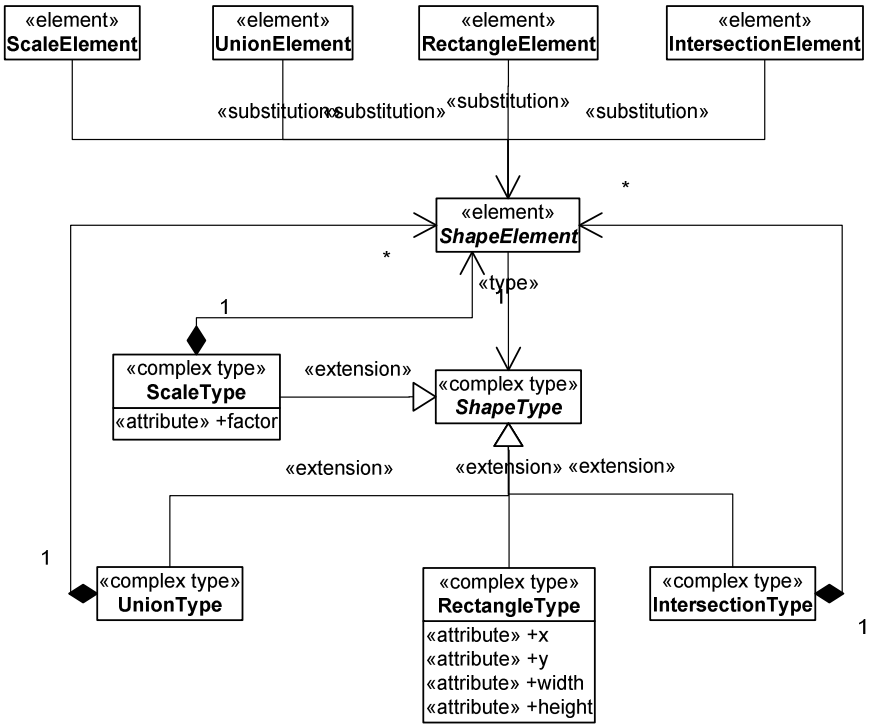


Figure 5.12 UML representation of the extended XML schema for geometrical shapes

We want to specify a new transformation definition for the new schema by reusing as much as possible the old one. The processing of the *Scale* operator can be done in two ways. The first one is to extend the target model with a new class *Scale* that specializes *Operator* class and implements the scaling functionality. This requires addition of one new rule in the transformation definition that maps scale elements to instances of class *Scale*. The second way is to keep the target application model unchanged and to apply the scaling during the transformation execution by multiplying the coordinates and the size of the shapes with the scale factor. We take the second approach to study how current transformation rules will change.

The application of the scale operator to a rectangle scales its dimensions and position by the scale factor. After applying the scale factor the transformation instantiates class *Rectangle* with the scaled position and dimensions.

Scale operators may be nested in an arbitrary depth. This means that the transformation has to multiply the scale factor and to pass it down through the composite hierarchy until the leaves (rectangles) are reached. A scale operator has to multiply the factor calculated by its parent with its own factor. The union and intersect operators have to copy the scale factor calculated by their parents in the hierarchy. Also, every scale operator has to pass the shape that its operand has generated to its parent operator in order to build the composite hierarchy of the target model.

The new rule that is added for the scale element is given below.

Scale element rule:

```
scaleRule (source[ScaleElement],
        target [newFactor=ScaleElement.factor * getParentScaleFactor(ScaleElement),
                Shape=getChildShape(ScaleElement)])
```

In this rule no element in the resulting target model is created. The rule only calculates the new scale factor and copies the shape generated from the child element in order to propagate it up in the hierarchy. We assume that two functions are available: *getParentScaleFactor* and *getChildShape*. These functions navigate to the parent and to the child element respectively and obtain the corresponding value for the scale factor and the shape object. The functions are also an example that a rule needs to access the elements created by other rules applied on the source model elements. This is a common functionality that most of the current transformation languages offer. In this transformation specification we are not concerned with the implementation of that functionality. This is discussed later in section 5.6.

Rule *scaleRule* should be added to the existing transformation. The rest of the rules, however, have to be changed to accommodate the processing logic related to the scaling operator. The new version of the rules is given below. The newly added constructs are underlined.

Shape transformation with application of scale operator:

```
rectangleRule (source[RectangleElement],
                target[Rectangle
               {x=RectangleElement.x * getParentScaleFactor(RectangleElement),
                y=RectangleElement.y * getParentScaleFactor(RectangleElement),
                width=RectangleElement.width * getParentScaleFactor(RectangleElement),
                height=RectangleElement.height * getParentScaleFactor(RectangleElement)}])

unionRule (source[UnionElement],
           target[Union,
                  factor=getParentScaleFactor(UnionElement)])

intersectionRule (source[IntersectionElement],
                  target[Intersection,
                         factor=getParentScaleFactor(IntersectionElement)])
```

The processing of scale operator changes the way how the attribute values of *Rectangle* are calculated (rule *rectangleRule*). *unionRule* and *intersectionRule* include new element *factor* in their targets. This new element has to be added to the rule for any new operator that eventually is added in the future. It crosscuts the target part of multiple rules and therefore this part has to be specified separately and composed with the rules that process the operators in the source document.

This scenario illustrates how a transformation definition may be affected if an additive change occurs in one of the intensions, in our example, the source XML schema. As a result the transformation logic is modified and all the rules are affected. Both the target of rules and the attribute value calculation are affected. There is a functionality introduced by the new construct that crosscuts other rules related to a set of source components: the elements that represent shape operators. Similarly to Scenario 3 this scenario illustrates the need for separating and integrating a crosscutting part in transformation definitions.

5.5 Transformation Language Requirements and Features

In this section we analyze the scenarios and identify a set of requirements that a language has to fulfill to implement the transformations described in the scenarios. We identify language features relevant for every requirement. These features are used in section 5.6 to evaluate a set of transformation languages.

5.5.1 Transformation Language Requirements

Scenario 1: Decomposition of Models in Multiple Dimensions

This scenario shows the need for constructs that allow decomposition of transformation definitions on the base of decompositions of the source and target intensions. The rules required in the scenario are based on *1-to-n* relations among the source and target model elements but we can generalize this to *m-to-n* relations. This means that it should be possible to specify rules that contain multiple model elements in rules source and target parts. Scenario 1 also showed that it should be possible to process a single element in the source model by more than one rule. In the scenario the elements for exam item types in source documents are selected and processed by at least three rules that create the model, the view and the controller objects respectively. This leads to the following requirement concerning the structure of transformation rules:

> *Requirement 1*: Transformation languages should support rules based on *m-to-n* relation between the source and target parts. It should be possible to apply more than one rule over a single model element.

Scenario 1 also showed that rules must be able to obtain model elements created by other rules. If this is not fulfilled then the rules in the decomposition proposed in Scenario 1 will not be able to build the collaboration between models, views and controllers in the application. The following requirement is formulated concerning interaction among rules:

> *Requirement 2:* Transformation languages should support interaction among rules that allows elements created by a given rule to be accessed by other rules.

Furthermore, we identified the need of separation of the logic that calculates attribute values into a single language construct. This construct can be further integrated with multiple rules that instantiate model elements. This is a requirement for a proper compositional operator. We derive two requirements from this example:

> *Requirement 3:* Transformation languages should support specification of functionality for calculation of attribute values in a separate construct.

> *Requirement 4:* Transformation languages should provide compositional mechanism that composes constructs for calculation of attribute values with multiple rules.

Finally, the required reuse of set of rules leads to a requirement for packaging mechanism that groups together transformation definition constructs in a single unit.

Requirement 5: Transformation languages should support packaging mechanism for organizing transformation constructs into modules.

Scenario 2: Multiple Hierarchies in Models

The second scenario showed that the transformation designer has to consider the execution order among rules. Rules may have dependencies among each other that influence the execution order. We put a requirement that the transformation language provides support in determining the execution order. In fact, this is a feature that may be supplied by the transformation engine and not directly by the language.

Requirement 6: Transformation languages/engines should support the identification of execution order among rules in case of dependencies among rules.

Dependencies among rules require communication among rules. Therefore, Requirement 2 is also relevant for this scenario.

Scenario 3: Tracing Information

This scenario showed the need for modularizing parts of transformation rules that do not form a complete rule. In the scenario, such parts were included in the target of other rules. The transformation language should provide compositional operator for the inclusion. This case is similar to Scenario 1 where the calculation of some attribute values crosscuts multiple rules. Similarly to Scenario 1 we formulate two requirements: one for the required modular constructs and one for the required compositional operator.

Requirement 7: Transformation languages should support specification of parts of rule target in a separate construct.

Requirement 8: Transformation languages should provide compositional mechanism that composes parts of rule target with targets of multiple rules.

Furthermore, this scenario requires reflective capabilities in transformation languages to specify generic functionality. In the example, the transformation language should provide a representation of the transformation rules at runtime. This representation should allow at least introspection of rules. A consequence is that the transformation definition under execution is navigable and information about the rules may be retrieved.

Requirement 9: Transformation languages should support reflection.

Scenario 4: Additive Evolution

In this scenario some of the transformation rules must be changed to accommodate the changes in the source intension. Changes concern replacement of constructs for calculation of attribute values. However, we can envisage other cases where other parts must be replaced, for instance, constructs that must be instantiated. We formulate the following requirement:

Requirement 10: Transformation languages should support replacement of parts of transformation rules.

In Scenario 4 we observe a situation similar to Scenario 3. The existing transformation rules must be enhanced with an additional part that crosscuts multiple rules. This is the part responsible for passing the scaling factor across the document hierarchy. This situation is already captured in Requirement 7 and 8. Requirement 5 is also relevant in this scenario.

5.5.2 Transformation Language Features

In this section we derive language features relevant for every requirement. The language features are modularity, rule interaction and ordering, compositional operators, adaptation of transformation definitions, and reflection.

The relation among language features, requirements and scenarios are summarized in Table 5.1. Most features are required in more than one scenario in different contexts and are related to a number of requirements.

Feature	Scenario 1: decomposition in multiple dimensions	Scenario 2: multiple hierarchies in models	Scenario 3: tracing information	Scenario 4: Additive evolution
Modularity	Req1, Req3, Req5		Req7	Req5, Req7
Rule interaction and ordering	Req2	Req2, Req6		
Compositional operators	Req4		Req8	Req8
Adaptation of transformations	Req4		Req8	Req10
Reflection			Req9	

Table 5.1 Relations among language features, requirements and scenarios

We describe each language feature below:

- *Modularity*: Modularity in the context of a programming language means ability to develop programs as assemblies of smaller units, usually called *modules*. A set of requirements may be imposed upon the modules. Meyer [64] treats modularity as a means to achieve two quality properties in software: extensibility and reusability. He considers the notion of modularity in the context of a software construction method and defines 5 criteria, 5 rules and 5 principles that must be met (respectively followed) in order to call a basic unit of decomposition a *module*. We select only those criteria, rules and principles that are relevant to our discussion. Modules must satisfy the criterion for *modular composability*. This means that modules may be combined with each other to produce other modules. Furthermore, Meyer defines the rule for *direct mapping*. The rule requires that the modular structure of a software system must be compatible with the model of the problem domain for which the system is built. Finally, we select the principle of *linguistic modular units* which states: “modules must correspond to syntactic units in the language used” [64]. The notion of modularity is related to requirements 1, 3, 5 and 7;
- *Rule interaction and ordering*: *Interaction* among transformation rules occur when one rule needs to use the effects of another rule. This effect may be an update, a dele-

tion or a creation of a model element. In the most common case a rule needs to access a model element created by another rule. This may be done in multiple ways, for example, by a direct rule invocation or by tracing the elements created from a given source element. *Rule ordering* is concerned with the determination of the execution order of transformation rules. This feature is related to requirements 2 and 6 and indirectly to all the requirements concerning changes in transformations.

- *Compositional operators*: Compositional operators are language mechanisms that allow construction of modules on the base of other modules. Compositional operators are related to the language modularity. As we mentioned above language modules must meet the criterion for *modular composability*. Another property that is relevant in that case is the *closure* property which indicates the capability of a result of a composition to be further composed with other modules [6]. Requirements 4, and 8 indicate a need of various compositional operators in transformation languages;
- *Adaptation of transformation definitions*: This language feature indicates the capability of a transformation language to express changes over existing modules in transformation definitions. Changes may be addition, deletion and replacement of various constructs. This language feature is indicated in requirements 4, 8, and 10;
- *Reflection*: Reflection is a capability of a language to express and execute programs that perform computations on themselves. The computation may be *introspection* (observing the structure of the program) and *intercession* (altering the structure and behavior of the program) [63]. Requirement 9 indicates a need for reflection in transformation languages;

5.6 Evaluation of Transformation Languages

In this section we evaluate a number of transformation languages described in various publications. The selected languages are: ATL [18], DSTC, IBM and CBOP submission to QVT RFP (abbreviated here as *DSTC*) [30], Great [2], MOLA [51], QVT Partners submission to QVT RFP (abbreviated here as *QVTP*) [89], TRL [8], UMLX [102], Viatra [100], and YATL [84]. We have to state that the evaluation is done on the base of the descriptions found in the listed references. Many of the languages are work in progress and the available documentation is not always complete.

Czarnecki and Helsen [27] give classification of transformation languages based on different criteria. Chapter 4 presented a summary of this classification. Classification categories will be used throughout the evaluation.

Evaluation is given in the following way. For every language feature we discuss the relevant requirements. For every requirement we analyze how it is fulfilled by the evaluated languages. Results are summarized in Table 5.2. For every requirement in the context of a language feature we assign a level of support of the requirement by a language. Three levels of support are used: *supported*, *partially supported* and *not supported*. If the reference that describes the language does not provide enough information to justify the evaluation we use *not clarified*. The evaluation is described in the remaining part of this section.

	Modularity				Rule Interaction and Ordering		Compositional Operator		Adaptations of transformation definitions			Reflection
	R1	R3	R5	R7	R2	R6	R4	R8	R4	R8	R10	R9
ATL	partially supported	supported	supported	supported	supported	partially supported	supported	supported	partially supported	partially supported	partially supported	supported
DSTC	supported	supported	supported	supported	supported	supported	supported	supported	partially supported	partially supported	partially supported	not supported
Great	supported	supported	not clarified	supported	supported	not supported	supported	supported	not clarified	not clarified	not clarified	not supported
MOLA	supported	supported	not clarified	supported	supported	not supported	supported	supported	not clarified	not clarified	not clarified	not supported
QVTP	supported	supported	not clarified	Supported	supported	partially supported	supported	supported	partially supported	partially supported	partially supported	not supported
TRL	supported	supported	supported	supported	supported	not supported	supported	supported	partially supported	partially supported	partially supported	not supported
UMLX	supported	supported	not clarified	supported	supported	not supported	supported	supported	not clarified	not clarified	not clarified	not supported
Viatra	supported	supported	not clarified	supported	supported	not supported	supported	supported	not clarified	not clarified	not clarified	not supported
Yatl	supported	supported	supported	supported	supported	not supported	supported	supported	partially supported	partially supported	partially supported	not supported

Table 5.2 Evaluation of the support provided by transformation languages for features and requirements

5.6.1 Modularity

Requirement 1

Transformation rule is the basic construct in transformation languages. Typically a rule has two parts: a left-hand side and a right-hand side. Rules may be interpreted as functions that accept arguments conforming to the left-hand side pattern and return result specified by the right-hand side. Other types of rules perform update and deletion of already existing elements in a model and can be considered as rules that do not strictly follow this two-part structure.

Most transformation languages allow definition of *m-to-n* rules where the source and the target of the rules contain multiple components that form a pattern. We assume that Requirement 1 is supported by all the languages.

Requirement 1 also states that a language should allow processing of a single source element by multiple rules. Among the evaluated languages ATL is the one that does not allow this. ATL is considered as partially supporting the requirement. The language requires that a source element participates in at most one input pattern. In the context of Scenario 1 this prevents the definition of multiple rules for every exam item type. In other words, the software engineer is forced to encode this part of the transformation in a single rule.

Requirement 3 and Requirement 7

Our scenarios show that apart from transformation rules other modules are required as well. Scenario 1, 3, and 4 show that parts of the right-hand side of rules must be separated and reused across rules. Requirement 3 is about the separation of attribute value calculations into a separate module. Requirement 7 is about the separation of parts that are included in the right-hand side of multiple rules.

Generally, the required units of transformation logic shown in the scenarios can be structured as transformation rules despite the fact that they do not follow the two-part structure of rules. Languages with internal explicit scheduling (ATL, MOLA, TRL, YATL) may specify transformation definitions with rules that invoke other rules in an explicit order. Requirement 3 can be met by specifying the calculation of attribute values in a rule that will be invoked by other rules. The exact source and target model elements may be passed as input parameters. Input parameters of rules are supported in almost every transformation language. Already created target elements may be obtained via traceability mechanism and their attribute values may be assigned by different rules.

Requirement 7 can be met in the same way: a rule is specified that creates only a part of the target model and that rule is invoked by multiple rules. In Scenario 3 all the rules must include an invocation to a rule responsible for the generation of trace records. This approach usually requires several passes over the source model, a feature that most of the imperative and hybrid languages support. This is also valid for the graph transformation-based languages Great, Viatra and UMLX that rely on external explicit scheduling structure over the application of the rules.

Apart from the transformation rule used as a basic module, some languages provide other types of modules. DSTC provides a construct for specifying patterns that can be referred to from other rules and can be used in both the left and the right-hand side of the rules. This can be used to satisfy requirements 3 and 7. Another interesting feature unique for this language is the implicit instantiation of the target elements. Different rules may

build different parts of the same target model element. It is not known in advance which rule will be executed first. The transformation engine creates the model element on demand when the first rule that contains an instantiation instruction is invoked. Subsequent rules that refer to the same target element do not cause new instantiation. Instead, the already created model element is used. This approach helps in decomposing a transformation into parts that reflect the different views over the target model. The attribute values of a single target element may be assigned by multiple rules.

In general, we may conclude that requirements 3 and 7 can be fulfilled by the evaluated transformation languages by using transformation rules as modular constructs. It should be noted, however, that the principle of linguistic modular units explained in the context of modularity is not always met. Different modules identified in the scenarios are mapped to transformation rules. To fully satisfy the principle of linguistic modular units we recommend that languages should support two additional modular units. The first one should allow specification of parts of rule target in a separate unit. The second one should allow specification of attribute values in a separate unit. DSTC is the language that supports these units at a highest degree among the evaluated languages.

Requirement 5

This requirement concerns packaging constructs that group multiple rules in a single module. Although this is a well-known mechanism in programming languages some of the available proposals do not address this issue yet in the available descriptions (Great, QVTP, MOLA, UMLX, Viatra). Therefore, we consider their support as *not clarified*.

5.6.2 Rule interaction and ordering

Requirement 2

As we saw in the scenarios rules has to interact with each other to produce a complete target model. There are two ways of rule interaction: using traceability links and rule invocation. Traceability links establish associations between the source elements and the target elements created by a given rule. Rule invocation mechanism allows one rule to invoke another rule and to use the result of the execution of the invoked rule.

In some languages the transformation engine provides dedicated support for creation and maintenance of traceability links (ATL, TRL, YATL, DSTC). Other languages leave the responsibility for maintaining traceability links to the user (Viatra, Great). The proposal by QVT Partners does not clarify this issue.

Most languages provide both traceability links and rule invocation as forms of rule interaction. In DSTC proposal there is no support for explicit rule invocation.

Generally, either of these mechanisms of interaction is enough to satisfy Requirement 2. Since all the languages support at least one of the mechanisms we consider Requirement 2 as supported by the languages. However, the mechanisms differ in the effect they have on the adaptability of a transformation definition. This issue is discussed further in the section on adapting transformation definitions.

Requirement 6

Requirement 6 calls for support for identification of the execution order among the rules in a transformation. The level of support varies in different transformation languages. It is related to the form of rule scheduling.

Explicit form of rule scheduling uses well known control flow structures. Transformation definitions written in this form are similar to programs written in an imperative language. Languages with this form of scheduling are: ATL, TRL, YATL, MTL, MOLA, Great, Viatra. These languages leave the determination of the execution order to the transformation developer. He must be aware about all the dependencies among the rules and has to specify the order of possibly multiple passes over the source model. Problems that the developer might encounter are similar to the problem presented in Scenario 2. We can conclude that the languages with an explicit form of scheduling do not support Requirement 5 since they leave the determination or execution order to the software engineer.

The implicit form of scheduling relies on dependencies among rules and is often found in declarative transformation languages. The transformation developer describes the relationships between the source and target intensions. Transformation engine determines the execution order on the base of the dependencies among rules. A typical example of such a language is DSTC. This language supports Requirement 6. Hybrid languages such as QVTP and ATL provide partial support of implicit rule scheduling. Therefore, we consider them as partially supporting the requirement.

The division between explicit and implicit form of scheduling is not absolute. Almost all the languages classified in one of the forms possess also features from the other. For example, Great and Viatra allows some form of non-deterministic execution of rules. ATL is a hybrid language that combines declarative and called rules (the latter ones have an imperative section based on control flow constructs). In the ideal case the transformation developer can completely leave the determination of the execution order to the system.

5.6.3 Compositional Operators

Compositional operators in transformation languages allow defining new transformation modules from other transformation modules. Generally, two compositional mechanisms are found in the transformation languages: composite rules and rule inheritance.

Composite rules are built from other rules. In languages with explicit scheduling form, rules invoke other rules to use their functionality. More complex rules are defined by calling simpler ones. Therefore, we can treat rule invocation as a compositional mechanism for building composite rules. Languages that support rule invocation are TRL, YATL, MOLA, ATL, and QVTP. In addition QVTP provides logical operators over existing rules. The language that does not provide rule invocation is DSTC.

A variant of this form of composition is observed in Viatra and Great where the scheduling is external to rules. A rule cannot directly invoke other rules. Rule execution order is specified with control flow mechanisms but the specification is external to the rules.

The second compositional mechanism is based on inheritance. Languages that support (or at least report) inheritance are TRL, ATL, DSTC, QVTP. This mechanism allows definition of new rules by inheriting the components in other rules such as the source and target patterns. New rules, in turn, may define new constructs and override inherited constructs. TRL provides also a mechanism for transformation unit extension. In TRL unit is

a packaging mechanism for transformation rules. A unit may be inherited by another unit and its rules may be redefined and overridden. Unfortunately, the description of the language does not give sufficient details on that feature and its usage.

Requirement 4 and Requirement 8

Scenarios 1, 3, and 4 show examples of composition of transformation functionality. We already discussed how this functionality can be modularized in transformation languages as rules. Here we consider how the rules may be composed together by using the compositional mechanisms of rule invocation and inheritance.

The first compositional mechanism relies on rule invocation. Assume that in Scenario 1 the calculation of attribute values is represented in a rule. This rule is invoked by other rules that provide the elements whose attributes will be assigned with values. The same approach is applicable to Scenario 3 where the generation of tracing information may be separated in a rule invoked by every rule in the transformation definition. The invoking rule must pass parameters to the invoked rule.

These considerations show that it is possible to implement the required composition by using rule invocation. However, scenarios also impose other requirements related to the adaptability of the transformation definitions. This issue is discussed in the next section where we will see that this solution does not satisfy the requirements for adaptability.

The second compositional mechanism is based on rule inheritance. It can be applied in three different ways. They are illustrated in Figure 5.13 taking Scenario 3 as an example. We assume that the trace generation functionality is expressed in a rule. The need of reflection is not considered.

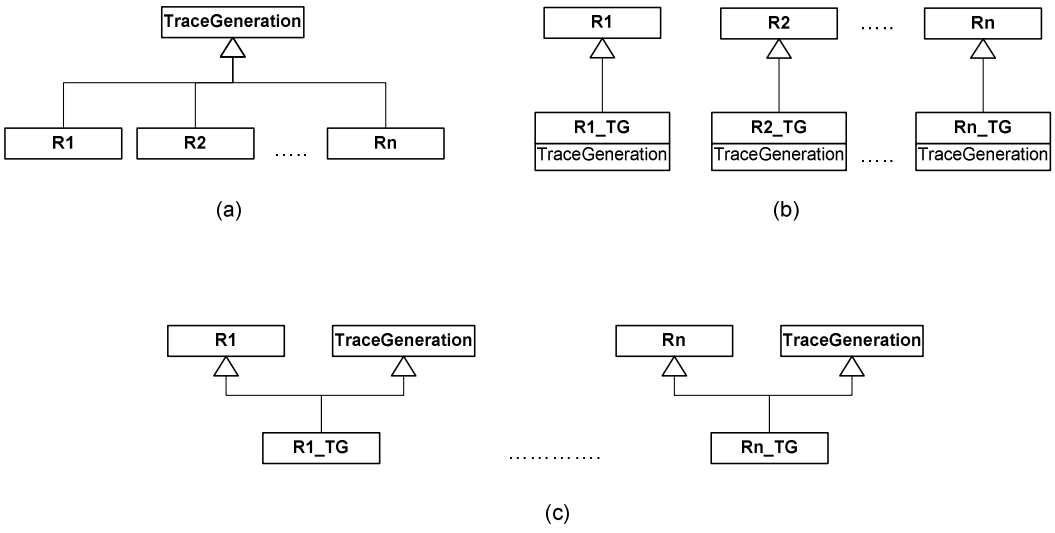


Figure 5.13 Three implementations of composition based on rule inheritance: implementation (a) based on a common parent rule, implementation (b) based on definition of separate rules, and implementation (c) based on multiple inheritance among rules

All the three implementations of the scenario expose some problems. Assume that a transformation definition contains *N* transformation rules: *R1, R2,…, Rn* that must be enhanced with trace generation functionality. Figure 5.13a shows the first possible implementation: all transformation rules inherit the rule that implement the trace generation. If a

new rule is added it will also inherit that rule. The problem with this solution is that all the rules are coupled with the trace generation and it is not possible to use them without that functionality. Recall that it should be possible to include and exclude the trace generation from a given transformation definition. However, the inheritance mechanism usually does not allow altering the parent class.

Figure 5.13b shows the second implementation. In contrary to the implementation (a), here rules *R1*, *R2*,…, *Rn* are inherited and the trace generation functionality is added in the inheriting rules. This implementation has two problems. The first one is caused by the repetition of the trace generation functionality in every inheriting rule. Eventual changes in that functionality will lead to changes in all the transformation rules. Of course, this can be avoided if trace functionality is implemented as a separate rule and the inheriting rules *Ri_TG* invoke it. However, this leads to a second problem: specification of an excessive number of rules that do not add new functionality and are created as 'glue' constructs.

The same problem of definition of excessive number of transformation rules is observed in the third implementation option shown in Figure 5.13c. It uses multiple inheritance among rules (assume it is supported by the transformation language). Again, we create a new rule *Ri_TG* for every rule *Ri* with the only purpose to 'glue' two existing rules.

In principle, inheritance can be applied to implement the compositions required in the scenarios. However, certain quality properties of the transformation definitions are deteriorated. The required compositions are examples of crosscutting of some functionality over multiple units. Application of inheritance leads to anomalies in the transformation definitions. Application of rule invocation has problems related to the adaptability of definitions. These problems are discussed in the next section.

In general, we may conclude that requirements 4 and 8 are supported by the described compositional mechanisms. However, this is often at the price of deterioration of various quality properties of transformation definitions.

5.6.4 Adaptation of Transformations

Adaptation of transformations is required in scenarios 1, 3 and 4. In this section we discuss how the requirements formulated for these scenarios may be fulfilled with the mechanisms in current transformation languages.

Requirement 4 and Requirement 8

Requirement 4 is formulated in the context of Scenario 1. In this scenario it should be possible to adapt the logic for attribute values calculation. A module that represents this logic must be integrated with multiple rules. Furthermore, this integration should allow replacement of the module with another one. Similar situation is observed in the case of Requirement 8. Requirement 8 is derived from Scenario 3. The adaptation of transformation definitions in Scenario 3 adds trace generation functionality to all the rules. It is required that the new functionality is implemented in a separate module and is reusable. There should be no strong coupling between the existing rules and the added module in the sense that the new module can be removed later.

In essence both cases require a composition between rules. In the section about compositional operators we discussed the available compositional mechanisms. Here we consider them again from the perspective of adaptability they provide.

Composition based on rule invocation. Assume that in Scenario 1 a new rule that provides other attribute values should be used in place of the old rule.

The replacement of rules is highly influenced by the form of rule scheduling. Languages with explicit scheduling rely on explicit rule calls and specification of control flow. This means that every replacement of a rule leads to potentially multiple changes in the transformation code. Current languages do not address this problem in the provided language descriptions. In the worst case the changes must be implemented manually.

It is worth to mention some constructs in TRL that can probably be used to handle replacement of rules. The first construct is for invocation of rules (*applyRule* construct). It relies on an algorithm for identification of the rule that will be executed. The concept of a virtual rule is mentioned. The second construct is module inheritance. The idea behind it is to allow redefinition of rules while still preserving their names. Unfortunately, TRL submission does not give enough details about these constructs.

Two languages define only a visual syntax: MOLA and UMLX. The first one is based on the control structures found in the traditional structured languages. The second is similar to graph transformation techniques. It is not clear if these languages provide any other means for adapting the control flow of a transformation beyond the direct manipulation of its visual representation.

In our view the languages with implicit form of scheduling handle replacement of rules in a better way. Languages that support that type of scheduling are mainly declarative and hybrid languages (DSTC, QVTP, and ATL). In these languages it is possible to have loosely-coupled rules that do not interact with each other via explicit calls. Furthermore, declarative languages usually do not specify an explicit flow of control. Rules interact via traceability links. This interaction results in loose coupling between rules and facilitates replacement of rules.

In DSTC for example, new rules may be added and rules may be replaced without affecting the rest of the transformation rules. The only requirement is to preserve the name of the link established for traceability. The name is used by other rules to obtain the result of the execution of a given rule. In other words, link names form an interface for communication between rules.

QVTP and ATL are hybrid languages that allow both declarative rules and explicit rule calls. QVTP, however, does not describe a traceability mechanism in its current proposal.

However, in Scenario 3 the usage of a declarative language with an implicit scheduling does not help since generation of the trace records uses elements that are selected by certain rules. To the best of our knowledge, current transformation languages cannot express rules that select only those source elements that have been selected by other rule(s). Such information is not known before runtime.

Composition based on rule inheritance. Applying inheritance as a composition mechanism also leads to problems. Three possible applications of inheritance have been shown in Figure 5.13. Solution (a) is not applicable since it does not allow the removal of the trace generation functionality. Solution (b) introduces new rules and therefore all the invocations to the inherited rules must be updated. Current transformation languages do not support polymorphic rule calls. This problem can be partially solved in DSTC language

since it allows rule superseding. Thus, rules *Ri* will be inherited and also superseded by rules *Ri_TG*. However, this solution still leads to excessive definition of new rules, a problem already explained in the previous section. Solution (c) has the same problems: it introduces new rules and leads to excessive definition of rules.

In summary we assign the following levels of support for requirements 4 and 8. For the languages with external explicit scheduling we do not have enough information how the flow of control can be adapted. These are Great, MOLA, UMLX, and Viatra. We indicate not clarified about how they support requirements 4 and 8. For the languages that support traceability there is a possibility for loose coupled rules. However, this is not enough in the case of Scenario 3. Therefore we consider these languages as partially supporting the requirements.

Requirement 10

Requirement 10 is derived from Scenario 4. In this scenario the calculation of some attribute values is changed. Usually, this functionality is encapsulated in rules that create the model elements with the attributes to be assigned with values. The only way to implement the changes is to override the rule. This can be done by applying rule inheritance or introduction of a new rule. These solutions introduce problems already discussed in the previous section.

Languages Great, MOLA, UMLX, and Viatra do not address this problem in the provided descriptions. The issue is therefore not clarified. Languages that support inheritance (ATL, DSTC, QVTP, TRL, YATL) allow overriding parts of the rules. However, inheritance introduces new rule and requires changing the invocations to the old rule. We decide to assign a partial support for these languages. DSTC does not support rule invocation. It allows inheritance and superseding between rules. Therefore, the inheriting rule may also cancel the inherited rule. However, the description of the language does not clarify if overriding of components in the inherited rule is possible. Therefore, we consider this language also as partially supporting the requirement.

5.6.5 Reflection

The need for reflection mechanism in transformation languages is illustrated in Scenario 3. To implement Scenario 3 we need to access the representation of a rule as an instance of its meta-class during the execution of the transformation. This requires at least introspection to be supported by the transformation language and its execution engine.

The issue of reflective capabilities in transformation languages is not well studied. Among the evaluated languages, only ATL claims to provide this functionality. In ATL the transformation rules and the transformation meta-model are navigable during the execution of transformations. We consider ATL as supporting the requirement. Other languages do not support it.

5.7 Implementing Language Extensions by applying Transformations

The previous section presented an evaluation of some transformation languages regarding their support for modularity, composition, and adaptation. We can conclude that there is hardly a perfect language. Languages usually have a focus on a given set of problems and perform strongly in solving these problems while they may perform weakly for another set of problems.

Some transformation scenarios shown in this chapter require adaptations in existing transformations. The desirable way to do the adaptations is through the constructs provided by the used language. However, in some cases the language may not have the required capabilities. Model transformation technology provides a way to solve this problem. A transformation definition may be adapted by executing a transformation on it. Generally, this can be done in two ways: by writing an ad-hoc transformation that solves only one particular problem and by implementing an extension of the transformation language.

In this section we investigate both approaches. Section 5.7.1 illustrates the case of applying a special purpose transformation for a concrete problem. Section 5.7.2 discusses a more general approach based on language extensions. Section 5.7.3 gives an example application of this approach.

5.7.1 Adapting Transformations by Applying other Transformations

Consider the problem explained in Scenario 3. It requires a change of a transformation definition that adds new constructs in the target part of all rules. Every addition is tuned to the concrete transformation rule. We saw that if the transformation language does not support reflection then the additional functionality cannot be specified in a generic way. A solution to overcome the lack of reflection in this case is to define a transformation (possibly written in the same language) that changes the transformation rules accordingly. This is possible because transformation definitions by themselves are models and therefore may be transformed. The approach is outlined in Figure 5.14.

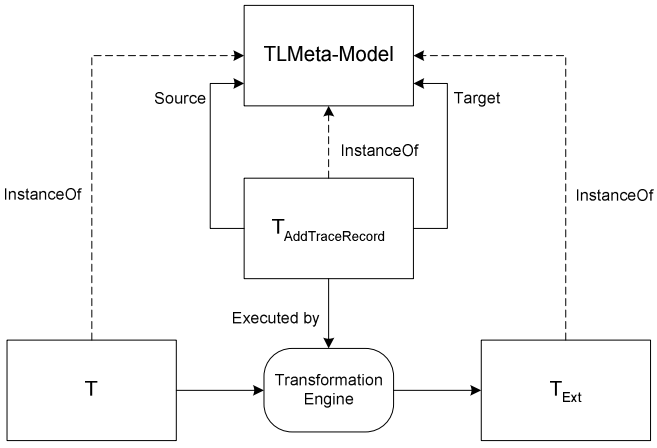


Figure 5.14 Extending a model transformation with functionality for tracing information

Transformation T is transformed by another transformation $T_{AddTraceRecord}$ to a new transformation T_{Ext} that contains the rules of T extended with functionality for generation of tracing information. The transformation $T_{AddTraceRecord}$ is written in the same language as T and T_{Ext}. It may also be written in any other transformation language.

This approach solves the problem at hand. Also, it is generic in the sense that it is applicable to every transformation T written in the given transformation language. However, the approach has three disadvantages. First, the generation of tracing information is not specified in the way it would appear if it was written in the transformation language. Instead, the meta-classes in *TLMeta-Model* are instantiated. If the transformation logic changes then the transformation developer should know the details of the transformation language meta-model in order to apply the changes. The second and more important drawback is that this is a rather ad-hoc approach. The transformation $T_{AddTraceRecord}$ implements operation that is useful in many other scenarios: it changes transformation rules by adding new functionality to them. This is a generic functionality and it is worth to reuse it and apply it in other situations. For example, we may want to add logging information instead of tracing information. Therefore, we aim at generalizing $T_{AddTraceRecord}$ by parameterizing it with respect to the rules that are modified and to the functionality that is added. A third drawback is that even though the transformation is generic enough it remains separated from the language. It is better to make its functionality available at the language level as an additional language construct. This gives an insight for the second approach shown in the next section. It is based on extending the transformation language.

5.7.2 Language Extensions

One possible way to improve a language that does not suite well for a given problem is to extend it with the required constructs. In the context of our example this means that the language is extended with new modular constructs and new compositional operators.

Designing languages in a modular and extensible way is not a new problem in computer science. This problem has also proved to be hard in the general case. It involves a number of sub-problems concerning the elements of a language. Extending a language requires extensions of the grammar, the parser and the interpreter of the language. These problems have been addressed in the literature on specifying modular grammars and parsers, and specifying modular semantic specifications of programming languages. Almost every framework for defining language semantics has treated the problem in some way. To name some of the approaches we mention modular attribute grammars [35], monadic semantics introduced to improve the denotational semantics [65], modular operational semantics framework [68], and the action semantics which is devised especially for solving the modularity problems in language semantic specifications [67].

This section presents an approach for extending transformation languages based on techniques that belong to the domain of model transformations. The extension of a language is done in its abstract syntax definition. Then, the added language construct are associated with a transformation that can be written in the language being extended or in another transformation language. When a transformation written in the extended version of the language is used, it is transformed to a transformation that does not contain the new constructs.

We assume that the concrete syntax of a language is described in a grammar and there is a translator (compiler or interpreter) for the language. The grammar is extended with

new constructs and this extension requires also an extension of the parser. In this chapter we do not address this problem. Some approaches can be found in [1][4][29]. We assume that the extension is done to the abstract syntax of the language. Since the abstract syntax is defined as a model then the extension of the syntax is an extension of the model.

The added language constructs can be handled in two ways: by extending the language translator and by keeping the translator intact. Changes in the translator may not always be possible. In MDE transformations we envisage scenarios where the transformation language translator is a third-party component that cannot be extended easily. Therefore, we limit ourselves to the second case where the language abstract syntax is extended but the language translator is kept intact.

Our approach is very similar to definition of a macro in the language being extended and the handling of the extensions is similar to the macro expansion.

Figure 5.15 shows the meta-model *TLMeta-Model* of a transformation language that defines the abstract syntax of the language. We refer to the non-extended version of the transformation language as basic language.

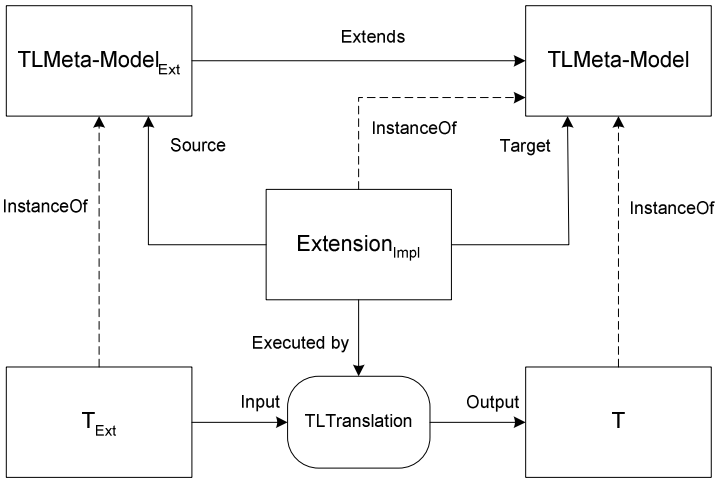


Figure 5.15 Language extensions based on transformations

The meta-model of the basic language is extended with new constructs and the extended meta-model is denoted as *TLMeta-Model*$_{Ext}$. The newly added constructs are associated with a transformation written in the basic language. In Figure 5.15 this transformation is denoted as *Extension*$_{Impl}$. It uses *TLMeta-Model*$_{Ext}$ as a source meta-model and *TLMeta-Model* as a target meta-model. This transformation is executed for every transformation T_{Ext} written in the extended transformation language and the result is a transformation T written in the basic language. *Extension*$_{Impl}$ replaces all the instances of the new constructs in *TLMeta-Model*$_{Ext}$ with their equivalents written in the basic language. In this way we reuse the transformation engine for the basic language without changing it. Moreover, transformation definition *Extension*$_{Impl}$ may be written in any other transformation language capable of transforming models. The designer of the language extension may select the language that suits best for the implementation of the transformation. Since the transformation definition *Extension*$_{Impl}$ operates on a transformation and generates another transformation as an output we can regard it as a *higher-order transformation*.

The approach illustrated in Figure 5.15 may be applied on the extended language thus extending it even further. The extensions may even rely on the previously extended lan-

guage. In that way we can form a series of extensions that rely on sequential transformation composition. The processing of the most complex language is based on a sequence of transformation applications until a transformation written in the language for which we have an engine is produced. An example of such a series of extensions is shown in Figure 5.16.

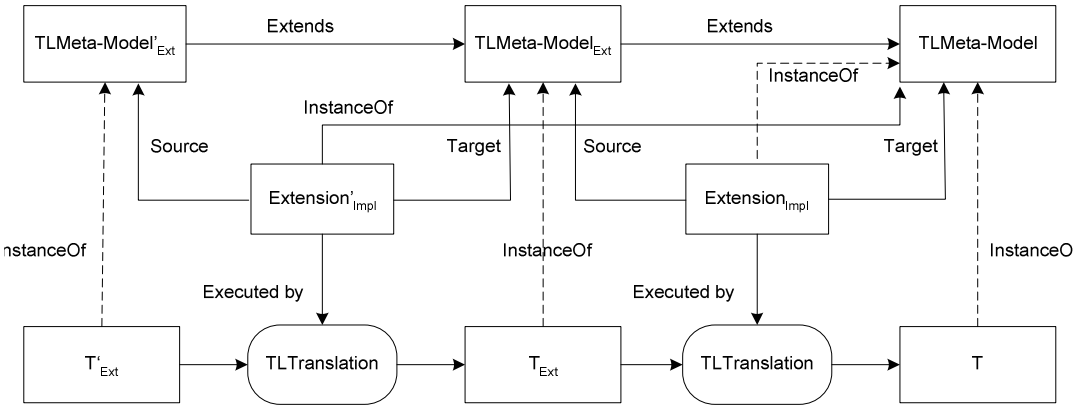


Figure 5.16 Composition of language extensions

Figure 5.16 shows two extensions of the basic language. The model *TLMeta-Model'*$_{Ext}$ extends *TLMeta-Model*$_{Ext}$, which in turn, extends *TLMeta-Model*. The extensions defined in *TLMeta-Model'*$_{Ext}$ are processed by a transformation *Extension'*$_{Impl}$ that transforms transformation *T'*$_{Ext}$ to transformation T_{Ext} written according to *TLMeta-Model*$_{Ext}$.

Furthermore, T_{Ext} is transformed to a transformation *T* written in the basic language. It should be noticed that both transformations that implement the extensions are written in the basic language and are executed by its engine (the process *TLTranslation*).

5.7.3 Example: Extending Transformation Language with New Compositional Operators

In this section we illustrate the approach by extending the transformation language presented in Chapter 4. Extension of the language is aimed at solving some of the problems revealed in scenarios 3 and 4.

In these scenarios some rules need to be extended with new functionality that is included in their target part. In Scenario 3 all the rules have to be extended while in Scenario 4 only the rules for the geometrical operators have to be extended to support the processing of the scale operator. Recall that this extension poses two requirements. The first one is about the possibility to express a part of the transformation functionality in a separate module. This functionality does not form a complete rule. Instead it is a part of the rule target. The second requirement is about the possibility to compose the module that contains the extension with the rules that must be extended. In other words, we need a proper compositional operator.

Our transformation language provides neither of these constructs. Therefore it is feasible to introduce the required constructs as extensions of the language by following the general approach explained in the previous section.

The first extension is a new modular construct that contains instantiation actions. We call the new construct *Particle*. A particle is not a complete rule since it does not specify a source. It is not an executable component. It is meant to specify instantiations that are integrated with the targets of some existing rules.

The integration of the particles is done by a compositional operator which is the second extension of our language. This operator is called *ParticleMerge*. It selects transformation rules from transformation definitions and adds a particle to their targets.

The following two figures show the extension of the abstract syntax of the language. They use diagrams from the abstract syntax of the language given in Appendix B.

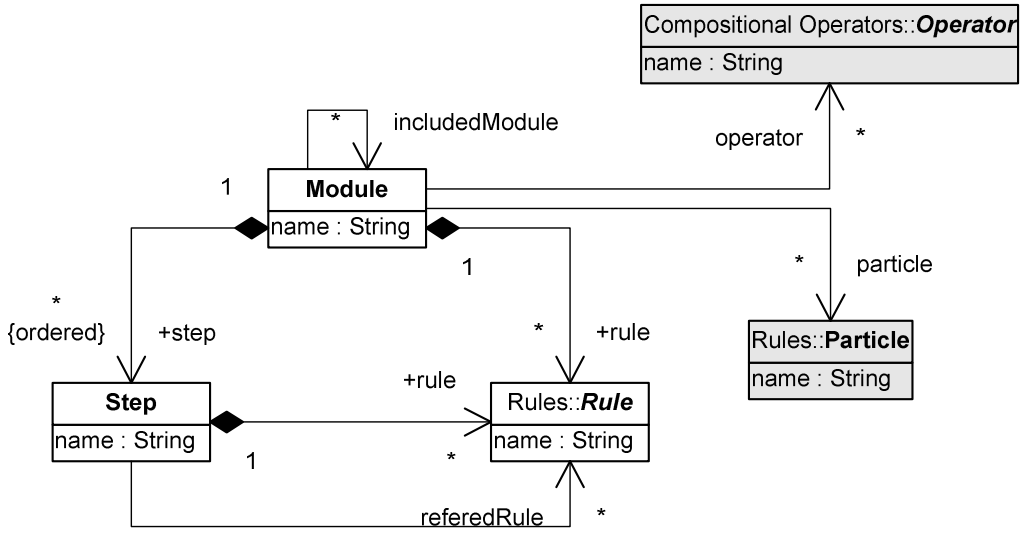


Figure 5.17 Extensions of the transformation language abstract syntax

Figure 5.17 shows the extensions made at the level of definition of transformation modules. The newly added elements are shown in gray. The extended abstract syntax allows transformation modules to contain definitions of particles and application of merging operators along with transformational rules. We do not show the definition of *Particle* element. It is similar to the definition of the target of a rule: particles contain a set of actions. Particles are not executable; they are merged with transformation rules.

Figure 5.18 shows the definition of *ParticleMerge* operator. Every merge operator has exactly one expression for selecting transformation rules and refers to exactly one particle.

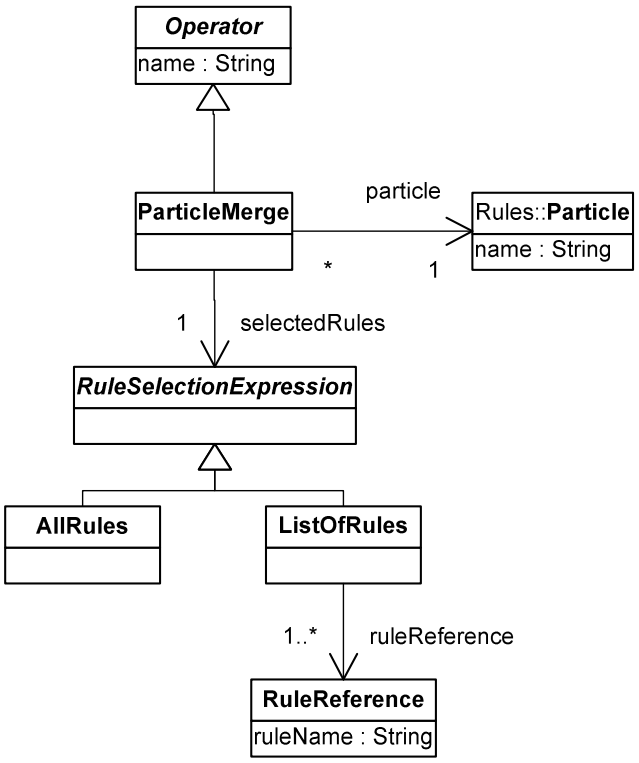


Figure 5.18 The definition of ParticleMerge operator

Rule selection expressions enumerate a set of rules by name or refer to all the rules in the transformation. Merge operators add the actions in the particle to the target of all the rules selected by the selection expression.

These extensions of the abstract syntax are associated with a transformation that takes as an input a transformation definition written in the extended syntax and updates it to reduce it to the basic syntax. The following code shows the rules of this transformation definition. It consists of two steps.

In the first step the merging operators are executed and some of (or all) the transformation rules are extended with new actions defined in particles. In the second step the merging operators and particles are deleted. Therefore, the result will not contain any element from the extended abstract syntax.

The following rule is used in the first transformation step. For every merging operator it obtains the model element rules selected by the operator and updates their targets by appending the actions in the particle to the current actions.

```
mergeParticle ModelElementRule inputParameters [inputModule : Module] {
  source [mergeOperator : ParticleMerge,
          selectedRule : ModelElementRule=
            if mergeOperator.selectedRules.oclIsTypeOf(AllRules) then
                inputModule.rule->select(oclIsTypeOf(ModelElementRule))
            else
                mergeOperator.selectedRules.ruleReference->collect(
                        ref | inputModule.rule->select(oclIsTypeOf(ModelElementRule)
                              and
                               name=ref.ruleName))
          ]
  target [update selectedRule.target {append  action=mergeOperator.particle.action} ]
}
```

The next two rules form the second step of the transformation. The first one deletes all the merge operations. The second one deletes the particles but keeps the actions specified in them. Actions are shared by the transformation rules. After this second step the transformation is reduced to the basic syntax and the model element rules are updated with new actions.

```
deleteMergeOperators ModelElementRule {
  source [mergeOperator: ParticleMerge]
  target [delete mergeOperator{delete mergeOperator.selectedRules}]
}
```

```
deleteParticle ModelElementRule {
  source [p: Particle]
  target [delete p]
}
```

5.8 Conclusions

In this chapter we studied the motivation for decomposition and composition of transformation definitions. Three cases were considered: decomposition for managing complexity, evolution of transformation definitions caused by evolution of models, and composition of transformation definitions driven by composition of models.

These three cases were investigated in more details in four transformation scenarios. The first scenario illustrated the impact of decompositions in the source and target intensions on the decomposition of transformation definitions into rules. The second scenario showed how different decompositions in models that are transformed influence the transformation definitions. The third scenario identified transformation functionality suitable for modularization that is not related to source and target models. The fourth scenario studied how changes in the intensions influence transformation definitions.

We derived a set of requirements that transformation languages should fulfill to provide support for composition and decomposition in transformation definitions. On the base of the requirements we evaluated a set of transformation languages. Languages were evaluated with respect to modularity, rule interaction, compositional mechanisms, adaptability of definitions, and reflection.

The first three scenarios showed possible ways for decomposition of transformation definitions. Software engineers should consider decompositions in the source and target intensions. It is important that all the decompositions are taken into account. Depicting models in a multidimensional space as in Scenario 1 can help in this task. Rules that group elements belonging to different dimensions of decomposition should be avoided. This reduces the scattering and tangling of elements in transformation rules. The positive effect of this approach is the improved adaptability of transformation definitions if models evolve over certain dimensions.

Scenario 2 illustrated another source for decomposition: the presence of multiple hierarchies in a given model that depend on each other. If transformation language is based on explicit form of scheduling, the software engineer should carefully consider dependencies and encode the rule execution order accordingly. If transformation language is declarative (or hybrid) the task of detection of dependencies is at large extent performed by the trans-

formation engine. We also observed the need of transformation language constructs different from transformation rules with left-hand and right-hand parts.

Scenarios also showed the possibility of crosscutting functionality in transformation definitions. In general, transformation languages provide two compositional mechanisms: rule invocation and inheritance. Both solutions exposed some anomalies in the resulting transformation definitions that handle crosscutting functionality. The anomalies are related to reduced adaptability of the definitions and to introduction of an excessive number of rules. We may conclude that the support for crosscutting transformation functionality should be improved.

Several scenarios required adaptations in transformation definitions. Adaptations were based on replacement of rules, addition of new constructs in existing rules and replacement of rule components. Adaptability of transformation definitions is improved if loosely coupled rules are defined. Whenever possible rules should interact with each other via traceability links. This is the main approach used in declarative and hybrid languages.

As an overall conclusion about the evaluated languages we can state that they provide a reasonably full set of modular constructs but still have problems in handling some compositional and adaptation scenarios. Another problem we experienced is that some languages are poorly documented.

In the end of the chapter we proposed an approach for extending transformation languages with new constructs. Extensions are done in the abstract syntax of a language and the new constructs are processed by transformations.

6

Model Driven XML Processing

This chapter presents an approach for processing XML documents. Current techniques for XML processing are considered in the broader context of Model Driven Engineering. We assume that the syntax of an XML language is defined in a schema and is interpreted via an application-specific model. Interpretation is driven by a transformation definition from the document model to the application model. This approach overcomes problems experienced in today's XML applications, such as lack of proper extensibility mechanisms in case of syntax changes.

6.1 Introduction[1]

In this chapter we present an approach for XML processing based on the transformation language presented in Chapter 4. We show how this transformational approach to XML processing improves adaptability and reusability of XML applications.

Extensible Markup Language (XML) [103] is nowadays a dominant data representation format used in many areas in computer science and industry such as World Wide Web (WWW), e-commerce and Web Services Architecture [110]. XML provides a grammar for XML documents and a mechanism for defining domain-specific markup languages. The conformance of XML documents to the XML grammar is called *well-formedness* of XML documents. A domain-specific markup language (sometimes referred to as *vocabulary*) provides syntactical constructs to express concepts in a given problem domain. These constructs are defined in a *schema* expressed in a *schema language*. The relation between a schema and a document that conforms to the schema is called *validity* relation.

[1] This chapter is based on work published in [58] and [61]

In the last few years many XML markup languages emerged focusing on various problem domains. This opens the possibility for reuse of existing languages into new ones (known as *hybrid languages*) and creating compound documents expressed in hybrid languages. This possibility is exemplified by the recent standards created within World Wide Web Consortium (W3C) based on composition and reuse of modules defined for Web languages such as XHTML [109], SMIL [107], MathML [111], and SVG [112].

The wide acceptance of XML motivates the need for techniques and tools that support the development of XML-based applications. Today, XML technology offers mature standards and tools that mainly facilitate the definition and processing of the syntactical part of XML applications. *XML parsers* are tools that check for well-formedness and validity of XML documents. Several languages exist for defining schemas of markup languages (e.g. Document Type Definition (DTD) [103], and XML Schema [106]). Other related languages are Extensible Stylesheet Language for Transformations (XSLT) [105] used to express document transformations, and XPath/XQuery [115] for navigation and extraction over XML documents.

Apart from the document parsing that checks for well-formedness and validity of XML documents, XML applications perform an application-specific document processing. This processing is an interpretation of the markup syntax used in the documents. The application usually has to transform XML documents into application-specific structures that implement the concepts in the domain for which an XML language is used. This is a recurring task and is a candidate for at least a partial automation.

Furthermore, today's XML applications often have to possess certain quality properties. In this chapter we focus on two quality properties: adaptability and reusability.

Adaptability of XML applications is a capability that allows them to be modified if the syntax of the markup language changes. Reusability of XML applications is motivated by the presence of compound documents based on multiple vocabularies. The ability to reuse the vocabulary is naturally followed by the need to reuse the XML application for that vocabulary. One possible reuse is in the composition of several XML applications in a new one.

In this chapter we focus on XML applications developed in object-oriented languages. Generally, software engineers may choose between two technologies to process XML documents: generic document interfaces (such as DOM and SAX), and data binding. Simple API for XML (SAX) [92] and Document Object Model (DOM) [104] provide interfaces to documents that reflect the document syntax. It is acknowledged that these technologies are too low level. Moreover, applications based on them are often designed in an ad-hoc manner and hardly possess the adaptability and reusability properties. For instance, a change in the syntax may lead to many changes in the code and recompilation of the whole application.

In data binding [90] a document schema is compiled into a set of classes in a given language and the processing of documents is automated (a process known as unmarshaling). This approach, however, is not applicable if the application classes already exist and differ significantly from the document syntax structure. Reusability and adaptability are deteriorated because every change in the schema requires schema recompilation.

The problems of adaptability and reusability caused by the need for redesign and recompilation of applications are strongly related to the fact that the XML technology does not provide a standard means for specifying semantics of markup languages. In current XML applications the relation between the syntax and its intended meaning is not explicit. It is often hard-coded in the application and it is difficult to reuse and maintain it. On the

other hand, the domain of programming language specification offers a number of frameworks for defining language semantics in an explicit way [120]. Issues like the evolution and composition of languages and their translators have been on the research agenda for a long time [67][68][35]. The experience gained in that area may be used to develop tools and techniques required for XML applications.

In this chapter we propose an approach for XML processing based on a declarative specification and execution of model transformations from the language syntax structures (the source model) to the application structures (the target model). These transformations can be regarded as a semantic specification for the markup language syntax. We assume that the document syntax is either defined in an XML schema or as a set of elements and attributes (schema-less approach). If a schema is present it is treated as a model of XML documents. The application classes employed by an XML application form the target model. A given transformation definition contains rules that encode how the syntax constructs defined in the source schema represent elements in the target model. We show how the language presented in Chapter 4 is applied in the context of XML processing.

By using transformations we aim to achieve a better separation of concerns. XML applications are decomposed in three components: syntax definition (schema), transformation definition and application classes. Application classes do not contain syntax processing code; this is captured in the transformation definition.

The benefits of our approach are the following:

- developers are freed from writing a low level syntax processing code;
- this approach opens a possibility for automatic generation of language translators similar to the compiler-compiler approach;
- syntax and application code may evolve independently;
- transformation rules can be designed at the granularity that provides good adaptability of the application. Only rules that reflect changes in the syntax are updated;
- reusability of the applications is improved. Using multiple vocabularies in a document is achieved by composing corresponding transformation rules and application classes.

This chapter is organized as follows. Section 6.2 gives a general introduction to XML processing. Section 6.3 presents techniques for XML processing and provides an evaluation of the techniques. Section 6.4 explains the proposal for model driven XML processing. Section 6.5 illustrates the approach by an example used further in Section 6.6 to present composition of XML applications. Section 6.7 discusses related work. Section 6.8 gives conclusions.

6.2 XML Processing

We distinguish between generic and application-specific XML processing. The first type of processing is concerned with the document syntax and does not consider the meaning a document may carry. Generic processing can be uniformly performed over any XML document no matter what a possible interpretation of the document is. In contrast, application-specific processing is usually applied on documents of a given markup language, e.g. XHTML documents, and gives an interpretation to them.

The processing tasks may be organized in a number of ways and there is a certain sequence among them. Processing tasks are explained in the next two sections.

6.2.1 Generic XML Processing

The XML specification defines a general model for XML processing. The model consists of three components: *XML Document*, *XML Processor* and *Application.* Figure 6.1 shows these components and the relationships among them.

Figure 6.1 Generic XML processing model according to the XML specification

XML Processor is a software component that reads XML documents and provides access to their content. *Application* component represents any software that uses the content of an XML document. The specification states that applications access XML documents through the services provided by the *XML Processor*.

The main task of the XML processor is to transform the sequence of characters in the XML document into a structure comprised of XML syntax constructs. XML processor performs the generic tasks of lexical and syntactical analysis of documents and isolates the application from the low level details of text processing. Two types of processors are defined: non-validating and validating. Non-validating processors check documents only for well-formedness while the validating ones perform checks for well-formedness and validity against an XML schema.

The XML specification [103] does not define a model for XML documents. Such a model has an impact on the interface between the processor and the application. A number of models are proposed in other standards: Document Object Model (DOM) [104], XPath 2.0 and XQuery data model [114], and XML Infoset [113].

Processors that perform the tasks described above are also known as XML parsers. A number of widely available parser implementations exist. In this chapter we make a distinction between XML Processor and XML Parser. XML Processor is regarded as a more general concept and may perform other tasks beyond the well-formedness and validity checking. In many sources these two terms are used as synonyms.

The presented model of XML processing does not incorporate the latest standards in this technology. It only takes into account the well-formedness and validity checking. However, there are other generic operations that may be included in XML processors functionality. Some of these operations may interfere with others thus putting a need for an explicit execution order. For example, a document may refer to a schema and may contain instructions for inclusion of an external document according to the XInclude recommendation [116]. The problem is which operation should be performed first: the inclusion or the validation. The original document may be invalid but the inclusion operation may turn it to a valid one. Consequently, the order of execution of the inclusion and validation is important and different orders may give different results. Apart from inclusion and validation other operations may be performed too. Examples are XSLT transformations,

XQuery extraction, and embedding a content referenced by links based on the XLink standard [28].

The XML processing model as defined in the specification is too limited. It is unable to specify the execution order of operations and to manage their dependencies. At the moment there is no standard processing model for generic XML processing. Some proposals suggest that the processing model should be organized in a pipeline of steps that operate on infosets [113]. Each step accepts an infoset as input and generates an output infoset. The final result is passed to an application, which performs application-specific processing. Figure 6.2 shows an example pipeline with two sequential steps.

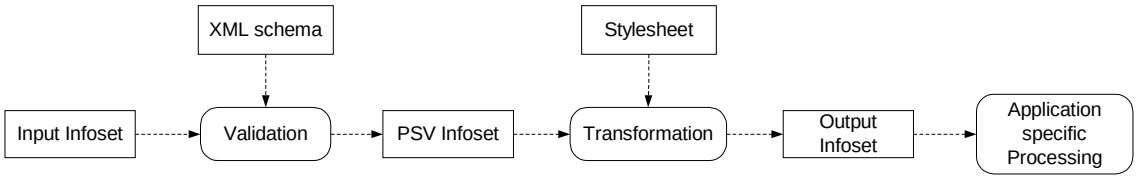


Figure 6.2 An example XML processing pipeline with two steps

In the first step a validating parser performs validation on the input infoset and produces another infoset (Post-Schema Validation (PSV) infoset) augmented with information from the schema that may include default attribute and element values and type information. In the second step, the PSV infoset is transformed by an XSLT processor on the base of a stylesheet and the result is passed for application-specific processing.

6.2.2 Application-specific XML Processing

Application-specific processing is performed by XML applications that give interpretation of documents. Applications usually operate only on documents written in a given XML language. Examples of such XML applications are SVG and MathML renderers, browsers, etc. In application-specific processing applications are not interested in the document structure but in the application model encoded in the documents. For example, Topic Maps technology defines an XML language for topic maps representation and a processing model for extracting topic maps from XML documents [119]. A topic map processor implements this processing model.

Scenarios of application-specific XML processing are transformations to another document, storing documents in a database, document rendering, e-commerce document exchange, etc. Some scenarios introduce problems general enough to be separated as a standalone area of research. Also, in many cases application-specific processing produces an XML document as a result and therefore it can be included in the pipeline along with generic processing tasks. Therefore, it is not always easy to judge if a particular task is a generic one or an application-specific.

In the next section we describe three technologies for XML processing. They are evaluated with respect to the adaptability and reusability of XML applications built with them.

6.3 Technologies for XML Processing

In this section we focus on application-specific processing with object-oriented languages since they are widely used in practice. It was mentioned that an application interprets the syntactical constructs and that interpretation relies on an application-specific model. The XML language provides syntax for expressing this model.

Object-oriented languages express models as classes that have properties and methods. Compared to that, XML documents are hierarchies of elements and attributes. In programming languages classes may have behavior, whereas XML is a data description standard without executable semantics. XML syntax is optimized for data serialization based on ordered trees. Object-oriented applications employ graph structures. There are multiple alternative serializations for a graph structure to a tree structure.

Apparently, there may be structural differences between the XML data model and its possible semantics implemented with object-oriented constructs. The application has to transform XML documents into instances of the application-specific classes that implement the concepts in the domain for which XML is used. This is a recurring task and is a candidate for automation.

In the next sections we discuss the three most commonly used technologies for XML processing: SAX, DOM, and Data Binding. An evaluation of these technologies is given in section 6.3.4.

6.3.1 Simple API for XML

Simple API for XML (SAX) [92] is an interface between XML parser and XML applications. During the parsing process a SAX-based parser generates parsing events triggered by the markups found in the document. Examples of events are the start and the end of the document, the start and the end of an XML element, and reading of a sequence of characters. The order of events follows the order of elements in the XML document.

SAX is referred to as an event-based API and the processing of XML documents based on SAX is called event-based processing. The architecture of an application that uses SAX to access the content of XML documents is shown in Figure 6.3:

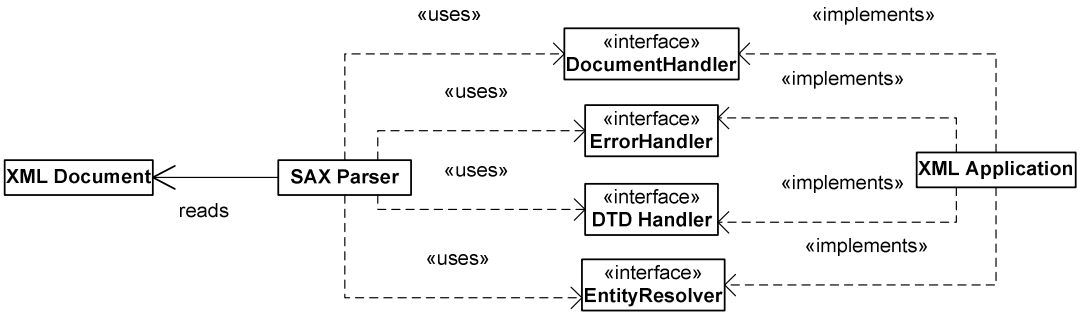


Figure 6.3 The architecture of an application based on SAX parser

Four standard interfaces for event handling are used: *DocumentHandler*, *ErrorHandler*, *DTDHandler*, and *EntityResolver*. The application implements these interfaces to handle the events. Event handlers are called back by the parser when an event occurs. An impor-

tant characteristic of this approach is that the parser does not build an internal structure of the document and leaves this task to the application.

6.3.2 Document Object Model

Document Object Model (DOM) defines a logical model for XML documents and an API for accessing the documents according to that model. The logical model represents XML documents as a hierarchy of nodes. The hierarchy reflects the syntactical structure of XML documents. DOM defines concrete node types for document, element, attribute, text, processing instruction, etc. Because of the tree-based logical model the processing based on DOM is called tree-based processing.

The architecture of an application based on DOM is shown in Figure 6.4.

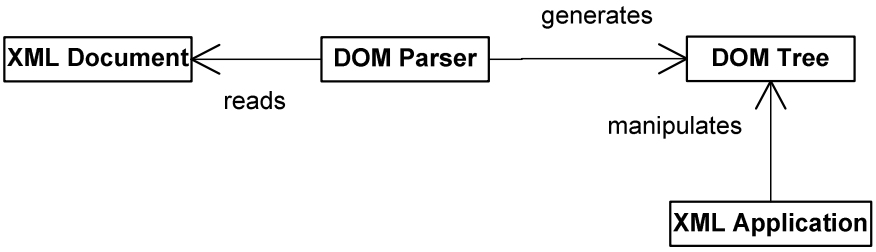


Figure 6.4 The architecture of an application based on Document Object Model

In this architecture XML documents are read by a DOM parser that generates a DOM tree and passes it to an application. The application accesses the DOM tree through the standard DOM API and performs the application-specific tasks. A part of the application code is responsible for translation of the generic document structure into application-specific objects. A typical task is the transformation of the un-typed string values to values of simple types. Another part of the code deals with the navigation through the document hierarchy to locate the required data. The application may also modify the DOM tree.

6.3.3 Data Binding

SAX and DOM-based processing are considered low-level because the parser only exposes the syntax structure of XML documents and leaves the task of traversing the document, instantiation of application objects and simple type conversion to the application.

Data binding approach [90] aims at automating some common processing tasks and isolating the application from the syntax details. The following example illustrates the idea of data binding.

Consider an element ‘person’ with one attribute that contains the age of the person and one sub-element that contains the name of the person:

```
<person age=”23”>
     <name>John Smith</name>
</person>
```

A DOM-based representation of this element is shown in Figure 6.5. The root of the tree is the element with name *person*. It contains one child element (element *name*) and one attribute named *age*. Element *name* contains a text node with value ‘John Smith’.

If an application uses the DOM interface to access the age of the person, it will first use the navigation API of the DOM to obtain the *age* attribute node and second, will convert the string value ‘23’ to the integer value 23.

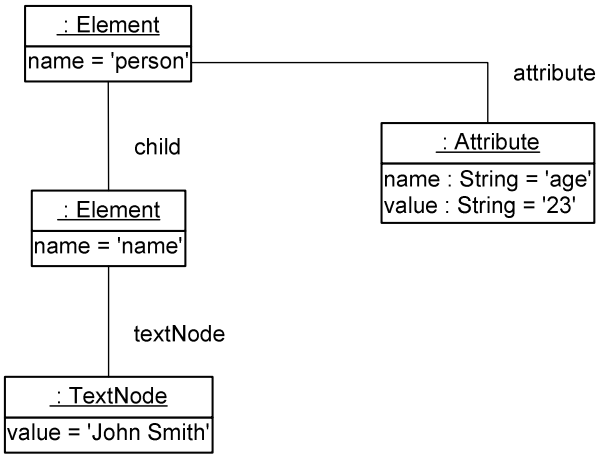
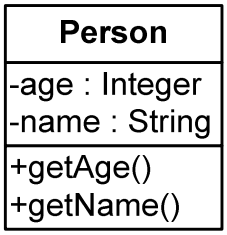


Figure 6.5 An example DOM tree

Instead of dealing with these details a more natural way is to use objects of the following class:



Data binding facility automates the generation of the class and instantiation of objects of that class. This approach considers the documents’ DTD and XML schema as a typing system. On the base of the DTD/schema a set of classes is generated by *schema compiler*. This is shown in Figure 6.6. Data access is facilitated through generated getter and setter methods.

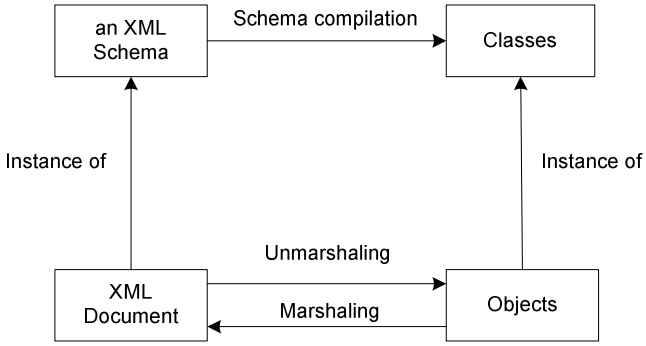


Figure 6.6 Model of data binding facility

At runtime the data binding facility creates objects from a given XML document. This process is known as unmarshaling. Unmarshaling can be driven by transformation rules but often they are hard coded in the unmarshaler implementation. Most data binding tools also support the reverse operation of marshaling objects to an XML document.

Since the classes are obtained from the DTD/schema by preserving its structure the generated classes are similar to the document structure. Classes do not have application specific behavior.

6.3.4 Evaluation

In this section we briefly evaluate the described techniques for XML processing. We are interested in two questions:

- how applications are supported in solving the problem of relating the XML syntax structures with the required application structures;
- at what extent the described technologies support adaptability and reusability of XML applications;

SAX and DOM provide access to the generic syntax structure. These technologies are applicable if applications have to maintain a representation of the document structure as in the case of XML editor. If applications aim at transforming documents to internal structures then these technologies are not suitable enough. They provide low level of automation and require a significant amount of tedious and error-prone coding. Computation is performed against generic labeled trees instead of domain specific concepts. Transformation to the internal application structures is not automated and is done by the application, usually in an ad-hoc way. As a result applications are bound to a particular syntax and every change in that syntax leads to changes in the processing code. This deteriorates adaptability of XML applications.

Data-binding approach provides certain degree of automation but still falls short to bridge the syntax structures and application structures in an adequate way. Most schema compilers take only the document schema as a source for generation of application classes. However, a given application construct may take multiple syntactical forms. Also, a given XML construct may represent different application constructs. For instance, an object and an object property may both be represented as elements. XML schemas do not provide means to distinguish between these representations. Therefore, the semantic information of the model is lost when it is expressed in XML. This makes the XML schema an unreliable source for inferring the application semantics. Some schema compilers can be driven by mapping rules or already existing classes can be used. The mapping rules, however, are not powerful enough to express the correspondence between a schema and classes, which are structurally different. Therefore, data binding is applicable in cases where the application model is similar to the schema structure. It is difficult to apply data binding if there is a difference between these structures.

Another problem of data binding is that usually the whole schema is compiled. It should be possible to select for compilation only the relevant schema components. Finally, data binding approach does not cope well with changes in the schema. Every change in the schema causes recompilation of the schema into new classes and redeploying them on all the applications including those not affected by the changes.

6.4 Model Driven XML Processing

This section presents an approach to XML processing that improves the adaptability and reusability of XML applications. The approach is based on specification of transformations between source models of XML documents and target application models. We benefit from the ability of transformation languages to express the actions that an application takes during the processing of XML documents.

This approach to XML processing applies the MDA transformation pattern defined in Chapter 2. In the context of XML processing, the transformation engine takes an XML document as input and generates an output. The output can be a set of rows in a relational database, another XML document or objects instances of classes written in a given programming language. In this chapter we focus on applications that instantiate objects on the base of XML documents. These objects may be implemented in any programming language. Our transformation language (see Chapter 4) is independent of concrete languages used to specify the models. We choose Java to illustrate our approach.

To apply the approach we have to identify the intensions used to define transformations. We distinguish between schema-less and schema-based processing. The type of processing determines intensions. These types of processing are discussed in the next two sections.

6.4.1 Schema-less XML Processing

In this type of processing XML documents are not validated by a schema. However, they have to be well-formed according to the XML grammar. To define transformations we need a model of XML documents. Any model that reflects the constraints in the XML grammar is suitable. Available standard alternatives are DOM and the XML Information Set. In this chapter we choose DOM.

The XML Document Model as defined by the DOM is shown in Figure 6.7 as an UML class diagram.

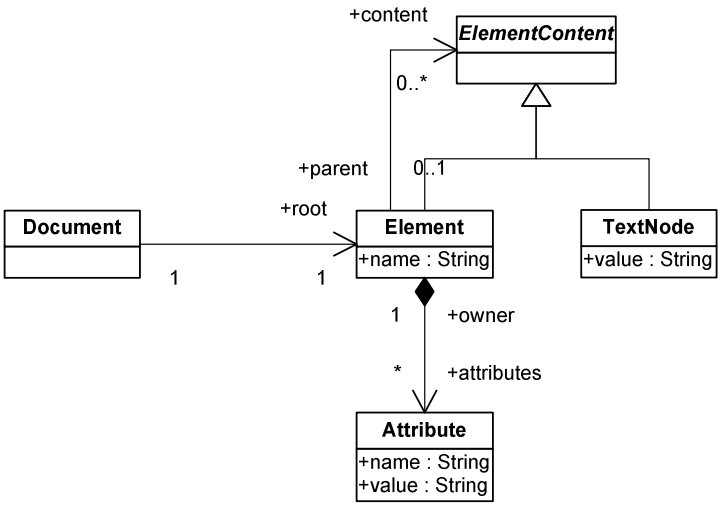


Figure 6.7 The XML document model

According to this model every XML document has exactly one root element. Elements have name and zero or more attributes. Every attribute has a name and a value. Furthermore, elements may have content that consists of other elements and text nodes. For simplicity, we do not include processing instructions and document type declarations in the model.

The transformation pattern for schema-less XML processing is shown in Figure 6.8. The source intension is the XML Document Model explained above. The model may be defined in the MOF language and therefore the XML documents are instances of this model according to the MOF instantiation mechanism. The target intension is a set of application classes written in a given programming language. The output of the transformation is a set of objects instances of the application classes according to the instantiation mechanism defined for the programming language.

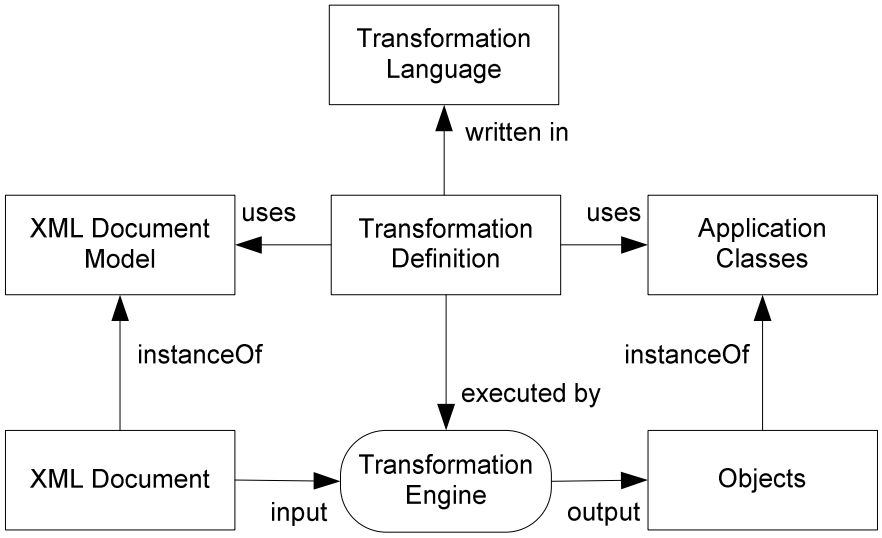


Figure 6.8 Transformation pattern for schema-less XML processing

Transformation definitions may select nodes in XML documents based on the three classes defined in the XML Document Model (see Figure 6.7): *Element*, *Attribute*, and *TextNode*. It should be noted that the schema-less processing does not require definition of a new language in the modeling space defined in Chapter 4. MOF language is enough to express XML documents in the modeling space.

6.4.2 Schema-based XML Processing

XML documents may conform to a schema. Many of today's XML languages are defined by XML schemas [106]. A schema can be perceived as a model of the documents that are valid against that schema. Moreover, the presence of schema does not cancel the conformance to XML Document Model. Schemas only impose additional constraints. Therefore the documents may be considered as instances of two different models: XML Document Model and the document schema. The *instanceOf* relationships are defined in different ways in these cases and may exist together. Working with both models is important and should be available in the transformation definitions. Software engineers should be able to

specify both generic document processing reflecting the XML document model and document processing that uses type information based on schema types. To employ schemas in our approach we include a model of XML Schema that can be derived from the specification. A fragment of that model is shown in Figure 6.9. The complete XML Schema model is referred to [106].

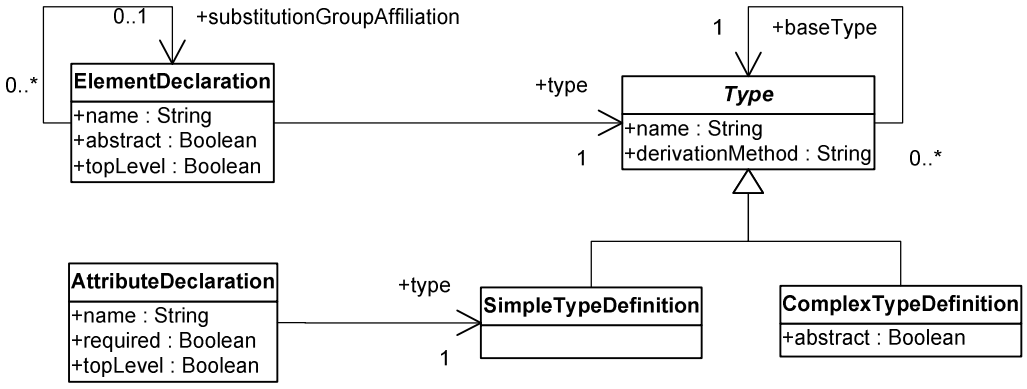


Figure 6.9 Part of XML Schema model

XML schemas contain a set of element and attribute declarations. Declarations define the names and types of elements and attributes used in XML documents. The content of elements and attributes is constrained by a type definition. There are two kinds of type definitions: *SimpleTypeDefinition* and *ComplexTypeDefinition*. Attribute declarations are always constrained by a simple type definition. Element declarations may be constrained by either simple or complex type definition. Element declarations may form substitution group hierarchies. Types may be related by the mechanism of derivation, which is similar to the inheritance mechanism in programming languages. Two methods of derivation are defined: by *extension* and by *restriction*. In the model, the derivation method is indicated as a value of attribute *derivationMethod* of class *ComplexTypeDefinition*.

The transformation pattern for schema-based XML processing is shown in Figure 6.10. In the pattern the source intension is a composition of two models: the XML Schema Model and the XML Document Model. The transformation specification may use intensional constructs from both models.

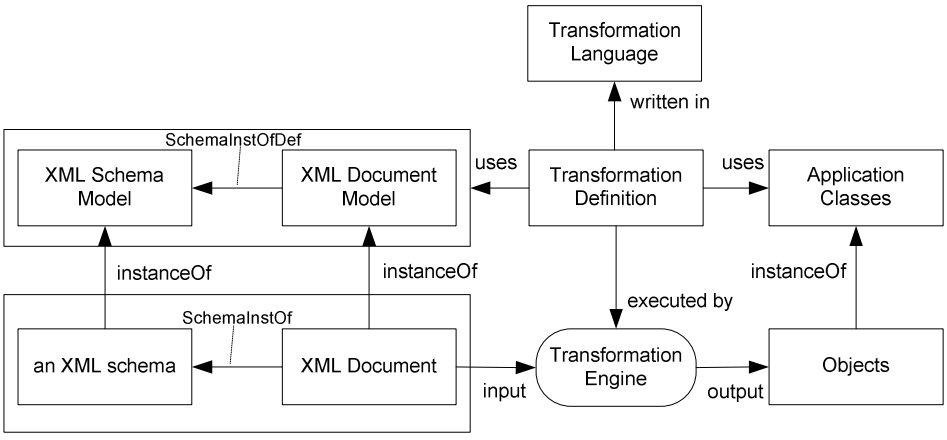


Figure 6.10 Transformation pattern for schema-based XML processing

Usually, after the validation of a document the parser augments the document with schema information. The augmented document provides information about the element declarations of element nodes and about attribute declarations of attribute nodes. Type information is also provided for every node. The XML Document Model is related to the XML Schema Model to define the relations between XML nodes and their element and attribute declarations. We may consider that the XML Document Model is composed with the XML Schema Model. Figure 6.11 shows how the two models are composed.

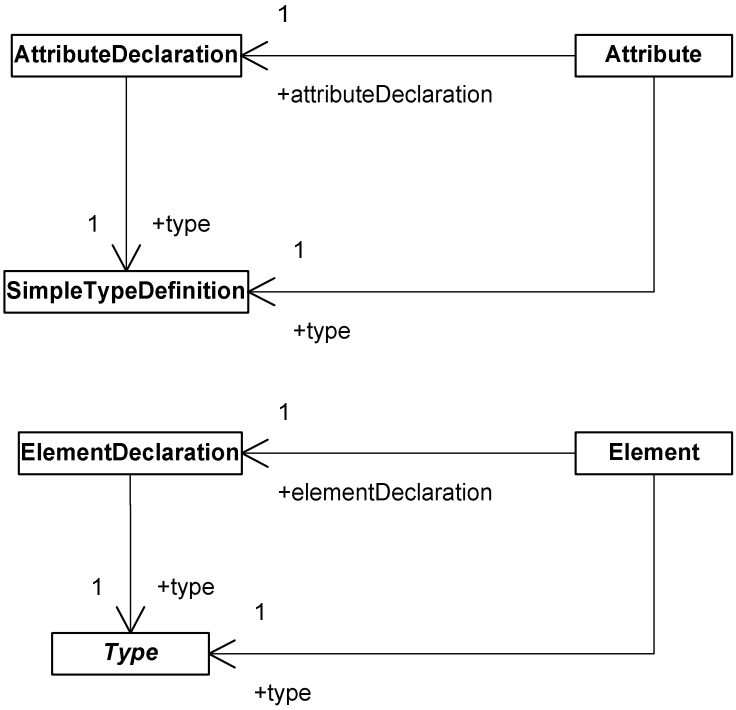


Figure 6.11 Relations between XML Document Model and XML Schema Model

This figure shows only the elements from the two models that participate in the composition. Classes *AttributeDeclaration*, *SimpleTypeDefinition*, *ElementDeclaration*, and *Type* are defined in the XML Schema Model (see Figure 6.9). Classes *Attribute* and *Element* are defined in the XML Document Model (see Figure 6.7). New associations are added between the classes. Class *Attribute* is related to class *AttributeDeclaration* to indicate the declaration that defines the name of the attributes in XML documents. Also, class *Attribute* is related to class *SimpleType* to indicate the type used to constrain the attribute value. In a similar way, class *Element* is related to classes *ElementDeclaration* and *Type*. These relations are based on the Post Schema Validation Infoset described in the XML Schema specification [106] that defines how XML documents are augmented with schema information.

After the validation of a document attribute nodes are provided with values of two new properties: *attributeDeclaration* and *type*. Element nodes also accept two new properties: *elementDeclaration* and *type*.

In the remaining part of this section we show how nodes in XML documents may be selected on the basis of schema information. XML Schema defines three transitive relations: *substitution* among elements, *extension* among types, and *restriction* among types. All the three relations may be used to select elements in source documents. To clarify this

we give an example based on one of the schemas shown in Chapter 3. It is shown again below.

```
<element name='examItem'
         type='examItemType'
         abstract='true'/>

<complexType name='examItemType'>
    ……….
</complexType>

<complexType name='openType'>
  <complexContent>
      <extension base='examItemType'>
       ……..
      </extension>
  </complexContent>
</complexType>

<complexType name='multipleChoiceType'>
  <complexContent>
      <extension base='examItemType'>
       ……..
      </extension>
  </complexContent>
</complexType>

<element name='open'
         type='openType'
         substitutionGroup='examItem'/>

<element name='multipleChoice'
         type='multipleChoiceType'
         substitutionGroup='examItem'/>
```

In this schema element declarations *open* and *multipleChoice* are related via substitution to the element declaration *examItem*. Complex types *openType* and *multipleChoiceType* are extensions of type *examItemType*.

We study four examples of selection of nodes from XML documents on the base of schema information:

Ex1: Selection of elements instances of a given element declaration;
Ex2: Selection of elements of a given type;
Ex3: Selection of elements that are substitutions of a given element;
Ex4: Selection of elements of types that are extensions of a given type;

Similar examples may be formulated for selection of attributes.

The implementation of the selection examples 3 and 4 requires usage of the transitive relations defined for XML schemas. In the modeling space described in Chapter 4, these relations may be defined in language configurations. Therefore, we have to define a language for the XML technology that includes the definitions of the relations in the language configuration. XML schemas are intensions of XML documents. The definition of the XML language takes the XML Schema Model as intension model and XML Document Model as extension model according to the approach shown in Chapter 2, Figure 2.8. We illustrate the application of this approach in Figure 6.12.

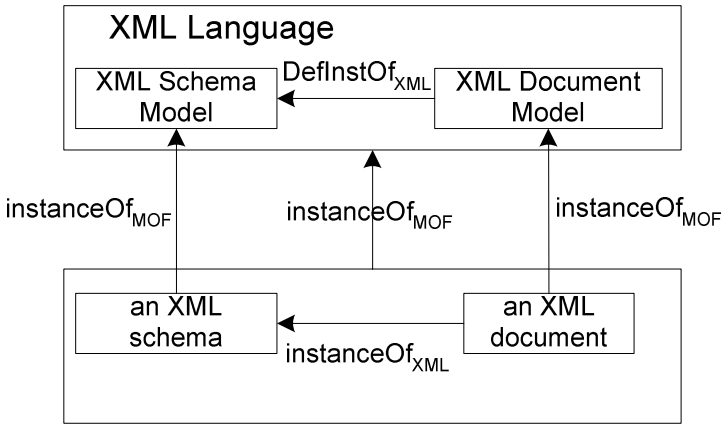


Figure 6.12 The definition of the XML language based on XML Schema Model as intension model and XML Document Model as extension model

In the XML language defined according to Figure 6.12 element and attribute declarations are the constructs that may be instantiated. Names of the instantiated elements and attributes are derived from the names specified in their declarations. Slots implied by the XML Document Model are reused. The language defines three transitive relations: *substitution*, *extension*, and *restriction*. In addition, a new slot called *type* is defined for attribute and element nodes.

Table 6.1 shows how selection examples are implemented in this approach.

N	**Implementation of selection examples**
Ex1	**source**[el : element(multipleChoice)]
Ex2.1	**source**[el: elDecl, elDecl **instanceOf**(MOF) ElementDeclaration, **condition**{elDecl.type=ComplexType(openType)}]
Ex2.2	**source**[el **instanceOf**(MOF) Element, **condition**{el.type=ComplexType(openType)}]
Ex3	**source**[el: element(examItem) **follow** substitution]
Ex4.1	**source**[el:elDecl, elDecl **instanceOf**(MOF) ElementDeclaration, **condition** {(ComplexType(examItemType) **follow** extension)-> includes(elDecl.type) }]
Ex4.2	**source**[el **instanceOf**(MOF) Element, **condition** {(ComplexType(examItemType) **follow** extension)-> includes(el.type) }]

Table 6.1 Implementation of the four example selection operations in the two approaches for schema-based XML processing

The table shows examples of source expressions written in MISTRAL. The first example selects all the instances of the element declaration *multipleChoice*. The second example selects all the elements of type *openType*. Two implementations are possible shown in rows 2.1 and 2.2 respectively. In the first implementation we determine all the element declarations of type *openType*. Since many declarations of that type are possible, in the

expression the element declaration is a variable (*elDecl*). In the second implementation we select instances of class *Element*, in contrast to the first implementation where instances of element declarations are selected. The second implementation is more concise. The third example selects all the elements that are related via substitution to *examItem* element. This selection is based on *substitution* transitive relation. The fourth example selects elements of types extensions examItemType. Similarly to the second example two implementations are possible shown in rows 4.1 and 4.2. Both implementations use the *extension* transitive relation.

6.4.3 Structure of XML Applications based on Model Transformations

The structure of XML applications based on model transformations is shown in Figure 6.13.

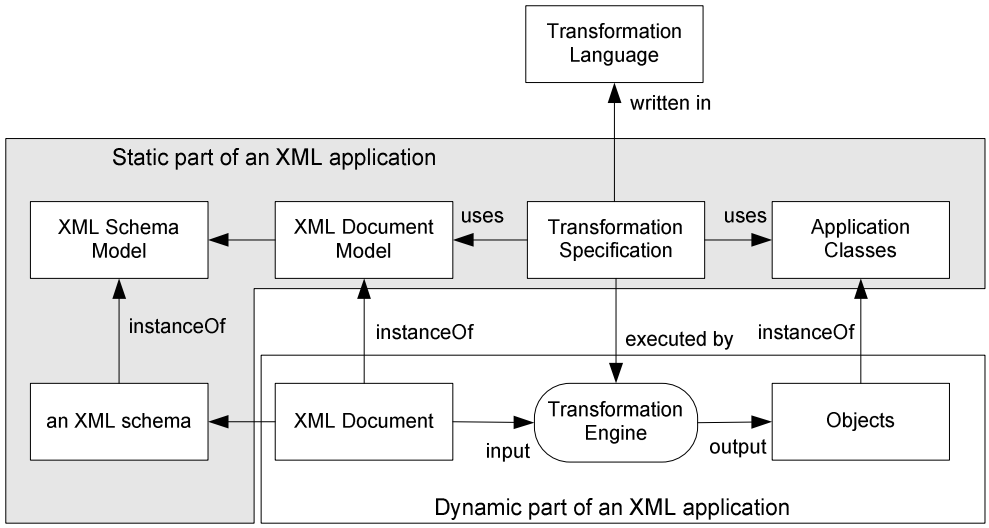


Figure 6.13. Structure of XML applications based on model transformations. Applications consist of a static and a dynamic part.

The static part of the application consists of the components surrounded by the gray area. In this part the optional XML schema, the transformation definition and the classes are the application-specific components. Application classes implement the intended meaning of the markup syntax constructs. The transformation definition specifies how the syntax is related to that meaning. Application classes do not contain syntax processing functionality. This functionality is captured in the transformation definition. The dynamic part of the application contains the components surrounded by the white rectangle in the lower right corner of Figure 6.13. The objects are part of the dynamic state of the XML application and are instantiated at runtime after the execution of the transformation.

6.5 Example of Model Driven XML Processing

We illustrate our approach on the basis of an example presented in this section. The example uses a simplified version of the Synchronized Multimedia Integration Language (SMIL) timing synchronization module [107] intended to be used together with other markup languages such as XHTML. A set of application classes written in Java is used to implement the behavior of the time dependency graph nodes according to the time model of SMIL. The structure of the example application is shown in Figure 6.14. It is an instance of the general pattern for XML processing in Figure 6.10.

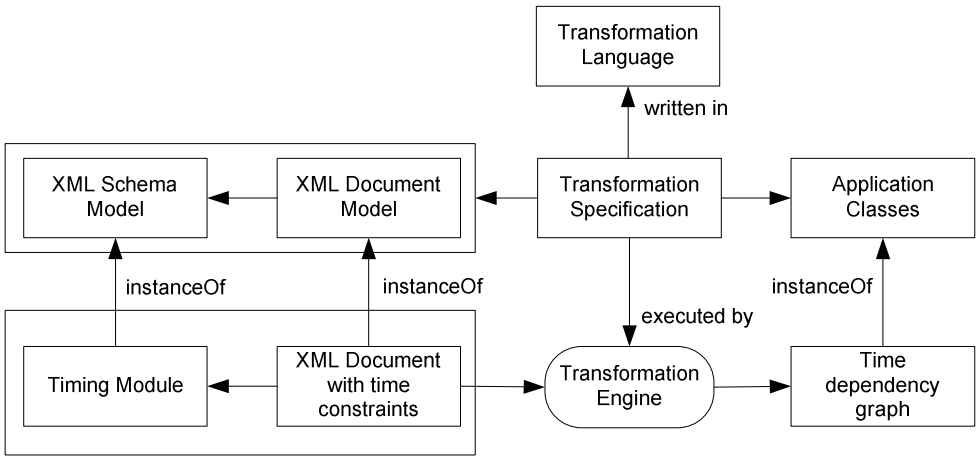


Figure 6.14 Structure of the example application

6.5.1 Example Source Schema and Application Classes

It should be noted that the SMIL timing model is rather complex and includes many capabilities. It is beyond the scope of the chapter to provide a detailed description of that model and the way how to design a time scheduler that implements the model. We limit ourselves to a very simplified subset of the timing module that uses a set of attributes indicating the type of the time element (*interval*, *parallel*, and *sequence*) and the *start*, *end* and *duration* properties. Our source schema contains four attributes taken from the SMIL specification. The following schema snippet shows the attribute definitions.

```
<attribute name='begin' type='string'/>
<attribute name='end' type='string'/>
<attribute name='dur' type='string'/>
<attribute name='timeContainer' type='string'/>
```

The attribute *timeContainer* may assume 3 values: *none*, *par* and *seq*. The first value *none* indicates that an element that has an attribute with value *none* is an atomic timed element (interval). The two values *par* and *seq* determine the element as a time container with a parallel and sequential scheduling of its children respectively.

An example XML document that uses these attributes is given below.

```
<a timeContainer='seq' begin='1' dur='20'>
  <b timeContainer='par' dur='10'>
    <c timeContainer='none' dur='10'/>
    <d timeContainer='none' dur='10'/>
  </b>
  <e timeContainer='none' dur='10'/>
</a>
```

Element *a* is a sequential time container that contains two children. The first one (element *b*) is a parallel container with two children (elements *c* and *d*). The second one (element *e*) is an interval element.

The processing of an XML document that uses time attributes results in creation of a time dependency graph. Such a graph captures the timing constraints and dependencies expressed in the document. The nodes of that graph reflect the semantics of interval and container nodes and implement their functionality. Time graph nodes are instances of application classes written in Java. A sketch of the classes is given below. Implementation of the timing functionality is not included since we focus only on the structure of the time dependency graph.

```
public interface ControlledObject {
  public void activate();
  public void deactivate();
}

public abstract class TimedElement{
  public int begin;
  public int end;
  public int dur;
  public ControlledObject ctrlObject;
  public void abstract start();
  public void abstract stop();
}

public class Interval extends TimedElement{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}

public abstract class TimeContainer extends TimedElement{
  public Vector components;
}

public class Parallel extends TimeContainer{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}

public class Sequence extends TimeContainer{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}
```

Class *TimedElement* is the abstract root class of the application hierarchy. Every node in the time graph is an indirect instance of that class. It has fields for the begin, end and the duration of the timed element. A node in the graph manipulates the behavior of an object. The object could be a text, picture, an audio clip or any other element. Timed ob-

jects must implement the interface *ControlledObject*. At the time of activation/deactivation of a node it invokes the operations *activate()* and *deactivate()* on the controlled object.

We have three types of time nodes: *Interval*, *Parallel* and *Sequence*. The latter two are concrete time containers that specialize the abstract class *TimeContainer*. Time containers have other nodes as children and impose a sequential or parallel order on their activation. Time containers may be nested. The execution semantics of the time graph is described in the SMIL specification [107].

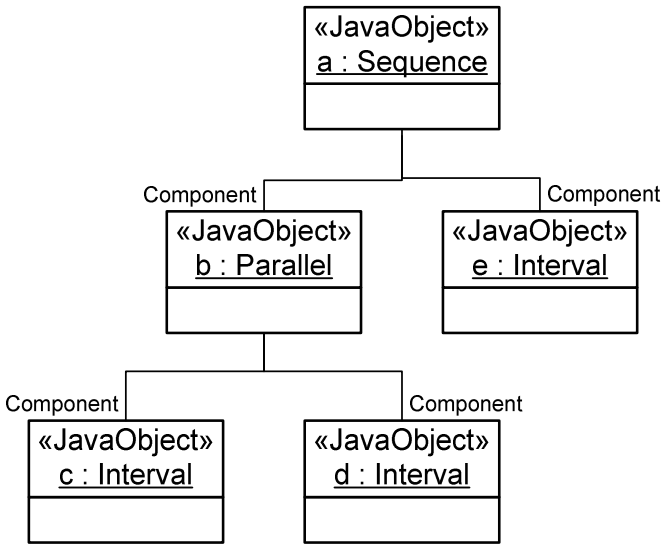


Figure 6.15 Example of a time dependency graph as a hierarchy of Java objects

Figure 6.15 shows an example of the time dependency graph built from the example XML document. Nodes in the graph are Java objects instances of the classes presented above.

6.5.2 Transformation Definition

The following transformation definition is meant for processing XML documents that contain timing attributes. After the execution of the transformation a time graph is built that captures the timing constraints specified in the document. The transformation definition does not process concrete controlled objects. They are provided by another markup language. An example of such a markup language is given in section 6.6.1.

```
1. transformation timeModule
2.    languages MOF, Java
3.    timeApplicationModel instanceOf(MOF) JavaModel
4.    source timedXMLDocument instanceOf(MOF) XMLModel default
5.    target timeGraph instanceOf(Java) timeApplicationModel default

6.    timedElementMapping abstract ModelElementRule {
```

```
7.      source[e:Element link-to(node), condition{XPath($e[@timeContainer])} ]
8.      target [node: TimedElement{begin, end, dur, ctrlObject}]

9.      SlotRules {
10.       beginValue
11.         source[beg:Attribute=XPath($e/@begin)]
12.         target [begin=toInt(beg.value)]
13.       endValue
14.         source[end:Attribute=XPath($e/@end)]
15.         target [end=toInt(end.value)]
16.       durValue
17.         source[duration:Attribute=XPath($e/@dur)]
18.         target [dur=toInt(duration.value)]
19.     }
20.   }
21.   parallelContainer ModelElementRule inherits timedElementMapping {
22.     source[ condition{XPath($e[@timeContainer='par'])} ]
23.     target[ node: Parallel{components} ]

24.     SlotRules{
25.       componentsValue
26.         source[timedChild:Element=XPath($e/*[@timeContainer])]
27.         target[components=target(timedChild, node)]
28.     }
29.   }
30.   intervalNode ModelElementRule inherits timedElementMapping {
31.     source[ condition{XPath($e[@timeContainer='none'])}]
32.     target[ node: Interval]
33.   }
```

The transformation definition is written in the language presented in Chapter 4 and is explained throughout this section. We apply the schema-less approach for XML processing presented in section 6.4.1. For the purposes of XML processing we allow the usage of XPath expressions in the conditions and in the initialization expressions of rule sources.

The first model element rule *timedElementMapping* (line 6) defines the common transformation functionality for all time nodes. The source contains a variable of type *Element* with an imposed condition (line 7). The evaluation of the rule source will return a set of element nodes that have an attribute *timeContainer* no matter what the attribute value is. In that way we select all the elements on which some time constraints are imposed. Note that the condition is written in XPath and refers to the variable *e* by using the notation *$e*.

Rule *timedElementMapping* contains one instantiation action based on class *TimedElement* assigned with the identifier *node* (line 8). Since this is an abstract class the rule cannot be executed and is declared as abstract. The instantiation enumerates the slots that must be assigned with values: *begin*, *end*, *dur* and *ctrlObject*. The rule has three slot rules that obtain values for slots *begin*, *end*, and *dur* (lines 9-19).

Rules *parallelContainer* (line 21) and *intervalNode* (line 30) inherit rule *timedElementMapping*. They specify the classes that will be instantiated for the parallel time container and interval node by overriding the instantiation labeled *node* in the parent rule. The identifier *node* is preserved, however, the classes are changed (see lines 23 and 32). The new classes are concrete and can be instantiated. Slot rules for obtaining the values of slots *begin*, *end* and *dur* are inherited. In this example rule inheritance follows the inheritance in the target intension. Rule inheritance allows reusing of the logic for calculating slot values defined in the super class. The rule for processing of sequential time container is skipped. It is similar to the rule for the parallel container.

Inheriting rules inherit the source element *e* and add new conditions. The conditions specified in the inheriting rules are logically and-ed to the inherited condition. Therefore, rule *parallelContainer* will be applied on all elements that have attribute *timeContainer* with value 'par' (line 22) and rule *intervalNode* will be applied on elements with attribute value 'none' (line 31).

An important feature of this definition is the usage of a common identifier (named *node*) in the rules for the instantiated time nodes (lines 8, 23, and 32). This allows time containers to treat in a uniform way their children nodes no matter what their exact type is. Usage of a common identifier is a kind of interface between different rules.

It should be noted that our example transformation definition is not complete since there is no slot rule for the slot *ctrlObject*. It is not known in advance what the controlled object is and how it is located. This is determined when the timing module is used together with another module to form a full language. Only in that case the information about the controlled object is available. Therefore, a new slot rule should be added based on the specific composition between the timing module and another module. This is explained in section 6.6 where our example is completed.

6.5.3 Creating Java Objects with Transformations

In the approach for XML processing described in this chapter Java objects are created as a result of the execution of transformations. Generally, these objects are not part of the modeling space in which the transformation engine operates. The objects are handled by a Java virtual machine in which the XML application runs. This requires a mechanism for integration between the transformation engine and the Java virtual machine. Furthermore, there are several issues related to manipulation of Java objects by the transformation engine: creation of objects, accessing slot values, and setting slot values. These issues are clarified in this section.

In Chapter 4 we outlined three approaches for handling model elements that are outside of the modeling space. The third approach proposes integration of the modeling space and the transformation engine with the native environment in which the model elements are handled. Furthermore, in Chapter 4 we described a prototype of a transformation engine for an early version of the language designed for XML processing [58]. The prototype is implemented in Java and invoked by the XML application also written in Java. Therefore, the modeling space, the transformation engine, and the generated Java objects are all in a single Java virtual machine.

The issues raised in this section are handled in the prototype in the following way:

- *creation of Java objects*: creation of objects is done by invoking the constructor of the Java class specified in the transformation definition. Transformation engine

uses Java reflection mechanism to invoke the constructor. If the constructor requires parameters then these parameters are declared as constructor slots in the transformation specification. This ensures that their values are available prior to the invocation of the constructor. The configuration of the Java language is responsible for implementing this mechanism. Also, if an object is created by invoking a method (e.g. Singleton design pattern) then this is specified as an additional information used by the operations in Java language configuration;

- *accessing and setting slot values*: in general, slots are Java fields. If the field is public then it is directly accessed. If the field is private and has getter and setter methods, then these methods are used. If the field is private without any way for direct access it must not be used as a slot. In case of other mechanism for accessing and setting slot values (e.g. via a method of the class) this mechanism is indicated in a file with additional information used by the Java language configuration. The implementation of the operations in Java language configuration relies on Java reflection mechanism;

6.6 Reuse and Composition of Transformations

In this section we show how our approach supports reuse of XML applications in the context of processing of compound documents. For this purpose we introduce another XML language that will be composed with the timing constructs in our previous example. This second language describes widget objects such as labels and images. For this new language we specify a second transformation definition that will be composed with the first one to form a new definition capable of processing compound documents that use markups from both languages.

The structure of the application that processes documents with time constrained widgets is shown in Figure 6.16. The schema of the hybrid language contains constructs in the timing module and the widget language. Transformation definition is a composition of the transformation definitions for the two languages. The application model is a composition of the widget classes and time graph classes.

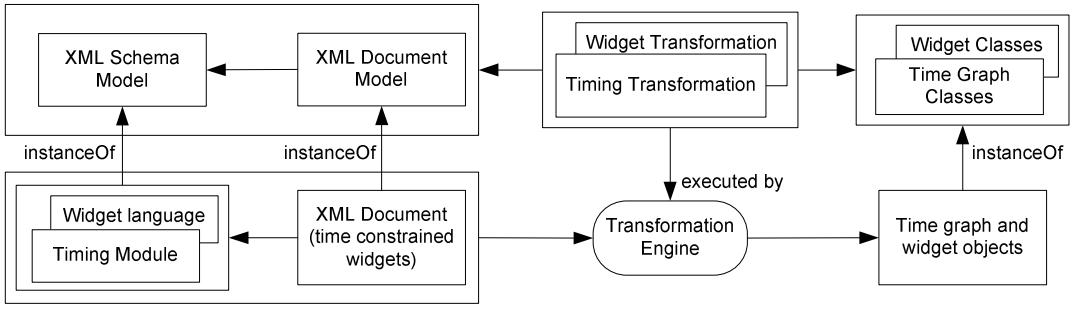


Figure 6.16 Composition of timing and widget markup languages and their processors

6.6.1 The Second Example Language

The second language contains simple elements that encode widgets and container elements that organize the widgets horizontally and vertically. We have two simple widgets for labels and images and two containers. The element names are *label, image, hbox,* and *vbox* respectively. For simplicity we skip the details about the size, font and color of the elements. A document written in that widget language is visualized by building and showing a hierarchy of layout objects following the nesting hierarchy of the document. Layout objects are instances of Java classes. The following code gives a sketch of the application class hierarchy.

```
public abstract class LayoutElement {
  public boolean visible;
  public boolean displayed;
  public abstract void draw();
  public abstract void refresh();
}

public abstract class Container extends LayoutElement{
  public Vector components;
}

public class Label extends LayoutElement{
  public String labelText;
  public void draw() {//implementation}
}

public class Image extends LayoutElement{
  public String imageFile;
  public void draw() {//implementation}
}

public class HBox extends Container{
  public void draw() {//implementation}
}

public class VBox extends Container{
  public void draw() {//implementation}
}
```

LayoutElement is an abstract class that defines the common functionality of widgets. The classes for concrete widgets specialize this class. Containers specialize class *Container,* which provides functionality for management of component widgets. There is one class for every concrete widget: *Label*, *Image*, *HBox*, and *VBox*.

6.6.2 Transformation Definition

The transformation definition for transforming documents into a hierarchy of layout widget objects is given below. Some slot rules are omitted.

```
transformation widgetModule
  languages MOF, Java
  widgetApplicationModel instanceOf(MOF) JavaModel
  source widgetXMLDocument instanceOf(MOF) XMLModel default
  target widgetHierarchy instanceOf(Java) widgetApplicationModel default

  labelRule ModelElementRule {
    source [e: Element link-to(widget), condition{e.name='label'}]
    target [widget: Label{labelText}]
  }
  imageRule ModelElementRule {
    source [e:Element link-to(widget), condition{e.name='image'}]
    target [widget: Image{imageFile}]
  }
  containerRule abstract ModelElementRule {
    source [e: Element link-to(widget)]
    target [widget: Container{components}]

    SlotRules {
    componentsRule
      source[child:Element=XPath($e/*)]
      target[components=target(child, widget)]
    }
  }
  hBoxRule ModelElementRule inherits containerRule {
    source[condition{e.name='hbox'}]
    target[widget: HBox]
  }
  vBoxRule ModelElementRule inherits containerRule {
    source[condition{e.name='vbox'}]
    target[widget: VBox]
  }
```

The transformation definition contains rules for every widget: *labelRule*, *imageRule*, *hBoxRule*, and *vBoxRule*. Rule *containerRule* is an abstract rule that defines the common transformation functionality for widget containers. This rule is inherited by *hBoxRule* and *vBoxRule*. Similarly to the transformation definition for timing module, we use a common identifier name (*widget*) in the target of the rules.

6.6.3 Composing Languages

A required step for integration of the two markup languages is to establish an interpretation of the activation and deactivation events in terms of the widget language. Following the ideas of XHTML+SMIL [108] we choose two possible interpretations of activation. The first one affects the *visible* property of widgets and the second one affects the *displayed* property of widgets. Making a widget invisible means that it is not shown on the screen but still generates a layout element and affects the layout of other elements. Turning off the *displayed* property means that the widget is not shown and no layout element is created for it. This interpretation is specific for the composition of the two languages and needs an explicit indication. We introduce a new attribute to indicate the exact action. Similarly to XHTML+SMIL the attribute name is *timeAction* with two possible values: *visibility* and *display*.

The next step is to integrate the two applications in order to obtain a processor for compound documents. This requires composition of the transformation definitions and the application classes. We first discuss the composition of the application classes.

6.6.4 Composing Application Classes

Every node in the time dependency graph must hold a reference to the object it controls. After the composition of the languages it is known which the controlled objects are: these are the widgets objects. Controlled objects must implement the *ControlledObject* interface. However, class *LayoutElement* does not implement this interface. Therefore we have a type compatibility problem. To overcome the problem and to perform the required composition between the classes we turn to the Adapter design pattern [41]. We introduce a new Java class that implements *ControlledObject* interface and holds a reference to the actual layout element being controlled. The invocations of *activate* and *deactivate* methods lead to changing the *visible* and *displayed* properties of the layout objects. Since there are two possible interpretations of activation/deactivation we create one adapter class per interpretation. The first adapter class is called *ChangeVisibility* and is shown below:

```
public class ChangeVisibility implements ControlledObject{
  public LayoutElement obj;
  public void activate(){
    obj.visible=true;
    obj.refresh();
  }
  public void deactivate(){
    obj.visible=false;
    obj.refresh();
  }
}
```

The second adapter class *ChangeDisplay* is implemented in a similar way.

6.6.5 Composing Transformation Definitions

This section explains the composition of the transformation definitions for our two languages (section 6.5.2 and section 6.6.2). The composition of the transformation definitions results in a new definition named *timedWidgetsModule*. All the rules in the two transformation definitions are reused (see the *include* clause) and new rules are added to instantiate the adapter classes shown in section 6.6.4. The resulting transformation definition is shown below.

```
transformation timedWidgetsModule
  include timeModule, widgetModule
  languages MOF, Java
  composedApplicationModel instanceOf(MOF) JavaModel
  source hybridDocument  instanceOf(MOF) XMLModel default
  target timedWidgetHierarchy instanceOf(Java) timedWidgetsApplicationModel default

  visibilityRule ModelElementRule {
    source[e:Element link-to(adapter), condition{XPath($e[@timeAction='visibility'])}]
    target[adapter:ChangeVisibility{obj=target(e, widget)}]
  }

  displayRule ModelElementRule{
    source[e:Element link-to(adapter), condition{XPath($e[@timeAction='display'])}]
    target[adapter: ChangeDisplay {obj=target(e, widget)}]
  }

  ctrlObjectValue SlotRule owner=timedElementMapping {
    target[ctrlObject=target(e, adapter)]
   }
```

Two model element rules are added to instantiate the adapter classes: *visibilityRule* and *displayRule*. The slot rule *ctrlObjectValue* determines the value of the slot *ctrlObject*. This slot rule is associated to *timedElementMapping* rule from *timeModule* definition.

6.7 Related Work

Explicit specification of the XML semantics can be done in one of the formalisms for programming language specification. A number of papers adapt techniques for specification of computer language semantics in the context of XML as a syntactical framework. In [87] the semantics of an XML language is given in the form of an attribute grammar [54]. This opens the possibility for applying the results and tools of extensive research available in that area. In attribute grammars translation is performed over attributed trees. The difference with our approach is that in our approach translation is performed as a transformation from a document tree to a graph.

RelaxNGCC [70] is based on compiler-compiler techniques to build a processor for a language conforming to a RelaxNG schema. This provides more flexibility in bridging between the application model and document syntax and in associating behavior with XML documents. It allows reuse of already existing classes and deals better with structural differences between the syntax and classes. In addition, it generates a dedicated parser for a given XML language. Our transformation rules can be seen as similar schema annotations for W3C schemas and for schema-less documents. The approach for modular and extensible processors presented in [94] is inspired by denotational semantics of computer languages. Processors operate on document trees and do not rely on a schema. Our approach permits both types of processing: document-based and schema based.

Another dimension of the approach presented in this chapter is the explicit specification of the relation between the syntax and another type of structure (a model, a database, an ontology). The approach presented in [10] maps XML documents to domain ontologies. Mapping rules rely on XPath. The primary goal of mapping is to allow translation of queries over ontologies to queries over source documents. In our work the rules are used to transform source documents into a set of objects. Other papers that discuss the problem of bridging between XML syntax and ontology are [71] where a mapping ontology is presented that transforms XML documents to their RDF representation and [83] where the authors suggest a unification approach for XML and RDF based on model-theoretic semantics. In [31] a framework for expressing the semantics of markup is defined. The semantics of markup is a set of inferences that can be drawn from the document. PROLOG is used as an implementation language for inference rules. In the context of this work our transformation rules are particular types of inference rules. However, we rely on a domain-specific language for transformation specification and aim at a closer integration with object-oriented programming languages.

There exist a number of languages dedicated to XML processing: XSLT, XDuce [46], XL [40]. All of them transform XML documents to other XML documents and their type system is based on XML types. Our approach is focused on transformations to application objects and uses types from a programming language, in our case Java.

The problem of reuse of language processors and building new languages by composing existing modules has been addressed in research of programming language development and generally the problem proved to be hard. Existing work studies the composability properties of frameworks for semantics specification: attribute grammars [35], denotational semantics, operational and action semantics [68][67]. These techniques rely on mathematical formalisms to specify the semantics. Transformation rules in our approach may be perceived as specification of the language semantics in a domain-specific transformation language that has features closer to programming languages. In this respect it is more familiar to software developers than the enumerated formal techniques.

There are tools supporting XML processing in browsers. XSmiles [88] is a browser that supports some of today's web languages. Mozilla browser [69] provides a framework for client-side web applications relying on a set of XML languages. Both tools provide an extensible architecture for XML applications. In contrast, our approach does not define an architecture nor a tool but stresses on the explicit specification of the language semantics that can be further employed in a tool. XVM [62] is an extensible architecture for XML processing based on associations between XML elements and software components that implement their behavior. Our approach allows similar association of processing logic to more complex structures in the document (e.g. a tuple of elements and attributes). In

XVM many aspects of the processing may remain hard-coded in the components while our approach declares this explicitly.

6.8 Conclusions

We presented an approach for development of XML applications based on specification of model transformations from a document model (or document schema) to a set of application-specific classes.

The approach frees the software engineer from writing low level and error-prone code against a generic syntax document model such as DOM and SAX. Instead, a declarative transformation definition is written that establishes the relations between syntax constructs and application structures. Transformation definitions have an operational semantics and are executed by an interpreter. Another possible approach is to apply a compilation on transformation definitions to produce a dedicated XML processor for a given language. This is inspired by the compiler-compiler approach mentioned in the chapter. Transformation rules are considered as semantic annotations.

The main purpose of the approach is to improve adaptability and reusability properties of XML applications. The strength of the approach is that it makes processing logic of the application explicit by expressing it in a set of rules that can be manipulated. However, we observe also problems in achieving these properties.

Adaptability is required to respond to changes in the application in an easy and cheap way without a redesign and recompilation of the whole application. Changes may occur in the language syntax, in the transformation definition and in the application classes. These three aspects are clearly separated from each other and may evolve independently. The most difficult case is the change of the syntax since it usually brings changes in the other two components. In our approach the required changes can be isolated in specific rule(s) and classes. The adaptability in that case is derived from the finer control over the application components. Additive adaptations seem to be easier to handle. They usually require additions of new rules and classes that must be integrated with their counterparts. Deletions and replacement of components may be more difficult since this will require replacement of classes and refactoring the transformation rules. However, these changes are still isolated and may be done without recompilation and redeployment of the whole application.

Another quality property being pursuit is the reusability of XML applications motivated by the need of hybrid languages and compound documents. Usually, an application is reused and composed together with other applications. In that case there are two distinct problems: the composition of transformation rules and the composition of application classes. They are driven by the composition of the XML languages. Composition of transformation definitions is achieved by the available operators in the transformation language. Every language, however, has limitations with respect to the available composition operators. Chapter 5 suggested a technique for extending transformation languages.

The second problem related to the composition of XML applications is the composition of the application classes which is a case of software composition. In general, this is a problem that proved to be difficult. In our example it was easily solved by using the Adaptor design pattern. This is possible if the composition is anticipated and the application is properly designed. In many cases, however, the composition is not anticipated and

the application classes may not be composable. Composition may be done on source code and on already compiled classes. We can benefit from research in the area of aspect-oriented software development that provides advanced software composition techniques beyond the aggregation and inheritance [39]. This is the main direction for future research.

One problem that remains open is the scalability of the approach. It works on examples that are relatively simple but it is not clear what happens in complex cases with larger number of languages involved. A possible approach is to start with a small stable set of languages and to create modular processors for them that can be composed with each other. This could be the domain of the Web languages. The composition of markup languages is complicated further by the requirement that software engineers should know the details of the language semantics which is not always simple.

In this chapter we focused on the possibility to utilize model transformations for the specification of XML language semantics. A possible further step is developing a tool that supports the tasks not discussed here such as deploying, updating and composing transformations and classes in a browser-like environment. This is similar to the architectures that XSmiles and Mozilla provide.

7

Conclusions

Sometimes, the journey is more important than the final destination.

7.1 Introduction

This chapter gives the overall conclusions of the thesis. First, in section 7.2 we summarize the core concepts presented in Chapter 2. In Section 7.3 we reflect on the main research issue in this thesis: adaptability of model transformations in the context of Model Driven Engineering. Three problems were derived from this issue: identification and selection of alternative transformation definitions, definition of transformations between models expressed in multiple languages, and decomposition and composition of transformation definitions. We give the conclusions for these problems in sections 7.4, 7.5, and 7.6 respectively.

7.2 Concepts: Models, Meta-models, Intensions, and Extensions

Chapter 2 presented a set of concepts used throughout the thesis. The concepts of *model*, *meta-model*, *intension*, *extension*, and *model transformation* were analyzed and a set of definitions was given based on literature study. In this thesis models are considered as symbolic entities expressed in a modeling language. Models provide knowledge about other entities (object systems). Meta-models are models of modeling languages.

Apart from the distinction between models and meta-models we distinguish between *intensions* and *extensions*. In this thesis we consider intensions as models of other models.

An intension is associated to an extension. The extension is a set of all the possible models modeled by the given intension. A given model may be an instance of multiple intensions. We consider *instanceOf* relation in the context of a given modeling language.

The dichotomy intension/extension is more general than the dichotomy meta-model/model. A meta-model is always an intension. Its extension is the modeling language modeled by the meta-model. However, not all intensions are meta-models. An important set of intensions are models of domains expressed in a given modeling language. The *instanceOf* relation between an intension and the members of its extension plays the central role in the model transformation language MISTRAL proposed in Chapter 4 of the thesis.

7.3 The Problems of Adaptability of Model Transformations

The need for adaptability of model transformations is motivated by an analysis of the model transformation pattern given in Chapter 2, Figure 2.28. We repeat this figure below (Figure 7.1). It shows a transformation definition expressed in a model transformation language. The transformation definition is based on multiple source and target intensions. Transformation execution is a process that accepts a source model and generates a target model both satisfying a set of requirements.

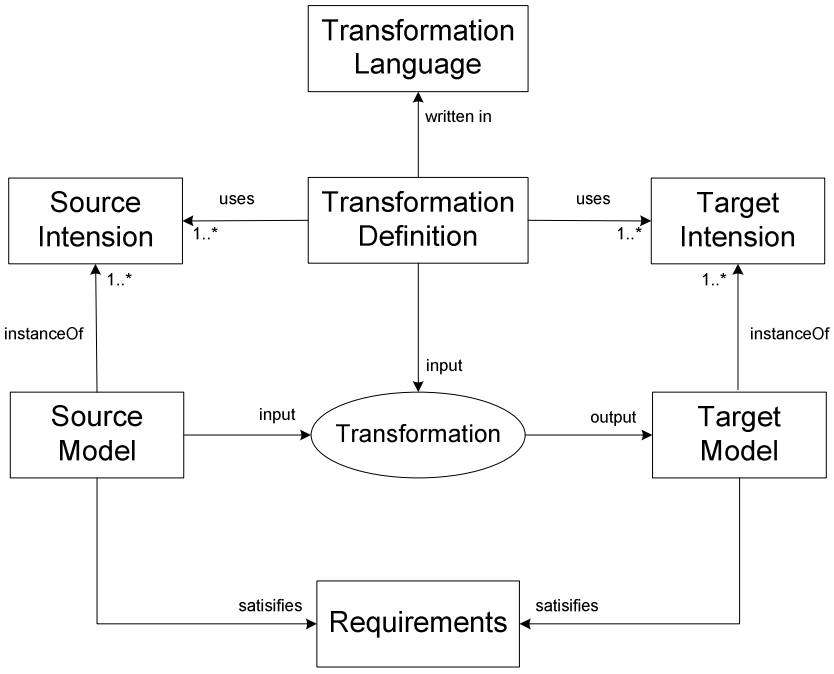


Figure 7.1 The model transformation pattern (Chapter 2, Figure 2.28)

Various changes may occur in the components shown in the figure that affect other components. The following list is a summary of the discussion about these changes given in Chapter 2. We focus on the changes concerning model transformation definitions and model transformation languages:

- *Changes in requirements.* Changes in requirements may affect the transformation definition. We have two possibilities. First, an alternative transformation definition may be required that produces a new target model, eventually with an equivalent functionality to the previous one but with different quality properties. Second, there may be a need for adapting the transformation definition in order to produce a target model that satisfies the new requirements;
- *Changes in intensions.* Two possibilities exist. First, an intension is replaced by another intension because the modeling language is changed. In this case the intension is the meta-model of the new modeling language. Therefore, the transformation language should be capable of working with multiple modeling languages. Second, an intension evolves and this evolution brings new elements in the intension or causes removal of existing elements. Since the transformation definition is based on intensions, the definition must be adapted accordingly;
- *Changes in models.* Changes in the source model may also affect the transformation definition. Some of the newly added elements may introduce new rules in the definition or may require changes in already existing rules. Again, there may be a need of adaptation of the transformation definition;

On the basis of this analysis three problems were identified:

- Identification and comparison of alternative transformations for a given source model;
- Definition of transformation languages that may transform models expressed in multiple modeling languages;
- Coping with changes in transformation definitions;

To cope with the first problem we proposed a formal technique for transformation space analysis. Conclusions related to this problem are presented in subsequent Section 7.4. To cope with the second problem we proposed a model transformation language called MISTRAL capable of working with multiple modeling languages. Conclusions about this problem are given in Section 7.5. To cope with the third problem we studied ways for decomposition and composition of transformation definitions and the required language support. Section 7.6 gives conclusions on this problem.

7.4 Identification and Comparison of Alternative Transformations

In Chapter 3 we illustrated the possibility for alternative transformation definitions for a given source model. Alternative transformations may produce functionally equivalent target models that may differ in the quality properties they possess.

Two problems were identified. The first problem is the identification of alternative transformations. The second problem is how to compare and select among alternative transformations in order to obtain the transformation definitions that produce a model with the desired properties.

Software engineers are often faced with these problems. However, the identification and comparison of alternatives is more implicit than explicit. Software engineers usually

rely on their experience and intuition. A systematic approach to these two problems is required.

In Chapter 3 we suggested that the analysis of alternative transformations should be included as an explicit activity in an MDA based development process. A working product in this activity is a *transformation space*. A transformation space models a set of alternative transformations for a given source model.

A technique for defining transformation spaces was presented in Chapter 3. This technique requires a source model, its meta-model and the meta-model of the target as input and generates a transformation space as output. The technique provides operations for reduction of spaces. Two operations are defined: for selection and for exclusion of alternatives from spaces based on certain criteria. In this technique the desired quality properties of the target models are captured in a quality model. Quality properties are explicitly included in transformation spaces. By this way, the software engineer is able to select transformations from transformation spaces on the base of the quality requirements.

In Chapter 3 we presented a case study on transforming UML class models into XML schemas. Target XML schemas had to be extensible in order to cope with expected changes in the source model. The application of the technique resulted in two alternative schemas, both satisfying the quality requirement for extensibility

Although transformation spaces tend to be rather large even for simple models, they are purely conceptual. The software engineer does not need to generate the full set of alternatives in a transformation space unless a number of reduction steps are applied and the size of the space is reduced sufficiently. The structure of a transformation space provides a framework to reason about the alternatives in general instead of per alternative individually.

To reduce transformation spaces software engineers may utilize various knowledge sources. One possible source is the potential correspondences among the constructs in the meta-models. For example, an UML class may be mapped to a complex type whereas mapping to a simple type may be unfeasible. The second possible source is available heuristic rules related to the quality model being used. In the presented case study both types of knowledge were used. We applied rules from XML schema best practices to obtain extensible schemas [118].

The technique described in Chapter 3 was applied on a rich source model that contains enough information to generate the target model. The case study may be regarded as a PIM to PSM transformation in the context of MDA. PIM to PSM transformations are examples of refinement transformations. Usually, refinement transformations add additional details in the target model. The applicability of the technique in this type of transformations needs further study. We envisage the usage of patterns in such a refinement process. Patterns may be used as coordinates in the dimensions of transformation spaces.

The process of definition and reduction of transformation spaces may benefit from tool support. We are working on a prototype of a tool that supports developers in definition of transformation spaces and performing reduction steps. Such a tool may also maintain a knowledge base with heuristic rules that can be applied during the reduction process.

7.5 Transforming Models expressed in Multiple Languages: Transformation Language MISTRAL

According to the MDA guide [72] transformation definitions are based on source and target meta-models expressed in the MOF language. In Chapter 4 we presented four transformation scenarios in the MOF meta-modeling architecture. Two of them were based on transformation definitions using models expressed in languages different from the MOF language. Analysis of the scenarios revealed that it is not possible to apply current model transformation languages for all the scenarios. The main reason for this is the coupling between transformation languages and the MOF language. This coupling is mainly caused by the inability of the MOF meta-modeling architecture to explicitly define important features of modeling languages such as instantiation and generalization mechanisms. Therefore, current model transformation languages cannot cope with the second problem formulated in this thesis: defining transformations on models expressed in various modeling languages.

To decouple a transformation language from particular modeling languages we analyzed five common operations performed in model transformations: *selection* of model elements on the base of their meta-construct, *instantiation* of model elements from a meta-construct, *accessing* slot values, *setting* slot values, and *deletion* of model elements. The analysis showed that these operations are influenced by the instantiation and generalization mechanisms for a given modeling language. These mechanisms were represented as a set of functions. These functions may be used by a transformation language to perform the five operations enumerated above. Functions form an interface called *language configuration* for the interaction between a modeling language and a transformation language. Different modeling languages should provide an implementation of their own language configuration.

In Chapter 4 we proposed a model transformation language called MISTRAL (Multiple Intension Transformation Language) and a modeling space in which the language can be used.

The modeling space organizes models and modeling languages. It provides simple constructs for representing models as sets of model elements related with each other via slots. On top of these simple constructs various modeling frameworks may be represented. We showed how a simplified MOF model may be represented in the modeling space. By defining the configuration of the MOF language it is possible to instantiate meta-models and models via the MOF instantiation mechanism.

The transformation language is based on the dichotomy between intension and extension explained in Chapter 2. Transformation definitions are specified on the base of intensions and executed over models members of the extensions of these intensions. Transformation definitions may be specified between models expressed in more than one modeling language. This is possible because the transformation language and its transformation engine are parameterized with respect to the implementation of language configurations. Adaptability of the transformation language with respect to new modeling languages, therefore, depends on the possibility to capture the modeling language features in a language configuration.

The examples in Chapter 4 and Chapter 6 showed how the first two transformation scenarios presented in Chapter 4 may be supported in the modeling space by using the language MISTRAL. The third scenario requires further investigation. It is an example of language translation where a transformation definition between language models is used to derive a transformation definition for intensions expressed in these languages. The fourth scenario uses two standard transformations in MOF architecture (XMI [79] and JMI [50]). We consider these transformations as future case studies that would further test the expressiveness of the transformation language.

7.6 Decomposition and Composition of Transformation Definitions

The third problem formulated in this thesis is about coping with changes in transformation definitions. A possible approach to handle changes in software artifacts is to represent them as a composition of simpler units. In the context of model transformations we studied different ways for decomposition and composition of transformation definitions. Decomposition of transformation definitions requires language constructs capable of expressing the units of decomposition. Composition of such units requires compositional operators. In Chapter 5 we set the research objective to identify requirements for such constructs and compositional operators and to evaluate how current transformation languages satisfy these requirements.

In Chapter 5 we described three general cases that motivate decomposition and composition of transformation definitions: decomposition for managing complexity, evolution of transformation definitions caused by evolution of models, and composition of transformation definitions driven by composition of models.

These cases were studied in more details in four transformation scenarios. The first scenario illustrated the impact of decompositions in the source and target intensions on the decomposition of transformation definitions into rules. The second scenario showed how different decompositions in models that are transformed influence the transformation definitions. The third scenario identified transformation functionality suitable for modularization that is not related to source and target models. The fourth scenario studied how changes in the intensions influence transformation definitions.

For every scenario we derived a set of requirements that a transformation language must fulfill in order to handle the scenarios. Several representative transformation languages were evaluated against these requirements. Evaluation was organized around five language features: *modularity, rule interaction and ordering, compositional operators, adaptation of transformation definitions*, and *reflection*.

The following conclusions may be derived on the base of the analyzed scenarios:

- Decomposition of transformation definitions is influenced by the decompositions found in the source and target intensions. Intensions may be decomposed in multiple ways. Software engineers have to consider all relevant decompositions in the source and target intensions and how they interact with each other. Taking only one decomposition and neglecting the others leads to anomalies in transformation definitions such as tangling and scattering of transformation functionality. This ultimately hinders the adaptability and reusability of transformation definitions;

- Decomposition of transformation definitions is influenced by possible multiple mutually dependant hierarchies in source and target models. These hierarchies have an impact on the order of execution of transformation rules. Software engineers must carefully plan the order of execution of transformation rules with respect to the dependencies among the hierarchies;
- There is a need for modularizing pieces of transformation functionality that do not follow the common rule structure of left-hand and right-hand parts. Reflective capabilities of a transformation language may support specification of functionality in a generic way;

The following conclusions may be derived from the evaluation of transformation languages:

- Transformation rules are the basic modular units in current transformation languages. In general, it is possible to express the required decompositions in the scenarios by using transformation rules;
- Current transformation languages provide two mechanisms for rule composition: inheritance and rule invocation. In principle, both can be used to implement the required compositions identified in the scenarios. However, these mechanisms may expose some anomalies and may reduce the adaptability of transformation definitions. Using rule inheritance may lead to an excessive number of rule definitions. Using rule invocation may lead to a tight coupling between rules that deteriorates adaptability of definitions when rules are replaced. Whenever possible, software engineers should prefer interaction between rules based on traceability links since it leads to loose coupling between rules.
- Declarative and hybrid languages provide better support in dealing with rule execution order compared to imperative transformation languages. Declarative languages encode relations among model elements. Detection and handling the dependencies among model elements is done by the transformation engine. In contrast, imperative languages require explicit encoding of rule execution order and leave handling the dependencies to the software engineer;
- Some scenarios showed the presence of crosscutting functionality in transformation definitions. Application of two compositional mechanisms exposed some anomalies in the resulting transformation definitions. The anomalies are related to a reduced adaptability of the definitions and to introduction of an excessive number of rules. We may conclude that the support for expressing and composing crosscutting transformation functionality should be improved;
- One scenario showed a need for reflective capabilities in transformation languages. Only one language among the evaluated ones declares such a support. The applicability of reflection in model transformations is an issue that needs further investigation;

As an overall conclusion about the evaluated languages we can state that they provide a reasonably full set of modular constructs but still have problems in handling some compositional and evolution scenarios.

In the end of Chapter 5 we proposed an approach for enhancing the transformation language MISTRAL described in Chapter 4 with new constructs. The enhancement is done in the abstract syntax definition of the language. Newly added constructs are associated to

transformation definitions that process these constructs. A direction for future research is studying the applicability of this approach on modeling languages in general.

In this thesis we studied adaptability of model transformations in the context of Model Driven Engineering. We identified three problems: identification of alternative transformations for a given model, definition of adaptable model transformation languages, and specification of adaptable transformation definitions. The main contributions in the thesis are the proposed technique for analysis of transformation spaces, a transformation language MISTRAL for expressing transformation definitions on models expressed in multiple languages, and a study on decomposition and composition of transformation definitions. These contributions are possible solutions for the overall problem of adaptability of model transformations.

References

[1] Adams, S. Modular Grammars for Programming Language Prototyping. PhD Thesis, University of Southampton, 1991

[2] Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.The Design of a Simple Language for Graph Transformations, Journal in Software and System Modeling, in review, 2005

[3] Akehurst, D., Kent, S. A relational approach to defining transformations in a metamodel. In proceedings of UML2002, Germany, 2002

[4] Aksit, M., Mostert, R., Haverkort, B. Compiler generation based on grammar inheritance. Report 90-07, University of Twente, the Netherlands, 1990

[5] Aksit, M., Tekinerdogan, B. Deriving design alternatives based on quality factors. In M. Aksit (Ed.), *Software Architectures and Component Technologies* (pp. 225-257). Kluwer Academic Publishers, 2002

[6] Aksit, M. The C'7 for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering. Turkish Journal of Electrical Engineering, vol. 12 (2), 2004

[7] Alanen, M., Lilius, J., Porres, I., and Truscan, D. Realizing a Model Driven Engineering Process. TUCS Technical report No 565, 2003

[8] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Technologies Corp. Response to the MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2003-08-05, 2003

[9] Álvarez, J., Evans, A., Sammut, P. Mapping between levels in the metamodel architecture. In Proceedings of UML2001, Springer-Verlag Heidelberg, Toronto, Canada, 2001

[10] Amann, B., Beeri, C., Fundulaki, I., Scholl, M. Querying XML sources using an ontology-based mediator. In proceedings of CoopIS/DOA/ODBASE, 2002

[11] Apostel, L. Towards the formal study of models in the non-formal sciences. In H. Freudenthal (Ed.), *The concept and the role of the model in mathematics and natural and social sciences*. D. Reidel Publishing Company, Dordrecht, the Netherlands, 1960

[12] Atkinson, C., Kühne, T. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12 (4), pp. 290-321, 2002

[13] Atkinson, C., Kühne, T. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5), pp. 36-41, 2003

[14] Beckett, D. RDF/XML syntax specification. W3C Document, 2003

[15] Bézivin, J., Lemesle, R. Ontology-based layered semantics for precise OA&D modeling, In ECOOP'97 Workshop Reader, Finland, 1997

[16] Bézivin, J., Gerbe, O. Towards a precise definition of the OMG/MDA framework. In Proceedings of Automated Software Engineering, USA, 2001

[17] Bézivin, J., Farcet, N., Jezequel, J-M, Langlois, B., Pollet, D. Reflective model driven engineering. In Proceedings of UML2003, USA, 2003

[18] Bézivin, J., Dupe, G., Jouault, F., Pitette, G., Rougui, J. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In Workshop on generative techniques in the context of MDA. OOPSLA2003, Anaheim, California, USA, 2003

[19] Bézivin, J. In search of a basic principle for model driven engineering. Novatica, March, 2004

[20] Bird, L., Goodchild, A., Halpin, T. Object-Role modeling and XML-Schema. In Int. Conf. On Conceptual Modeling (ER2000), Salt Lake City, UT, USA, 2000

[21] Boman, M., Bubenko Jr., J.A., Johannesson, P., Wangler, B. Conceptual Modelling. Prentice Hall, 1997

[22] Bonestroo, W. Evaluation of a Model Driven Architecture model transformation language: TRL. MSc Thesis, University of Twente, 2004

[23] Bowers, S., Delcambre, L. On modeling conformance for flexible transformation over data models. In Proceedings of the Workshop on Knowledge Transformation for the Semantic Web at the 15th European Conference on Artificial Intelligence (KTSW-2002), Lyon, France, 2002

[24] Brickley, D., Guha, R. V. RDF vocabulary description language 1.0: RDF Schema. W3C Document, 2003

[25] Brinkkemper, S. Formalisation of information systems modeling. PhD Thesis, University of Nijmegen, 1990

[26] Carlson, D. Modeling XML vocabularies with UML. Addison-Wesley, 2001

[27] Czarnecki, K., Helsen, S. Classification of model transformation approaches. OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003

[28] DeRose, S., Maler, E., Orchard, D. XML Linking Language (XLink). W3C Document, 2001

[29] Dobson, S. Modular Parsers. Technical Report TCD-CS-1998-19, University of Dublin, 1998

[30] DSTC, IBM, CBOP. MOF Query/Views/Transformations. Second Revised Submission. OMG document ad/2004-01-06, 2004

[31] Dubin, D. Object mapping for markup semantics. In T. Usdin (Ed.), Proceedings of Extreme Markup Languages 2003, Montreal, Quebec, 2003

[32] Duke, R., Rose, G., and Smith, G. Object-Z: a specification language advocated for the description of standards. Technical report 94-45, Software Verification Research Center, University of Queensland, Australia, 1994

[33] Embley, D., Mok, W.Y. Developing XML documents with guaranteed "good" properties. 20th Int. Conf. on Conceptual Modeling (ER2001), Yokohama, Japan, 2001

[34] Falkenberg, E.D., Hesse, W., Lindgreen, P., Nilsson, B.E., Han Oei, J.E., Rolland, C., Stamper, R.K., van Assche, F.J.M., Verrijn-Stuart, A.A., Voss, K. A framework of information system concepts. The FRISCO report. 1998

[35] Farrow, R., Marlowe, T.,J., and Yellin, D.,M. Composable attribute grammars: support for modularity in translator design and implementation. 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, US. 1992

[36] Favre, J-M. Foundations of meta-pyramids: languages and metamodels – Episode II: story of Thotis the Babbon. Available at http://www-adele.imag.fr/~jmfavre/. 2004

[37] Favre, J-M. Towards a basic theory to model Model Driven Engineering. 3d Workshop in Software Model Engineering. In conjunction with UML2004. Portugal, 2004

[38] Favre, J-M., Nguyen, T. Towards a megamodel to model software evolution through transformations. Workshop on Software Evolution through Transformations (SETRA2004). In conjunction with 2nd Intl. Conference on Graph Transformations, Rome, Italy, 2004

[39] Filman, R., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley. 2004

[40] Florescu, D., Grunhagen, A., and Kossmann, D. XL: an XML programming language for web service specification and composition. 11th international conference on WWW, Honolulu, Hawaii, USA, 2002

[41] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[42] Gardner, T., Griffin, C., Koehler, J., and Hauser, R. A review of OMG MOF 2.0 Query/Views/Transformations submissions and recommendations towards the final standard. 1st International Workshop on Metamodeling for MDA, York, UK, 2003

[43] Geisler, R., Klar, M., Pons, C. Dimensions and dichotomy in metamodeling. In D.J. Duke and A.S.Evans (Eds.), 3rd BCS-FACS Northern Formal Methods Workshop. Springer-Verlag, New York, USA, 1998

[44] van Gigch, J.P. System design modeling and metamodeling. Plenum Press, New York, 1991

[45] Hausmann, J., H. Metamodeling relations – relating metamodels. In 1st International Workshop on Metamodeling for MDA. York, UK, 2003

[46] Hosoya, H., Pierce, B. XDuce: A typed XML processing language. In Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, (pp. 226–244), 2000

[47] Hughes, R.I.G., The Ising model, computer simulation, and universal physics. In M. Morgan and M. Morrison (Eds.), *Models as mediators. Perspectives on natural and social science.* Cambridge University Press, 1999

[48] ISO 9126, Information Technology, Software Product Quality – Part I: Quality model, International Organization for Standardization, FCD 1998.

[49] Jacobson, I., Booch, G., and Rumbaugh, J. The unified software development process. Addison-Wesley. 1999

[50] Java Metadata Interface (JMI). Available at http://java.sun.com

[51] Kalnins, A., Barzdins, J., Celms, E. Model transformation language MOLA. In U. Asmann (Ed.), *Proceedings of Model Driven Architecture: Foundations and Applications 2004*. Linkoping, Sweden, 2004

[52] Kent, S. Model Driven Engineering. In Proceedings of IFM2002, LNCS 2335, Springer, 2002

[53] Kleppe, A., Warmer, J., Bast, W. MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003

[54] Knuth, D. Semantics of context free languages, 1968

[55] Kurtev, I., Bézivin, J., Aksit, M. Technological Spaces: an initial appraisal. CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, CA, USA, 2002

[56] Kurtev, I., van den Berg, K. A synthesis-based approach to transformations in an MDA software development process. In A. Rensink (Ed.), *Model Driven Architecture: Foundations and Applications 2003*, CTIT Technical Report TR-CTIT-03-27, University of Twente, the Netherlands, 2003

[57] Kurtev, I., van den Berg, K., Aksit, M. UML to XML-Schema transformation: a case study in managing alternative model transformations in MDA. Forum on specification and Design Languages (FDL2003), Frankfurt, Germany, 2003

[58] Kurtev, I., van den Berg, K. Model Driven Architecture based XML processing. Proceedings of ACM Symposium on Document Engineering, Grenoble, France, 2003

[59] Kurtev, I., van den Berg, K. Unifying approach for model transformations in the MOF metamodeling architecture. In M. van Sinderen and L. Pires (Eds.), *1st European MDA Workshop on Industrial Applications (MDA-IA),* CTIT Technical report TR-CTIT-04-12, Enschede, the Netherlands, 2004

[60] Kurtev, I., van den Berg, K. A language for model transformations in the MOF meta-modeling architecture. In U. Asmann (Ed.), *Proceedings of Model Driven Architecture: Foundations and Applications 2004*. Linkoping, Sweden, 2004

[61] Kurtev, I, van den Berg, K. Building adaptable and reusable XML applications with model transformations. To appear in Proceedings of 14th International World Wide Web Conference (WWW2005), Chiba, Japan, 2005

[62] Li, Q., Kim, M.Y., So, E. Wood, S. XVM: a bridge between XML data and its behavior. 13th international conference on WWW, New York, USA, 2004

[63] Maes, P. Concepts and experiments in computational reflection. In N.K. Metrowitz (Ed.), *Proceedings of OOPSLA'87*, (pp.147-156), Orlando, Florida, USA, 1987

[64] Meyer, B. Object-oriented software construction. Second edition, Prentice Hall PTR, 1997

[65] Moggi, E. An abstract view on programming languages. LFCS Report, ECS-LFCS-90-113, University of Edinburgh, 1989

[66] Molenaar, J. Mathematical modeling and dimensional analysis. In A. van den Burgh and J. Simonis (Eds.), *Topics in Engineering Mathematics. Modeling and Methods*, Kluwer Academic Publishers, 1992

[67] Mosses, P. Action semantics. Cambridge University Press. 1992

[68] Mosses, P. Pragmatics of modular SOS. 9th International Conference on Algebraic Methodology and Software Technology, (pp.21-40), 2002

[69] Mozilla Organization, http://www.mozilla.org

[70] Okajima, D. RelaxNGCC - Bridging the gap between schemas and programs. Available at: http://www.xml.com

[71] Omelayenko B. and Fensel D. A two-layered integration approach for product information in B2B E-commerce, In K. Bauknecht, S. -K. Madria, G. Pernul (Eds.), *Proceedings of the 2nd Int. Conference on Electronic Commerce and Web Technologies*, Germany, 2001

[72] OMG. MDA Guide version 1.0.1. OMG document omg/2003-06-01, 2003

[73] OMG. Common Object Request Broker Architecture

[74] OMG. Common Warehouse Metamodel (CWM) Specification. OMG document formal/03-03-02, 2003

[75] OMG. Meta Object Facility (MOF) Specification. OMG document formal/02-04-03, 2002

[76] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002

[77] OMG. OMG Unified Modeling Language Specification v. 1.4. OMG Document, 2001

[78] OMG. UML 2.0 Infrastructure Specification. OMG document ptc/03-09-15, 2003

[79] OMG. XML Metadata Interchange (XMI) Specification. OMG document formal/03-05-02, 2003

[80] OMG. Object Constraint Language (OCL). OMG Document ptc/03-10-14

[81] OMG. Software Process Engineering Metamodel Specification. OMG document formal/05-01-06

[82] Ossher, H., and Tarr, P. Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit (Ed.), *Software Architectures and Component Technology* (pp. 293-323). Kluwer Academic Publishers, 2002

[83] Patel-Schneider, P., Siméon, J. The Yin/Yang web: XML syntax and RDF semantics. 11th International WWW Conference, Hawaii, USA, 2002

[84] Patrascoiu, O. YATL: Yet Another Transformation Language. In M. van Sinderen and L. Pires (Eds.), *1st European MDA Workshop on Industrial Applications (MDA-IA)*, CTIT Technical report TR-CTIT-04-12, Enschede, the Netherlands, 2004

[85] Pigoski, T.M. Software Maintenance. In Swebok (Software Engineering Body of Knowledge), IEEE version 1.00, Chapter 6, Retrieved at http://www.swebok.org, 2004

[86] Precise UML Group (pUML). http://www.puml.org

[87] Psaila, G., Crespi-Reghizzi, S. Adding semantics to XML. In Second Workshop on Attribute Grammars and their Applications, WAGA'99, 1999

[88] Pihkala K., Honkala M. and Vuorimaa P. A browser framework for hybrid XML documents. 6th International Conference on Internet and Multimedia Systems and Applications, (pp 164-169), Kauai, Hawaii, USA, 2002

[89] QVT-Partners. Revised submission for MOF 2.0 Query/Views/Transformations RFP. 2003

[90] Reinhold, M. An XML data-binding facility for the Java platform. 1999

[91] Rosheuvel, A. XML processing based on model transformations: design, implementation, and testing of unmarshaler. MSc Thesis. University of Twente. 2003

[92] SAX Project home page: http://www.saxproject.org/

[93] Seidewitz, E. What Models Mean. *IEEE Software*, 20(5), 2003.

[94] Sierra, J., L., Fernandez-Manjon, B., Fernandez-Valmayor, A., Navarro, A. An extensible and modular processing model for document trees. Extreme Markup Languages 2002, Montreal, Canada, 2002

[95] Sommerville, I. Software Engineering. 7th Edition, Pearson & Addison Wesley, 2004

[96] Starfield, M., Smith, K.A., and Bleloch, A.L. How to model it: Problem Solving for the Computer Age. McGraw-Hill, New York, 1990.

[97] Tarr, P., Ossher, H., Sutton, S. M. Jr., and Harrison, W. *N*Degrees of Separation: Multi-Dimensional Separation of Concerns. In R. Filman, T. Elrad, S. Clarke, and M. Aksit (Eds.), *Aspect-Oriented Software Development*, (pp. 37-62), Addison-Wesley, 2004

[98] Tekinerdogan, B., Aksit, M. Adaptability in Object-Oriented Development. In M. Muhlhauser (Ed.), *Special Issues in Object Oriented Programming*, 1996

[99] Tekinerdogan, B. Synthesis-based software architecture design. PhD thesis, University of Twente, the Netherlands, 2000

[100] Varró, D., Varró, G., Pataricza, A. Designing the automatic transformation of visual languages. *Journal of Science of Computer Programming*, vol. 44, pp. 205-227, Elsevier, 2002

[101] Varró, D., Pataricza, A. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Software and System Modeling* 2(3), (pp. 187-210), Springer-Verlag, 2003

[102] Willink, E. UMLX: A graphical transformation language for MDA. In A. Rensink (Ed.), *Model Driven Architecture: Foundations and Applications 2003*, CTIT Technical Report TR-CTIT-03-27, University of Twente, the Netherlands, 2003

[103] W3C. Extensible Markup Language (XML) 1.0 (Third Edition), 2004

[104] W3C. DOM Level 1 Specification, 1998

[105] W3C. XSL Transformations (XSLT), 1999

[106] W3C. XML Schema Parrt 1: Structures, 2001

[107] W3C. Synchronized Multimedia Integration Language (SMIL 2.0), 2001

[108] W3C. XHTML+SMIL, 2002

[109] W3C. XHTML 1.0, 2002

[110] W3C. Web Services Activity. Available on http://www.w3.org/2002/ws/.

[111] W3C. MathML 2.0, 2003

[112] W3C. SVG 1.1, 2003

[113] W3C. XML Information Set (Second edition), 2004

[114] W3C. XQuery 1.0 and XPath 2.0 Data Model, 2004

[115] W3C. XQuery 1.0: An XML Query Language, 2004

[116] W3C. XML Inclusions (XInclude) Version 1.0, 2004

[117] W3C. XML Path Language (XPath) 2.0, 2005

[118] XML Schemas: Best Practice. Available at: http://www.xfront.com/BestPracticesHomepage.html

[119] XML Topic Maps (XTM) 1.0, available at http://www.topicmaps.org/xtm/index.html, 2001

[120] Zhang, Y., and Xu, B. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, vol. 39, 3, pp. 14-30. 2000

A

Grammar for the Transformation Language MISTRAL

This appendix describes the grammar of the transformation language presented in Chapter 4. The grammar is given in simple Extended Backus-Naur Form (EBNF) notation. Every rule in the grammar defines one non-terminal symbol. The defining symbol in rules is '::='. In the right-hand side of the rules the following expressions may be used:

- terminal symbols which are quoted strings (e.g. 'string');
- grouping of expressions. A group is surrounded by '(' and ')' (e.g. (Expr));
- optional expression (e.g. Expr ?);
- sequence (e.g. A B);
- alternative (e.g. A | B);
- repetition with at least one occurrence (e. g. Expr+);
- repetition with zero or more occurrences (e.g. Expr *);

The language is explained in Chapter 4 of this thesis.

A.1 General Structure

```
[1]    Transformation        ::= TransformationHeader
                                 IncludeClause ?
                                 LanguageDeclarations
                                 ModelDeclaration +
                                 TransformationRule+
                                 HelperRules ?
                                 Steps ?
```

[2] TransformationHeader ::= '**transformation**' TransformationName

[3] IncludeClause ::= '**include**' TransformationName (',' TransformationName) *

A.2 Declarations

[4] LanguageDeclarations ::= '**languages**' LanguageName (',' LanguageName) *

[5] ModelDeclaration ::= ('**source**' | '**target**')? ModelName '**instanceOf**' '(' LanguageName ')' IntentionModelName ('**default**')?

A.3 Rules

[6] TransformationRule ::= ModelElementRule | SlotRule

A.3.1 Model Element Rule

[7] ModelElementRule ::= RuleName ('**abstract**')? '**ModelElementRule**' InheritedRule ? CanceledRules ? InputParameters ? '{' RuleSource ModelElementRuleTarget SlotRules ? '}'

[8] InheritedRule ::= '**inherits**' RuleName

[9] CanceledRules ::= '**cancels**' RuleName (',' RuleName) *

[10] InputParameters ::= '**inputParameters**' '[' InputParameter (',' InputParameter)* ']'

[11] InputParameter ::= VariableName ':' TypeName

[12] ModelElementRuleTarget ::= '**target**' ('**ordered**')? '[' Action (',' Action)* ']'

[13] Action ::= Instantiation | Update | Delete

Instantiation Action

[14] Instantiation ::= (VariableName InstanceRelation)? TypeName ('=' Expression)? ConstructorSlotList? SlotList?

[15] InstanceRelation ::= ':' | '**instanceOf** (LanguageName)'

[16] ConstructorSlotList ::= '**constructor** {' SlotAssignment (',' SlotAssignment)* '}'

[17] SlotList ::= '{' SlotAssignment (',' SlotAssignment)* '}'

[18] SlotAssignment ::= SlotName ('=' Expression)?

Update Action

[19] Update ::= '**update**' SourceComponent UpdateSlotList

[20] UpdateSlotList ::= '{' UpdateSlotAssignment (',' UpdateSlotAssignment)* '}'

[21] UpdateSlotAssignment ::= UpdateInstruction? SlotName ('=' Expression)?

[22] UpdateInstruction ::= '**replace**' ('{' Expression '}')?
| '**insert**' ('{' Expression '}')?
| '**append**'
| '**delete**' ('{' Expression '}')?

Delete Action

[23] Delete ::= '**delete**' SourceComponentName DeleteActions?

[24] DeleteActions ::= '{' DeleteAction (',' DeleteAction)* '}'

[25] DeleteAction ::= ('**delete**' Expression)
| RuleInvocation

Slot Rules Section

[26] SlotRules ::= '**SlotRules** {' InPlaceSlotRule+ '}'

[27] InPlaceSlotRule ::= RuleName (SingleForm | AlternativesForm)

A.3.2 Slot Rule

[28] SlotRule ::= RuleName '**SlotRule**' OwnerRule '{' (SingleForm | AlternativesForm) '}'

[29] OwnerRule ::= '**owner**=' RuleName

Slot Rule in Single Form

[30] SingleForm ::= RuleSource SlotRuleTarget

[31] SlotRuleTarget ::= '**target** [' TargetSlot (',' TargetSlot) * ']'

[32] TargetSlot ::= (VariableName '.')? SlotName '=' Expression

Slot Rule in Form with Alternatives

[33] AlternativesForm ::= SlotRuleTarget

('**alt** {' RuleSource? SlotRuleTarget '}') +

A.3.3 Rule Source

[34]	RuleSource	::= '**source** [' SourceComponent (',' SourceComponent) * Condition? ']'
[35]	SourceComponent	::= (Identifier | Variable) LinkStatement?
[36]	LinkStatement	::= '**link-to** {' VariableName (',' VariableName)* '}'
[37]	Variable	::= VariableName InstanceRelation Expression ('=' Expression) ?
[38]	Condition	::= '**condition** {' Expression '}'

A.4 Helper Rules

[39]	HelperRules	::= '**helperRules**' TransformationRule+

A.5 Transformation Steps

[40]	Steps	::= Step+
[41]	Step	::= '**Step**' StepName '{' (TransformationRule | RuleName) + '}'

A.6 Names and Expressions

Some non-terminals used in the rules are left undefined. They are explained below.

- *Names*: Non-terminals TransformationName, LanguageName, ModelName, RuleName, VariableName, TypeName, SlotName, StepName, and Identifier are not defined. Generally, they are identifiers that start with letter and are followed by any combination of letters and digits. We assume that their definition is trivial. Identifier and TypeName follow the syntax described in Chapter 4, section 4.3.4.
- *Expressions*: Non-terminal Expression is an OCL expression [80]. OCL expressions are enhanced with the capability to follow transitive relations defined in a language. The keyword **follow** is defined for this purpose (see Chapter 4, section 4.4.3).
- *Rule invocation*: Non-terminal RuleInvocation used in rule 25 is according to the syntax explained in Chapter 4, section 4.4.7.

B

Abstract Syntax for the Transformation Language MISTRAL

This appendix presents the abstract syntax definition of the transformation language MISTRAL. The abstract syntax is expressed as MOF model presented in a series of diagrams in the following sections.

B.1 General Structure

Abstract syntax definition is organized in 5 packages shown in *Figure B.1*.

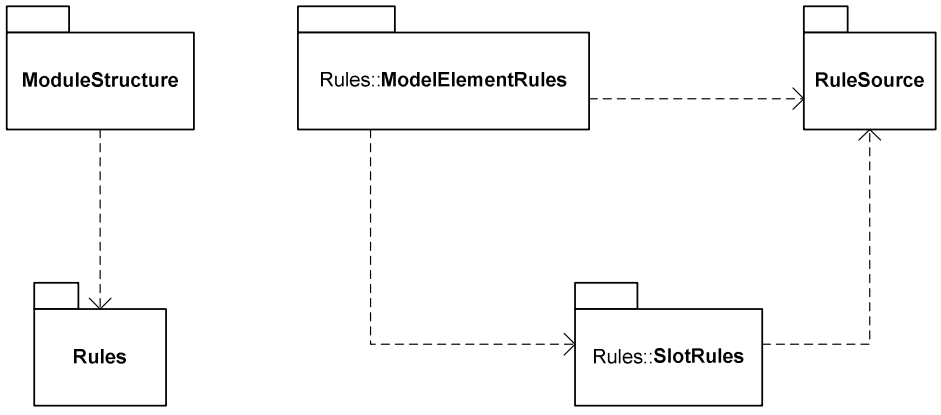


Figure B.1 Package structure of the abstract syntax of the transformation language MISTRAL

Package *ModuleStructure* contains constructs for defining the structure of transformation modules. It depends on package *Rules*. Package *Rules* has two sub-packages: *ModelElementRules* and *SlotRules*. The first sub-package defines the structure of model element rules and the second one defines the structure of slot rules. These two packages use constructs from package *RuleSource*, which defines the stricture of rule source expressions.

B.2 Structure of Transformation Modules

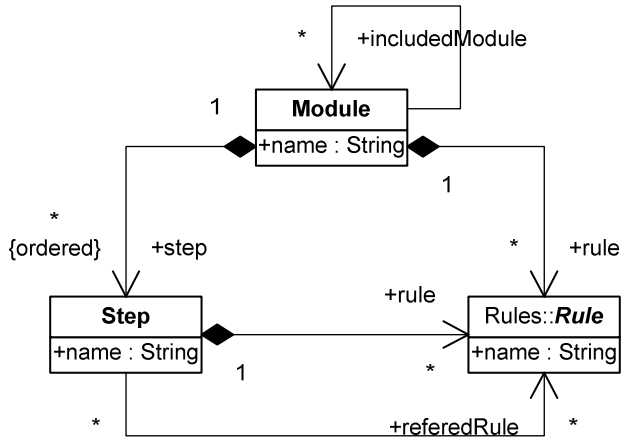


Figure B.2 The structure of transformation modules

Transformation modules have a name and a number of ordered steps (*Figure B.2*). Modules also contain transformation rules. Rules may be defined in the module and may be included from other modules.

Every step may define transformation rules and may refer to rules defined in the module. In this way single rule may be used in more than one step.

B.3 Transformation Rules

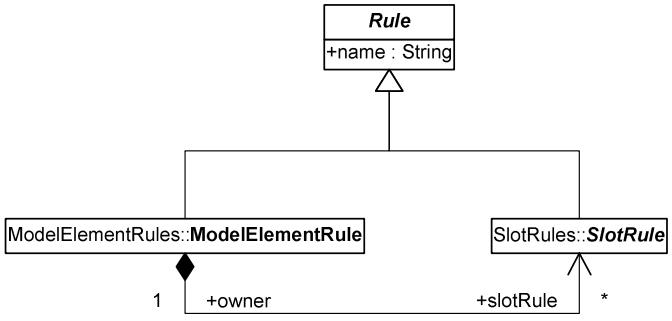


Figure B.3 Types of transformation rules

Package *Rules* defines an abstract class *Rule* and two specializations corresponding to model element rules and slot rules (*Figure B.3*). Model element rules have a set of slot rules. The structure of the two types of transformation rules is defined in the sub-packages *ModelElementRules* and *SlotRules*.

B.3.1 Model Element Rules

Figure B.4 shows the structure of model element rules. A model element rule may inherit other rule and may cancel more than one model element rule. Rules may have zero or more input parameters.

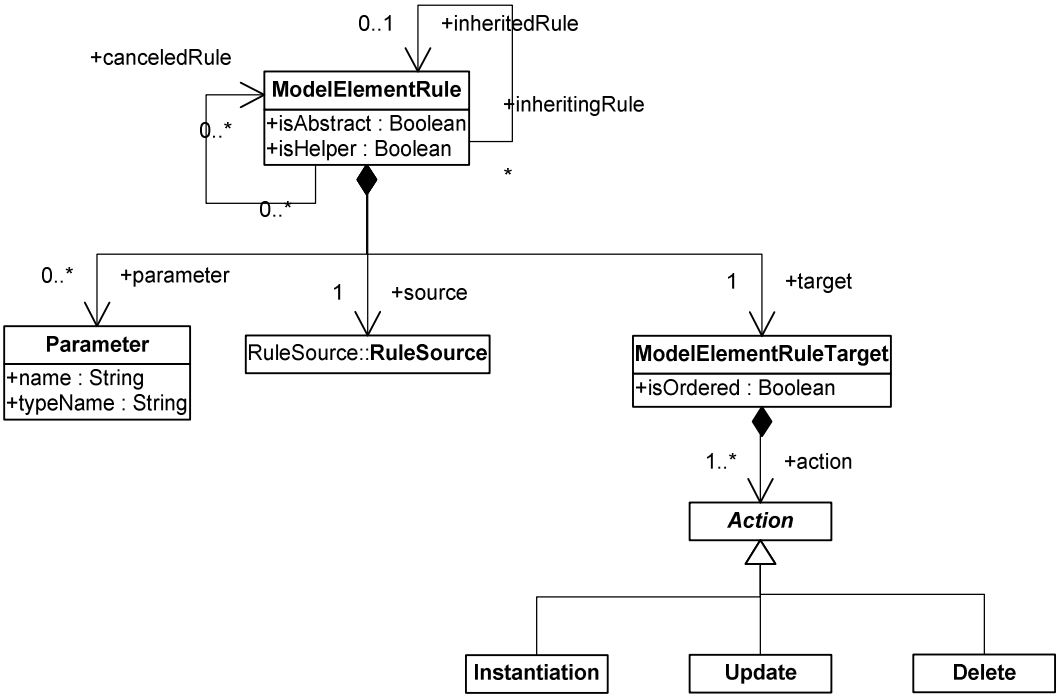


Figure B.4 Structure of model element rules

In Chapter 4 we described that model element rules execute actions for every match of its source expression. Model element rules organize actions in the rule target. Actions may be ordered. Three types of actions are supported: *Instantiation*, *Update*, and *Delete*. The structure of these actions is shown in *Figure B.5*.

Instantiation action instantiates a meta-construct from the target intension. The meta-construct is determined after an evaluation of an expression. The instantiation mechanism is specified as the value of the attribute *instanceRelationName*. Instantiation may be associated to an initialization expression. Furthermore, instantiations have two lists of slot assignments: one for constructor slots and one for non-constructor slots (or just slots).

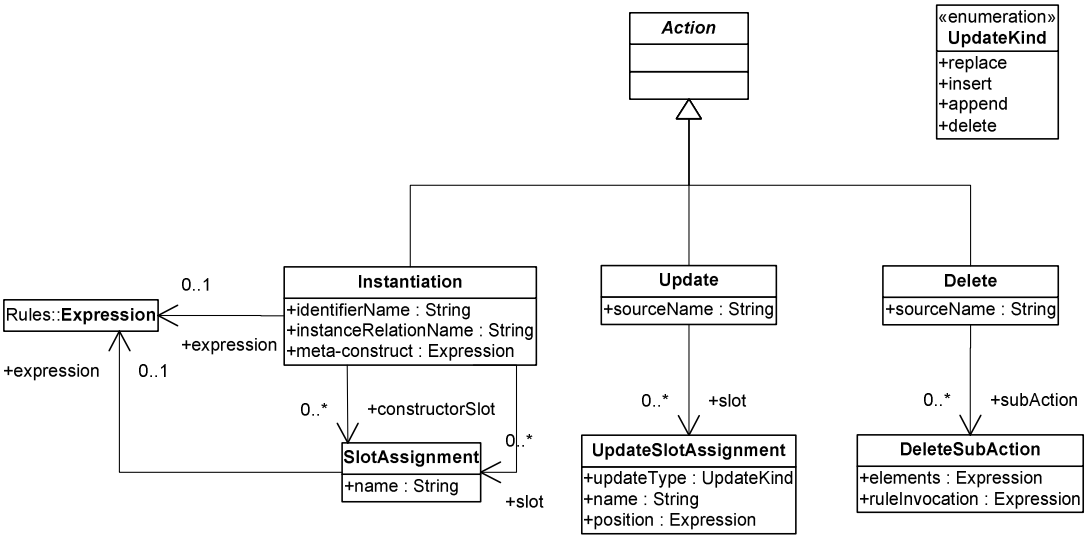


Figure B.5 Structure of actions used in model element rules

Update action refers by name to a source component (attribute *sourceName*). Update actions change the slot values of the selected source element. The change is indicated by the value of *updateType* attribute. Possible values are defined as attributes in the enumeration class *UpdateKind*.

Delete action also refers by name to a source component (attribute *sourceName*). This action deletes the selected source element and may execute delete sub-actions represented by class *DeleteSubAction*.

B.3.2 Slot Rules

Figure B.6 shows the structure of slot rules. Two forms of slot rules are defined as classes that specialize abstract class *SlotRule*: *SingleForm* and *AlternativesForm*. Both types of slot rules may specify values for more than one slot. Slots are represented by class *TargetSlot.* The value of a slot is calculated by an expression. This expression may refer to elements selected from the source model. Selection is done by a rule source expression. Slot rules in single form have only one source expression. Slot rules in form with alternatives may have more than one source expression associated to class *Alternative*.

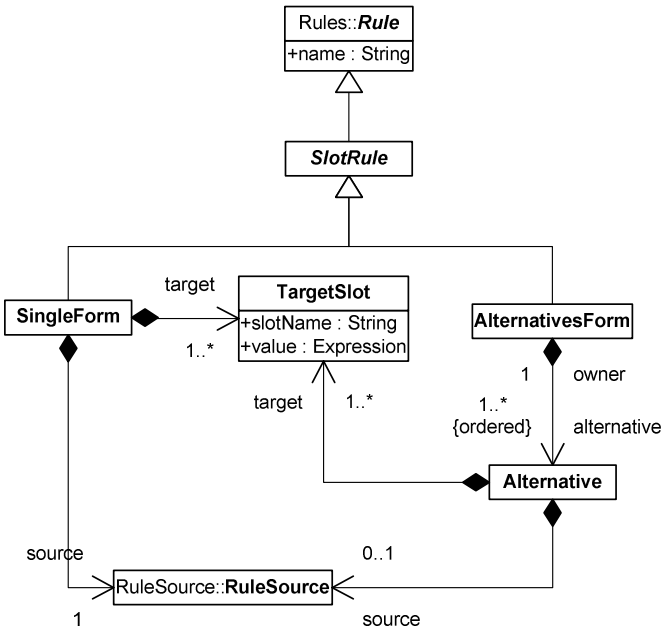


Figure B.6 Structure of slot rules

B.4 Rule Source

The structure of rule source expressions is shown in *Figure B.7*.

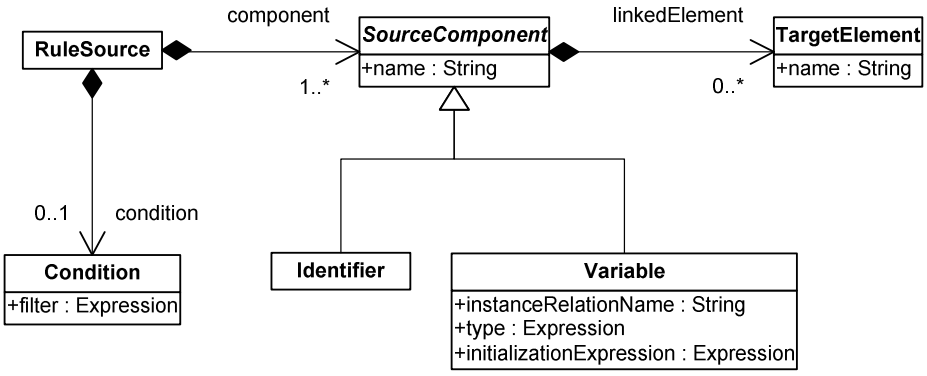


Figure B.7 Structure of rule source expressions

Index

Titles in the IPA Dissertation Series

J.O. Blanco, *The State Operator in Process Algebra*, Faculty of Mathematics and Computing Science, TUE, 1996-01

A.M. Geerling, *Transformational Development of Data-Parallel Algorithms*, Faculty of Mathematics and Computer Science, KUN, 1996-02

P.M. Achten, *Interactive Functional Programs: Models, Methods, and Implementation*, Faculty of Mathematics and Computer Science, KUN, 1996-03

M.G.A. Verhoeven, *Parallel Local Search*, Faculty of Mathematics and Computing Science, TUE, 1996-04

M.H.G.K. Kesseler, *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*, Faculty of Mathematics and Computer Science, KUN, 1996-05

D. Alstein, *Distributed Algorithms for Hard Real-Time Systems*, Faculty of Mathematics and Computing Science, TUE, 1996-06

J.H. Hoepman, *Communication, Synchronization, and Fault-Tolerance*, Faculty of Mathematics and Computer Science, UvA, 1996-07

H. Doornbos, *Reductivity Arguments and Program Construction*, Faculty of Mathematics and Computing Science, TUE, 1996-08

D. Turi, *Functorial Operational Semantics and its Denotational Dual*, Faculty of Mathematics and Computer Science, VUA, 1996-09

A.M.G. Peeters, *Single-Rail Handshake Circuits*, Faculty of Mathematics and Computing Science, TUE, 1996-10

N.W.A. Arends, *A Systems Engineering Specification Formalism*, Faculty of Mechanical Engineering, TUE, 1996-11

P. Severi de Santiago, *Normalisation in Lambda Calculus and its Relation to Type Inference*, Faculty of Mathematics and Computing Science, TUE, 1996-12

D.R. Dams, *Abstract Interpretation and Partition Refinement for Model Checking*, Faculty of Mathematics and Computing Science, TUE, 1996-13

M.M. Bonsangue, *Topological Dualities in Semantics*, Faculty of Mathematics and Computer Science, VUA, 1996-14

B.L.E. de Fluiter, *Algorithms for Graphs of Small Treewidth*, Faculty of Mathematics and Computer Science, UU, 1997-01

W.T.M. Kars, *Process-algebraic Transformations in Context*, Faculty of Computer Science, UT, 1997-02

P.F. Hoogendijk, *A Generic Theory of Data Types*, Faculty of Mathematics and Computing Science, TUE, 1997-03

T.D.L. Laan, *The Evolution of Type Theory in Logic and Mathematics*, Faculty of Mathematics and Computing Science, TUE, 1997-04

C.J. Bloo, *Preservation of Termination for Explicit Substitution*, Faculty of Mathematics and Computing Science, TUE, 1997-05

J.J. Vereijken, *Discrete-Time Process Algebra*, Faculty of Mathematics and Computing Science, TUE, 1997-06

F.A.M. van den Beuken, *A Functional Approach to Syntax and Typing*, Faculty of Mathematics and Informatics, KUN, 1997-07

A.W. Heerink, *Ins and Outs in Refusal Testing*, Faculty of Computer Science, UT, 1998-01

G. Naumoski and W. Alberts, *A Discrete-Event Simulator for Systems Engineering*,

Faculty of Mechanical Engineering, TUE, 1998-02

J. Verriet, *Scheduling with Communication for Multiprocessor Computation*, Faculty of Mathematics and Computer Science, UU, 1998-03

J.S.H. van Gageldonk, *An Asynchronous Low-Power 80C51 Microcontroller*, Faculty of Mathematics and Computing Science, TUE, 1998-04

A.A. Basten, *In Terms of Nets: System Design with Petri Nets and Process Algebra*, Faculty of Mathematics and Computing Science, TUE, 1998-05

E. Voermans, *Inductive Datatypes with Laws and Subtyping -- A Relational Model*, Faculty of Mathematics and Computing Science, TUE, 1999-01

H. ter Doest, *Towards Probabilistic Unification-based Parsing*, Faculty of Computer Science, UT, 1999-02

J.P.L. Segers, *Algorithms for the Simulation of Surface Processes*, Faculty of Mathematics and Computing Science, TUE, 1999-03

C.H.M. van Kemenade, *Recombinative Evolutionary Search*, Faculty of Mathematics and Natural Sciences, UL, 1999-04

E.I. Barakova, *Learning Reliability: a Study on Indecisiveness in Sample Selection*, Faculty of Mathematics and Natural Sciences, RUG, 1999-05

M.P. Bodlaender, *Scheduler Optimization in Real-Time Distributed Databases*, Faculty of Mathematics and Computing Science, TUE, 1999-06

M.A. Reniers, *Message Sequence Chart: Syntax and Semantics*, Faculty of Mathematics and Computing Science, TUE, 1999-07

J.P. Warners, *Nonlinear approaches to satisfiability problems*, Faculty of Mathematics and Computing Science, TUE, 1999-08

J.M.T. Romijn, *Analysing Industrial Protocols with Formal Methods*, Faculty of Computer Science, UT, 1999-09

P.R. D'Argenio, *Algebras and Automata for Timed and Stochastic Systems*, Faculty of Computer Science, UT, 1999-10

G. Fabian, *A Language and Simulator for Hybrid Systems*, Faculty of Mechanical Engineering, TUE, 1999-11

J. Zwanenburg, *Object-Oriented Concepts and Proof Rules*, Faculty of Mathematics and Computing Science, TUE, 1999-12

R.S. Venema, *Aspects of an Integrated Neural Prediction System*, Faculty of Mathematics and Natural Sciences, RUG, 1999-13

J. Saraiva, *A Purely Functional Implementation of Attribute Grammars*, Faculty of Mathematics and Computer Science, UU, 1999-14

R. Schiefer, *Viper, A Visualisation Tool for Parallel Program Construction*, Faculty of Mathematics and Computing Science, TUE, 1999-15

K.M.M. de Leeuw, *Cryptology and Statecraft in the Dutch Republic*, Faculty of Mathematics and Computer Science, UvA, 2000-01

T.E.J. Vos, *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*, Faculty of Mathematics and Computer Science, UU, 2000-02

W. Mallon, *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*, Faculty of Mathematics and Natural Sciences, RUG, 2000-03

W.O.D. Griffioen, *Studies in Computer Aided Verification of Protocols*, Faculty of Science, KUN, 2000-04

P.H.F.M. Verhoeven, *The Design of the MathSpad Editor*, Faculty of Mathematics and Computing Science, TUE, 2000-05

J. Fey, *Design of a Fruit Juice Blending and Packaging Plant*, Faculty of Mechanical Engineering, TUE, 2000-06

M. Franssen, *Cocktail: A Tool for Deriving Correct Programs*, Faculty of Mathematics and Computing Science, TUE, 2000-07

P.A. Olivier, *A Framework for Debugging Heterogeneous Applications*, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2000-08

E. Saaman, *Another Formal Specification Language*, Faculty of Mathematics and Natural Sciences, RUG, 2000-10

M. Jelasity, *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*, Faculty of Mathematics and Natural Sciences, UL, 2001-01

R. Ahn, *Agents, Objects and Events a computational approach to knowledge, observation and communication*, Faculty of Mathematics and Computing Science, TU/e, 2001-02

M. Huisman, *Reasoning about Java programs in higher order logic using PVS and Isabelle*, Faculty of Science, KUN, 2001-03

I.M.M.J. Reymen, *Improving Design Processes through Structured Reflection*, Faculty of Mathematics and Computing Science, TU/e, 2001-04

S.C.C. Blom, *Term Graph Rewriting: syntax and semantics*, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2001-05

R. van Liere, *Studies in Interactive Visualization*, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2001-06

A.G. Engels, *Languages for Analysis and Testing of Event Sequences*, Faculty of Mathematics and Computing Science, TU/e, 2001-07

J. Hage, *Structural Aspects of Switching Classes*, Faculty of Mathematics and Natural Sciences, UL, 2001-08

M.H. Lamers, *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*, Faculty of Mathematics and Natural Sciences, UL, 2001-09

T.C. Ruys, *Towards Effective Model Checking*, Faculty of Computer Science, UT, 2001-10

D. Chkliaev, *Mechanical verification of concurrency control and recovery protocols*, Faculty of Mathematics and Computing Science, TU/e, 2001-11

M.D. Oostdijk, *Generation and presentation of formal mathematical documents*, Faculty of Mathematics and Computing Science, TU/e, 2001-12

A.T. Hofkamp, *Reactive machine control: A simulation approach using* χ, Faculty of Mechanical Engineering, TU/e, 2001-13

D. Bošnački, *Enhancing state space reduction techniques for model checking*, Faculty of Mathematics and Computing Science, TU/e, 2001-14

M.C. van Wezel, *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*, Faculty of Mathematics and Natural Sciences, UL, 2002-01

V. Bos and J.J.T. Kleijn, *Formal Specification and Analysis of Industrial Systems*, Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e, 2002-02

T. Kuipers, *Techniques for Understanding Legacy Software Systems*, Faculty of Natural Sciences, Mathematics and Computer Science, UvA, 2002-03

S.P. Luttik, *Choice Quantification in Process Algebra*, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-04

R.J. Willemen, *School Timetable Construction: Algorithms and Complexity*, Faculty of Mathematics and Computer Science, TU/e, 2002-05

M.I.A. Stoelinga, *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*, Faculty of Science, Mathematics and Computer Science, KUN, 2002-06

N. van Vugt, *Models of Molecular Computing*, Faculty of Mathematics and Natural Sciences, UL, 2002-07

A. Fehnker, *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*, Faculty of Science, Mathematics and Computer Science, KUN, 2002-08

R. van Stee, *On-line Scheduling and Bin Packing*, Faculty of Mathematics and Natural Sciences, UL, 2002-09

D. Tauritz, *Adaptive Information Filtering: Concepts and Algorithms*, Faculty of Mathematics and Natural Sciences, UL, 2002-10

M.B. van der Zwaag, *Models and Logics for Process Algebra*, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-11

J.I. den Hartog, *Probabilistic Extensions of Semantical Models*, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2002-12

L. Moonen, *Exploring Software Systems*, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2002-13

J.I. van Hemert, *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*, Faculty of Mathematics and Natural Sciences, UL, 2002-14

S. Andova, *Probabilistic Process Algebra*, Faculty of Mathematics and Computer Science, TU/e, 2002-15

Y.S. Usenko, *Linearization in μCRL*, Faculty of Mathematics and Computer Science, TU/e, 2002-16

J.J.D. Aerts, *Random Redundant Storage for Video on Demand*, Faculty of Mathematics and Computer Science, TU/e, 2003-01

M. de Jonge, *To Reuse or To Be Reused: Techniques for component composition and construction*, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2003-02

J.M.W. Visser, *Generic Traversal over Typed Source Code Representations*, Faculty of Natural Sciences, Mathematics, and Computer Science, UvA, 2003-03

S.M. Bohte, *Spiking Neural Networks*, Faculty of Mathematics and Natural Sciences, UL, 2003-04

T.A.C. Willemse, *Semantics and Verification in Process Algebras with Data and Timing*, Faculty of Mathematics and Computer Science, TU/e, 2003-05

S.V. Nedea, *Analysis and Simulations of Catalytic Reactions*, Faculty of Mathematics and Computer Science, TU/e, 2003-06

M.E.M. Lijding, *Real-time Scheduling of Tertiary Storage*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2003-07

H.P. Benz, *Casual Multimedia Process Annotation - CoMPAs*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2003-08

D. Distefano, *On Model Checking the Dynamics of Object-based Software: a Foundational Approach*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2003-09

M.H. ter Beek, *Team Automata - A Formal Approach to the Modeling of Collaboration Between System Components*, Faculty of Mathematics and Natural Sciences, UL, 2003-10

D.J.P. Leijen, *The λ Abroad - A Functional Approach to Software Components*, Faculty of Mathematics and Computer Science, UU, 2003-11

W.P.A.J. Michiels, *Performance Ratios for the Differencing Method*, Faculty of Mathematics and Computer Science, TU/e, 2004-01

G.I. Jojgov, *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*, Faculty of Mathematics and Computer Science, TU/e, 2004-02

P. Frisco, *Theory of Molecular Computing - Splicing and Membrane systems*, Faculty of Mathematics and Natural Sciences, UL, 2004-03

S. Maneth, *Models of Tree Translation*, Faculty of Mathematics and Natural Sciences, UL, 2004-04

Y. Qian, *Data Synchronization and Browsing for Home Environments*, Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e, 2004-05

F. Bartels, *On Generalised Coinduction and Probabilistic Specification Formats*, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-06

L. Cruz-Filipe, *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*, Faculty of Science, Mathematics and Computer Science, KUN, 2004-07

E.H. Gerding, *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*, Faculty of Technology Management, TU/e, 2004-08

N. Goga, *Control and Selection Techniques for the Automated Testing of Reactive Systems*, Faculty of Mathematics and Computer Science, TU/e, 2004-09

M. Niqui, *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*, Faculty of Science, Mathematics and Computer Science, RU, 2004-10

A. Löh, *Exploring Generic Haskell*, Faculty of Mathematics and Computer Science, UU, 2004-11

I.C.M. Flinsenberg, *Route Planning Algorithms for Car Navigation*, Faculty of Mathematics and Computer Science, TU/e, 2004-12

R.J. Bril, *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*, Faculty of Mathematics and Computer Science, TU/e, 2004-13

J. Pang, *Formal Verification of Distributed Systems*, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-14

F. Alkemade, *Evolutionary Agent-Based Economics*, Faculty of Technology Management, TU/e, 2004-15

E.O. Dijk, *Indoor Ultrasonic Position Estimation Using a Single Base Station*, Faculty of Mathematics and Computer Science, TU/e, 2004-16

S.M. Orzan, *On Distributed Verification and Verified Distribution*, Faculty of Sciences, Division of Mathematics and Computer Science, VUA, 2004-17

M.M. Schrage, *Proxima - A Presentation-oriented Editor for Structured Documents*, Faculty of Mathematics and Computer Science, UU, 2004-18

E. Eskenazi and A. Fyukov, *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*, Faculty of Mathematics and Computer Science, TU/e, 2004-19

P.J.L. Cuijpers, *Hybrid Process Algebra*, Faculty of Mathematics and Computer Science, TU/e, 2004-20

N.J.M. van den Nieuwelaar, *Supervisory Machine Control by Predictive-Reactive*

Scheduling, Faculty of Mechanical Engineering, TU/e, 2004-21

E. Abraham, *An Assertional Proof System for Multithreaded Java-Theory and Tool Support*, Faculty of Mathematics and Natural Sciences, UL, 2005-01

R. Ruimerman, *Modeling and Remodeling in Bone Tissue*, Faculty of Biomedical Engineering, TU/e, 2005-02

C.N. Chong, *Experiments in Rights Control - Expression and Enforcement*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2005-03

H. Gao, *Design and Verification of Lock-free Parallel Algorithms*, Faculty of Mathematics and Computing Sciences, RUG, 2005-04

H.M.A. van Beek, *Specification and Analysis of Internet Applications*, Faculty of Mathematics and Computer Science, TU/e, 2005-05

M.T. Ionita, *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*, Faculty of Mathematics and Computing Sciences, TU/e, 2005-06

G. Lenzini, *Integration of Analysis Techniques in Security and Fault-Tolerance*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2005-07

I. Kurtev, *Adaptability of Model Transformations*, Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2005-08