University of Latvia

ELĪNA KALNIŅA

MODEL TRANSFORMATION DEVELOPMENT USING MOLA MAPPINGS AND TEMPLATE MOLA

Thesis for the PhD Degree
at the University of Latvia

Field: Computer Science
Section: Programming Languages and Systems

Scientific Advisor:
Prof., Dr. Habil. Sc. Comp.
AUDRIS KALNINS

Riga – 2011

IEGULDĪJUMS TAVĀ NĀKOTNĒ

This work has been supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

Scientific Advisor:

Professor, Dr. Sc. Comp. Audris Kalniņš
Latvijas Universitāte

Referees:

Professor, Dr. Sc. Comp. Guntis Bārzdiņš
University of Latvia

Professor, Dr. Sc. Ing. Oksana Ņikiforova
Riga Technical University

Professor, Dr. Olegas Vasilecas
Vilnius Gediminas Technical University (Vilnius, Lithuania)

The defence of the thesis will take place in an open session of the Council of Promotion in Computer Science of the University of Latvia, in the Institute of Mathematics and Computer Science of the University of Latvia (Room 413, Raina Boulevard 29, Riga, Latvia) on March 7, 2012 at 4 PM.

The thesis and its summary are available at the library of the University of Latvia (Kalpaka Boulevard 4, Riga, Latvia).

ABSTRACT

Model transformation development for three specific domains: Model-Driven Software Development (MDSD), DSL tool development and transformation synthesis has been studied in the thesis. It is concluded that transformation development in domain-specific transformation languages is more straightforward and faster compared to traditional transformation languages. A domain-specific model transformation language has been developed for each studied domain. Two of them are based on mappings. In both cases it was concluded that mappings better fit for typical tasks and transformations better fit for non-standard tasks. Therefore a close integration between mappings and transformations is required.

The research results have been published in 15 papers (6 of them have been included in SCOPUS).

Keywords

Model transformations, Domain-Specific Languages (DSL), Model-Driven Software Development (MDSD), DSL tool development, Higher-Order Transformations (HOT)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

This work has been partially supported by the European Social Fund within the project «Support for Doctoral Studies at University of Latvia».

The author of the thesis would like to thank:

- supervisor prof. Audris Kalnins;
- current and former members of MOLA team: Edgars Celms, Agris Sostaks, Janis Iraids, Oskars Vilitis;
- ReDSeeDS project partners;
- colleagues in LUMII Research Laboratory of System Modeling and Software Technologies;
- prof. Rusins Martins Freivalds;
- Maija Treilona;
- Lāsma Začesta;
- Valdis Kalniņš;
- Lolita Nahodkina;
- Maiga Reinharde;
- family;
- all others who have helped me in any way.

INTRODUCTION

The present PhD thesis has been worked on from 2007 to 2011 in the Institute of Mathematics and Computer Science (UL IMCS), and the Faculty of Computing established as an independent unit on the basis of the Faculty of Physics and Mathematics, University of Latvia. The thesis supervisor is professor Audris Kalnins. The thesis elaborates further the UL IMCS DSL (*Domain-Specific Language*) tool development and language design traditions that started already in the year 1986.

Relevance of the Thesis:

Lately *Model-Driven Software Development* (MDSD) is gaining popularity. The idea of elaborating all software development steps on models defined in specialised modelling languages lies at the basis of the approach. Models, defined at higher abstraction levels, are ever more detailed in each step of Model-Driven Software Development. Model transformations are used to automate transitions from one model to another. Use of model transformations allows using models as a direct part of the software development process instead of using them only as documentation.

The origin of MDSD was the *Model-Driven Architecture* (MDA) [111] initiative by *Object Management Group (OMG).* The first document about the MDA was published in 2000 [116]. In 2002 OMG concluded that model transformation languages are required [119], to easily describe the required model transformations. Most of the modelling languages are defined by using the means of metamodelling; therefore model transformations were built to transform the models defined according to metamodels. Metamodels were defined by using the metamodelling standard MOF (*Meta Object Facilities*) [120].

OMG activities led to the creation of a new model transformation standard MOF-QVT (*MOF Queries/Views/Transformations*) [128]. Moreover, many new model transformation languages were developed, e.g., *ATL* [63], *GReAT* [7], *GrGen* [48], *Epsilon* [92] and the model transformation language *MOLA* [76] that was developed in UL IMCS. This was also a new application area for graph transformation languages, e.g., *PROGRES* [144], *AGG* [163], *VIATRA* [31] and also *Fujaba* [43], previously used in a narrower context. The variety of model transformation languages could be explained by two reasons: lack of complete MOF-QVT implementation and different model transformation application domains. In different software development areas there are different requirements for a model transformation language.

Today model transformations are a serious software component in large software development projects. Transformation development requires a considerable amount of resources. Transformations should be projected, tested, maintained, etc. Currently the transformation development is rather chaotic and every developer develops transformations according to one's own wishes. It could be explained by the poor experience in adaption of the classic software development steps (testing, etc.) to transformations. Consequently, studying of the transformation development is a popular research direction.

In the same way there are attempts to adapt the classic software development methods to the model transformation development. One of such methods is to build a *Domain-Specific Language* (DSL) to be applied to the software development in a specific class of tasks. The thesis is devoted to researching domain-specific transformation languages. Usage of domain-specific transformation languages could improve transformation development, the same as the use of the *domain-specific languages* helps to reduce the software development time and costs. However, it should be noted that the use of *domain-specific languages* is cost-effective only in case of developing multiple similar solutions.

Aim of the Research:

The aim of the research is to investigate the ways of defining transformations for classes of similar tasks, requiring development of many transformations of the same type.

- Explore transformation development for *Model-Driven Software Development*.
- Explore the nature of the transformations for DSL tool development.
- Explore the opportunities of defining *Model-Driven Software Development* and tool building transformations in specialised languages (higher abstraction level) and using mappings.
- Explore the definition possibilities of transformation generating transformation. Develop a higher-order transformation language which is specialized for transformation synthesis.

Main Results of the Thesis:

- Developed and implemented the transformation supported path from the requirements to the code. The research has been carried out as a part of the ReDSeeDS project. Transformations for Model-Driven Software Development have been analyzed. It is concluded that some of the transformations could be defined more effectively by using a specialised (higher abstraction level) language.
- Developed the first version of the *MOLA 2 tool* within the *METAclipse* framework. A conclusion has been drawn that part of the transformations are very simple and uniform and it would be more convenient to define them in a mapping language. Likewise, it is concluded that it would be impossible to define everything by using a mapping language; therefore, integration between the mappings and transformations is required.
- Developed the mapping language MALA4MDSD, which is especially adapted for transformation development in Model-Driven Software Development.
- Outlined the mapping language for DSL tool development.
- Developed the language Template MOLA, which is a domain-specific language for transformation synthesis.
- Analysis of three particular problem areas leads to the conclusion that the transformation development in a domain-specific language is possible at a higher level of abstraction. Thus, transformations can be developed faster. If the transformation is defined by a higher level of abstraction and the use of mapping, then less-skilled users can define the transformations as well.

Scientific and Practical Significance of the Thesis:

Model transformation development for three specific domains, namely, *Model-Driven Software Development* (MDSD), *Domain-Specific Language* (DSL) tool development and transformation synthesis has been studied in the thesis.

One of the areas under research in the present thesis is a specification of transformations for Model-Driven Software Development. While working on the ReDSeeDS project the author of the PhD thesis developed two transformation sets for

Model-Driven Software Development. This type of transformations typically contains a transformation from UML to UML and for facilitating the given transformation development, the mapping language MALA4MDSD is offered in the PhD thesis. The language MALA4MDSD is also of practical importance, since it makes it significantly easier to develop transformations for Model-Driven Software Development. This could encourage a wider use of model-driven development methods in industry, as transformations could be defined by less experienced users - those who are experts in the transformed problem area, but do not know anything about metamodelling. In addition, the transformation development would become faster.

The second researched area is the model transformations for DSL tool development. It was concluded that the best way for defining a tool for graphical DSL is by combining mappings with transformations. Using of mappings allows a less skilled user to configure tools as well; the tool development would become significantly faster. However, using mappings makes it impossible to provide convenient instruments for all possible cases of non-standard treatment; therefore there is a need for a way of processing non-standard cases in a transformation language. Many of the existing DSL tool development platforms offer processing the non-standard cases in a programming language, but a transformation language for this task would be more appropriate, because the data are model-driven, and transformation languages are adapted for processing this type of data.

The third problem area brought an observation that a domain-specific language is more convenient for defining transformations. However, here is chosen a different type of language that does not use mappings. This is a specific area which describes transformation synthesis. The task is very specific, and the existing means are very inadequate and are difficult to use, therefore the domain-specific language has been created. The language Template MOLA is a higher-order transformation language, specifically adapted to the tasks of transformation synthesis. It is the first language in the world of such a type. Later an extension, specifically for transformation synthesis, has been developed for the language ATL [182]. It should be noted that comparing to the language MOLA, ATL is a textual language, therefore the synthesis of ATL is an easier task. Nevertheless, the basic idea used in the ATL extension is the same as in the Template MOLA - using fragments of concrete syntax.

The language Template MOLA helps to solve a very important issue in the model transformation world, namely, metamodel independent transformation development. Since almost all transformations are linked to metamodels, building of a library of transformations and reuse of transformations is still an open problem.

The research results of the thesis suggest that model transformations is a sufficiently vast area, making it possible to choose more limited problem areas – domain-specific transformations - and domain-specific transformation languages have to be created for these areas. The research focused on studying mapping languages as it is the most user-friendly way of defining transformations. Nevertheless, the existing mapping languages are not quite appropriate as usually they can process only very simple cases. Therefore, the research offers a new idea for defining transformations – use of domain-specific mapping languages instead of a universal mapping language.

Publications of the Research Results and Presentations in Scientific Conferences:

The main results of the PhD thesis are presented in 10 publications; each containing a significant (70-80%) contribution of the author of the present thesis:

- *"DSL Tool Development with Transformations and Static Mappings"* [67] The publication outlines the role of mapping in the DSL tool development.
- *"DSL Tool Development with Transformations and Static Mappings"* [68] The publication discusses the use of the mapping language in the DSL tool development.
- *"Graphical Template Language for Transformation Synthesis"* [69] The publication describes the language Template MOLA.
- *"Transformation Synthesis Language – Template MOLA"* [71] The publication describes in detail the language Template MOLA.
- *"Generation Mechanisms in Graphical Template Language"* [70] The publication discusses a merge mechanism in the language Template MOLA.
- *"From Requirements to Code in a Model Driven Way"* [79] The publication outlines transformations used for the model-driven development process realization within the ReDSeeDS project.

- *"A Model-Driven Path from Requirements to Code"* [80] The publication describes in detail the development of transformations for Model-Driven Software Development within the ReDSeeDS project.
- *"Model Migration with MOLA"* [72] The publication describes a transformation design in the language MOLA for transforming UML 1.X activity diagrams to UML 2.3 activity diagrams.
- *"Hello World with MOLA - A Solution to the TTC 2011 Instructive Case"* [74] (accepted for publication). The publication discusses solutions of simple transformation tasks in the language MOLA.
- *"Tree Based Domain-Specific Mapping Languages"* [73] (accepted for publication). The publication describes the mapping language MALA4MDSD and the methodology of constructing a domain-specific mapping language.

The author of the thesis has participated in the preparation of 5 more publications with the contribution of 5-25%.

- *"Building Tools by Model Transformations in Eclipse"* [86] The publication outlines the principles of the METAclipse DSL tool development framework and its use in the MOLA 2 tool development.
- *"Behaviour Modelling Notation for Information System Design"* [78] The publication describes the experience, gained while working with the UML sequence diagrams within the ReDSeeDS project.
- *"Comprehensive System for Systematic Case-Driven Software Reuse"* [153] The publication describes a platform developed within the ReDSeeDS project and highlights the role of transformations in this platform.
- *"Domain-driven Reuse of Software Design Models"* [82] The publication discusses software reuse facilitatation by the transformations, developed within the ReDSeeDS project.
- *"Solving the TTC 2011 Reengineering Case with MOLA and Higher-Order Transformations"* [155] The publication discusses the transformation development for transforming the Java code (coded with a model) to a state chart model.

The author has reported on the results of the work in a number of scientific conferences:

- "*Graphical Template Language for Transformation Synthesis"* International conference SLE (Software Language Engineering), 2009; Denver, USA
- "*From Requirements to Code in a Model Driven Way*" MDA (Model-Driven Architecture: Foundations, Practices and Implications) workshop of ADBIS (Advances in Databases and Information Systems), 2009; Riga, Latvia
- *"DSL Tool Development with Transformations and Static Mappings*" Doctoral Symposium of MODELS (International Conference on Model-Driven Engineering Languages and Systems), 2008; Toulouse, France
- *"Domēn-specifiskas attēlojumu valodas"* 69th Scientific Conference of the University of Latvia, Information Technology Section, 2011; Riga, Latvia.
- *"Valoda Template MOLA un tās realizācija"* 68th Scientific Conference of the University of Latvia, Information Technology Section, 2010; Riga, Latvia.
- *"MDA transformācijas ReDSeeDS projekta kontekstā"* 67th Scientific Conference of the University of Latvia, Information Technology Section, 2009; Riga, Latvia.
- *"Transformāciju un attēlojumu kombinēšanas lietojumi rīku būvē"* 67th Scientific Conference of the University of Latvia, Information Technology Section, 2009; Riga, Latvia.
- *"MOLA-2 rīka būve, izmantojot METAclipse platformu"*, 66th Scientific Conference of the University of Latvia, Information Technology Section, 2008; Riga, Latvia.
- The developed MOLA tool has been demonstrated at the international conference ECMDA-FA Tool Demonstration Section (see [85]).

Structure of the Thesis:

The thesis is a logical conclusion of the previously described investigational and practical work, thus forming a complete research. The structure of the thesis is as follows:

- CHAPTER 1 briefly describes the main ideas of MDSD and the role of model transformation languages in the software development process. A reader is offered the basic knowledge required for understanding the research carried out by the author, as well as the significance of the results achieved. In this chapter a reader is familiarized with the concept of model transformation language.
- CHAPTER 2 contains a detailed description of the model transformation language MOLA, developed in IMCS.
- CHAPTER 3 discusses the role of model transformations in MDSD and Model-Driven Software Development related experience gained while working on the ReDSeeDS project.
- CHAPTER 4 offers the mapping language MALA4MDSD which facilitates the development of this type of transformation.
- CHAPTER 5 describes another practical application of model transformations – the DSL tool development. The DSL tool development frameworks and the role of transformations in the DSL tool development are outlined.
- CHAPTER 6 contains a description of the higher-order transformation language Template MOLA which should be used for transformation synthesis.
- CHAPTER 7 describes different applications of the Template MOLA. Special attention is paid to the development of the mapping language compilers and metamodel independent transformations.
- CHAPTER 8 lists the conclusions drawn while working on the thesis, including possible directions of future research.

CHAPTER 1

Motivation - MDSD and Model Transformation Languages

CHAPTER 1 embraces clarification of the main terms used in the thesis and outlines the research field and the main results in the field under discussion. Results by other researchers used while working on the present thesis are described.

Section 1.1 of this chapter is devoted to the description of modelling. The terms model and metamodel are defined. Application of modelling in software development is discussed in Section 1.2. In Section 1.3 the term model transformations is defined alongside with related to the thesis the latest research results in the area of model transformations.

1.1 Modelling

This section is devoted to the definition of the terms model and metamodel, starting with defining what model is.

1.1.1 What is a Model?

Let us look at this issue in a little broader context, not only as a part of the software development process. Models are used in many areas of our everyday life. Maps are a great example of it. Compared to the original, maps are simplified representations. They contain the necessary information, but skip unimportant details. For example, in metro schemes the lines between stations are drawn as straight lines; however, it is not always true in the reality. A real Paris metro map is shown in Fig. 1. The reader may compare this map with the Paris metro scheme used in maps and tourist guides. An example of a metro scheme is given in Fig. 2. The real metro trajectories do not matter for metro passengers as they can leave the metro only in stations. The things that do matter are locations of metro stations and where it is possible to change from one metro line to another. Metro schemes are drawn keeping in mind what is important and skipping unimportant details.

Models are used in other areas as well and they are widely used in physics. Models are built for physical systems to be used extensively for predicting behaviour of a physical system. Results obtained using models are compared to experimental results. If the experimental results differ from the results obtained using a model it means that the model is false. Consequently, the model of physical systems is either modified or extended.

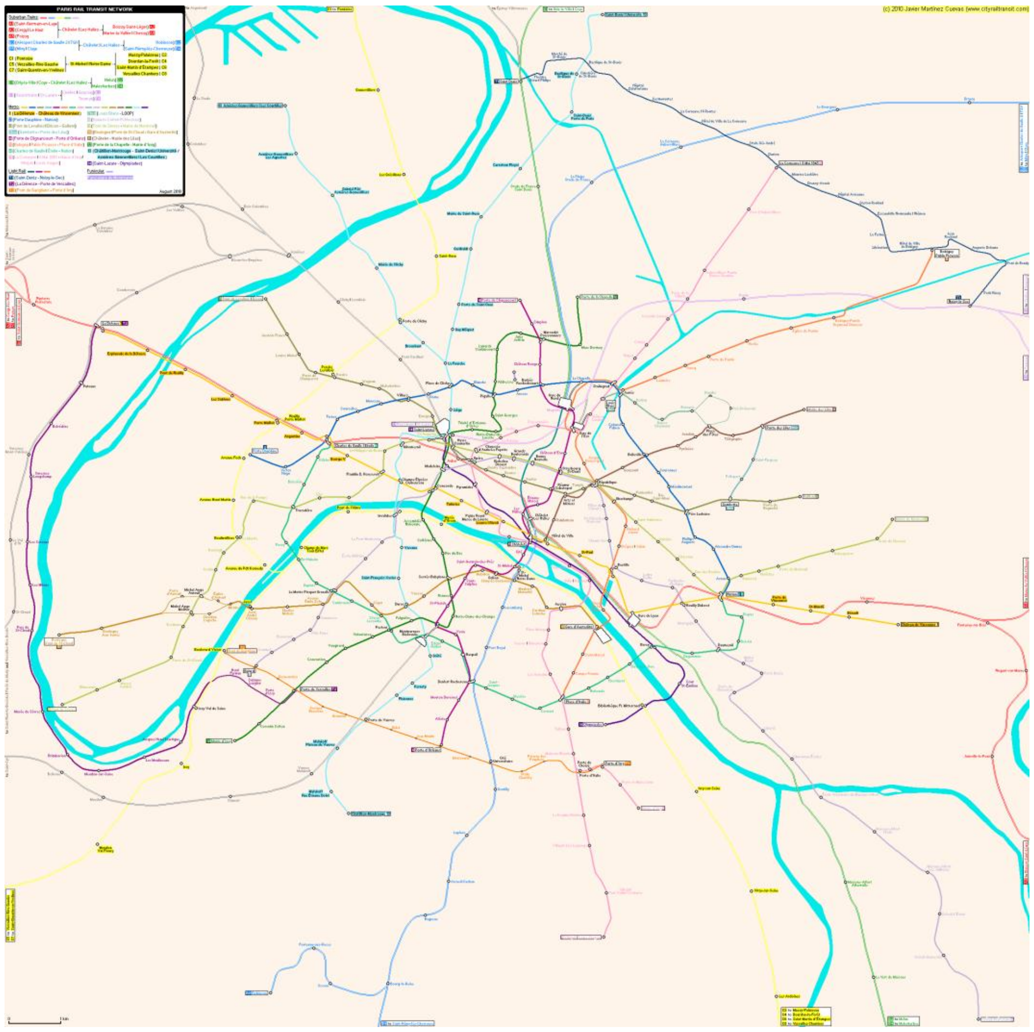


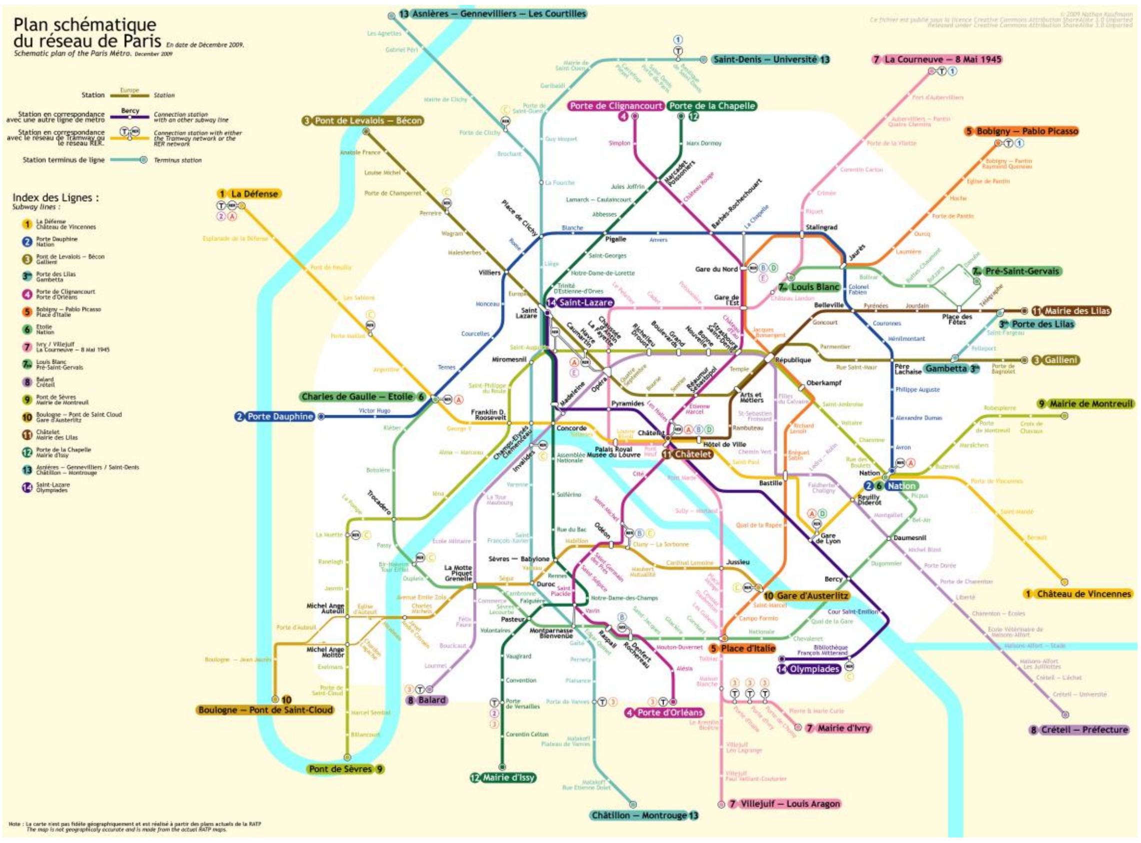
Fig. 1. Real distance map of the Paris metro [27]

Irrespective of the wide use of models in different areas of our life there is no common understanding what a model is.

„Nobody can just define what a model is, and expect that other people will accept this definition; endless discussions have proven that there is no consistent common understanding of models." Jochen Ludewig [103]

Though common understanding of a model is lacking, many definitions of it are available and some of them are listed in Table 1. In the author's opinion a model is simplification of a system which could be used instead of the original for some purpose.

As a result, it is possible to use *model*, which is simpler, safer, and also cheaper, instead of something else that is more complicated, dangerous or more expensive. This is exactly the case of metro schemes. For metro passengers the real metro trajectory and distance does not matter as the stations are the only exit points for them.



Plan schématique du réseau de Paris
En date de Décembre 2009.
Schematic plan of the Paris Métro. December 2009
Index des Lignes :
Subway lines :

Fig. 2. Paris metro schema [196]

Table 1. Model definitions

Author	Definition
Oxford Dictionaries	1. a three-dimensional representation of a person or thing or of a proposed structure, typically on a smaller scale than the original; o (in sculpture) a figure or object made in clay or wax, to be reproduced in another more durable material; 2. a thing used as an example to follow or imitate; o a person or thing regarded as an excellent example

Author	Definition
	of a specified quality; o an actual person or place on which a specified fictional character or location is based; o (the Model) the plan for the reorganization of the Parliamentary army, passed by the House of Commons in 1644-5. 3. a simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions; 4. a person employed to display clothes by wearing them; o a person employed to pose for an artist, photographer, or sculptor; 5. a particular design or version of a product; o a garment or a copy of a garment by a well-known designer. [131]
Jeff Rothenberg	*"Modeling in its broadest sense is the cost-effective use of something in place of something else for some purpose. It allows us to use something that is simpler, safer, or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality."* [143]
Marvin L. Minsky	*"To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A."* [112]
Jean Bézivin	*"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."* [18]
Alan W. Brown	*"Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones."* [24]
Liliana Favre	*"A model is a simplified view of a (part of) system and its*

Author	Definition
	environments." [*40*]
Michael Jackson	*"Here the word 'Model' means a part of the Machine's local storage or database that it keeps in a more or less synchronised correspondence with a part of the Problem Domain. The Model can then act as a surrogate for the Problem Domain, providing information to the Machine that can not be conveniently obtained from the Problem Domain itself when it is needed."* [61]
Thomas Kühne	*"A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made."* [89]
Jochen Ludewig	*"Models help in developing artefacts by providing information about the consequences of building those artefacts before they are actually made."* [103]
OMG	*"A model of a system is a description or specification of that system and its environment for some certain purpose."* [111]
Ed Seidewitz	*"A model is a set of statements about some system under study (SUS)."* [147]
Bran Selic	*"Engineering models aim to reduce risk by helping us better understand both a complex problem and its potential solutions before undertaking the expense and effort of a full implementation"* [148]
Wilhelm Steinmüller	*"A model is information: on something (content, meaning), created by someone (sender), for somebody (receiver), for some purpose (usage context)."* [160]
Thomas Stahl, Markus Völter	*"A model is an abstract representation of a system's structure, function or behaviour."* [*159*]

In software development models are used to describe a system to be built. Models allow analyzing a system before it is really built and looking at the system in different abstraction levels. Systems are very complex. It is not possible to represent all aspects of a system in one diagram. Different models may contain information about different aspects of a system to be built. For example, UML sequence diagrams describe behaviour of a system. UML use case diagrams describe usage scenarios of a system. UML class diagrams contain information about the structure of a system.

On the other hand the information level about a system in diagrams may have a different degree of elaboration. For example, class diagrams may be used to describe the conceptual model of a system as well as the class hierarchy of a system.

Models may be used only as documentation or as an essential part of software development. In MDSD (see Section 1.2) formal models are used. Stahl and Völter describe a model in MDSD:

"Models are abstract and formal at the same time. Abstractness does not stand for vagueness here, but for compactness and a reduction to the essence. MDSD models have the exact meaning of program code in the sense that the bulk of the final implementation, not just class and method skeletons, can be generated from them. In this case, models are no longer only documentation, but parts of the software, constituting a decisive factor in increasing both the speed and quality of software development." [159]

This type of models is going to be discussed in the present PhD thesis. These models are developed by using modelling languages which may be graphical or textual. The focus will be on graphical and formal modelling languages as they are more popular.

1.1.2 Meta-modelling

It is necessary to model modelling languages. A model of a modelling language is called metamodel. Traditionally a metamodel describes the syntax of a modelling language. OMG defines a metamodel similarly: *"A metamodel is a model used to model modeling itself."* [125] *"The typical role of a metamodel is to define the semantics for how model elements in a model get instantiated."* [127]

Stahl and Völter define a metamodel more precisely: *"Metamodels are models that make statements about modelling. More precisely, a metamodel describes the possible structure of models – in an abstract way, it defines constructs of a modelling language and their relationships, as well as constraints and modelling rules – but not the concrete syntax of the language"* [159]

The most popular meta-modelling language is MOF. *"The MOF 2 Model is used to model itself as well as other models and other metamodels (such as UML 2 and CWM 2 etc.). A metamodel is also used to model arbitrary metadata (for example software configuration or requirements metadata)."* [125]

"A model that is instantiated from a metamodel can in turn be used as a metamodel of another model in a recursive manner." [127] It is possible to go further this way and introduce a metametamodel – a model of metamodelling language. It is possible to introduce even more meta-levels. However, in practice we don't need to introduce more meta-levels. A scheme of meta-levels is shown in Fig. 3.

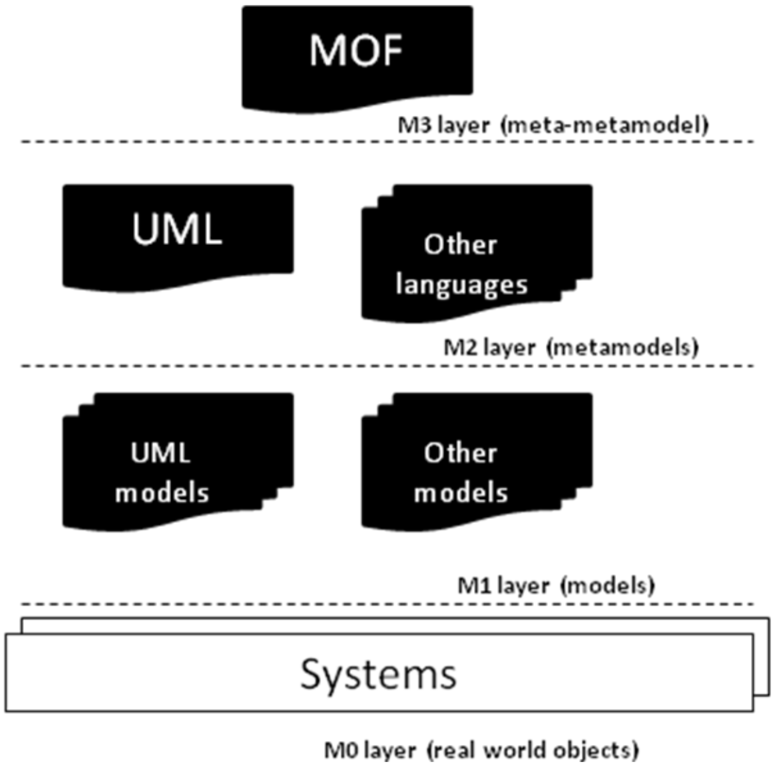


Fig. 3. Example of OMG MOF meta-level hierarchy [130]

Layer M3: *"The meta-metamodeling layer forms the foundation of the metamodeling hierarchy. The primary responsibility of this layer is to define the language for specifying a metamodel." "MOF is an example of a meta-metamodel."* [127]

Layer M2: *"A metamodel is an instance of a meta-metamodel, meaning that every element of the metamodel is an instance of an element in the meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models." "UML and the OMG Common Warehouse Metamodel (CWM) are examples of metamodels."* [127]

Layer M1: *"A model is an instance of a metamodel. The primary responsibility of the model layer is to define languages that describe semantic domains, i.e., to allow users to model a wide variety of different problem domains, such as software, business* *processes, and requirements. The things that are being modeled reside outside the metamodel hierarchy." "A user model is an instance of the UML metamodel."* [127]

"The metamodel hierarchy bottoms out at M0, which contains the run-time instances of model elements defined in a model. The snapshots that are modeled at M1 are constrained versions of the M0 run-time instances." [127]

OMG MOF 1.4 standard explains meta-levels as follows: *"the MOF meta-metamodel is the language used to define the UML metamodel, the UML metamodel is the language used to define UML models, and a UML model is a language that defines aspects of a computer system."* [118]

The most popular meta-modelling standard (language) is MOF *(Meta-Object Facility),* developed by the international standards organisation OMG. Currently the actual MOF version is 2.4.1 [129]. Of course, MOF is not the only meta-modelling language, there are others, for example, *KM3* [62] and *EMF Ecore* [166].

1.2 Model-Driven Software Development

Today software becomes more and more complicated. Software development and management has become more challenging, especially if it refers to large-scale systems which are developed and used by hundreds, even thousands of people. In order to ease the development of software, particular models are used to describe different aspects of the system to be developed. [130]

Different terms are used to refer to the use of models in software development. This section outlines different approaches to the use of models in software development and the role of models in each approach to the software development process. The most popular approaches in model use are described below.

1.2.1 MD*

Several terms are used regarding model use in software development. The most popular terms are listed in Table 2, starting from the narrowest to the broadest formulation. Term relationship is given in Fig. 4.

Table 2. Terms for MD*

Term	Definition
MDA – Model Driven	*"MDA is the OMG's particular vision of MDD and*

Architecture	*thus relies on the use of OMG standards. Therefore, MDA can be regarded as a subset of MDD."* [113]
MDSD – Model Driven Software Development	*"Model-Driven Software Development is a software development approach that aims at developing software from domain-specific models."* [190] The same as MDD.
MDD – Model Driven Development	*"MDD is a development paradigm that uses models as the primary artefact of the development process. Usually, in MDD, the implementation is (semi)automatically generated from the models."* [113] *"Model-driven development is a style of software development where the primary software artifacts are models from which code and other artifacts are generated."* [161] The same as MDSD.
MDE – Model Driven Engineering	*"Software Engineering paradigm where models play a key role in all engineering activities (forward engineering, reverse engineering, software evolution,...)"* [113]
MD* - Model Driven Everything	*"I use MD* as a common moniker for MDD, MDSD, MDE, MDA, MIC, LOP and all the other abbreviations for basically the same approach."* [189]

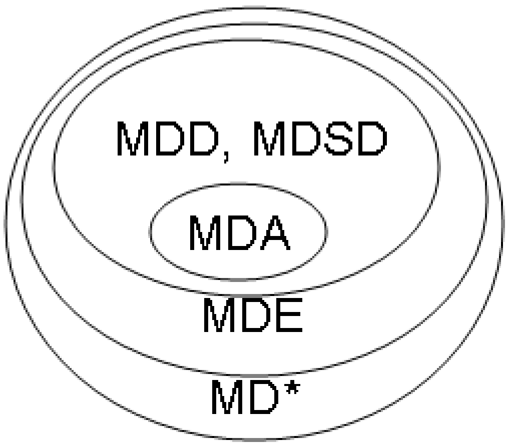
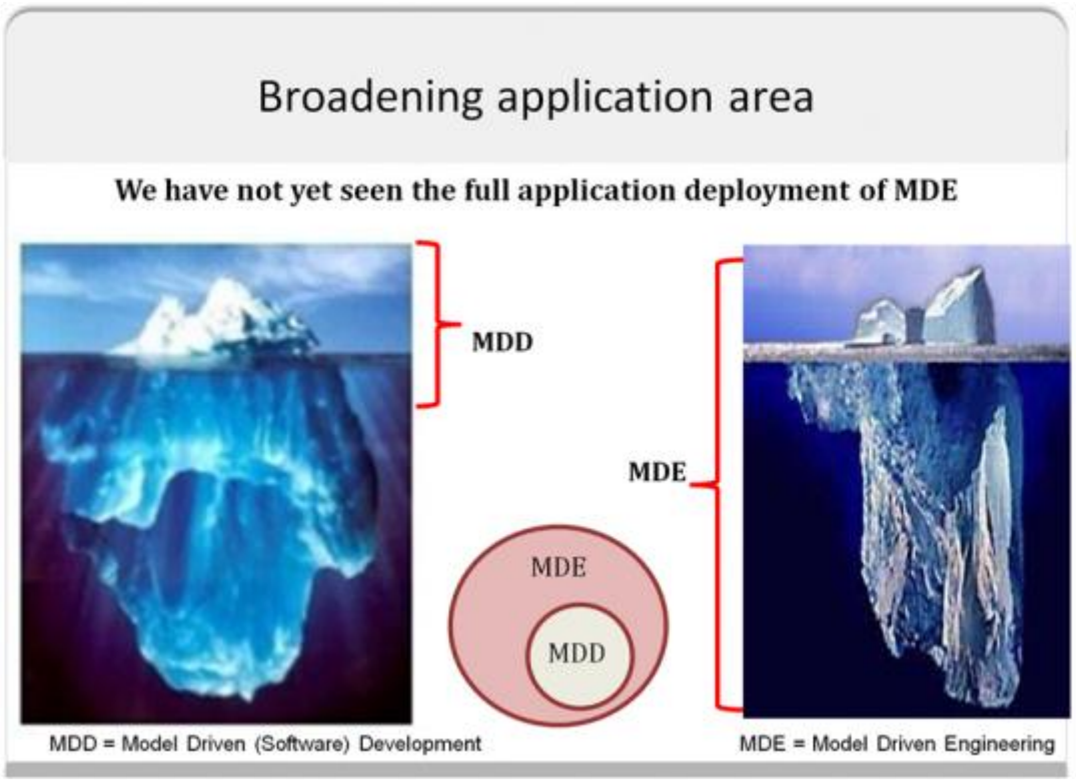


Fig. 4. Relationship between MD* terms

MDA was the first term applied regarding the use of models in software development. It was launched by OMG (*Object Management Group*) in 2000. In MDA a chain of three consecutive models is used. More information on MDA is given in Section 1.2.2. Today MDA is considered an obsolete term. The usage of exactly three consecutive models seems too restrictive.

The terms MDD or MDSD, carrying approximately the same meaning, are used as well. The usage of one or another depends on the taste of the author.

Another term is MDE which has a wider application than MDD and MDSD. See Fig. 5 for the way Jean Bezivin presents the relationship between MDD and MDE. MDE could be applied to any usage of models, including even those we are not yet familiar with.



Broadening application area

We have not yet seen the full application deployment of MDE

MDD = Model Driven (Software) Development

MDE = Model Driven Engineering

Fig. 5. MDE versus MDD [17]

1.2.2 Model Driven Architecture

Model Driven Architecture (MDA) was launched by OMG in 2000. It was the first attempt to formalize the use of models in software development. The first version of MDA manual [117] was published in 2000 by OMG. The updated version of the MDA guide was published in 2003 [111].

"The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MDA provides an approach for, and enables tools to be provided for:

- *specifying a system independently of the platform that supports it,*
- *specifying platforms,*
- *choosing a particular platform for the system, and*
- *transforming the system specification into one for a particular platform.*

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns." [111]

The MDA guide proposed to use three consecutive models. Each of them described a system on a different level of details, starting from a more abstract definition and gradually elaborating the details. The following three models where offered:

- **CIM** - *"A computation independent model is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification."* [111] This model does not contain information about the system implementation. *"The CIM helps to bridge the gap between the experts about the domain and the software engineer."* [40] This model could be treated as requirements for a system to be built. *"A CIM could consist of UML models and other models of requirements."* [40] However there is no common understanding what and how should be modelled in CIM.
- **PIM** - *"A platform independent model is a view of a system from the platform viewpoint. A PIM exhibits a specified degree of platform independence suitable for use with a number of different platforms of similar type."* [111] This model describes the architecture and high-level behaviour of a system to be built. However this description could be adapted for different implementation frameworks.
- **PSM** - *"A platform specific model is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the*

details that specify how that system uses a particular type of platform." [111] This model is an extension of PIM, adding specific details for the implementation platform.

Computation Independent Model was proposed for starting software development and continued with *Platform Independent Model*. Today most of industrial approaches propose to start with PIM as there is no common understanding of CIM. Some authors even have a disparaging attitude towards CIM; some propose to treat CIM as requirements [101]. In case of using CIM some suggest it to be automatically transformed to PIM. However, as it is not possible to obtain automatically all the necessary information in *Platform Independent Model*, it was proposed that this model should be extended manually. It is easy to see that it is not possible to automatically obtain system architecture from requirements.

Already the MDA guide proposed transition from PIM to PSM to be done by using automatic transformation. A model is not an executable system. Therefore one more transition step from *Platform Specific Model* to a code is necessary. MDA application scheme is shown in Fig. 6.

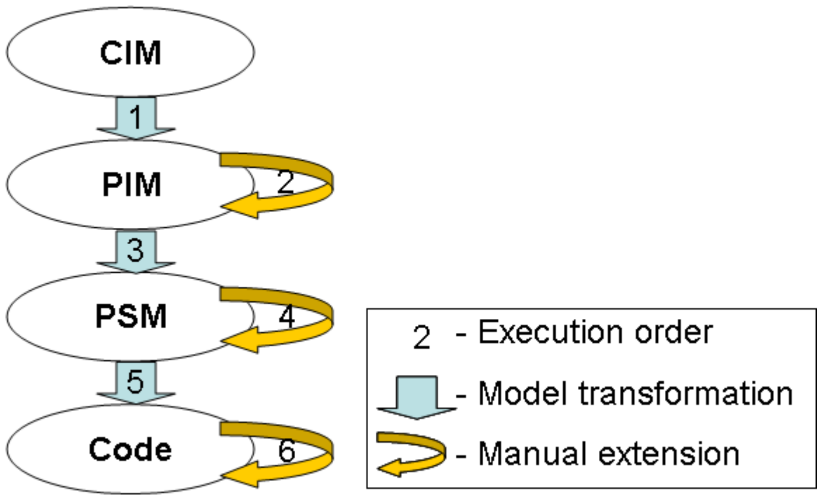


Fig. 6. MDA application schema with one execution environment

One of the goals for MDA introduction was to support reusability and application development for different frameworks as there are cases when it is necessary to create the same application for different frameworks. Applications for mobile phones may serve as an example. Different phone developers support different application execution environments. This is one of the reasons why *Platform Independent Model* is separated from *Platform Specific Model*. When using the same *Platform Independent Model* it is possible to develop application for different frameworks. MDA application scheme with the support of multiple execution environments is given in Fig. 7.

It should be noted that MDA allows using only the UML language for a model description.

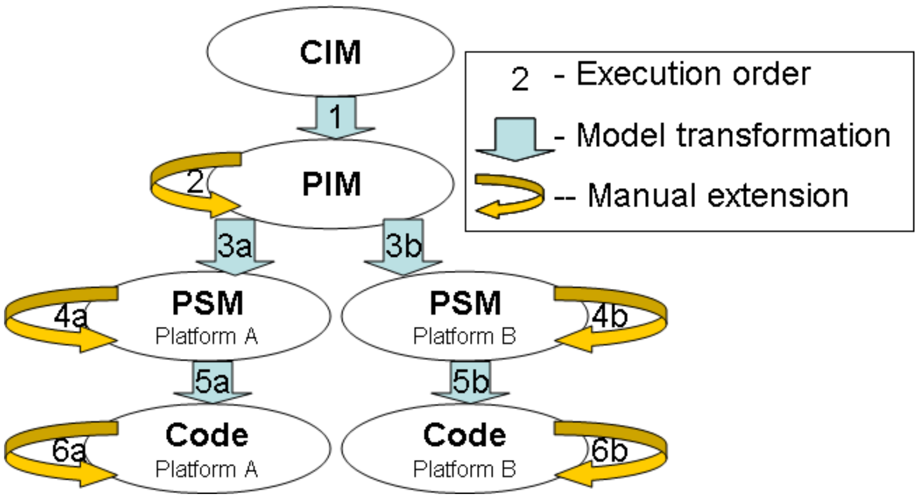


Fig. 7. MDA application schema with multiple execution environments

As already stated above the MDA guide proposed to implement transition from PIM to PSM by using automatic model transformation. In the context of MDA the term model transformation was introduced. *"Model transformation is the process of converting one model to another model of the same system."* [111] The term model transformation is described in detail in Section 1.3.

1.2.3 Model Driven Software Development

MDA process is too restrictive. This is a reason why it has not been widely accepted in industry. Nowadays MDA is treated as obsolete term. However, the good ideas behind MDA as models and model transformations are employed in *Model-Driven Software Development*.

Compared to MDA in MDSD it is possible to use any chain of models. In MDA there was the restriction that the UML language should be used to define models. In MDSD there is no such restriction.

One specific type of MDSD is *Domain-Specific Modelling* (DSM). In DSM only one model is used. Code is generated directly from this model which is defined in specialised *Domain-Specific Modelling Language*. *Domain-Specific Modelling* is described in detail in Section 1.2.4.

1.2.4 Domain-Specific Modelling Languages

Another specific case of MDSD have become exceedingly popular - the specialized modelling languages. It is a common practice to create and use specialized modelling languages for a domain area and they are called *Domain-Specific Modelling Languages* (DSML). They are developed for users specialized in a concrete area, e.g. a language for automotive software development (*AUTOSAR* [10]), mobile telephone software development [88], and many others.

Domain-Specific Modelling Languages (DSML) is a subset of a more general set of languages, namely, *Domain-Specific Languages (DSL)*. When using *Domain-Specific Languages* users can operate with familiar terms. The use of a DSL increases the efficiency of software development in the field. DSLs are applied in many areas of software development. A popular DSL, for example, is SQL – a specialised language for working with databases.

Software development using DSML is called *Domain-Specific Modelling* (DSM). Commonly, when applying this approach, only one model developed in DSML is used. This model is directly transformed into an executable code. However, approaches exist of using chains of domain-specific models when each model covers different aspects of a system. Relation between DSM and other software development approaches is shown in Fig. 8.

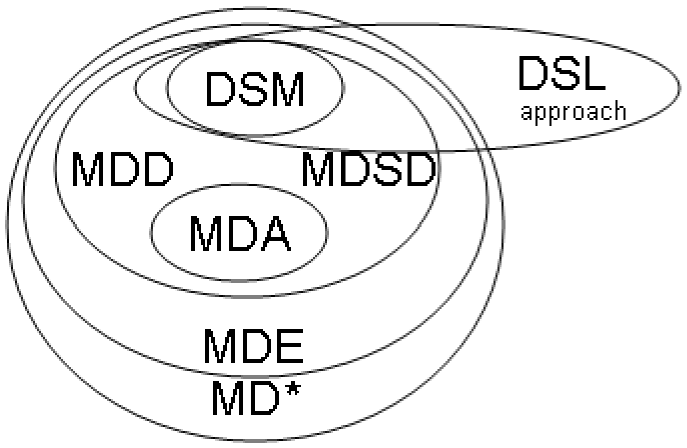


Fig. 8. Relation between MD* and DSL approaches

There can be graphical or textual *Domain-Specific Modelling Languages*. However, DSMLs are more often graphical. (Nevertheless it is not true for DSLs in general.) Only graphical *Domain-Specific Modelling Languages* will be considered here.

A visual *Domain-Specific Modelling Language* basically consists of two parts – the domain part and the presentation (visual) part. Sometimes they are called also the abstract and concrete syntax respectively. The domain part of the language is defined by means of the *domain metamodel*, where the relevant language concepts and their relationships are formalized. The domain metamodel is also used for a precise definition of language semantics. Standard MOF [120] or similar notations are used for the definition of domain metamodel.

As regards the presentation part (concrete syntax) definition there is no universally accepted notation. The same meta-modelling techniques are used, but with various semantics. Most frequently, instances of classes in the presentation type metamodel are *types of diagram elements* to be used in the diagram. A concrete set of graphical element types for a diagram definition is called the *presentation type model* (a typical example is the graphical definition model in GMF [172]).

Tool development for graphical *Domain-Specific Languages* is time consuming and expensive. Due to the growing popularity of *Domain-Specific Modelling Languages* various graphical tool building frameworks have been developed to improve the tool (editor) building process. Two different approaches are used in these environments. The first option is to use a mapping-based approach. During the tool design this mapping assigns a fixed presentation type model element (a node type, edge type or label type) to a domain metamodel element, by means of which the latter must be visualized. This solution is quite appropriate for simple cases, where no complicated mapping logic is required. In this case tools for simple DSMLs can be developed even during a presentation session. However, frequently DSML support requires much more complicated and flexible mapping logic. One of the reasons is the lack of fixed correspondence between the domain metamodel and presentation types. In this case the second approach is used: to define the correspondence by *model transformation languages*. Transformations define the synchronisation between the domain and presentation models and the tool behaviour in general.

Mapping based frameworks are *MetaEdit+* [109], *GMF framework* [172], *Microsoft DSL Tools* [28], *Generic Modeling Tool* [26] and some other. A pure transformation based framework is *METAclipse framework* [86]. The other transformation based frameworks *Tiger GMF project* [37], *ViatraDSM framework* [133] and *GrTP* [15] provide also some elements of the mapping based approach.

There exist mapping based and transformation based tools, but usually some parts of the same DSL are suitable for mappings and some for transformations. It means none of the solutions is optimal. The absence of a good combined solution creates the problem which is discussed in detail in CHAPTER 5.

1.3 Model Transformations

This Section focuses on defining the term model transformation; sketching a brief introduction into the history of model transformations; listing the popular model transformation languages and discussion of the need of model transformations as DSLs for specific transformation domains. For introduction a definition of transformation is offered:

Transformations can easily be understood when thinking about what happens in nature: an ugly caterpillar is transformed into a beautiful butterfly (Fig. 9); tadpoles into frogs; leaves change their colours in autumn. These transformations occur always in the same way. It means that the occurrence and the way of transformation is predefined somewhere in nature, most probably in DNA.

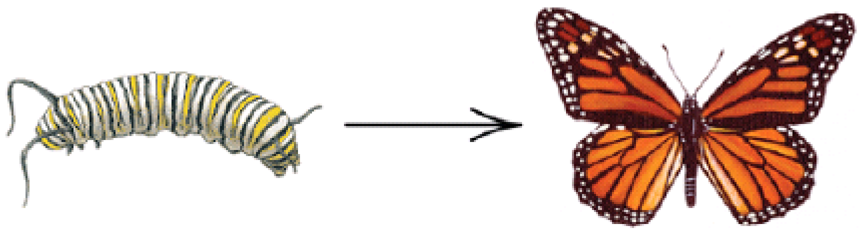


Fig. 9. Transformation in the nature [30]

"A transformation is the automatic generation of a target model from a source model, according to a transformation definition." [90]

"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. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language." [90]

Although this definition could be applied to caterpillars and butterflies in terms of this thesis we will be concerned with transformation of data or, more precisely, transformation of models. Model transformation execution scheme is given in Fig. 10. This scheme directly corresponds to the definition of transformation. The source model is transformed into a target model according to a transformation definition. It should be added that model transformations are defined in terms of source and target metamodels. It means that the same transformation could be used for all source models confirming to the source metamodel. As transformation works in terms of metamodels all target models will confirm to the target metamodel. Of course, it is possible that source and target models coincide; such transformations are called in-place transformations.

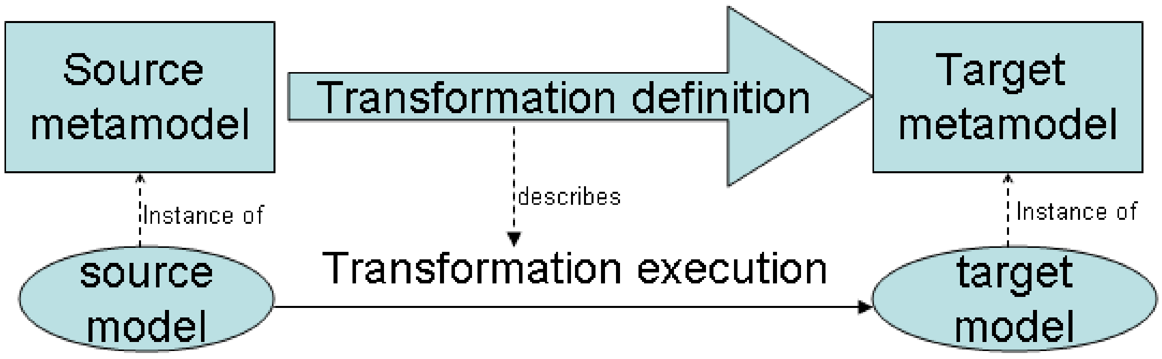


Fig. 10. Execution scheme of model transformations

Model transformation languages are used for writing down a model transformation definition. The most popular model transformation languages are listed in the following sub-Section.

1.3.1 Model Transformation Languages

As already mentioned above the term model transformation for the first time was introduced in the MDA Guide [117]. At that point there were no appropriate means for writing down model transformations. Of course, general purpose programming languages could be used, however, they did not have appropriate means to support working with models. Therefore OMG requested to submit proposals on model transformation language *QVT (Queries/ Views/ Transformations)* [119]. The development of QVT standard was very slow and the first version of QVT standard was published only in April, 2008 [122]. Currently the actual version is QVT 1.1. [128].

As a result of the slow QVT development many independent model transformation languages were developed, for example, *MOLA* [76, 59], *Lx* [13], *GReAT* [7], *UMLX* [197, 179], *ATL* [63, 165], *Tefkat* [98, 35], *MTF* [56], *ATOM*^{3} [96, 107], *VMTS* [99, 25], *BOTL* [105, 58], *Fujaba* [42, 45], *RubyTL* [32, 185].

In CHAPTER 2 the model transformation language MOLA is discussed in detail as it is used in model transformation applications described in the present PhD thesis.

There already existed many graph transformation languages before OMGs RFP. The first graph transformation language *PROGRESS* was developed as early as the beginning of the 1990s [145]. Influenced by OMGs RFP many graph transformation languages were adapted for the development of model transformations, for example, *AGG* [163], *PROGRES* [144], *TGG* [146, 46], *VIATRA* [31, 180]. In fact, there is no big difference between typed-attributed graphs and models. At present distinguishing between a model and a graph transformation language is sometimes quite difficult.

Model transformation language alone is not sufficient for developing model transformation as tool support for the language is required as well. Tool support for independent model transformation languages was mainly developed by research groups closely associated with the authors of the language. As a result tool support for many languages is mainly experimental and is devoid of industrial qualities. The first language with good enough tool support was ATL. Most probably this is the reason why ATL is the most popular model transformation language.

The situation with tool support of the QVT standard is even worse. There is no tool supporting the QVT language completely. There are some tools supporting parts of MOF QVT. *MOF-QVT Operational* is supported by *SmartQVT* tool [150]. *Eclipse M2M* project partially implements *QVT Operational* and *QVT Declarative* (*Core, Relational*) [175]. *MOF-QVT Relational* is partially supported by *MediniQVT* [57]. UML modelling tool *MagicDraw* [115] uses *QVT Operational* plug-in implemented by *Eclipse M2M project* [175].

The limited tool support of QVT and understanding that for different domains different transformation languages are needed are the reasons for developing new transformation languages even now, among them being *Epsilon* [92, 169], *Henshin* [9, 173], *GreTL* [55], *lQuery* [100], *UML-RSDS* [95], *Edapt* [168].

Examination of application areas of model transformations reveals that for each different domain a different language is more appropriate. Actually many transformation languages are developed, keeping a certain domain in mind. For example, MOLA was developed for transformation development in the MDA process. *Viatra* specializes in transformation development for simulators. *lQuery* is suitable to develop transformations for the DSL tool development. *Epsilon* actually is a transformation language family where each language is suitable for a definite set of tasks. There are domain-specific transformation languages applicable in certain domains. One well studied domain is model transformation for model migration.

1.3.2 Mapping Languages

When highly abstracting in the consideration of model transformations, we can treat them as mapping that is done from the source to the target. That is the way transformations were treated in the MDA guide [111]. However, transformations can be subject to complicated execution conditions. It is hard to represent these conditions as mappings. Therefore mappings can be used only in simple and declarative parts of transformations. Hence mappings can be used as a transformation language for simple cases.

“A mapping is specified using some language to describe a transformation of one model to another. The description may be in natural language, an algorithm in an action language, or in a model mapping language.” [111]

Attempts to create universal mapping languages as a certain alternative to traditional transformation languages have been started sufficiently early. The term mappings are used already in the MDA guide [111].

List of mapping languages is given in the Section 4.1.2.

1.3.3 Higher-Order Transformations

MDD can be naturally applied also to transformation development. It means that transformations are used to create transformations. This special kind of transformations is named *Higher-Order Transformations* (HOT). These are transformations modifying/ reading/creating model transformations. In the HOT approach transformations must be treated as models conforming to the relevant metamodel.

Though the HOT idea can be applied to any transformation language, the largest amount of HOTs has been created for the ATL language [63]. A comprehensive survey of HOT applications is given in [183] where the four main types of HOTs have been identified. One of the HOT application types is transformation synthesis. Transformation synthesis means transformation generation from various sources of information, including model mappings. Such a mapping between two models can be considered as a high level specification of the required model transformation. A large set of such mappings has been obtained by applying the *ATLAS Model Weaver* (AMW) [39]. The idea of obtaining a transformation from a mapping can be applied to many other transformation languages, for example MOLA. In CHAPTER 6 a special language for transformation synthesis Template MOLA is proposed. It is the first language [69] built specially for the development of *higher-order transformations*. Afterwards a special extension of ATL for transformation synthesis was developed as well. [182]. However ATL is textual, while MOLA and Template MOLA are graphical languages.

One of the popular research directions related to the HOTs approach is the development of metamodel independent transformations. In most of the model transformation languages a transformation is attached to the metamodel it is defined for. This makes transformation reuse almost impossible. An approach for solving this problem is proposed by [33] and [139]. It should be noted that Template MOLA could be used to develop metamodel independent libraries for MOLA. See Section 7.4 for details.

CHAPTER 2

MOLA Language

As the model transformation language MOLA was used to develop transformations described in the thesis an overview of the MOLA language is given in this chapter. More about the MOLA language can be found in [76], [75] and [77]. A formal description of MOLA as well as the *MOLA tool*, can be downloaded at [59].

2.1 MOLA Overview

MOLA is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts of transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed.

A MOLA program transforms an instance of a source metamodel into an instance of a target metamodel. The two metamodels are specified using the *EMOF* [120] compliant metamodelling language (*MOLA MOF*). These metamodels, which may also coincide, both are parts of a transformation program in MOLA. Mapping associations may be added to link the corresponding classes in the source and target metamodels.

MOLA is a model transformation language which combines the imperative (procedural) programming style with declarative means of pattern specification. A transformation written in MOLA consists of several *MOLA procedures,* one of them being *the main.* An example of a MOLA procedure is given in Fig. 11 (p.54). The execution of a MOLA program starts with the main procedure. Procedures in MOLA may be called from the body of another procedure by using *call statements*. Like in most transformation languages, class instances, primitive and enumeration-typed variables can be passed on to the called procedures as parameters. There are other types of statements in MOLA as well, i.e. *rule*, *foreach loop, text statement*, etc. The execution of a MOLA procedure starts with the *start symbol.* The next statement to be executed is determined by the outgoing control flow.

The rule in MOLA represents the classical branching (*if-then-else*) construct of imperative programming. The rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Only the first valid pattern match is considered. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using *class elements* in the MOLA rule. The traditional UML instance notation (*instance_name:class_name*) is used to identify a particular class element and specify the class the instance must belong to. Class elements included in a pattern may have attribute constraints – simple OCL-like expressions. Expressions are also used to assign values to variables and attributes of class instances. Additionally, the rule contains *association links* between class elements. A class element may represent an instance, matched previously by another pattern. Such class element is called a reference class element and is specified using the name of the referenced class element, prefixed with the symbol“@”.

Typical transformation algorithms require iteration through a set of the instances, satisfying the given constraints. In order to accomplish this task, MOLA provides the *foreach loop* statement. The *loophead* is a special kind of the rule used to specify a set of instances to be iterated in the foreach loop. The pattern of the loophead is given by using the same pattern mechanism as for an ordinary rule, but with an additional important construct. It is the *loop variable* – the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages.

2.2 MOLA Elements

Table 3 presents a list of MOLA elements. The application context and semantics of each element is described.

Table 3. List of MOLA elements

Image	Element	Description
●	Start symbol	Execution of a MOLA procedure starts with a start symbol. Execution of a MOLA transformation starts

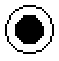
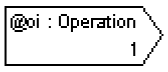
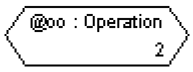
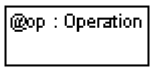
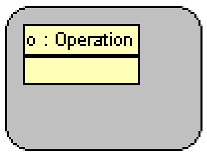
Image	Element	Description
		from the start symbol of the main procedure.
	End symbol	Execution of a MOLA procedure ends with an end symbol. When the end symbol is reached in the main procedure execution of transformation is completed. In other procedures control is returned to the procedure calling this procedure.
	Input parameter	MOLA procedures may have parameters, defined by name and type (@<*name*>:<*type*>). The name should be unique in the procedure (different from class element names). The type is a reference to a class defined in MOLA MOF or a primitive type. Parameters are ordered. The order is represented by numbers. Values of input parameters are passed to the procedure; if the value is changed it is not passed back.
	In/out parameter	The same as the input parameter: the only difference is that the value of parameter is passed back to the calling procedure.
	Variable	It is possible to define variables in MOLA procedures. For variables the name and the type is defined (@<*name*>:<*type*>). Variables are used in the same way as parameters.
	Rule	MOLA rule consists of a pattern to be matched and an action part. Both are defined by means of class elements and association links. The pattern in the rule is matched only once. If a rule without a valid match is to be executed and it has no *ELSE*-exit, then the current procedure is terminated (if this occurs outside a loop) or the next iteration of the loop is started

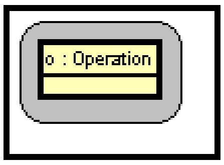
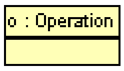
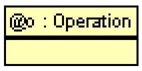
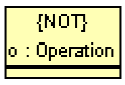
Image	Element	Description
		(within a loop body).
	Loop	MOLA loop contains a loophead (the first rule) and a loop body (0 or more loop elements whose execution order is defined by control flows). The loophead is a rule which contains a loop variable. The loophead and the loop body are executed for each distinct match of loop variable.
	Class element	A class element is a metamodel class, prefixed by the element (*role*) name. A class element may also contain a constraint – a Boolean expression in a simplified subset of OCL. Assignments in class elements may be used to set the attribute values of the instances. When a pattern in a rule is matched for each class element, an instance satisfying constraints is found and attached to a class element (constraints are defined in a class element and by a pattern, e.g., connections with other class elements).
	Class element, reference	References are marked with the symbol "@". The previously matched instances, as well as the parameters and the variables, may be used as references. In this case, an instance already attached to a referenced element is used in a pattern matching.
	Class element with NOT constraint	Equivalent to NAC (negative application condition) in graph transformation languages, e.g., AGG [163]. A pattern is matched if there are no instances in

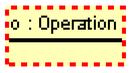
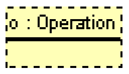
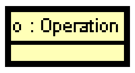
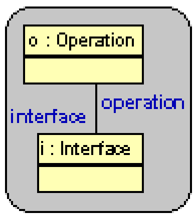
Image	Element	Description
	(NOT-element)	the model corresponding to the NOT-element. NOT-elements are typically connected to other class elements by using association links. Such a pattern matches if there is no instance corresponding to the NOT-element which fulfills conditions defined to NOT-element and has all specified links to the instances of “normal part”.
	Class element, creation	It is possible to create instances in the rules. Creation is marked with a red dashed line. Assignments may be used to set the attribute values of the newly created instances.
	Class element, deletion	It is possible to delete instances in the rules. Such class elements may be references or they are matched before deletion. Deletion of a class element causes automatic deletion of the related links.
	Loop variable	Loop variable is an iterator of foreach loop. A foreach loop iterates through all possible instances of the loop variable class that satisfies the constraint imposed by the pattern in the loophead. There is only one loop variable in a loop.
	Association link	An association link, connecting two class elements, corresponds to an association linking the respective classes in the metamodel. Class elements at the ends of links are matched to the instances connected with a link of this type.

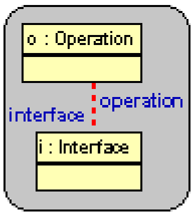
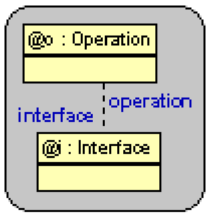
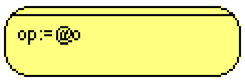
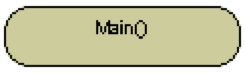
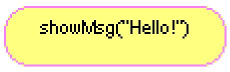
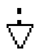
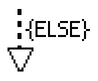
Image	Element	Description
	Association link, creation	It is possible to create instances of association links. An end of a create-link may be attached to a *class element* included in the pattern or to the *class element, creation*.
	Association link, deletion	It is possible to delete instances of association links. An end of a delete-link may be attached to a *class element* included in the pattern (also the *class element, deletion*). Association links are deleted before the class element deletion.
	Text statement	Text statements consist of a constraint and assignments. It is possible to assign values to parameters, variables and class element references. Assignments are skipped if the constraint fails. Mainly text statements are used to process primitive-typed elements. A text statement containing a constraint (a Boolean expression) may also have an *ELSE*-exit and serve as an *if-then-else* construct.
	Call statement	Call statements are used to invoke sub-procedures. Parameters are passed to the invoked procedures. If the parameter is of the type in/out to pass the value to this parameter a referencable element (variable, parameter, class element reference) should be used.
	External call statement	Besides MOLA procedures, external (coded in an OOPL) procedures can also be invoked; this feature is used for low-level data processing (e.g., model data import). Parameters may be passed to external procedures.
	Control flow	Control flow arrows determine the execution order of MOLA statements. The element that

Image	Element	Description
		follows the use of the control flow is executed as the next one. (If the execution of the previous element – rule, text statement – had succeeded.)
	Alternative control flow	Certainly, there may be a situation when no match exists – then the rule is not executed at all. To distinguish this situation, the rule may have a special ELSE-exit (alternative control flow), which is traversed in this situation. Alternative control flow may be added also to text statements. This control flow is used if the constraint in the text statement fails.

2.3 MOLA Example

In order to illustrate the basic MOLA concepts, briefly listed in the previous section, a simple MOLA transformation example is provided in Fig. 11. This example is taken from transformations developed in the ReDSeeDS project (see CHAPTER 3). UML (+ ReDSeeDS specific traceability framework) is used as a source and target metamodel of the transformation.

This procedure copies the interface and all operations it contains to the provided package in the target model. ReDSeeDS specific traceability information is created between the original interface and its copy.

This MOLA procedure has four parameters. Three of them are input parameters and one in/out parameter. The first parameter (*@int*) is the interface to be copied. The second parameter (*@pt*) is a package for the copy of the interface to be placed. The third parameter (*@sa*) is ReDSeeDS specific. It is a logical model (*Software Artifact*) processed. All traceability links between the elements are attached to this logical model. The fourth (in/out) parameter (*@i*) is used to return the reference to the newly created copy of the interface.

Execution of the MOLA procedure starts with a start symbol, followed by the execution of the rule (using control flow). As already stated previously, the MOLA rule may consist of a declarative pattern and an action description. In this case the pattern is trivial as all class elements with black solid borders are references. Nothing is matched; the values attached to the references are used directly. Therefore execution of the rule starts directly with the execution of actions defined in the rule. This rule creates a new instance of an interface (*newint*) and the latter is set the same name as the name of the interface to be copied (*name=@int.name*). To assign values in MOLA simple OCL like expressions are used. (For details see MOLA reference manual [6].) In the same rule ReDSeeDS specific traceability information is created (*id:isDependentOn*) for which the original interface is set as a source and the copy of the interface - as a target. The traceability information is attached to ReDSeeDS logical model (@sa). This rule uses references to the provided parameters (*@int*, *@sa*, *@pt*) and creates appropriate instances (*newint*, *id*) and association links.

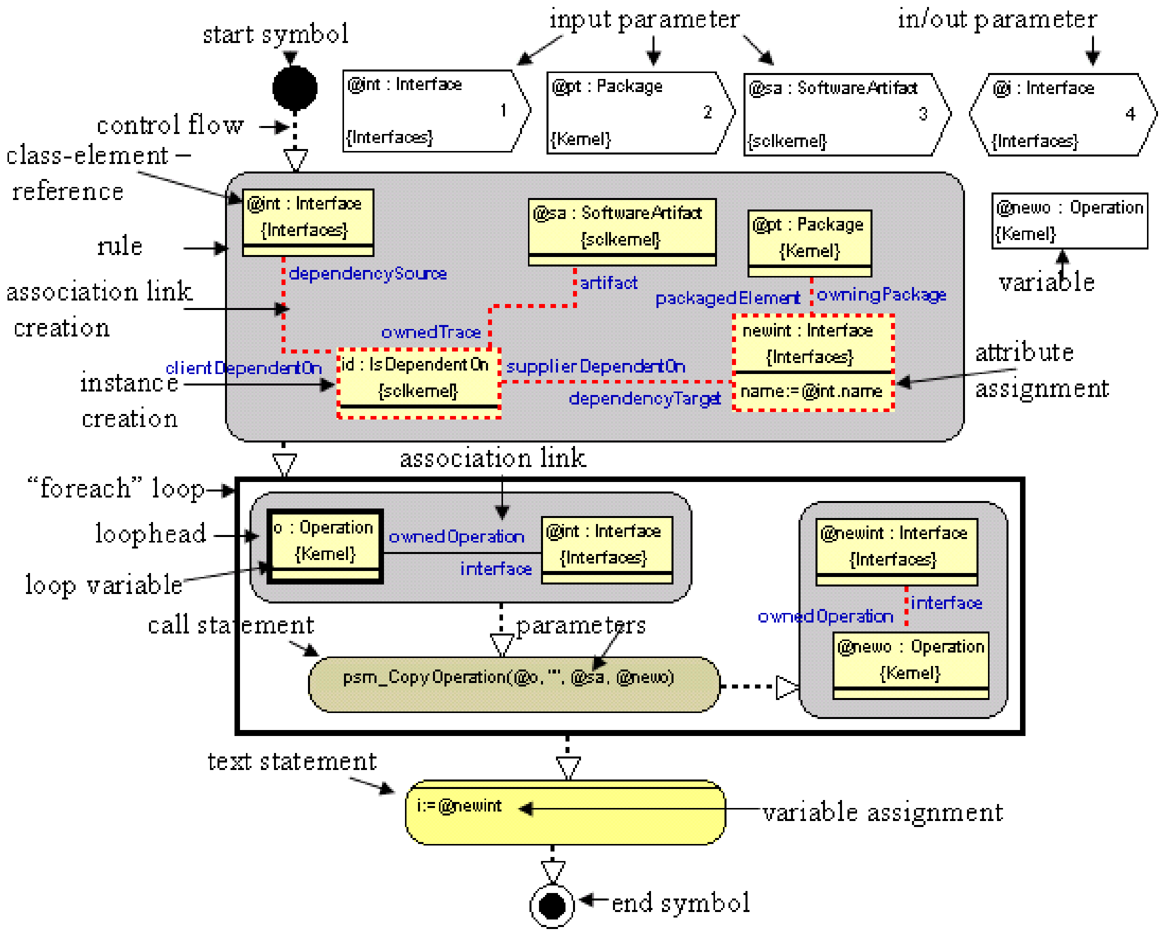


Fig. 11. MOLA example

The rule is followed by a foreach loop which iterates through all operations of the interface to be copied. The operation is used as a loop variable (*o*). It is checked that the operation is connected to the interface using the association link *ownedOperation – interface*. Only the operations satisfying this condition are processed.

For each such operation procedure “pim_CopyOperation” is called (using the call statement). This procedure contains four parameters as well. The first is the operation to be copied (*o*). The second is simply an empty string and it is not important in this context. The third is again ReDSeeDS logical model, used to attach the traceability between the original and the copy in the same way as in this procedure. The fourth is a reference to the variable (*@newo*) defined in this procedure. This actually is in/out parameter and is used to return the newly created copy of operation.

After the call statement the MOLA rule is executed. The copy of operation (*@newo*) returned by the call statement is attached to the copy of the interface (*@newi*). Association link (*ownedOperation – interface*) is created.

The loop and actions in it are executed while there are operations satisfying constraints in the loophead. After execution of the loop completes the text statement is executed. This text statement assigns a value to in/out parameter. The value of the parameter is set to the created copy of the interface. As a result, when reaching the end symbol, the parameter will return the reference to the newly created copy of the interface.

Reaching of an end symbol is the last element of the MOLA procedure and it completes its execution. Control is returned to the calling procedure. The value of in/out parameter is also returned.

To get a more detailed understanding about the usage of different MOLA elements see the next section.

2.4 Hello World with MOLA

This section is dedicated to describing a solution for the *Hello World* case [106] of the TTC 2011 [5] contest, implemented in the MOLA model transformation language: “*Saying Hello World with MOLA - A Solution to the TTC 2011 Instructive Case*” [74]. This use case demonstrates the application of MOLA constructs for solving typical transformation tasks. This section provides a more detailed understanding about the usage of different MOLA elements in transformation development. If a reader is familiar with the MOLA language he/she can skip this section.

The *Hello World* case consists of several very simple tasks. It confirms the assertion that simple tasks can be solved in a straightforward and easy readable way in MOLA. In most cases the basic part of the task is performed by one rule (or loophead).

2.4.1 Greeting Tasks

The first group of tasks is "Greeting" transformations. The first task is to *"provide a constant transformation that creates the example instance of the "Hello World" metamodel given in Fig. 12."* [106] The next task is based on *"slightly extended metamodel given in Fig. 13."* [106] It is required to *"provide a constant transformation that creates the model with references also shown in Fig. 13."* [106] The last task in this group is to *"provide a model-to-text transformation that outputs the GreetingMessage of a Greeting together with the name of the Person to be greeted. For instance, the model given in Fig. 13 should be transformed into the String "Hello TTC Participants!"* [106]

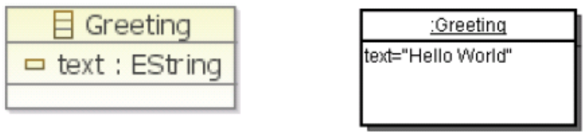


Fig. 12. The "Hello World" metamodel and the example instance [106]

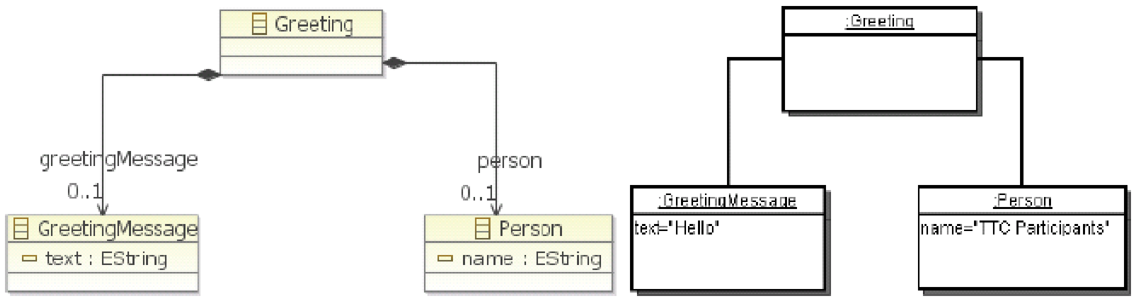


Fig. 13. The extended "Hello World" metamodel and the example instance [106]

In these transformations the MOLA pattern used is very similar to the corresponding instance diagram given in the task specification. Greeting transformations are given in Fig. 14, Fig. 15 and Fig. 16. The transformation logic for these tasks is described by using one MOLA rule (the grey rounded rectangle). The only requirement in the first two tasks is to create elements (marked with red dashed lines). In the third task an instance of the class *"StringResult"* is created, if the pattern (the elements with black solid lines) is matched with the MOLA rule.

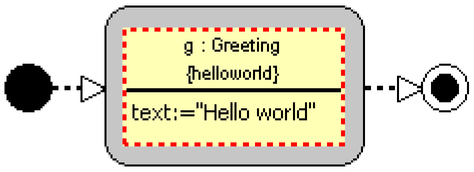


Fig. 14. Transformation creating a constant *Greeting* instance

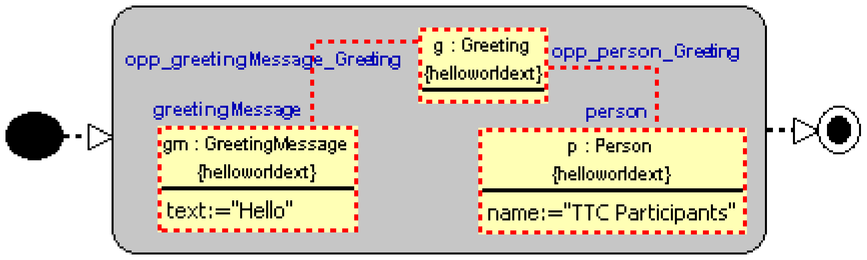


Fig. 15. Transformation creating a constant *Greeting* instance with references

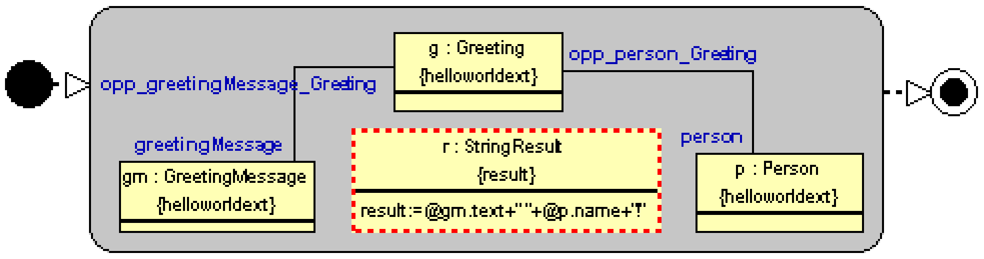


Fig. 16. Model-to-text transformation creating a greeting message

2.4.2 Instance Counting

The next group of tasks in the task specification is the instance counting tasks. The input models are simple graphs conforming to the metamodel given in Fig. 17 [106]. The task specification is as follows

- *"Provide a model query that counts the number of nodes in a graph.*
- *Provide a model query that counts the number of looping edges in a graph, i.e. edges where the source and the target node coincide.*
- *Provide a model query that counts the number of isolated nodes in a graph, i.e. nodes that are neither the source nor the target of any edge.*
- *Provide a model query that counts the number of matches of a circle consisting of three nodes, i.e. the pattern shown in Fig. 18 where n1, n2 and n3 are pairwise distinct. Note that each circle in the model should be matched three times.*
- *Optional: Provide a model query that counts the number of dangling edges in a graph, i.e. edges where either the source or the target node is missing."* [106]

Transformation counting nodes in a graph is given in Fig. 19. Transformation counting looping edges is given in Fig. 20. Transformation counting isolated nodes is given in Fig. 21. In MOLA the counting is implemented by using an integer counter and a foreach loop (a rectangle with a bold border) where the counter is increased. In most cases the loophead pattern directly specifies the set of instances to be counted.

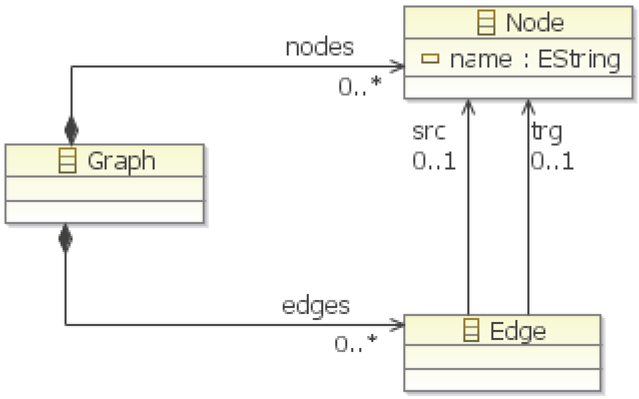


Fig. 17. The simple graph metamodel [106]

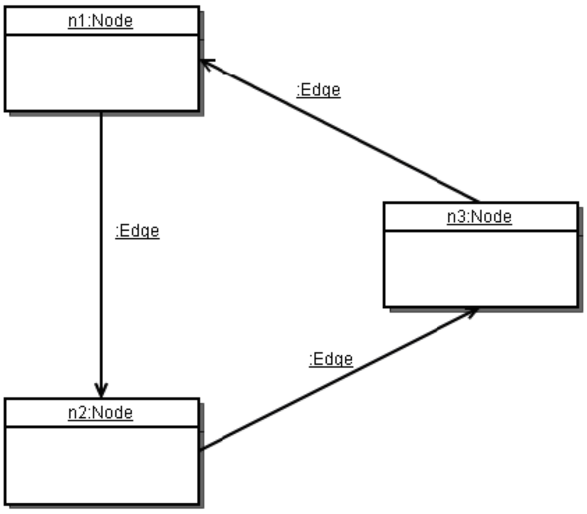


Fig. 18. Circle of three nodes (simplified representation of edge objects) [106]

A MOLA variable “*sk*” (a white rectangle) of type integer is used as a counter. Each loop iteration increases the instance count by one. Text statements (yellow rounded rectangles) are used to modify the values of the counter. Finally, to save the counting result in the resulting model the MOLA rule creating an instance of the class *"IntResult"* is used.

For all these tasks it was required to count elements in a graph. As it was not defined whether the model contains only one graph or multiple graphs, we admitted the worst case of many graphs in the model. For transformations to work properly when there is more than one graph in a model we provide the graph to be processed as a parameter. Consequently, we use another MOLA procedure where we iterate through all graphs in a model (using a foreach loop) and from here we call the transformation (using the call statement) for processing the current graph. An example of such transformation is given on the left side of Fig. 19. (The only thing that changes is the called procedure.) A similar graph processing is done for all tasks where the phrase "in a graph" is used. If there is always only one graph in a model this step could be omitted. The same could be said about transformations in Fig. 25- Fig. 32 as well.

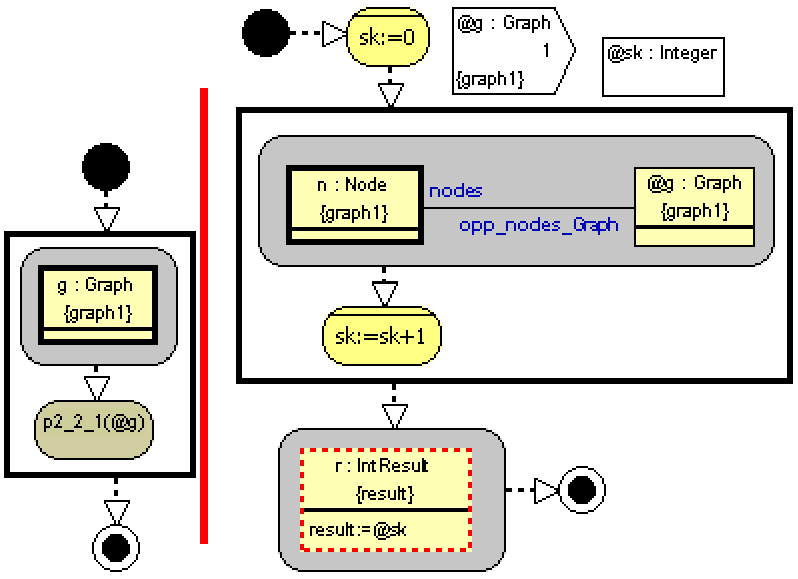


Fig. 19. Transformation counting nodes in a graph

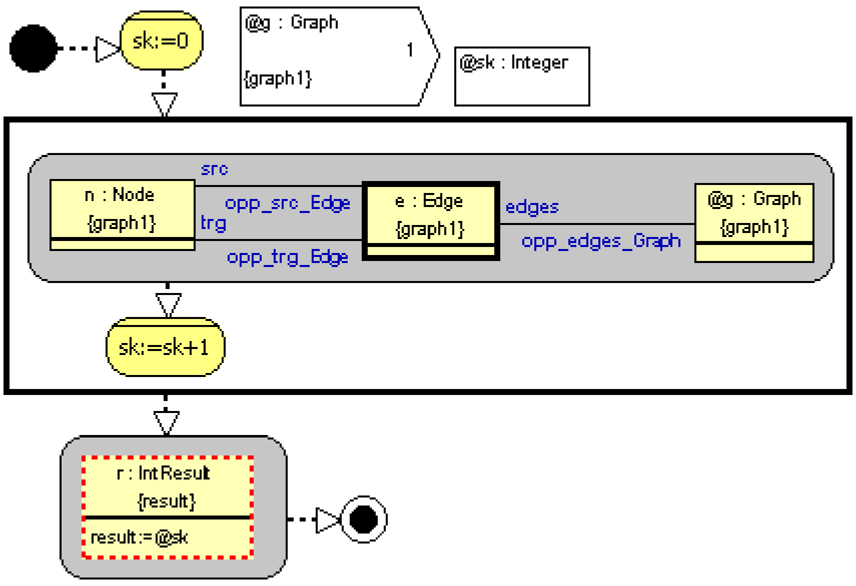


Fig. 20. Transformation counting looping edges in a graph

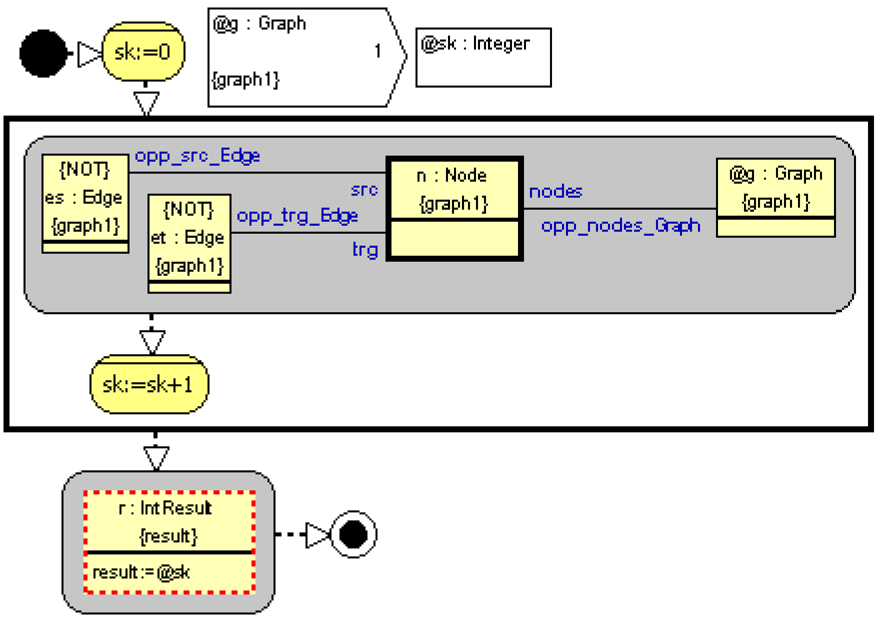


Fig. 21. Transformation counting isolated nodes in a graph

The only counting task, processed differently, is the circle counting. In MOLA there are two loop types: the foreach loop and the *while loop* (rule + appropriate control flow). In the while loop, to ensure only distinct matches, an explicit marking of the already found matches (using a NAC construct) is required, claiming the usage of temporary metamodel elements to solve the task. An alternative is to use three nested foreach loops, since multiple loop variables are not supported in MOLA. We provide solutions using both loop types as each has some advantages and disadvantages.

We start with the solution using the foreach loop, as this loop type was used in the previous tasks. The solution of this task is different from the previous one because we want to find all different circles. In this case one loop variable is not sufficient and, consequently, several loops are required.

The task specification did not clearly state whether graphs or multi-graphs should be considered (i.e., is it possible to have multiple edges between two nodes.) As the provided metamodel supports multi-graphs and graphs are a subclass of multi-graphs, we decided to build our solution, providing support to multi-graphs. This being the case, if there is a circle "*n1;n2;n3*" and two edges between "*n1*" and "*n2*", then there will be two circles "*n1;n2;n3*" (and 2*"*n2;n3;n1*" + 2*"*n3;n1;n2*"). The solution of this task is given in Fig. 22. To distinguish different edges between the same nodes, the edges are used as loop variables. There are three nested loops used in the solution. Each loop selects one edge for the circle. Actually, finding of circles is defined in the loophead of the first loop, however, when using this loop we are only able to find all edges which are a part of some circle, but we do not have information in how many circles this edge is used. Adding the second and the third loop we count all circles that have different edges three times, as required in the task specification.

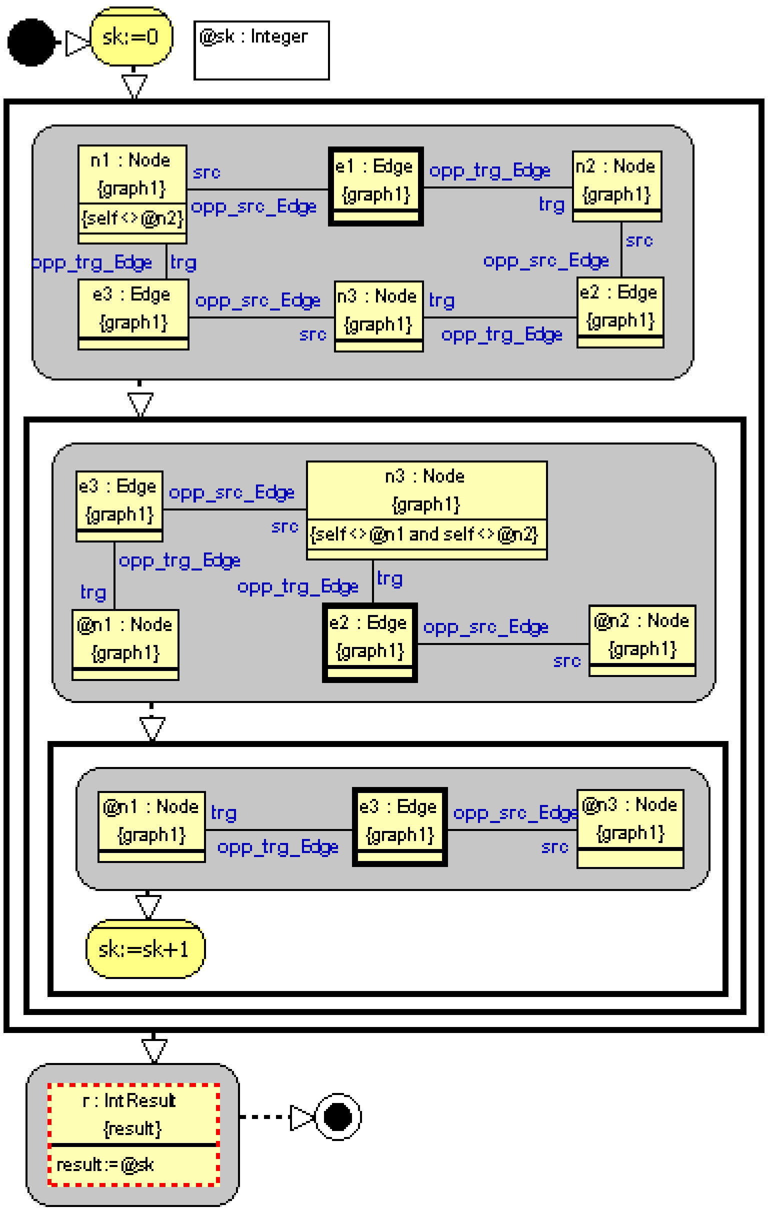


Fig. 22. Transformation counting circles consisting of three nodes

If we know that there are no multi-graphs, then the last loop can be omitted because the existence of the third edge is already validated by the patterns in the first and the second loop. However, understanding of this case is probably easier if nodes are used as loop variables, but anyway three loops are needed again.

Solving of the task by using the foreach loop is quite lengthy; however, if we add temporary classes it is possible to create a shorter and more elegant solution. In this case we will use the *while loop*. We extend the metamodel by adding the temporary class *"Circle"* and connecting it to the class *"Edge"*. The metamodel extension is shown at the bottom of Fig. 23. If such extended metamodel is used then we can simply write a MOLA rule looking for circles and marking the found circles: connecting all edges of a circle to a new instance of the "Circle" class. To ensure that each circle is found exactly once a NOT constraint (an equivalent to NAC in graph transformation languages, e.g., in AGG [163]) is used, stating that this circle has not been marked previously. As in this solution we do not care about the order of edge finding, the loop counter is increased by 3, to ensure that each circle has been counted three times. The above mentioned solution is presented in Fig. 23.

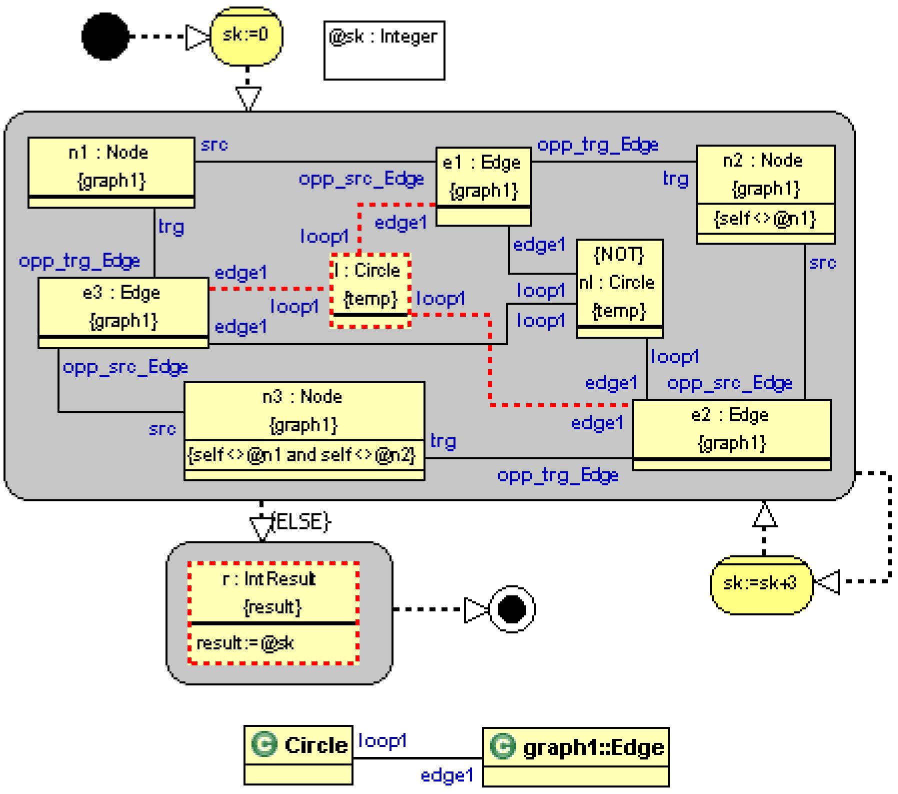


Fig. 23. Transformation counting circles consisting of three nodes, using temporary metamodel elements

Next was an optional task to count the dangling edges. The solution is given in Fig. 24. In this case two loops are used. The first one counts the edges without a source.

To ensure that the edges without a source and without a target are counted only once the second loop counts only the edges with a source and without a target.

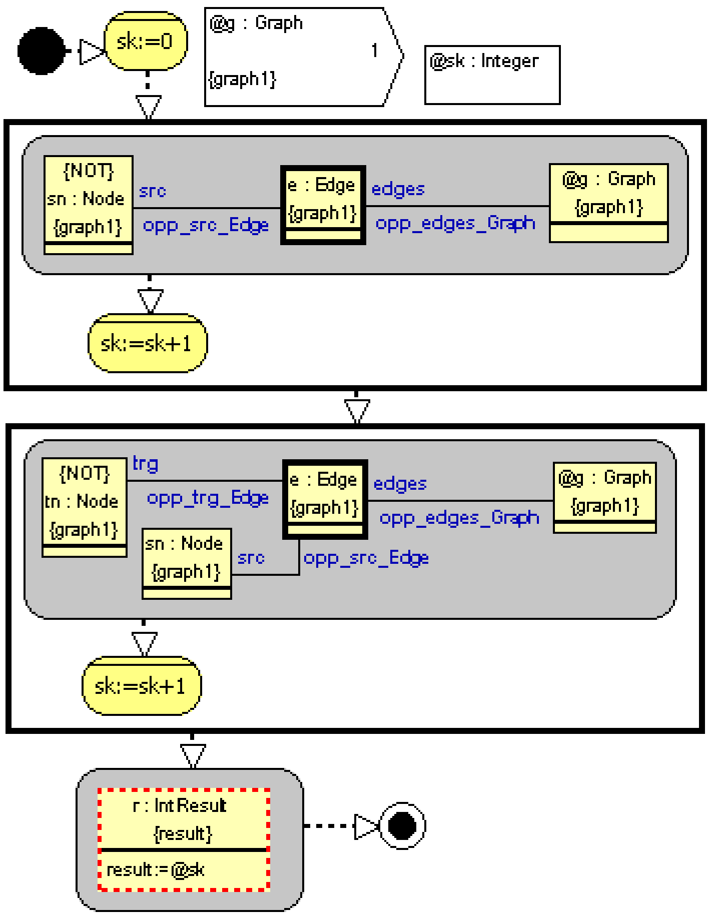


Fig. 24. Solution of optional task: counting of dangling edges

2.4.3 Reversion

The next task to be considered is edge reversing. It was required to *"provide a transformation that reverses all edges in a graph conforming to the simple graph metamodel given in Fig. 17 (p.58). This is an update operation."* [106]

We selected a solution where a new reverted edge is created and the old edge is deleted (delete is marked by using a black dashed line). The solution is displayed in Fig. 25. Actually, a shorter solution in MOLA is possible; however, it is not supported by the current version of the MOLA tool.

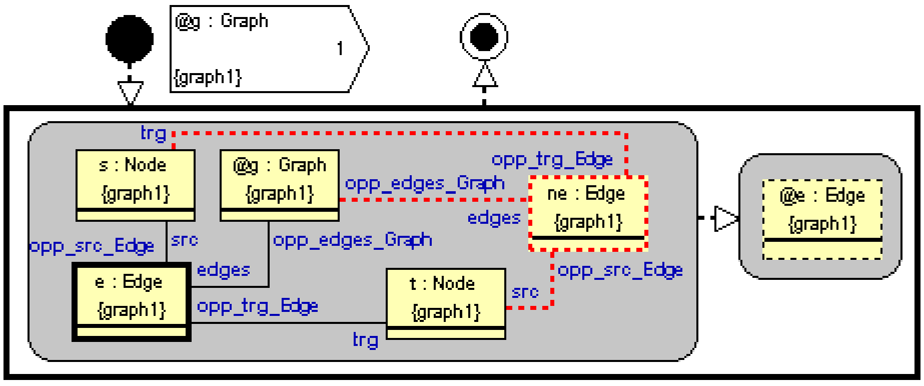


Fig. 25. Transformation inversing edges

2.4.4 Model Migration

The next group of tasks was model migration tasks. The first task was to *"provide a transformation that migrates a graph conforming to the metamodel given in Fig. 17 (p.58) to a graph conforming to the metamodel given in Fig. 26. The name of a node becomes its text. The text of a migrated edge has to be set to the empty string."* [106]

The second optional task was to *"provide a topology-changing migration that transforms graphs of the metamodel given in Fig. 17 (p.58) to graphs as defined by the metamodel in Fig. 27."* [106]

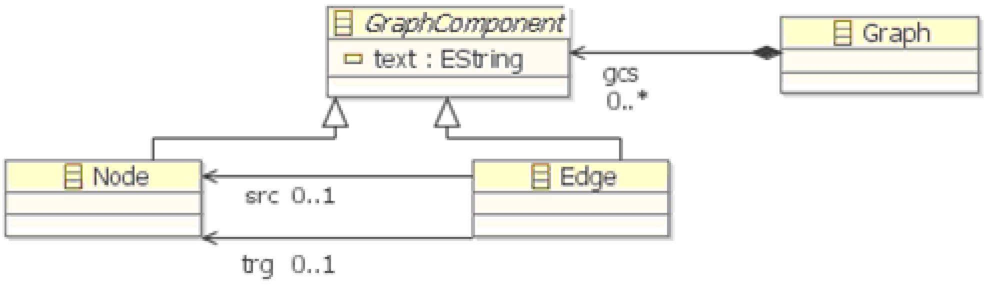


Fig. 26. The evolved graph metamodel [106]

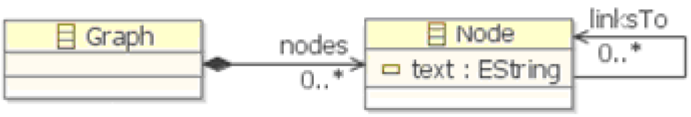


Fig. 27. The even more evolved graph metamodel [106]

Implementation of such tasks requires adding of temporary traceability relations to the metamodel. In this case it is sufficient to have an association between nodes in both metamodels (see Fig. 28). The migration transformation from the metamodel *graph1* to the metamodel *graph2* is given in Fig. 29 and from the metamodel *graph1* to the metamodel *graph3* in Fig. 30. At first a new graph in the target model is created in both cases. After that all nodes are cloned and traceability links added. (To ensure it a foreach loop iterating through all nodes in the source graph is used.) Finally, all edges are transformed by using the traceability information to find the appropriate source and target nodes in the migrated model. (To ensure it a foreach loop iterating through all edges in the source graph is used.)

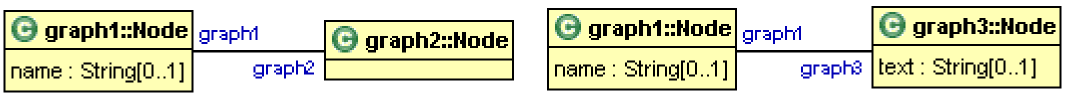


Fig. 28. Metamodel extensions for model migration tasks

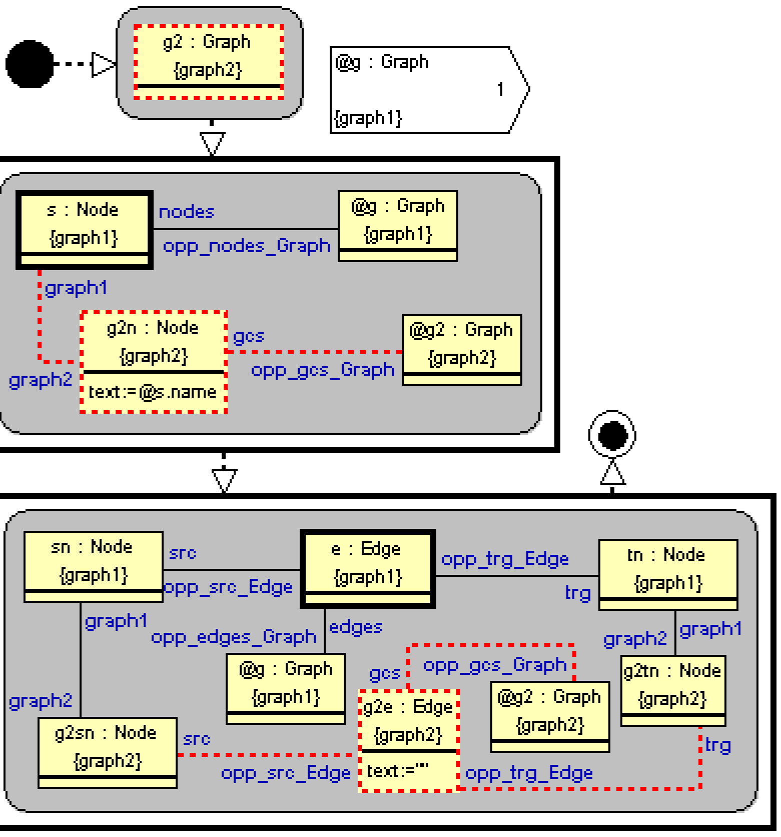


Fig. 29. Model migration transformation. Migrates graph from encoding *graph1* (Fig. 17) to encoding *graph2* (Fig. 26).

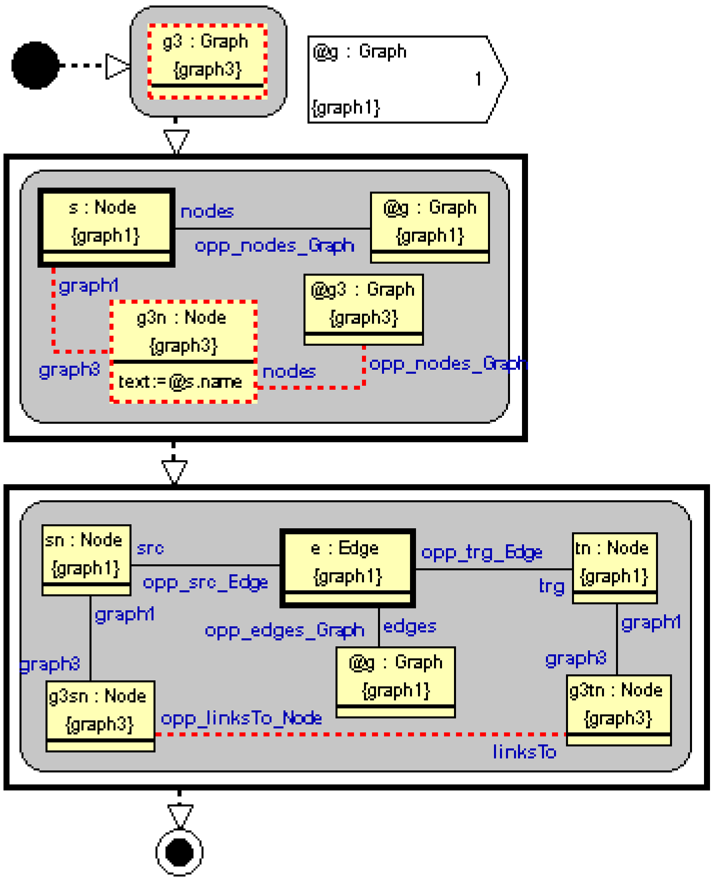


Fig. 30. Solution of optional model migration task. Migrates graph from encoding *graph1* (Fig. 17) to encoding *graph3* (Fig. 27).

2.4.5 Deletion Tasks

Deletion tasks constitute the last group of tasks. The task definition was as follows:

"Given a simple graph conforming to the metamodel of Fig. 17 (p.58), provide a transformation that deletes the node with name "n1". If a node with name "n1" does not exist, nothing needs to be changed. It can be assumed that there is at most one occurrence of a node with name "n1".

Optional: Provide a transformation that removes the node "n1" (as above), but also all its incident edges." [106]

The last mandatory transformation is deletion of the node named "n1". This transformation is very straightforward (see Fig. 31). We try to find such a node by using a MOLA pattern and delete it, in case of finding it. Deletion is represented by a black dashed line. It was required to delete all incident edges in the extension as well. The solution of extension is given in Fig. 32. In this case the sequence of deletions is as follows – at first the node is found, all outgoing edges deleted, followed by deletion of all incoming edges and finally the node itself is deleted.

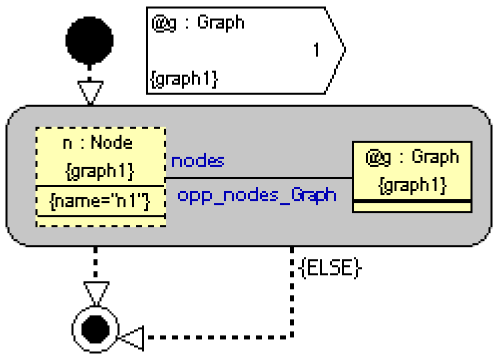


Fig. 31. Transformation that deletes the node named "n1" (if such a node exists) in a graph

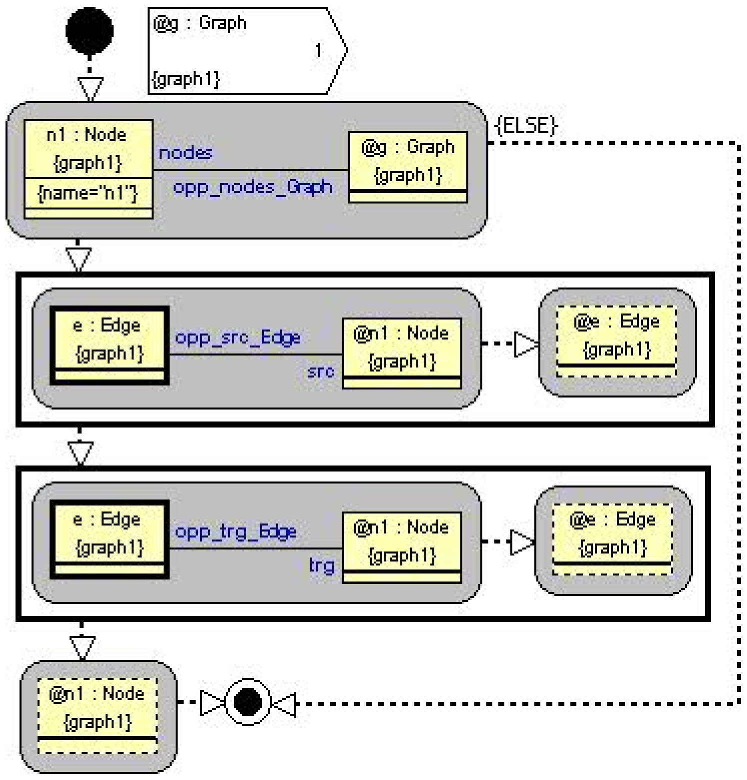


Fig. 32. Transformation that deletes the node named "n1" (if such a node exists) and its incident edges in a graph

2.4.6 MOLA Tool Support

This section describes the technical details regarding the solution of the task.

MOLA has an Eclipse-based graphical development environment (*MOLA tool* [59]), incorporating all the required development support. A transformation in MOLA is compiled via the low-level transformation language *L3* [13] into an executable Java code which can be run against a runtime repository containing the source model. For this case study *Eclipse EMF* is used as such a runtime repository, but some other repositories can be used as well (e.g., *JGraLab* [64], *mii_rep* [11]).

The MOLA tool has a facility for importing existing metamodels, in particular, in EMF (*Ecore*) format. Though the MOLA metamodelling language (*MOLA MOF*) is very close to EMOF, and consequently *Ecore*, there are some issues to be solved. The current version of MOLA requires all metamodel associations to be navigable both ways (this permits to perform an efficient pattern matching by using simple matching algorithms). Since a typical *Ecore* metamodel has many associations navigable one way, the import facility has to extend the metamodel. Another issue is the variable coding of references to primitive data types.

Metamodel import facilities in MOLA are able to perform all these adjustments automatically. In such a way the provided metamodels were imported into the MOLA tool. Transformation development of some tasks in MOLA requires additional metamodel elements, for example, in migration tasks to store relations between the source and target models. These metamodel elements have to be added manually. In migration tasks, these are the associations between the node classes in different graph encodings.

Since the metamodels have been modified during import, the original source model does not conform directly to the metamodel in the repository mainly due to the added association navigability. Therefore a source model import facility is required. The *MOLA execution environment* (*MOLA runner*) includes a generic model import facility, which automatically adjusts the imported model to the modified metamodel. Now the transformation can be run on the model. Similarly, a generic export facility automatically strips all elements of the transformed model which do not correspond to the original target metamodel. Thus, a transformation result is obtained which directly conforms to the target metamodel. (For an *inplace* transformation the source and target metamodels coincide, as a result nothing has to be stripped.) The transformation user is not aware of these generic import and export facilities, he/she directly sees the selected source model transformed.

An executable version of the solution is available online, using the SHARE [186] system. A SHARE image of the solution is provided in [4]. By using the SHARE image a reader can access an executable version of this case study. All transformation sources are available in the transformation definition environment. It is also possible to compile and execute all “Hello World” transformations in MOLA. To access the SHARE image a reader should register in the SHARE system and require access to the SHARE image in [4]. When the access is granted a reader should connect to the SHARE server by using *Remote Desktop Protocol* (RDP). It is possible to work with a copy of the image, using a remote desktop connection.

2.5 MOLA Metamodel

In CHAPTER 6 the Template MOLA language is defined. This language is based on MOLA. To facilitate a reader’s understanding of the Template MOLA language the MOLA metamodels are provided in this section.

As already mentioned above the transformation definition in MOLA consists of a metamodel definition and a transformation procedure definition. The metamodel of MOLA MOF, MOLA meta-modelling language is given in Fig. 33. This package of the MOLA metamodel is named “*Kernel*”. The metamodel of MOLA procedure elements is given in Fig. 34. This package is named “*MOLA*”.

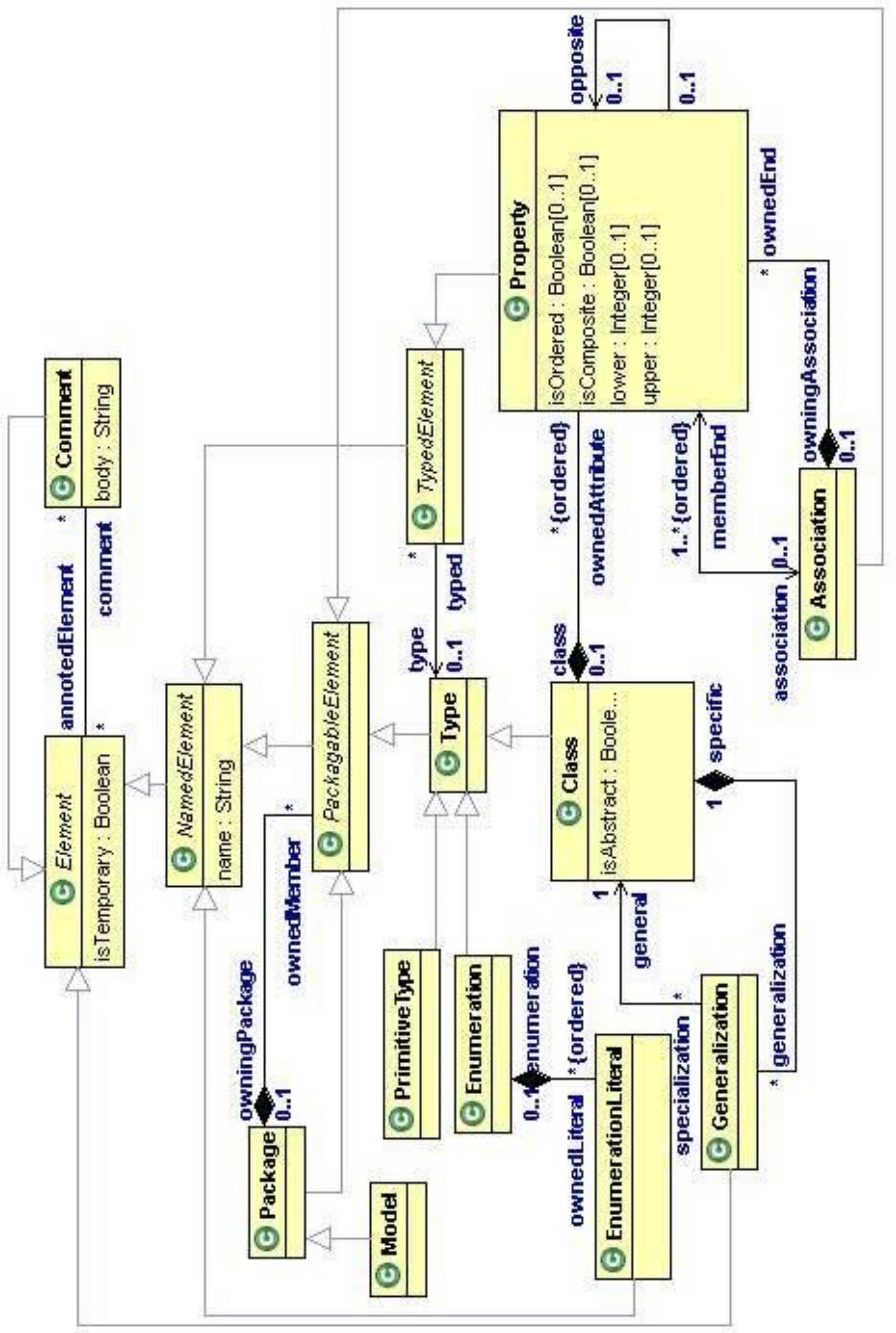


Fig. 33. The metamodel of the MOLA meta-modelling language [130]

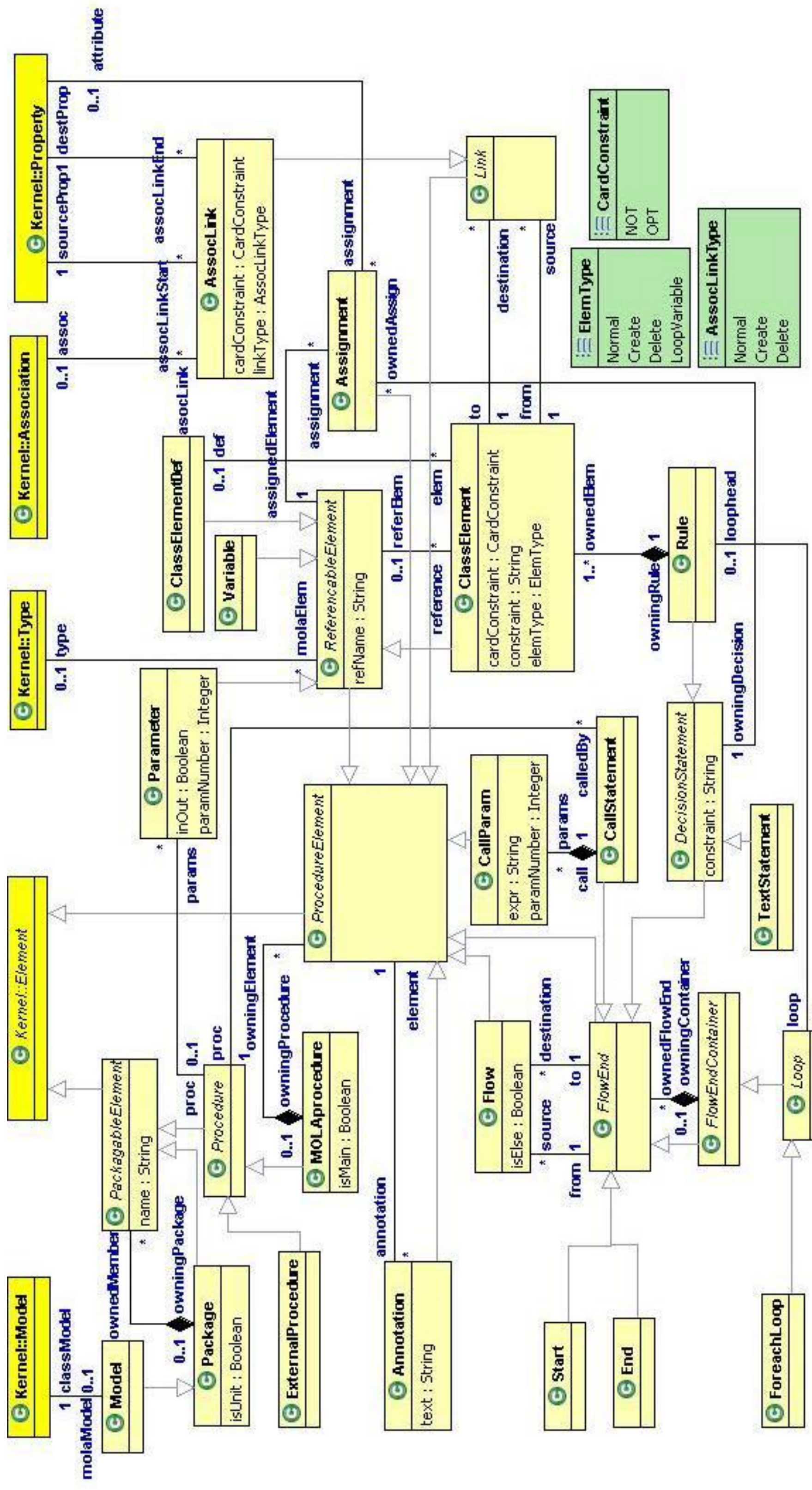


Fig. 34. The metamodel of the MOLA procedure elements [130]

CHAPTER 3

Transformations for Model-Driven Development in ReDSeeDS

In this chapter transformations for Model-Driven Software Development are analyzed. Transformations described in this chapter are developed within the ReDSeeDS project [3] therefore a short overview on ReDSeeDS seems appropriate. Further on Requirements Specification Language (RSL) used in the ReDSeeDS project is described, this being the entry point for transformations. General principles regarding MDD in ReDSeeDS are outlined in Section 3.3, continued with the description of two transformation supported paths from the requirement to the code (Section 3.4 and 3.5). These paths are based on different architecture styles. The chapter is concluded with implementation aspects in Section 3.6 and conclusions in Section 3.7.

3.1 ReDSeeDS Overview

Requirements Driven Software Development System (ReDSeeDS) [3, 38] is an EU funded project (Contract No. IST-2006-33596 under 6FP). The project was realized from September 2006 till December 2009 and it was coordinated by Infovide (Poland) with the technical lead of Warsaw University of Technology (Poland) and University of Koblenz-Landau (Germany); Vienna University of Technology (Austria); Fraunhofer-Gesellschaft (Germany); Institute of Mathematics and Computer Science, University of Latvia (Latvia); Hamburger Informatik Technologie Centre e.V., University of Hamburg (Germany); Heriot-Watt University (United Kingdom); PRO DV Software AG (Germany); C/S Enformasyon Teknolojileri Limited Sirketi (Cybersoft, Turkey) and Algoritmu Sistemos (Lithuania).

The author of the thesis was involved in the project from January 2007 till the end of the project (December, 2009). The author’s responsibility was to develop model transformations to support a full model-driven path from the requirements to the code. The author of the thesis participated in the development of 11 project deliverables [65, 83, 84, 81, 94, 19, 8, 136, 134, 135, 151], as well as in the preparation of 4 publications related to the project results [79, 80, 153, 82].

The motto of the ReDSeeDS project was as follows: *"Fulfilling the promise of comprehensive software reuse by bringing it to the level of requirements linked with precise model-based solutions."* [3]

"The main objective of the project is to create an open framework consisting of a scenario-driven development method (precise specification language and process for the "how-to"), a repository for reuse and tool support throughout. The basic reuse approach will be case-based, where a reusable case is a complete set of closely linked (through mappings or transformations) software development technical artefacts (models and code), leading from the initial user's needs to the resulting executable application." [3]

The following were the main elements in the ReDSeeDS project:

- Use (and development) of formal and at the same time easily usable, understandable *Requirements Specification Langua*ge (RSL). (See Section 3.2.1.)
- Transformation supported model-driven part from the requirements to the code. All artefacts produced were related with traceability information.
- *Software Case Repository* for storing artefacts of past software cases (models and code).
- *Query engine* to find similar past software cases in the repository.
- *Slicing* to extract appropriate parts (models and code) from past software cases to the one under development.

In the ReDSeeDS project a prototype of the ReDSeeDS system was developed. The usage scenario of the system could be as follows:

- Requirements for the system to be built are sketched in RSL.
- The *Software Case Repository* is queried for similar past software cases. Sketched requirements are used as a query.
- A list of similar software cases is presented. For each software case a similarity coefficient is given. A user can analyze similar software cases.
- Similar slices are found and imported in the current software case. A slice is set of related elements from the requirements through all models to the code. The set of requirements is selected and all elements implementing these requirements (in the models and the code) are automatically added to the slice.

- Imported slices are adapted, if necessary.
- Requirements specification is improved, if necessary.
- The Model-Driven Software Development path described in Section 3.4 or Section 3.5 is applied.
- The developed software case is saved to the *Software Case Repository* for reuse.

If not needed, reuse of the previously defined software cases could be skipped. The usage scenario without reuse is as follows:

- Requirements for a system to be built are specified in RSL.
- The Model-Driven Software Development path described in Section 3.4 or Section 3.5 is applied.

In the thesis a model-driven path from the requirements to the code skipping reuse aspects will be described in detail. It should be noted that the ReDSeeDS approach for Model-Driven Software Development has a value of its own even without reuse aspects. It is a real example of a MDSD path with several models. Most of MDSD approaches proposed in commercial tools use only one model, e.g. *Model2code* [21]. In commercial tools there is no path from the requirements. Typical MDSD approaches in these tools start with PIM. Nevertheless, there exist other approaches starting from the requirements, e.g. [101].

We will start with a short description of *Requirements Specification Language*, continued with a discussion of the possible use of these requirements in a model transformation supported path from the requirements to the code. Two different model-driven paths supporting different *architecture styles* will be considered.

3.2 Requirements Specification in ReDSeeDS

In the ReDSeeDS project *Requirements Specification Language* (RSL) was introduced. In this section RSL is described and the usage of RSL is demonstrated.

3.2.1 Requirements Specification Language in ReDSeeDS

RSL [66, 65, 152] is a semiformal language for specifying requirements for a software system. The elements of RSL which can be directly transformed into the system design are described below.

RSL employs use cases for defining precise requirements for the system behaviour. Each use case is detailed by one or more scenarios, in turn consisting of special controlled natural language sentences. The main sentence type is the *SVO(O)* sentence [152], consisting of a subject, a verb, and a direct object (optionally, also an indirect object). These sentences express the actions to be performed in the scenario. In addition to SVO(O), there can also be conditions, rejoin sentences ("*gotos*" to a point in the same or another scenario) and invoke sentences (invoke another use case). Alternatively, the set of scenarios for a use case can be visualized in a natural way as a profile of an UML activity diagram. SVO(O) sentences serve as the nodes of the diagram, and conditions and rejoins as control flows (in addition to the natural "next sentence" control flow).

Another part of RSL is the domain definition which consists of actors (system users), system elements, and notions. A reader may think about actors and system elements as actors in UML use case diagrams. Notions correspond to the elements (classes) of the conceptual model of the future system. It is also possible to define notion generalization and simple associations between notions. In the second version of RSL [65] it is possible to define one notion as an attribute of another notion. Actually, the notion part in the second version of RSL describes a conceptual model of the system to be built, only alternative syntax is used instead of traditional class diagrams.

The precise syntax of RSL is defined by means of a metamodel [66]. All elements of requirements specification in RSL are stored as model elements, corresponding to the metamodel. Even *SVO(O)* sentences are processed as model elements, although they seem to be a plain text to a user. The behaviour and domain parts in a valid RSL requirements model must be strictly related. The subject of a SVO(O) sentence must be an actor or a system element. An object (direct or indirect) must be a notion. In principle, an object is an element of the conceptual model, affected by the action described in a SVO sentence.

A SVO(O) sentence is given in Listing 1. The syntax used in the RSL editor is used here. In this sentence the nouns (or noun phrases) - user, facility, reservable facility list - are coloured blue, the verb (selects) is coloured red. The preposition (from) is coloured green. "User" is the subject of the sentence. In this case the actor is used as a subject. "Facility" is a direct object. "Reservable facility list" is an indirect object. For both objects the notions should be defined in RSL.

Listing 1. SVO(O) sentence

User *selects* **facility** from **reservable facility list**

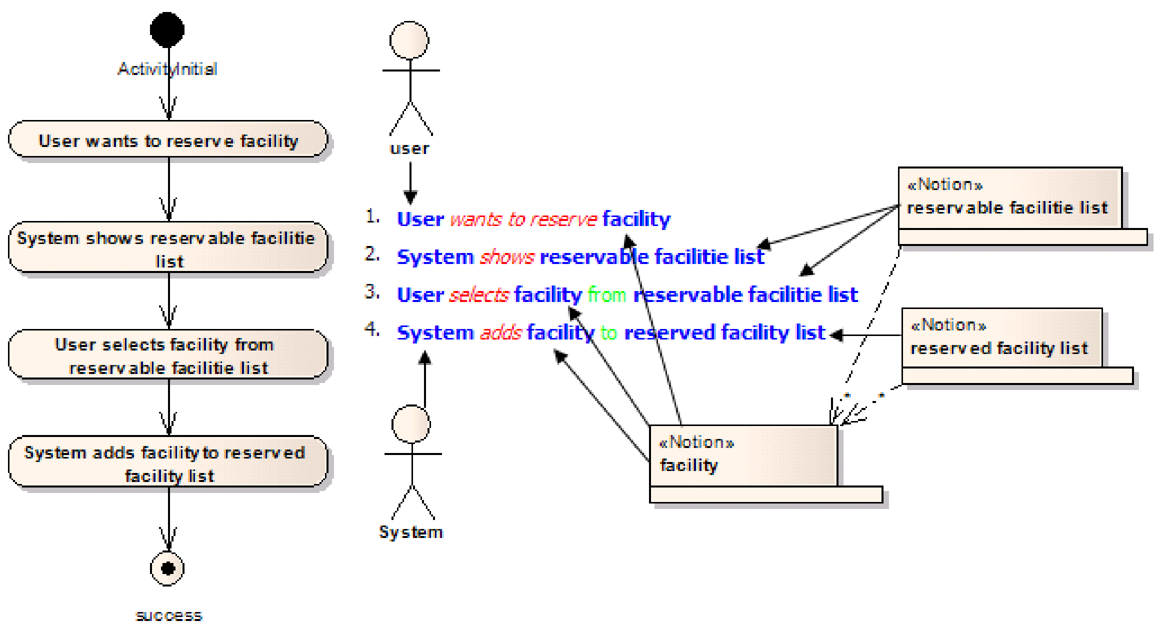
The informal meaning of each noun and verb must be defined in a vocabulary (currently, *WordNet* [41]). In the ReDSeeDS tool support it is possible to extend *WordNet* by adding new words and new meanings. Typically complex notions as "reservable facility list" should be added manually to the vocabulary which is used as the domain dictionary, describing the meaning of domain terms. In addition to the vocabulary keywords are introduced in the second version of RSL. Compared to the vocabulary, elements for keywords predefined semantics is introduced in the RSL profile (see Sections 3.3.2 and 3.5.3).

3.2.2 Example of Requirements

The proposed ideas are illustrated on a fragment of an example of the Fitness Club system. One use case *Reservations* is taken – how a club customer can book regular access to the selected fitness facility of the club. A simple example of this type is given in Fig. 35. The activity diagram representation of the requirements is on the left side of the figure. The right side of the figure contains the textual representation of the requirements where notions and system elements, related to the SVOO sentences, are also given.

This scenario consists of four consecutive SVO sentences. *Actor* or *system element* is used as the subject of these sentences. There is one *actor* "User" and one *system element* "System". There are three *notions* "facility", "reservable facility list" and "reserved facility list". They are used as direct and indirect objects in the SVO(O) sentences.

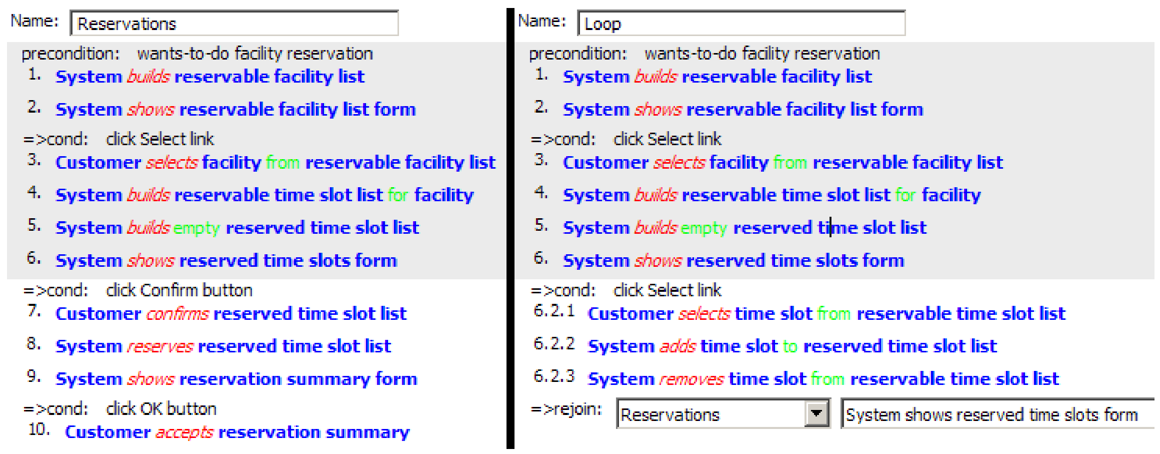
Textual representation was used as the main representation of the requirements in ReDSeeDS. The colour marking in the textual representation of the requirements helps to distinguish more clearly the parts of the SVO(O) sentences – subjects, verbs, and objects. The subjects and the objects are blue. The verbs are red. The prepositions preceding the indirect object are marked green. The whole following group of words marked blue is an object with a complex name (there must be an equally named notion in the domain part of the requirements). Note that in the textual syntax, each scenario is one continuous path in the diagram.



ActivityInitial
User wants to reserve facility
System shows reservable facilitie list
User selects facility from reservable facilitie list
System adds facility to reserved facility list
success
user
1. User wants to reserve facility
2. System shows reservable facilitie list
3. User selects facility from reservable facilitie list
4. System adds facility to reserved facility list
System
«Notion» reservable facilitie list
«Notion» reserved facility list
«Notion» facility

Fig. 35. RSL example

Fig. 36 provides a fragment from a more elaborated example from this use case displaying two scenarios from it. They are given in a textual form as they were entered by using the RSL editor. This to be a correct requirements model, the relevant notions must also be defined (facility, reservable facility list, etc.). The activity diagram for this use case is given in Fig. 41 (p.90).



Name: Reservations

precondition: wants-to-do facility reservation
1. System builds reservable facility list
2. System shows reservable facility list form
=>cond: click Select link
3. Customer selects facility from reservable facility list
4. System builds reservable time slot list for facility
5. System builds empty reserved time slot list
6. System shows reserved time slots form
=>cond: click Confirm button
7. Customer confirms reserved time slot list
8. System reserves reserved time slot list
9. System shows reservation summary form
=>cond: click OK button
10. Customer accepts reservation summary

Name: Loop

precondition: wants-to-do facility reservation
1. System builds reservable facility list
2. System shows reservable facility list form
=>cond: click Select link
3. Customer selects facility from reservable facility list
4. System builds reservable time slot list for facility
5. System builds empty reserved time slot list
6. System shows reserved time slots form
=>cond: click Select link
6.2.1 Customer selects time slot from reservable time slot list
6.2.2 System adds time slot to reserved time slot list
6.2.3 System removes time slot from reservable time slot list
=>rejoin: Reservations | System shows reserved time slots form

Fig. 36. Requirements – two scenarios in a textual form

3.3 Model-Driven Development in the ReDSeeDS Project

The ReDSeeDS approach covers a complete chain of models for model-driven development – from the requirements to the code. Each transition in this chain is to a great degree assisted by formal model transformations. Although two specific chains of models are described here, the approach could be applied to any similar setting of models.

The first in the both chains is the Requirements Model built in a special semiformal requirement language RSL (described in Section 3.2). The required behaviour specification in this controlled natural language is defined by the model; therefore, this specification can be processed by model transformations in order to generate initial versions of the next models.

Both architecture styles, implemented in the ReDSeeDS project, contain "Architecture" and "Detailed Design" models corresponding to PIM and PSM in the MDA approach. In the *Keyword-Based Style* additional "Analysis" model between "RSL" and "Architecture" (PIM and CIM) is used. All transitions between the models are assisted by model-to-model transformations.

It should be noted that we use for our models a pre-selected consistent set of design patterns and other design rules, called an architecture style in our approach (this concept is described in Sections 3.3.1 and 3.5.2). Transformations are adjusted to this style to get maximum results in extracting the required behaviour from RSL. The best results are obtained if the requirements are specified in RSL in an appropriate way – there is used an RSL profile, associated with the architecture style (see Section 3.5.3).

All model-to-model transformations in our approach are implemented in the model transformation language MOLA [76]. If the selection of patterns and the architecture style are changed, the transformations should be rebuilt, too.

Another issue to be solved by transformations is the inevitable modifications of models and the necessity to reapply the transformations and merge the results. Transformation development is discussed in Section 3.6.

3.3.1 Design Patterns and the Architecture Style

Today large enterprise systems are developed by using a set of *design patterns* as a rule. There are two types of design patterns: *platform-independent* and *platform-specific*. The traditional *GoF design patterns* [47] represent the former type. The modern Java EE environments (based on the POJO [158] idea and declarative ORM) also share a large set of common enterprise patterns (and so do the latest .NET environments based on POCO [114]). On the other hand, low level patterns, such as an adequate usage of *Spring* framework annotations, are still platform-specific.

Usage of design patterns is vital to efficient application of MDD and transformations. However, patterns alone are not sufficient for deciding what the generated models look like. Therefore, we use the concept of *architecture style*, which includes the structure of the system and the model, a related set of design patterns (with indications where they should be used), the applied general design principles, and finally, the rules by which model elements are obtained from the models preceding in the development chain. This last feature is formalized by a model transformation set associated with the architecture style. The most important content of an architecture style is the selected set of design patterns, tied up to the chosen model structure. Namely, patterns are the style element which helps most in specifying efficient transformation rules. In addition, for transformations supporting the given architecture style to produce maximum results, the requirements must be specified in an appropriate style, too; therefore, the concept of RSL profile (associated with the given architecture style) is introduced.

Two different architecture styles are considered in the thesis: the *Basic style* (see Section 3.4) and the *Keyword-Based Style* (see Section 3.5).

The goal of the *Basic Style* is to prove the feasibility of the approach in which the model-driven development, starting from the requirements, is combined with the requirement-based reuse of software. The initial version of the ReDSeeDS tool support was based on this style. However, the possibilities to extract behaviour from the requirements in the *Basic Style* are significantly weaker than in the *Keyword-Based Style*.

The main goal of the *Keyword-Based Style* is to extract as much as possible behaviour from the requirements. The in-depth analysis of requirements is based on keywords to be found in the RSL sentences which the style is named after. The RSL profile associated with the *Keyword-Based Style* is described in Section 3.5.3.

In no case the described architecture styles should be considered the only possible solutions; other styles are also possible. To a great degree, the choice of the most appropriate architecture style depends on the domain of the system to be created. For example, the *Keyword-Based Style* could be an adequate solution for simple web-based information systems. The selection of architecture style could be formalized on the basis of non-functional requirements for the system; however, this topic is completely out of the scope of the thesis. Furthermore, it should be reminded that creation of a new architecture style also requires creation of an appropriate transformation set.

3.3.2 The RSL Profile

Transformations can be applied to any valid set of requirements in RSL for a system. Nevertheless, in order to ensure that these transformations generate a really substantial fragment of the software system to be built, some more constraints on the requirements should be put. Thus, a concept of the RSL profile is introduced. The profile defines the set of keywords with predefined semantics to be used in the scenario sentences (verbs, nouns, and prepositions) and some rules on how these keywords should be used. Moreover, there are constraints on the order of these sentences (or nodes in the activity form). All these rules are “soft” rules in the sense that the requirements do not become invalid if they violate some of these rules; simply, the transformations can do less. At the same time, profiles are defined so that they never make the requirements less readable to domain area specialists (however, more skills may be required by requirement engineers to create them). A profile is always associated with an architecture style so that the corresponding transformation set can produce the largest possible part of the PIM and PSM models from the requirements.

When defining requirements, keywords are not specially marked, they are used as all other words in the scenario sentences. The same could be said about a specific order of sentences in an architecture style. So it is completely left to the requirements definer to follow or disregard these soft rules. The use of RSL profile is analysed only by transformations. If the rules are followed, the transformation produces more detailed models of the system to be built. If the rules are ignored, the following models in the model driven path are of lower quality. It means that more manual work is required for adding the missing information.

In the RSL profile for the ReDSeeDS *Basic Style* only assumptions about the order of sentences are used. The full power of the RSL profile mechanism is used in the *Keyword-Based Style*. A detailed description of the RSL profile and keywords used in this architecture style is given in Section 3.5.3.

3.4 ReDSeeDS Basic Style

This architecture style was defined by Warsaw University of Technology (Poland) and transformations for it were implemented by the author of the thesis. This was the first architecture style defined in the ReDSeeDS project.

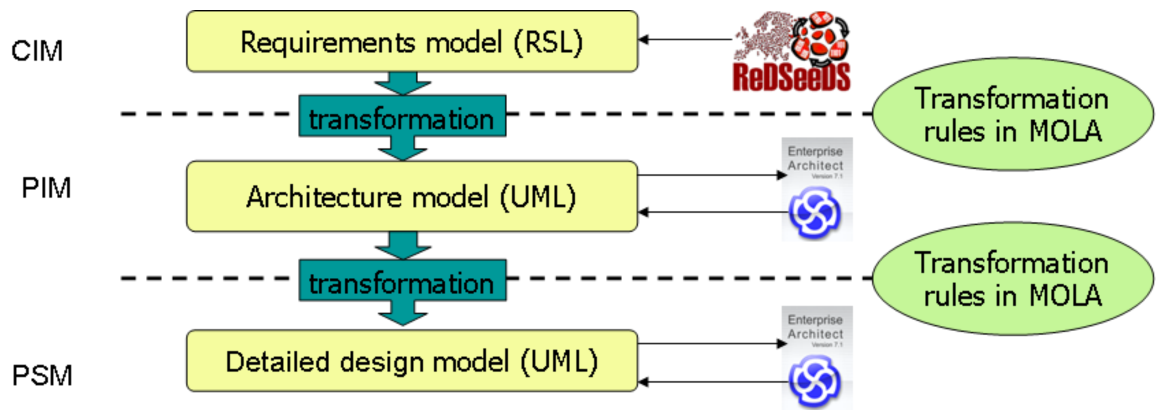


Fig. 37. Model chain in the ReDSeeDS *Basic Style*

The RSL profile for this architecture style has no keywords, only some constraints on sentences. The usage of this style has confirmed the feasibility of the used technologies and approaches; however, the part of a system, generated by transformations in this style, is small. The model chain used in this architecture style is presented in Fig. 37.

3.4.1 The Platform-Independent Model

The PIM model is going to be described in greater detail since its relation to the CIM model in RSL is the most interesting in our approach. PIM defines the static structure of the system to be built by means of classes, components and interfaces. Draft behaviour of the system is described by means of sequence diagrams.

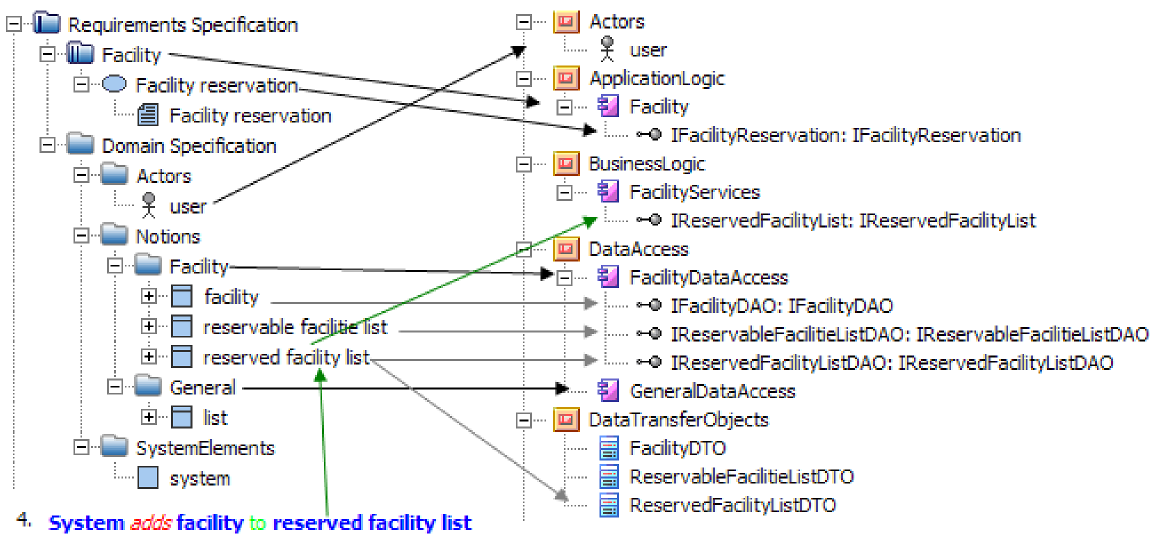
According to the chosen architecture style, a four-layer architecture is used with the following layers: *Data Access*, *Business Logic*, *Application Logic* and *User Interface*. Additionally, *Data Transfer Objects (DTOs)* are used as data containers for data exchange between the layers. Component and interface based design style is used at all layers. Components encapsulate groups of related elements of the system. Interfaces appear as provided interfaces of the respective components. The main patterns used in this architecture style are data access objects (DAO) for the *Data Access* layer and MVC for the *Application Logic* layer.

There are seven static structure packages in PIM, one for each layer, one for the DTOs, one for the Interfaces and one for the Actors. The package *Actors* contains actors of the system to be built. They are directly copied from the requirements. The package *Data Transfer Objects* contains DTOs created from notions. Each notion is transformed into one DTO class. Thus, this package serves also as a sort of conceptual domain model.

The package *Data Access* contains data access objects (DAO) for the persistence related operations. Each lowest level notion package is transformed in one DAO component. Each notion contained in this package is transformed into an interface of this component. The relevant CRUD (*create-read-update-delete*) operations are added for each interface.

The package *Business Logic* contains business level components and interfaces. Components and interfaces are created in the same way as in the *Data Access* layer. However, only notions, participating in business level operations, are used therein. In other words, only interfaces, containing business level operations, are created. Creation of the latter will be described together with the behaviour sequence diagram creation.

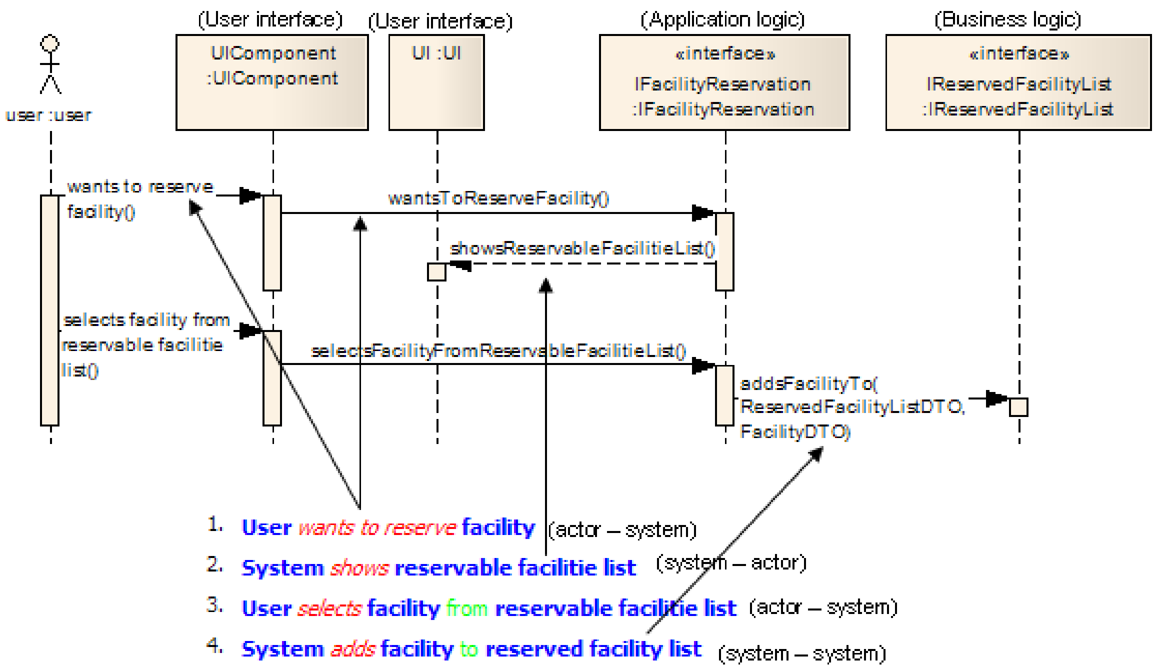
The packages *Application Logic* and *UI* are based on the MVC (model-view-controller) pattern. Components in *Application Logic* are created from use case packages of the lowest level in the package tree. Provided interfaces of these components are created from use cases written in RSL. One interface is created for each use case. Methods of these interfaces are created by analyzing the system behaviour. This will be described together with the sequence diagram creation. Currently, only a placeholder for the UI part is created. It could be replaced by a real UI support, but it is out of the scope of this chapter.



Requirements Specification
Facility
Facility reservation
Facility reservation
Domain Specification
Actors
user
Notions
Facility
facility
reservable facilitie list
reserved facility list
General
list
SystemElements
system
4. System adds facility to reserved facility list
Actors
user
ApplicationLogic
Facility
IFacilityReservation: IFacilityReservation
BusinessLogic
FacilityServices
IReservedFacilityList: IReservedFacilityList
DataAccess
FacilityDataAccess
IFacilityDAO: IFacilityDAO
IReservableFacilitieListDAO: IReservableFacilitieListDAO
IReservedFacilityListDAO: IReservedFacilityListDAO
GeneralDataAccess
DataTransferObjects
FacilityDTO
ReservableFacilitieListDTO
ReservedFacilityListDTO

Fig. 38. Static structure processing example

The above rules for generating the static structure of the system introduced in Fig. 35 (p.78) are illustrated in Fig. 38. On the left side of Fig. 38 the static structure of requirements in RSL (as in the RSL editor) is given. On the right side of Fig. 38 the static structure of PIM (*Architecture*, as displayed in EA) is shown. Both sides are connected with mappings relating the source and the target of some transformation rule described above. These mappings are similar to the ones used in the *Model Transformation by Example* (MTBE) approach [199]. However, when replacing concrete instances with patterns for finding relevant instances, a new mapping language could be obtained (similar to the one described in CHAPTER 4).



(User interface)
(User interface)
(Application logic)
(Business logic)
user :user
UIComponent :UIComponent
UI :UI
«interface» IFacilityReservation :IFacilityReservation
«interface» IReservedFacilityList :IReservedFacilityList
wants to reserve facility()
wantsToReserveFacility()
showsReservableFacilitieList()
selects facility from reservable facilitie list()
selectsFacilityFromReservableFacilitieList()
addsFacilityTo(ReservedFacilityListDTO, FacilityDTO)
1. User wants to reserve facility (actor – system)
2. System shows reservable facilitie list (system – actor)
3. User selects facility from reservable facilitie list (actor – system)
4. System adds facility to reserved facility list (system – system)

Fig. 39. Behaviour example

Certainly, the most complicated part is the description of the system behaviour. The sequence diagrams, describing the system behaviour, are created by analyzing scenario sentences. There can be three types of SVO sentences. The first one is an *actor – system* sentence. In this case the subject of the SVO sentence is the actor. For two other sentence types the subject of the sentence is a system element. The sentence types are distinguished by using the recipient link. Recipient is a SVO sentence element; it defines where to the behaviour described in the sentence is directed. The second type of the sentence is *system – actor*. In this case the subject is the system and the recipient is the actor. The third sentence type is *system – system*. In this case the subject and the recipient is the system. It is used to describe the internal actions of the system. The type of the particular message generated in the sequence diagram depends on the sentence type. Fig. 39 illustrates the behaviour sequence diagram of the example described above. It shows that the operations in the *Business Logic* layer are created only for the *system – system* sentences. The actor-system sentences are used for the creation of the *Application Logic* methods. UI methods are created from the *system – actor* sentences.

PIM can be manually extended after the initial generation. Afterwards it is transformed to PSM.

3.4.2 The Platform-Specific Model

The same four layers and DTOs are used in PSM. In this model the factory pattern is used, enabling the management of classes and interfaces. Each component in PIM is transformed into a package and a factory class in PSM. Every interface is transformed into an interface and an implementing class. Classes and interfaces are located in packages, created from components. Factory classes, created from components, have methods for getting provided interfaces. For each layer one more factory class is created. It manages all other factory classes in this layer.

The platform-specific model can be extended manually in the same way as the platform-independent one. Then this model can be transformed to the code.

Transformation, creating PSM, uses two transformation libraries. The copy library was used to copy DTOs from PIM to PSM. For the other layer the transformation library, converting components with its interfaces to factory classes, was used. Here the copy library for interfaces was used as well.

3.5 The Keyword-Based Style

This style is defined by the author of the thesis in cooperation with her supervisor and the UL IMCS ReDSeeDS team. All model-to-model transformations have been implemented by the author of the thesis. Model-to-text transformations in EA CTF have been implemented by Agris Šostaks.

In this section only the main ideas the Keyword-Based Style rests upon are outlined. A detailed description of the Keyword-Based Style model structures and transformation algorithms is given in Section 6.3 of ReDSeeDS deliverable D3.2.2 [84].

Introduction into the Keyword-Based Style starts with the description of the model and system structure and some general design rules. We have chosen a four-layer architecture because it is the most popular and accepted information system architecture style today. As already mentioned we use the following layers: *Data Access* or Repository layer, *Service* or Business layer, *Application Logic*, and *User Interface*. We also have domain objects as data containers (available to any layer, former DTOs [104]). Another general principle of our approach is based on a declarative *Object-Relational Mapping* (ORM). The particular ORM in our approach is Hibernate [16]. Whenever possible, we use an interface-based design style for all layers, meaning there is an interface (where the operations are specified) and its implementation class.

3.5.1 Models

In this section we present a short rationale behind our selection of the specific model chain. The selected model chain is given in Fig. 40.

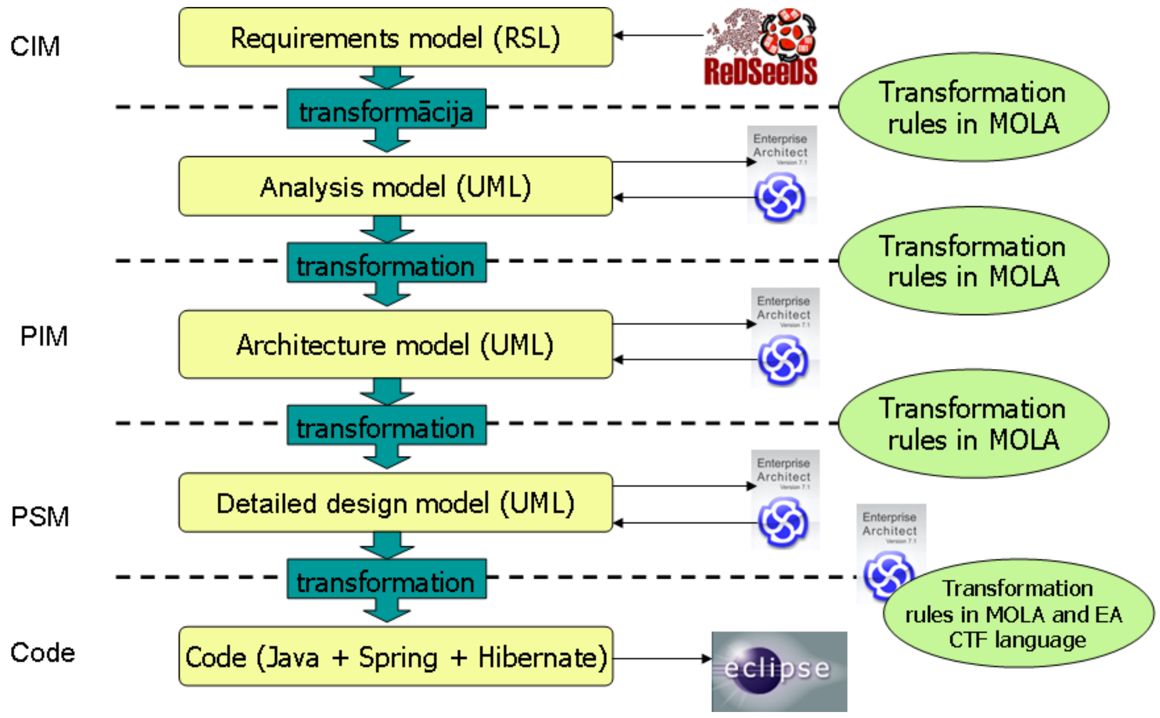

CIM
Requirements model (RSL)
ReDSeeDS
transformācija
Transformation rules in MOLA
Enterprise Architect
Analysis model (UML)
transformation
Transformation rules in MOLA
PIM
Architecture model (UML)
transformation
Transformation rules in MOLA
PSM
Detailed design model (UML)
transformation
Transformation rules in MOLA and EA CTF language
Code
Code (Java + Spring + Hibernate)
eclipse

Fig. 40. Model chain used in ReDSeeDS Keyword-Based Style

Requirements are specified in the requirement specification language RSL [66] [152] which lies at the basis of the approach. We are interested mainly in the requirements for the system behaviour specified by use case scenarios and draft domain concepts (which are called notions in RSL).

Starting from the requirements, a chain of models for a model-driven development of the software system is proposed. To a great degree, this chain has been inspired by the classical MDA approach. However, the specific structure and construction principles of the models in our approach are determined by the chosen *architecture style*, the most important feature of which is the set of the selected *design patterns*. A more precise description of the concept of the architecture style is given in Section 3.3.1. All the models are built in UML2 [121], using an appropriate profile.

Initially the *Analysis* model is extracted by transformations from the requirements. This model has no direct counterpart in the classical MDA chain. It corresponds more to the *Analysis* model in the standard OOAD [97] approach. Therefore, we call this model the *Analysis* model. The most important part of it is the class diagram, describing the main concepts of the software system to be created (the *Domain Model*). Stereotypes are used to distinguish different types of concepts according to the Analysis Profile. The Analysis Model is described in a greater detail in Section 3.5.4.

The most important model in the proposed model chain is PIM, which is very close to the corresponding model in the MDA approach. This model is built according to the selected design patterns and contains the description of structure and detailed behaviour of the would-be system in a platform-independent way. In this model the implementation structure is represented according to the behaviour extracted from use case scenarios. This model is platform-independent and could be used as a basis for the development of a code on any enterprise platform (Enterprise Java, .NET, etc.). This is the model where the selected design patterns and sophisticated analysis of the requirements permit to generate a non-trivial part of solution behaviour. Transformations which generate the initial version of this model use both *Requirements* and *Analysis* as inputs. In the whole chain of transformations, this step contributes most to the rich system functionality inferred directly from the requirements. The contents of PIM are described in Section 3.5.6.

The final model in the chain is the PSM in a fairly standard MDA style (Section 3.5.8). It is built by transformations from PIM by adding platform-relevant details. Currently the chosen target platform is *Java* in the *Spring/Hibernate* framework, but any similar platform can be used as well. In this model stereotypes corresponding to *Spring*-specific annotations are used. Finally, PSM is transformed to the *Java* code with *Spring/Hibernate* annotations. The main value of the approach lies in the fact that a large fraction of a non-trivial prototype of the system can be obtained from the requirements without a manual extension of intermediate models. Certainly, a true model-driven development should follow, where in each step the required details of the real system are filled in manually. PSM is described in detail in Section 3.5.8.

It should be noted that in the ReDSeeDS project an alternative model naming is used – PIM is also called the *Architecture* model and PSM the *Detailed Design* model.

3.5.2 Selected Design Patterns for the Keyword-Based Style

In this section we will describe the design patterns chosen for the Keyword-Based architecture style. The patterns are grouped according to models and system layers chosen for the style. The patterns used at the PIM level are as much platform-independent as possible. Since we have chosen *Java + Spring + Hibernate framework* as the target platform, the design patterns popular in the *Spring* community are used at the platform-specific level. This choice has also slightly influenced our PIM level, when we had to choose one of several equivalent options.

We use the *DAO design pattern* [138] at the *Data Access* layer. Data access objects are introduced as the main actors for explicit ORM-related actions. Therefore, each DAO has the basic CRUD and typical Find operations. A data access object is created for each persistent domain concept. The DAO classes are assumed to have the standard transaction support for their operations.

Manager is the main design pattern used for *Business Logic* (see [108] for its version in the .NET world). It means that for each domain concept participating in *Business Logic*, a class (and interface) is created, which encapsulates all business level operations related to this concept.

The *Application Logic* and *User Interface* layers are governed by the MVC pattern, which is used in almost every four-layer architecture. Moreover, the façade pattern [47] [104] is used for the *Application Logic*. For each Use Case in the requirements, we create one *Application Logic* interface and an implementing class. This class implements all operations invoked by the MVC controllers within this use case.

The UI part is kept as simple as possible. It contains only calls to the application layer. This research does not include the specific issues of building user interfaces from the requirements, which is a separate topic in the ReDSeeDS project (see [87]).

We also use the domain object design pattern. It means we use domain objects as data containers, in other words, as standard “POJO” (not mandatory Java) objects. Persistent domain objects are treated as the basis for the ORM definition; therefore, platform-independent ORM features, such as identifying attributes and persistent relations, are included.

The design in general relies on the *Dependency Injection Pattern* (which will appear later as platform-specific dependency annotations) for referencing other classes; therefore, the Factory Pattern is not used explicitly.

Platform-specific design patterns are used in PSM and in the code. These are domain objects that have the most of platform-specific features. The POJO pattern is used, adapted to the *Spring* style. We use the declarative ORM definition (*Spring* + *Hibernate*) based on annotations which are coded as appropriate stereotypes in PSM. The transactionality of relevant classes is also defined by annotations. For reference initialization, the dependency injection pattern is used.

For UI layer, the MVC design pattern is used in a standard ("Spring-Basic") way.

3.5.3 RSL Profile for the Keyword-Based Style

As stated in the previous section *Java* + *Spring* + *Hibernate framework* was chosen as a target platform for this architecture style. This decision is closely related to typical application areas of this architecture style which is suitable for web application development. Examples of typical applications are online shops, online reservation systems, etc.

Terms related to this type of systems are selected as keywords. Actions typical to this type of systems are selected as verb keywords. Objects used in these systems are selected as noun keywords. When selecting some terms as keywords, predefined semantics is added to them.

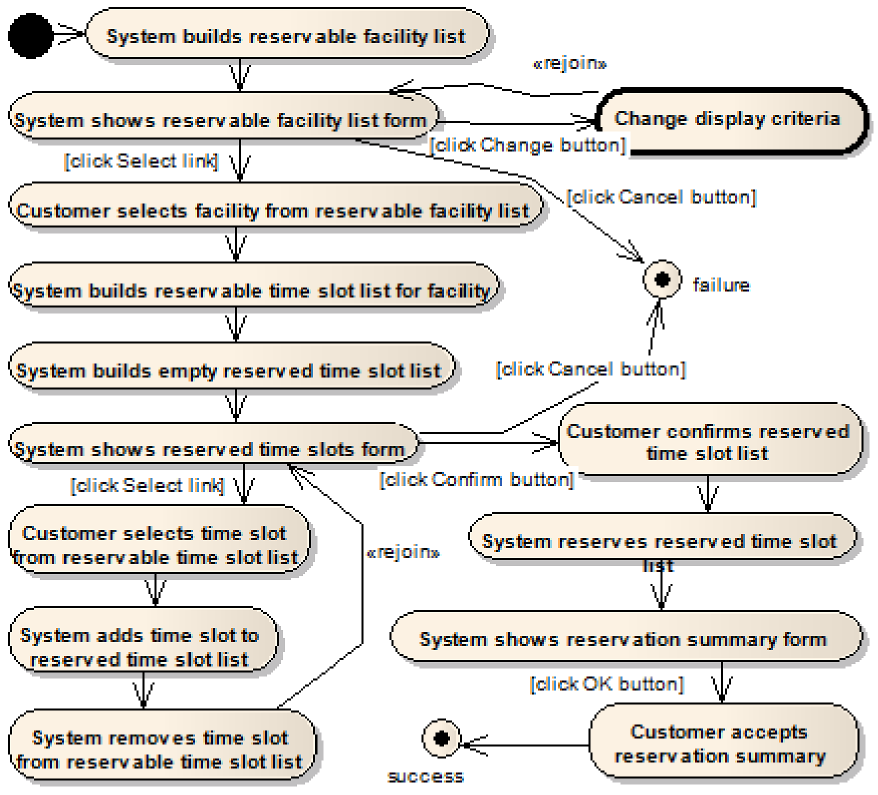
In this profile the verb keywords for SVO(O) sentences are *show*, *select*, *build*, *add,* and *remove*. The noun keywords are *form* and *list* – when used as parts of complex notion names (and, consequently, objects in SVO(O) as well). Conditions (which otherwise are arbitrary sentences in RSL) can contain the verb keyword *click* and the noun keywords *button* and *link*. The adjective (modifier in RSL terms) *empty* is also treated as a keyword.

A brief description of the meaning of keywords and some context rules in scenarios is given below. The keyword *show* means that the system must display a form defined by the direct object of this sentence. This object, in turn, must correspond to a notion whose complex name ends with the noun keyword *form*. For example, the SVO(O) sentence "System *shows* reservable facility list *form*" specifies that the form "reservable facility list *form*" must be displayed at this point.

Similarly, the sentence "System *builds* reservable time slot *list* for facility" uses the verb *build*, which means data creation. The direct object "reservable time slot *list*" denotes a list, since the last noun in it is *list*.

The sentence "Customer *selects* facility from reservable facility *list*" means that the user has performed element selection from the data table in the form. The indirect object (preceded by the preposition "from") specifies the data table contents ("reservable facility *list*", i.e., a *list* notion), the selected element is an instance of the notion "facility".

The condition "*click* Select *link*" means that the user clicks on an active element (link) in a form table with selectable rows. Normally this condition should be on the control flow, which goes from the *shows* sentence/node (see the example above) to the *selects* sentence (the previous example). This order of sentences should be followed to enhance the following models produced by transformation. A recommended order of sentences is a part of the RSL profile.



System builds reservable facility list
«rejoin»
System shows reservable facility list form
Change display criteria
[click Change button]
[click Select link]
[click Cancel button]
Customer selects facility from reservable facility list
System builds reservable time slot list for facility
failure
System builds empty reserved time slot list
[click Cancel button]
System shows reserved time slots form
Customer confirms reserved time slot list
[click Select link]
[click Confirm button]
Customer selects time slot from reservable time slot list
«rejoin»
System reserves reserved time slot list
System adds time slot to reserved time slot list
System shows reservation summary form
[click OK button]
System removes time slot from reservable time slot list
success
Customer accepts reservation summary

Fig. 41. Requirements – scenarios of the use case in a graphical form

The condition "*click* Confirm *button*" means that the form button has been clicked. The meaning of the remaining keywords is self-explanatory. The example in Fig. 41 completely complies with the rules described above.

It should be noted that the use of keyword and predefined order of sentences is voluntary. However, it affects the quality of the following models. If keywords are used appropriately, more complete models are obtained in the following steps.

The described profile for the *Keyword-Based Style* is supported in the current version of the ReDSeeDS tools. Currently for a term to be treated as a keyword exactly this predefined term should be used. Nevertheless, extending keyword support and using *WordNet* [41] it should be possible to treat synonyms of predefined terms as keywords as well.

3.5.4 The Structure of the Analysis Model

The main part of the *Analysis* model in the *Keyword-Based Style* is the Domain Model – a conceptual class model for the system to be built. The Domain Model is generated by appropriate transformations from the domain (notion) part of *Requirements*. It contains classes corresponding to all notions in *Requirements*. Class attributes and associations are also extracted from the notions part of *Requirements* (if they have been defined there). A special *Analysis* profile is defined in ReDSeeDS which contains stereotypes to be applied to the Domain Model. Classes generated from persistent notions would have the <<*entity*>> stereotype (there also are some heuristic rules how to find persistent notions when they have not been properly marked in the requirements). Other classes with the stereotype <<*form*>> would correspond to forms – notions with the suffix form in their names. In a similar way, collection classes (for example, *ReservableFacilityList*) will have the <<*list*>> stereotype. In the design stage, these classes will be converted into generic list classes. Control elements in forms (such as buttons and links) are also represented by stereotyped classes in the Domain Model, with stereotypes <<*button*>>, <<*gridLink*>>, <<*link*>>, and some others. Additional associations, having a special meaning for the design model (e.g. aggregations linking a form to a list to be visualised as a data grid in this form), can also be generated. These associations are also given special stereotypes (<<*owned*>>, <<*formElement*>>, a.o.). See more on the principles how the Domain Model is generated from *Requirements* by transformations in Section 3.5.5. Fig. 42 presents a part of the generated Domain Model in the Fitness club example. It shows that the proposed approach can transfer a significant part of the intended semantics of the requirements into the stereotyped Domain Model (this, in turn, will guarantee a rich behaviour to be generated into the PIM model).

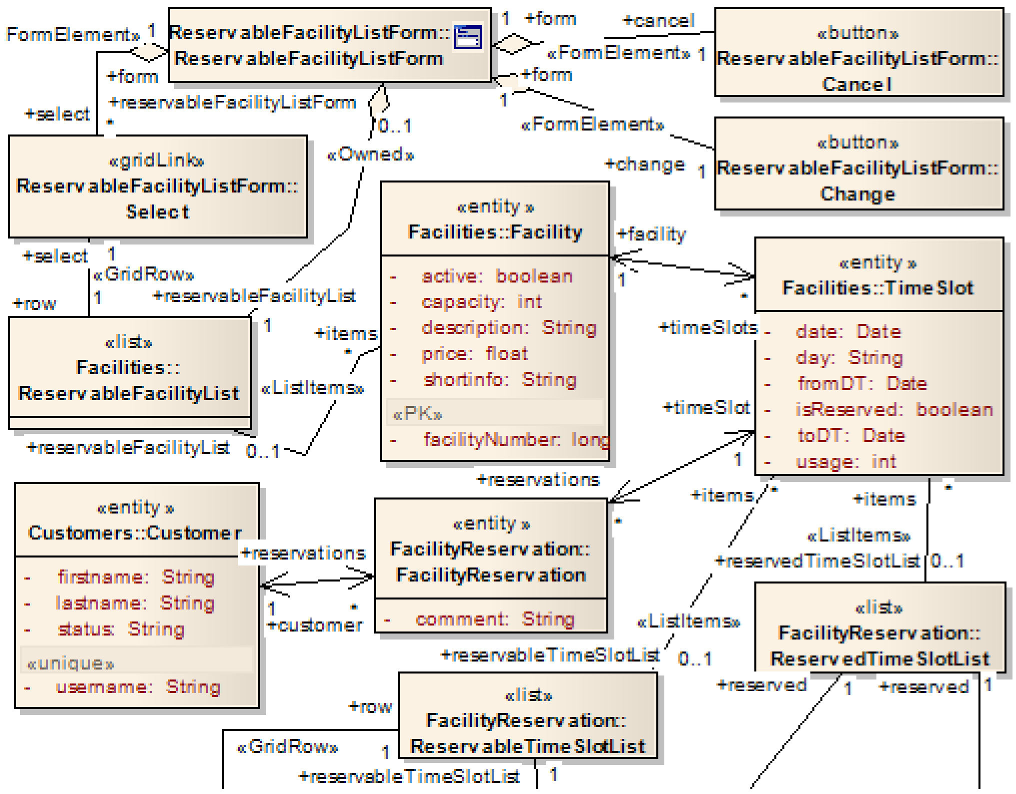


Fig. 42. Fragment of the generated Domain Model

The full strength of the transformations is revealed only if requirements are built in RSL according to the appropriate RSL profile **(**see 3.5.3). If requirements in RSL cannot provide sufficient information for building this Domain Model, it is highly recommended to extend this model manually in the *Analysis* step. Only in this case the next steps will provide the desired results.

The structuring of the Domain Model is based on notion packaging (provided in RSL).

3.5.5 Transformation of Requirements to Analysis

The main task of this transformation is to create the Domain Model from the notion part of Requirements, taking into account some elements of scenarios as well. The basic transformation is very straightforward since notions, their attributes, and relationships in RSL actually are in one-to-one correspondence to the class model. The stereotypes <<*list*>> and <<*form*>> are added if the respective keywords are present in the notion names. An additional analysis is done for list classes. If an entity name is contained within the list notion name (such as “facility” within “reservable facility list”), the entity class is assumed to be the element of that list (a <<*listItems*>> association is generated).

Classes for control elements can be generated from scenarios. We are looking for a *click*-condition (*click … link* or *click … button*) which follows a *show*-sentence (*… shows … form*). If such (new) situation is found, a class is generated with the name equal to the name in the *click*-condition and the stereotype <<*gridLink*>> or <<*button*>>, respectively. The association (with the stereotype <<*formElement*>>) linking the control element to its form is also generated.

More form-related associations can be generated from scenarios. *Select*-sentences (such as ... *selects* facility from reservable facility *list*) allow us conclude that the relevant form (that in the preceding *show*-sentence) permits to select elements exactly from this kind of list. Hence, this list (here, *ReservableFacilityList*) is visualized in the form (the <<*owned*>> association can be built), and each *gridLink* element in the form corresponds to a row in the list (the <<*gridRow*>> association is built).

Using these relatively simple principles, the Domain Model in the example in Fig. 42 can be generated from notions and the scenario in Fig. 41 (p.90). Implementation of these transformations in the MOLA language is also quite straightforward.

3.5.6 The Platform-Independent Model

This model is the most important to our approach since all platform-independent functionality is generated in this model. This is done by revisiting the use case scenarios and analyzing them repeatedly, taking into account the (possibly manually extended) Domain Model from Analysis. In combination with the keyword-based sentence analysis, a significant part of application and especially Business Logic can be generated. This model is created according to the platform-independent design patterns described in Section 3.5.2.

The main result of the PIM step is the design class model: packages and classes (and interfaces) with all attributes and operations. The operations will have all parameters defined. All the other data such as persistence info for ORM-related classes are coded by platform-independent stereotypes, which constitute the PIM profile.

The other essential results of this analysis are stored as sequence diagrams, also covering a significant part of the Business Logic method bodies. All method invocations with appropriate parameters that can be generated are coded this way. Whenever possible, the invocation logic up to the DAO level is documented. These sequence diagrams are kept in the *behaviour* package and are grouped in the same way as use cases in the Requirements Model. Some small practical extensions of sequence diagram syntax are used, for example, *FOREACH* iterator in *loop* fragments.

The design class model is split into the following packages: *applicationlogic*, *businesslogic*, *dataaccess* and *domainobjects*. The first three are further subdivided into *Interfaces* and *Implementation* parts, containing interfaces and implementing classes, respectively. Each interface name has the prefix "I" added to the corresponding class name.

For application logic, the façade design pattern is used. For each use case, a class corresponding to this use case is generated (with the suffix "Service" added to the name). Further structuring of the *applicationlogic* package is done according to the use case packages.

The content of *businesslogic* is generated according to the Manager Pattern. Here classes correspond to persistent classes (entities) whose usage in *Business Logic* can be inferred from sentences with keywords and the Domain Model. Classes/interfaces have the suffix "Service" added to the entity name.

For *dataaccess,* an updated version of the DAO pattern is used, and practically applicable methods are generated for DAO classes. Each class corresponds to a persistent domain object; the class name is generated from the object name with the suffix "DAO". Classes are grouped in the same way as domain objects. For each class, CRUD and some typical find operations are generated. Bodies of these operations are similar in all classes, only the types vary. Therefore, we propose to implement them once in a template class which contains parameterized types. All the other classes will inherit them from this template class (with parameters set to the relevant values in each case). We remind that this specialization of the classical DAO pattern is platform-independent since it can be directly implemented in most of typical platforms.

For the *domainobjects* package, the domain object design pattern is used. This package represents a platform-independent *Object Relational Mapping* (ORM) model for all entities with platform-independent annotations. Associations (relations) are also included in a way typical of an ORM definition. A database schema for a specific platform can also be easily generated from this model (in the next PSM step). Names of domain objects are taken from the corresponding domain concepts. For each persistent class, a unique identifier attribute is defined as well.

3.5.7 Transformation of Requirements and Analysis to PIM

Transformations for building a platform-independent model are more complicated than for building the Domain Model in *Analysis*. They use the behaviour part of the *Requirements* model as input, as well as the updated Domain Model.

The transformation of domain objects is very straightforward. Domain classes are transformed to PIM domain objects, retaining all attributes. For each persistent class without a primary key, an artificial primary key is created. Here the copy library is used.

For each persistent domain class, a DAO class and its interface is created in the *dataaccess* package. They specialise the template-based implementation of CRUD and filter operations.

In the *Business Logic* layer, classes and interfaces have a structure, similar to that in DAO, with the exception that classes, devoid of business level methods, are excluded. The generation of business methods is done in the general context of behaviour generation by analyzing scenarios in the requirements.

In the *Application Logic* layer, for each use case, a class and interface is generated. For this interface/class, one “main” method is generated (which means invoking this use case from another one). Its name corresponds to the Use Case name. Other methods for this class are generated for UI-related sentences in the scenario that are detected by analyzing the subject of the sentence. If the subject of the sentence is an actor, then it is *actor-system* sentence (or UI-related sentence).

Behaviour generation, described below, is the most complicated part. Here we greatly rely on the meaning attached to the keyword. We use heuristics describing how the resulting model should look if one or another keyword is used. The transformation algorithm is complicated. A detailed description of the transformation algorithm is quite lengthy (see [84]).

Now we present the main ideas transformations rest upon and typical examples, representing the use of keywords in transformation algorithms.

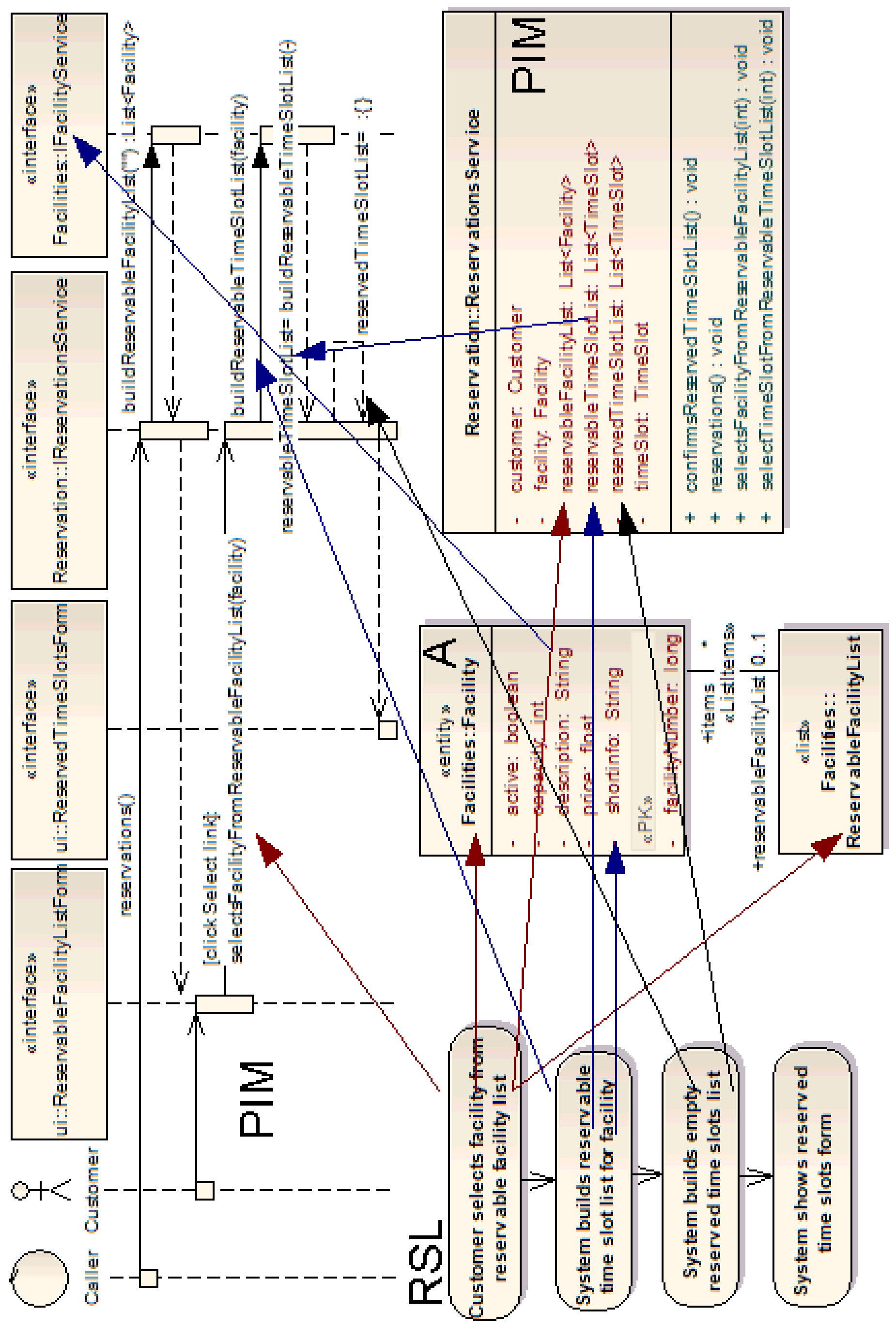
Behaviour is grouped in the same way as Use Cases. For one Use Case, one or more sequence diagrams are generated by processing its scenario. The behaviour of a Use Case begins with invocation of the "main" method of the *Application Logic* class corresponding to the Use Case.

In order to build an *Application Logic* method body, we look for consecutive scenario sentences with the subject System and the recipient system (in other words, any verb other than "System *shows* …). All these sentences correspond to calls to the *Business Logic* layer. At first the verb used in this sentence is analyzed. If the verb is a keyword, the sentence is analyzed according to the rules used for this keyword. If the verb used is not a keyword, the structure of the sentence alongside with the object keywords is analyzed. Default behaviour generation principles corresponding to the sentence structure are applied. The immediate recipient of this call depends on the sentence structure. If the indirect object (e.g., … *for* facility) is present, the call is directed to the manager of the corresponding entity (here, *FacilityService*). Another typical case is when an indirect object is absent and the direct object corresponds to a notion/class with the stereotype <<*list*>>. Then the invocation is created to the manager class corresponding to the entity class which is the list element. There are also some other "patterns" of sentences, corresponding to the *Business Logic* calls (or simple actions directly in the *Application Logic* layer).

The grouping of the generated *Business Logic* calls is done in a simple way – all these calls up to the next UI call (corresponding to the next "System *shows* …" sentence) are included in the body of the current *Application Logic* method body (see Fig. 43). The "System *shows* … *form*" sentence generates a call to the *User Interface* layer (to the controller of the relevant form), which completes the current body. The next sentence (which in fact follows the "*click* …" condition) corresponds to the invocation of another *Application Logic* method. Then building of the body of this method starts.

Fig. 43 illustrates in detail a typical application of the transformation rules described above by an informal "model mapping diagram", with arrows going from the source model instances (bottom) to the corresponding target model instances (top). The first sentence in the scenario fragment ("Customer selects facility from reservable facility list") follows the "click Select link" condition; therefore, it implies the method invocation *selectFacilityFromReservableFacilityList()* to the *Application Logic* class

(*ReservationsService*). The two following sentences in the scenario correspond to the actions in the body of this *Application Logic* method.



«interface» Facilities::IFacilityService
«interface» Reservation::IReservationsService
«interface» ui::ReservedTimeSlotsForm
«interface» ui::ReservableFacilityListForm
Caller
Customer
buildReservableFacilityList("") :List<Facility>
buildReservableTimeSlotList(facility)
reservableTimeSlotList= buildReservableTimeSlotList()
reservedTimeSlotList= :{}
reservations()
[click Select link]: selectsFacilityFromReservableFacilityList(facility)
PIM
RSL
PIM
A
Reservation::ReservationsService
- customer: Customer
- facility: Facility
- reservableFacilityList: List<Facility>
- reservableTimeSlotList: List<TimeSlot>
- reservedTimeSlotList: List<TimeSlot>
- timeSlot: TimeSlot
+ confirmsReservedTimeSlotList() : void
+ reservations() : void
+ selectsFacilityFromReservableFacilityList(int) : void
+ selectTimeSlotFromReservableTimeSlotList(int) : void
«entity» Facilities::Facility
- active: boolean
- capacity: int
- description: String
- price: float
- shortInfo: String
«PK»
- facilityNumber: long
+items *
«ListItems»
+reservableFacilityList 0..1
«list» Facilities::ReservableFacilityList
Customer selects facility from reservable facility list
System builds reservable time slot list for facility
System builds empty reserved time slots list
System shows reserved time slots form

Fig. 43. An example of informal mapping describing transformations to Detailed Design

Fig. 43 presents a detailed analysis of the first sentence. The sentence "System builds reservable time slot list for facility" implies the *Business Logic* method invocation *buildReservableTimeSlotList*(). According to the rules described above, there is an indirect object ("for facility"); therefore, the method must go to the corresponding manager class (to the class *FacilityService*). Because of *build*-semantics (*build* is the keyword) of the verb and *list*-semantics of the direct object, the return type of the method is *List<TimeSlot>*. The returned value must be stored in the attribute *reservableTimeSlotList* (of the same list type) of the invoking application class (*ReservationsService*). The next sentence corresponds to an action in the body (assignment to the attribute *reservedTimeSlotList*) because of the semantics of the keyword *empty*. Note that all lifelines correspond to the interfaces because any invocation goes via the corresponding interface in our style (certainly, the body behaviour relates to the relevant class).

There are some more rules in the approach quite similar to those explained in the example. We do not examine the interaction with the UI layer in a greater detail.

3.5.8 The Platform-Specific Model

This model is a specialisation of the platform-independent model to a specific platform. The choice was *Java* with *Spring + Hibernate 3* with the greatest possible declarative (annotation-based) style.

For this platform, the model is quite similar to the platform-independent model. The class structure in PIM corresponds more or less to the required structure in PSM. The main task is to convert annotations to the specific style required by *Spring* and *Hibernate*. However, some new model elements should be added as well. In this step the copy library is widely used which is characterized by the feature to do copying and make some modifications depending on the transformation type

A new model is the database diagram generated from the domain objects. This is a typical database design diagram (with tables, columns, PK, FK, etc.) in EA.

The domain objects are "copied" with the same package structure. They are used to describe Hibernate-specific ORM functionality. All Hibernate- and Spring-specific annotations are added (coded as stereotypes) to the domain classes, attributes, and operations. The relevant getters/setters and some predefined methods are added to the classes. Traceability links between PIM and PSM elements are generated by transformations and used to maintain various annotations related to mappings between different parts of the model.

For each DAO class, the annotation <<*@Repository*>>is added. These classes have also annotations describing the transactional mode, the default "required" is used. The template-based mechanism is directly taken from PIM.

The *Application Logic* layer classes are included in the *Business Logic* layer. Classes in these layers are given the annotation <<*@Service*>> (to mark them as *Spring* beans). The annotation <<*@Autowired*>> is used to initialize references to other beans.

The structure of PSM corresponds directly to the potential Java class structure typically used in *Spring* (with the packages *domain*, *repository* and *service*). These packages are further structured in accordance with the already defined model structuring.

In order to have a more or less complete design class structure and behaviour in sequence diagrams, some elements in the UI area have to be specified as well. The basic source for that – forms, attached data, and actions (buttons and links) are available in the *Analysis* model. Currently a rudimentary solution directly based on *Spring MVC* is proposed. In this solution, we can use JSP for data visualisation and controllers to manage user actions. We use one controller per form, adding a method for each user action in the form. Typically a controller method directly calls the appropriate *Application Logic* method. Nevertheless, this should be treated only as a "stub" which can be replaced by a more appropriate UI feature definition. Such a prototype form structure definition could be incorporated in the requirements since the RSL language contains features for that purpose. Some experiments in this direction have been performed.

Sequence diagrams, defining behaviour within method bodies, are also refined according to the Spring requirements. The most significant changes refer to the User Interface part. At this level, a simple version of UI and the *Application Logic* interaction can be precisely defined. In particular, a special "executable" solution (including DAO methods) could be provided for finding the object selected by the user via a data grid in a form. This way, the form behaviour sufficient for simple prototyping could be provided. We do not describe the UI aspects of PSM in a greater detail since the tool support for them has not been fully implemented.

3.5.9 The Java Code

The provided PSM can be used for the *Java* code generation. This generation is quite straightforward – at first all information must be transferred into a properly stereotyped class model using the MOLA transformations (the body behaviour must also be transferred from sequence diagrams to the code sections of operations in EA). Then the properly modified EA Java code generation scripts can be used. The main issue of modification concerns adding scripts for processing all relevant annotations.

The structure of the Java code directly corresponds to the structure of PSM. Methods are generated according to the model. Predefined method bodies are generated for some methods. This is widely used for domain objects (almost all methods are generated). Bodies of getters, setters, *hashCode*, *equals*, *toString* are generated in particular. A template-based generator is used and the method body vary according to the object properties for which the method is generated.

There are also generated predefined method bodies of the *TemplateDAO* class and concrete DAO classes extending the *TemplateDAO* class with appropriate types. Appropriate *Hibernate* configuration file describing, for example, the data base connection is also necessary. An initial version of this file can be generated. It should be noted that a data base script can also be generated from PSM.

The *Business logic*- and *Application Logic*-related functionality is generated according to the class structure. The behaviour (described in sequence diagrams) is generated as well. Concerning the UI part, currently only a placeholder is generated.

The generated *Java* project can be inserted into an *Eclipse IDE* project template containing references to the required *Spring* and *Hibernate* libraries. Thus, a ready-to-compile project is obtained. All this constitutes a significant part of a simple prototype – mainly the UI part has to be added manually. However, if the complete set of transformations described here was implemented, a “near to executable” prototype would be obtained.

Some examples of the generated *Java* code are given below. The example in Listing 2 presents apart of the code generated for the *Facility* entity.

Listing 2. Generated Java code for the entity class “Facility”.

```
@Entity
@Table(name="facility")
public class Facility {

private Boolean active;
private Boolean capacity;
private String description;
private String facilityNumber;
private String id;

@Override
public boolean equals(Object obj){
```

```
if (this == obj) return true;
if (!super.equals(obj)) return false;
if (getClass() != obj.getClass()) return false;
Facility other = (Facility) obj;
if (active == null) {
 if (other.active != null) return false;
} else if (!active.equals(other.active)) return false;
if (capacity == null) {
 if (other.capacity != null) return false;
} else if (!capacity.equals(other.capacity)) return false;
if (description == null) {
 if (other.description != null) return false;
} else if (!description.equals(other.description)) return false;
if (facilityNumber == null) {
 if (other.facilityNumber != null) return false;
} else if (!facilityNumber.equals(other.facilityNumber)) return false;
return true;
}

@Column(name = "active", nullable = false)
public Boolean get_Active(){
return active;
}
public void set_Active(Boolean p){
active=p;
}

}
```

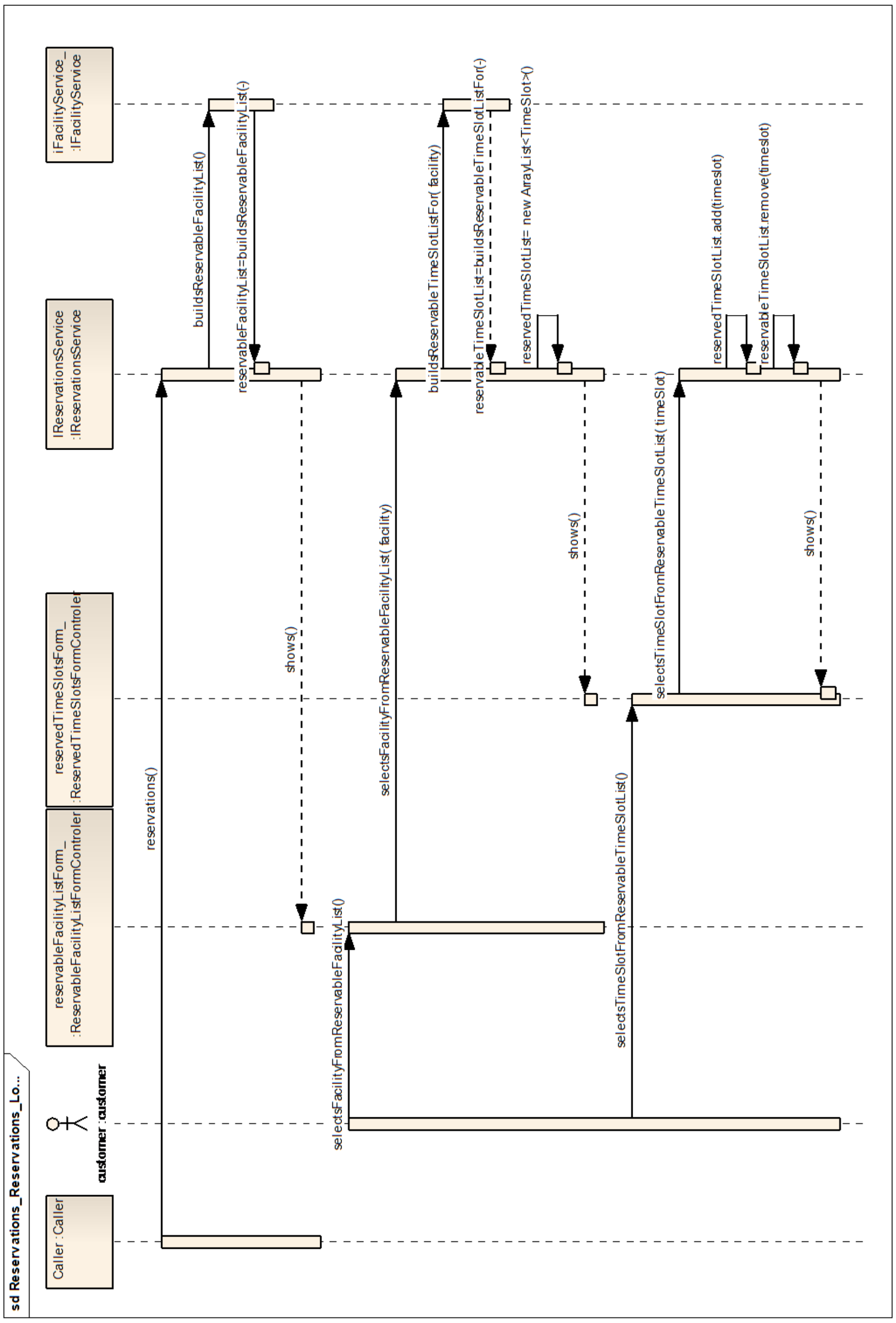
The code fragment in Listing 3 illustrates the code generated for the *Application Logic* methods. They represent three methods for the *Application Logic* class *ReservationsService*. To understand the context, one sequence diagram from the PSM model is given in Fig. 44. There are three method invocations on the *ReservationsService* lifeline (*reservations*, *selectsFacilityFromReservableFacilityList*, and *selectsTimeSlotFromReservableTimeSlotList*). The methods invoked within the corresponding fragments of the lifeline (until the return) appear within the corresponding body.

Listing 3. The generated code, describing the system behaviour for the *ApplicationLogic* class "*ReservatinService*"

```
@Service("ReservationsService")
public class ReservationsService implements IReservationsService {

@Autowired
private IChangeDisplayCriteriaService iChangeDisplayCriteriaService_;
@Autowired
private IFacilityService iFacilityService_;
@Autowired
private IReservedTimeSlotListService iReservedTimeSlotListService_;
private List<Facility> reservableFacilityList;
private List<TimeSlot> reservableTimeSlotList;
private List<TimeSlot> reservedTimeSlotList;

public void reservations(){
       reservableFacilityList=iFacilityService_.buildsReservableFacilityList();
}
public void selectsFacilityFromReservableFacilityList(Facility facility){
        reservableTimeSlotList=iFacilityService_.buildsReservableTimeSlotListFor(
facility);
        reservedTimeSlotList= new ArrayList<TimeSlot>();
}
public void selectsTimeSlotFromReservableTimeSlotList(TimeSlot timeslot){
        reservedTimeSlotList.add(timeslot);
        reservableTimeSlotList.remove(timeslot);
}
}
```



sd Reservations_Reservations_Lo...
Caller :Caller
customer :customer
reservableFacilityListForm_ :ReservableFacilityListFormControler
reservedTimeSlotsForm_ :ReservedTimeSlotsFormControler
IReservationsService :IReservationsService
iFacilityService_ :IFacilityService
reservations()
buildsReservableFacilityList()
reservableFacilityList=buildsReservableFacilityList(-)
shows()
selectsFacilityFromReservableFacilityList()
selectsFacilityFromReservableFacilityList(facility)
buildsReservableTimeSlotListFor(facility)
reservableTimeSlotList=buildsReservableTimeSlotListFor(-)
reservedTimeSlotList= new ArrayList<TimeSlot>()
shows()
selectsTimeSlotFromReservableTimeSlotList()
selectsTimeSlotFromReservableTimeSlotList(timeSlot)
reservedTimeSlotList.add(timeslot)
reservableTimeSlotList.remove(timeslot)
shows()

Fig. 44. An example of a sequence diagram for the *ReservationsService* class

3.6 Implementation

In the ReDSeeDS project an experimental tool support for the approaches described above has been built. The tool support is named the ReDSeeDS engine. It (and its sources) is available from SourceForge.net [2].

The ReDSeeDS engine contains the RSL editor, integrated transformation execution environment, and the entry point to the UML editor. The *Enterprise Architect* (EA) tool [156] used as the UML editor. A tool support for automatic data exchange with EA was built. For details see Section 3.6.4.

Model-to-model transformations supporting the MDSD path were implemented in the model transformation language MOLA [76]. More about transformation in general can be found in Section 3.6.1. The transformations algorithms used in the Keyword-Based Style are described in Section 3.6.2. Model-to-text transformations implementing code generation are described in Section 3.6.3.

3.6.1 Model-to-Model Transformations Implementation

Transformation algorithms described in style definitions (see Sections 3.4 and 3.5) are implemented in the model transformation language MOLA [76]. The transformations are implemented using the MOLA tool [59].

The metamodel used for transformations is the same as for other ReDSeeDS tool components – it consists of a RSL metamodel merged with the relevant parts of the standard UML metamodel and extended by special traceability elements. Transformations also build the relevant traceability links in every step.

Fig. 45 presents a MOLA transformation example which creates (or finds an existing) lifeline in a sequence diagram. The first rounded rectangle represents the most typical construct in MOLA – the rule (for details see CHAPTER 2). This concrete rule searches for a lifeline in a sequence diagram.

While implementing transformations, some transformation libraries were developed and they were reused in different layers of the models and in different transformation steps. The most powerful and most widely used library was the copy library, used to copy some element with all its child elements to another model. For each UML element type it was necessary to develop transformation in the library, implementing the copy logic. In MOLA it is not possible to define the copy logic independently of element types.

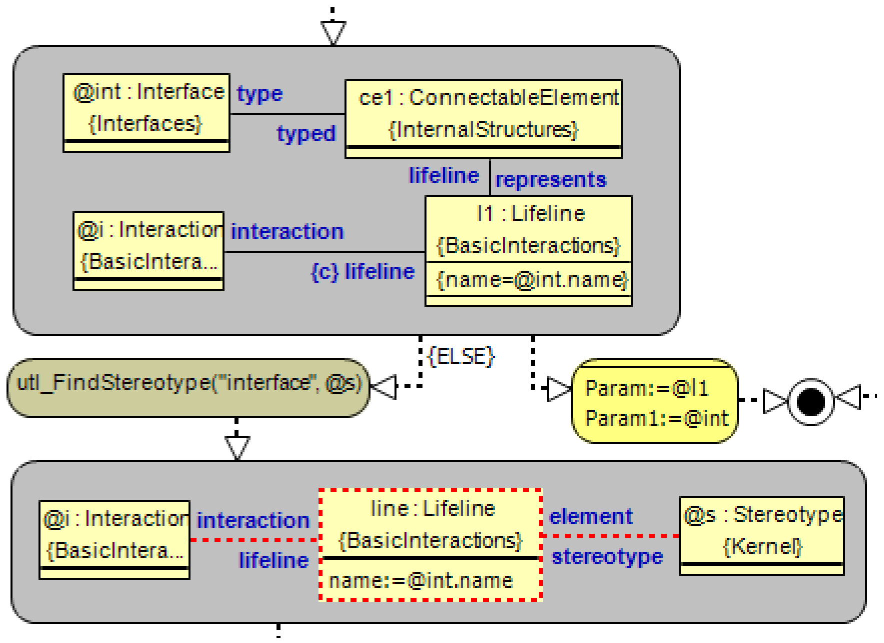


Fig. 45. Transformation example

In fact, when using this library it was possible to incorporate also typical changes of the resulting model. Each use of the library was given a name. By using this name a check-up was performed on the need for any adaption of the model elements. It was easy to combine the library with the extensions attached to the name of the library use.

The copy library was mainly used when working with a static structure. There were other transformation libraries used in the ReDSeeDS project, e.g., string processing, sequence diagram creation and processing, traceability creation, etc.

Another aspect of transformation implementation should be pointed out as well. All transformations in the chain must support repeated runs – the requirements always change. What is even more important, for the same transformations to be applicable to the manual model-driven development, all models in the chain should allow manual modification. Therefore, support for various result merge actions must be included in the transformation set. In our approach, this support mainly relies on traceability links. Currently one kind of the merge procedure – the so-called *Simple Merge* - is implemented, but more sophisticated merge procedures could be implemented, too.

Transformations were used not only to support a path from one model to another, but also to implement such technical tasks as merge or model import/export. As a result the following model-to-model transformations were developed: for the *Basic style*: RSL to PIM, PIM to PSM; for the *Keyword-Based Style*: keyword analysis, RSL to Analysis, Analysis to PIM, PIM to PSM, PSM to code; technical transformations: RSL scenario visualization by UML activity diagrams, export to EA, import from EA and *Simple Merge*. It should be noted that some transformation rules are reused in several transformations.

3.6.2 Model-to-Model Transformations in the Keyword-Based Style

In this section, we briefly describe the implementation of transformation algorithms for building the chain of models in the Keyword-Based Style.

Missing Features

Not all model transformation features outlined in the Keyword-Based Style description have been implemented. Mainly the features related to the generation of UI functionality are missing. The delay of transformation support for the UI functionality is due to the fact that it would be natural to combine the generation of UI features from scenarios with a direct specification of the UI structure in RSL (as is usually done during the requirements specification). Although this possibility exists in the RSL language, as already stated, currently there is minimum tool support for this.

Consequently, the UI part in the generated models is implemented minimally; only some basic UI classes and interfaces have been created. All the remaining details of UI, such as form elements, are not generated in the current version. Therefore, the code generation for the UI part is not supported either, although the generation of some code skeletons is technically feasible.

One deviation from clean usage of UML in models is also observable in some of the examples. Assignments in sequence diagrams are emulated by the message text and some tagged values because this feature is defined in UML in a very complicated way and virtually supported in no UML tools. This workaround has made some transformations more complicated.

Keyword-Based Analysis and Analysis Model

Some non-trivial aspects of transformation implementation are described below.

Transformation for keyword analysis (which is the first to be applied in the chain) scans nouns, verbs and modifiers used in the scenario sentences, and fills in the keyword field of the relevant RSL elements. This permits to specify the same keyword with several synonyms. It could be improved further by using the WordNet meaning as the keyword. This way it would be possible to distinguish different meanings of the same term and to use all synonyms with the same meaning.

The next transformation is from RSL to the Analysis Model. The logic of this transformation is relatively simple – it analyses the notion model in RSL and transforms it directly into an UML class diagram, adding stereotypes based on the keywords set by the previous transformation.

Creation of PIM

The most important transformation is from the *Requirements* and the *Analysis* model to PIM. This transformation has two logical parts. The first part is the creation of a static structure – package hierarchy, classes, and interfaces. The second part is the creation of behaviour stored as UML sequence diagrams.

For creation of a static structure, a universal "package hierarchy copier" library is used. The package hierarchy copier receives as input the root of the source package hierarchy, the target package, and the copy mode. The package copier copies a hierarchy of packages and their elements (classes, interfaces, etc.) in a way specific to the given model. For example, it is possible to define that for some mode either a suffix should be added to the class name or class attributes should be ignored, etc. The universal package hierarchy copier is used in several contexts during the creation of PIM and PSM models. In PIM the *Data Access* objects and the *Business Logic* objects are based on the Analysis class diagram. In PIM the *Data Access* class should be created for each persistent class in the *Analysis* model. This is ensured by using an appropriate copy mode. The same copy package hierarchy mechanism is even more widely used in the creation of PSM since it is based on the PIM model with some modifications.

Another important part of PIM is the behaviour description, using UML sequence diagrams. In this case the RSL scenarios are analyzed and sequence diagrams are created. For each scenario, one UML sequence diagram is created. The content of this sequence diagram depends on the RSL sentences, used in this scenario. Objects, generated from a sentence, depend on the kind of the sentence. There are three kinds of sentences: an *actor-system* sentence defines the interaction of an actor with the system. It can be recognized by the subject of the sentence – an actor. The subject of the two other kinds of sentences must be a system element. The next kind is a *system-actor* sentence. Such sentence typically means that the system shows something to the user or asks for some input from the user. The third kind is *system-system* sentences. These sentences are used to describe internal actions of the system, typically some *Business Logic*. There are different sub-kinds of these sentences, depending on the keywords used in the sentence.

The sequence diagram elements generated from a sentence depend on the kind and sub-kind of the sentence. At first the sub-kind of the sentence is determined, followed by the creation of elements of the sequence diagrams. Since the UML sequence diagram metamodel is quite complicated, a library has been created for the basic element creation and used accordingly. The procedure for one sub-kind of a sentence consists of calls to procedures for creating/finding the basic sequence diagram elements. It helped to separate the transformation algorithm from the technical sequence diagram metamodel processing.

Fig. 46 provides an example of the procedure for creating the sequence diagram elements for a *system-system* SVO sentence without keywords. At first the lifeline, corresponding to the object, is found or created. Then a message to this lifeline is created. Afterwards an operation corresponding to this message is found or created, followed by association of this operation with the message created. Then a return message is created. Each of these tasks is implemented as a MOLA procedure, invoked by the given procedure. These procedures for the sequence diagram element processing are used as building blocks. The content of one such MOLA procedure is shown in Fig. 47, which demonstrates the search of lifeline in a sequence diagram depending on the object used in the verb phrase. In the first rule, the notion corresponding to the noun used in the verb phrase is found (the long chain of associations necessary to locate this correspondence is implied by the RSL metamodel [66]). Then it is determined whether this notion or its parent should be used and the interface corresponding to this notion is found (it has been created during the static structure generation). In this case, the *Business Logic* interface is found. Finally, the lifeline for this interface is found or created. This procedure is very typical of transformation implementation in ReDSeeDS – it uses the MOLA patterns for finding complicated correspondences between model elements (such complicated correspondences are enforced by the structure of RSL and UML metamodels).

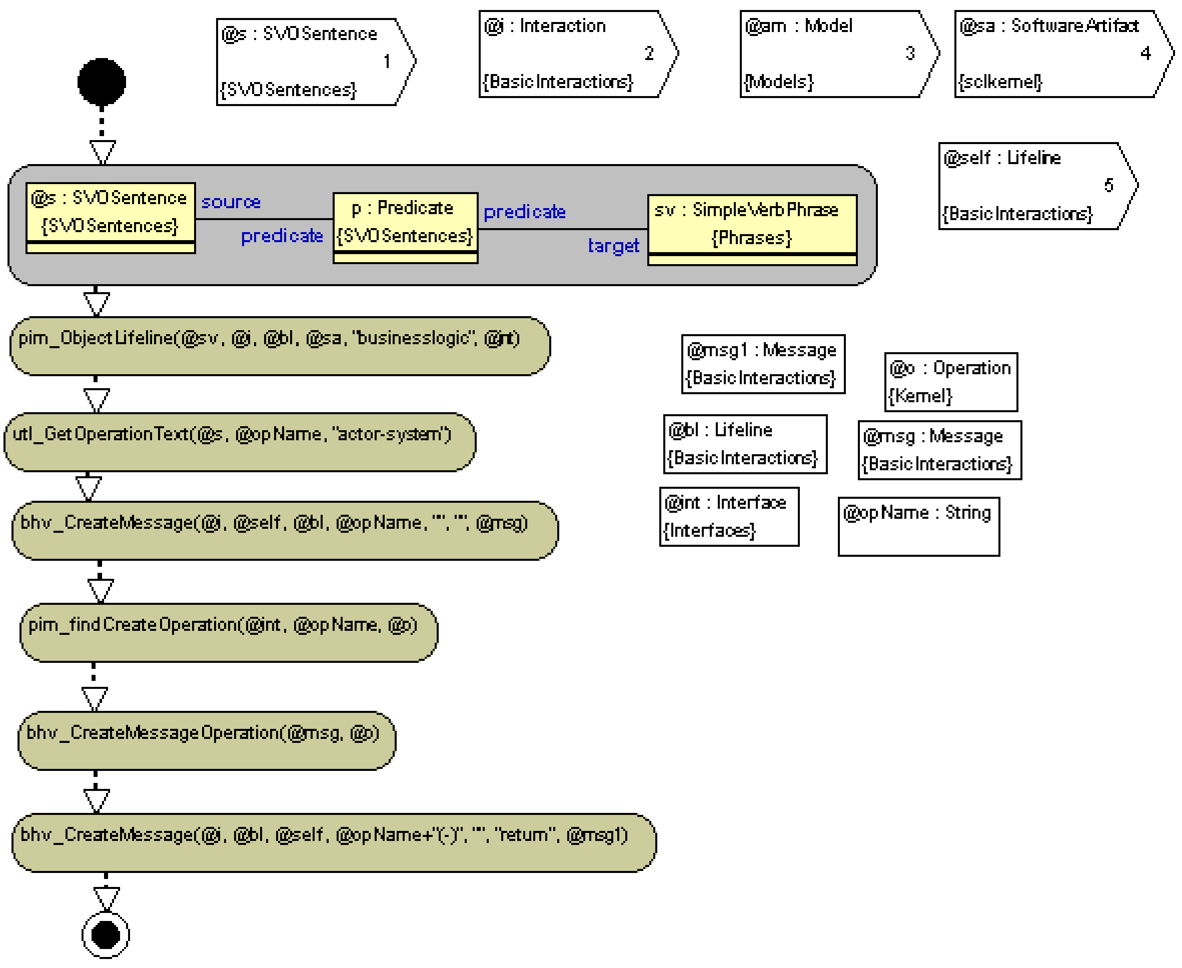
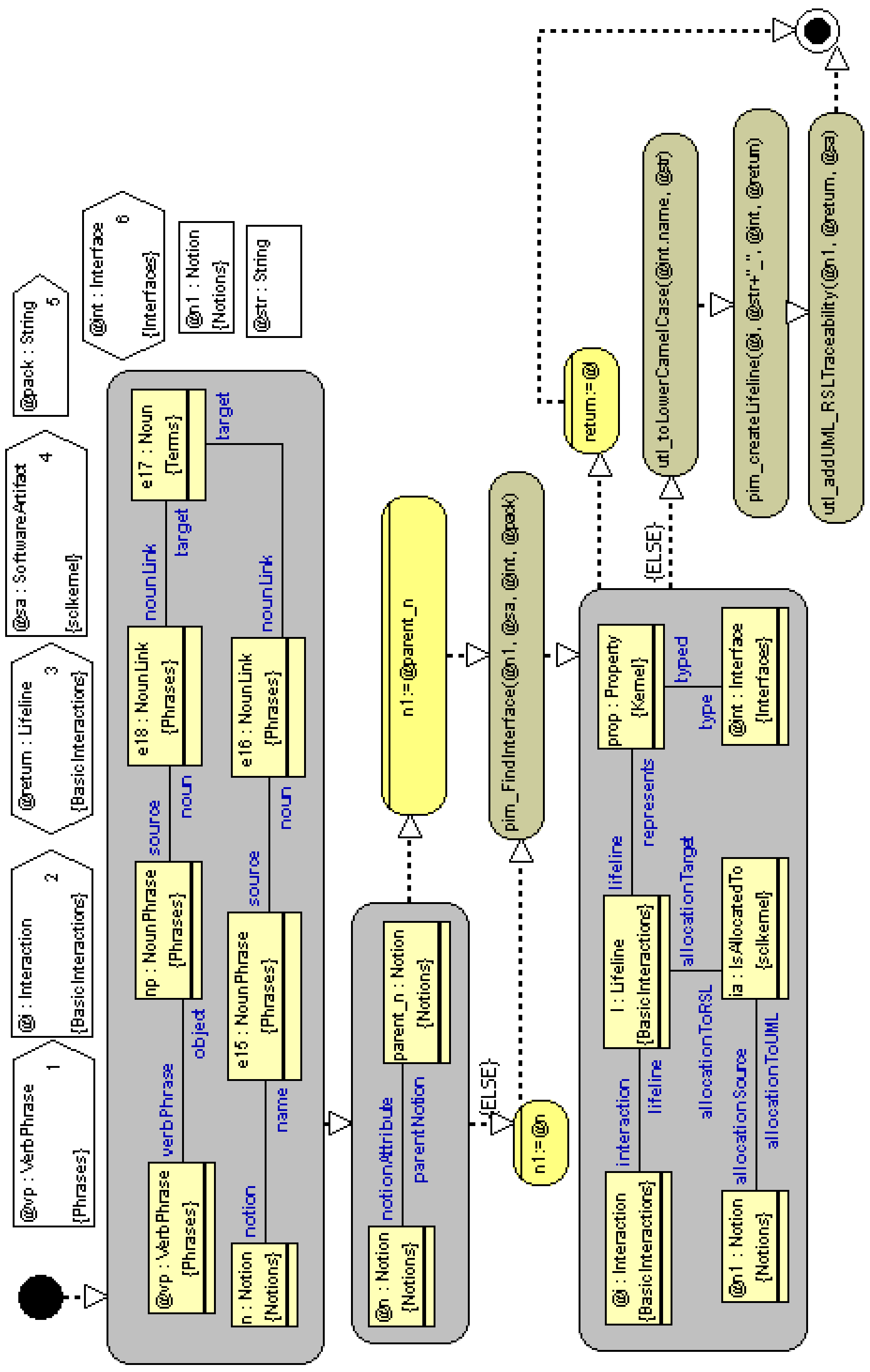


Fig. 46. Creation of a message for a “System-System” sentence without an indirect object



@vp : VerbPhrase {Phrases} 1
@i : Interaction {BasicInteractions} 2
@return : Lifeline {BasicInteractions} 3
@sa : SoftwareArtifact {scKernel} 4
@pack : String 5
@int : Interface {Interfaces} 6
@n1 : Notion {Notions}
@str : String
@vp : VerbPhrase {Phrases}
verbPhrase
np : NounPhrase {Phrases}
object
source
noun
e18 : NounLink {Phrases}
nounLink
target
e17 : Noun {Terms}
target
nounLink
e16 : NounLink {Phrases}
noun
source
e15 : NounPhrase {Phrases}
name
notion
n : Notion {Notions}
@n : Notion {Notions}
notionAttribute
parentNotion
parent_n : Notion {Notions}
n1:=@parent_n
{ELSE}
n1:=@n
pim_FindInterface(@n1, @sa, @int, @pack)
@i : Interaction {BasicInteractions}
interaction
lifeline
l : Lifeline {BasicInteractions}
lifeline
represents
prop : Property {Kernel}
typed
type
@int : Interface {Interfaces}
allocationTarget
allocationToRSL
ia : IsAllocatedTo {scKernel}
allocationSource
allocationToUML
@n1 : Notion {Notions}
{ELSE}
return:=@l
utl_toLowerCamelCase(@int.name, @str)
pim_createLifeline(@i, @str+"_", @int, @return)
utl_addUML_RSLTraceability(@n1, @return, @sa)

Fig. 47. The procedure of finding a lifeline in a sequence diagram, depending on the object used in the verb phrase

PSM Model and Initial Code

The next step in the chain is the transition from PIM to PSM. For the creation of PSM, the package hierarchy copier described above is widely used. Only appropriate modes are defined. The transformation algorithm, creating static structure of PSM from PIM, is mainly based on the package hierarchy processing. There are many repetitive steps. Most of transformations in this step could be defined by using a higher-level language than MOLA (see CHAPTER 4). Behaviour processing in this step actually is also copying of sequence diagrams with some small fine tuning.

The transformation from PSM to the initial code analyses the sequence diagrams and creates the initial code. The code is attached to each relevant method. All messages from a lifeline starting from a method invocation on the lifeline to the return message (a message describing return to the caller of this message or a message to UI) are transformed to actions in the code for this method. For storing a code, corresponding to an operation, the UML comments are used (the initial code is not a standard UML metamodel element). The transformation for code creation iterates through all messages in the sequence diagram. The search is performed in a recursive way (based on a stack). When it detects a call of some operation, it means the following messages will constitute the body of this operation. If a call to another operation follows this operation, the call to this other operation is added to the code body of this operation and this operation is added to the stack; and the newly created operation is set to be the current. If return from this operation to the previous operation is detected, the previous operation is popped out from the stack. If self messages are detected, an appropriate code is simply added to the message body. The stack is implemented by using the UML comments since it was not possible to extend the metamodel with temporary classes (due to the requirements of other tool components).

Summary

Implementation of these transformation rules in the Keyword-Based Architecture Style took approximately 3 person months. Implementation of these transformation rules consists of about 140 MOLA procedures (of a size similar to the one given Fig. 46, or Fig. 47, p.109). Implementation of rules, currently missing, would be a small part of the existing code.

3.6.3 Model-to-Code Transformation Implementation

Many MDD-based tools offer code generation from the UML models. The *Enterprise Architect* (EA), the modelling tool used in the ReDSeeDS project, has the *Code Template Framework* (CTF) which also provides code generation features. Just like most of code generation tools in the MDSD world, EA does not provide a full code generation, but code skeletons (classes, interfaces, fields and operation declarations) can be obtained. Only packages, classes, and interfaces are used by these templates, the other UML elements are ignored. These templates are called base templates. The latest versions of EA (not used in the project) provide some code generation features for behavioural UML diagrams as well (sequence, state).

Since the ReDSeeDS project uses EA for UML support, there is a possibility to reuse all CTF capabilities of code generation. It is a significantly easier way to obtain a code than to generate a *Java* model as the first step and then convert this model to a proper code.

Base templates can be used directly for the *ReDSeeDS Basic* style. These templates are applied to a *Detailed design* model of this *architecture style*. The package hierarchy, declarations of all classes (DAO, DTO, etc.), and methods are included in the generated code. Bodies of the obtained methods should be filled in manually since the detailed design model in this style contains no behaviour.

For the *Keyword-Based* style, significantly more code can be generated, including the behaviour aspects. Base templates do not generate the declarative annotations used in the Keyword-Based architecture style. We underline that these annotations are specified in the platform-specific model as appropriate stereotypes of classes, attributes, and associations. However, code generation templates are defined by using the model-to-text language (the CTF language) in EA. Thus, it is possible to customize the way in which CTF generates a source code. The extension of the Java code generation template for the *Spring* framework has been built. The generated code contains *Spring* annotations obtained from the stereotypes.

Although behavioural diagrams cannot be properly used for code generation in EA, they can be processed by model transformations before the code generation step. For example, a MOLA transformation converting a message and action sequence in a sequence diagram into a part of the code of the appropriate method body has been implemented by using an intermediate model. Then such an enriched intermediate model can be further processed by the code generation templates in EA. Since such pre-processing is done, a great portion of the code (for example, method invocations from sequence diagrams) is being generated, using EA. This way a meaningful executable prototype code could be obtained directly from the requirements. If the models in the software platform-independent and platform-specific models have been extended manually, a true model-driven development can be carried out by this approach.

3.6.4 Integration with the Enterprise Architect

As already stated the *Enterprise Architect* (EA) tool [156] was used as the UML editor in the ReDSeeDS project. However, it was necessary to exchange the UML models between the ReDSeeDS repository and EA, as EA was used to visualize and modify the UML models created by model transformations.

The data exchange was done by using import/export procedure. It was possible to export the data to EA and then the user could modify the data using EA. After that the data were imported back to the ReDSeeDS repository.

The data export to EA was done in two steps. In the first step the UML model was transformed to the EA encoding of UML. A metamodel describing the structure of EA *Application Programming Interface* (API) was used as the EA encoding of UML. The first step was implemented in model-to-model transformation. The second step was implemented by a *Java* program that was reading the data in the EA encoding of UML and feeding them in EA by using EA API.

The data import from EA was performed similarly in two steps. At first the data from EA API were transferred to the EA encoding of UML by using a Java program. As the second step the data from the EA encoding of UML were transformed to the UML model.

This two step data exchange was selected because the UML encoding in the ReDSeeDS repository and in EA was very different. These differences were mainly due to a strange encoding of the UML models in EA. For example, enumeration was encoded as a class with the stereotype “*enumeration*”.

The author of the thesis implemented model transformations from EA encoding of UML to UML and back. The transformation from UML to EA was implemented by using

23 of the MOLA procedures. The transformation from EA to UML was implemented by using 39 MOLA procedures.

3.7 Conclusions

In this chapter a model-driven path from the requirements to the code is studied. Two different paths built in the ReDSeeDS project are analyzed. Transformations supporting these paths are typical transformations used in Model-Driven Software Development. This is a great case study in building transformations for Model-Driven Software Development from which various conclusions can be drawn.

Almost each model consisted of a static structure description and behaviour description. When creating static structure descriptions, mainly the copy library for the selected UML subset was used. Creation of static structure usually meant copying elements from one model to another with some small modifications. Although the copy library helped a lot in static structure transformation development, here still was a lot of routine job and the amount of static structure transformations was big enough. Creation of static structure could be described by using the mapping similar to the ones used in Fig. 38 (p.83). The effort required to build all these different cases of static structure processing in the project was the main stimulus to develop the mapping languages to be described in the next chapter.

Transformations, creating the behaviour part of models, were more advanced. The most complicated part was creation of sequence diagrams from the requirements. This task required quite complicated analysis of the requirements to produce appropriate sequence diagrams. The algorithm was very complicated. Another issue was work with an annoying UML metamodel for sequence diagrams. To ease work with sequence diagrams a library for processing the sequence diagrams was created and widely used. The library helped to separate logical work from technical processing of the UML model. In general, the classical pattern and the rule based transformation paradigm seemed to be the most appropriate for this part of task – thus making MOLA a very adequate implementation language for it.

The most complex transformations were transformations generating the initial code. Here a stack was required to keep track in which operation the code, corresponding to this sequence diagram message, should be included. In MOLA there is no natural support for a stack. Therefore, it was necessary to emulate all stack operations by using transformations. It was also hard to determine whether this is a forward call or a call back when using sequence diagram metamodel instances. Though MOLA could be used for this task, clearly a specific language extension for collection processing (similar to such libraries in the OOP languages) would be of high value.

The number of MOLA procedures for each task is given in Table 4. The number of transformations related to static structure processing and behaviour processing is also provided.

Table 4. MOLA procedure count in different transformations. Classified as to processing static structure, behaviour or independent operations.

Type	Transformation	Static structure	Behaviour	Other	**Total**
Basic Style	RSL to PIM	12	19	3	**34**
	PIM to PSM	8		1	**9**
Keyword-Based Style	Keyword Analysis			4	**4**
	RSL to Analysis	8		2	**10**
	RSL, Analysis to PIM	16	32	5	**53**
	PIM to PSM	14	2	2	**18**
	PSM to Code			9	**9**
Libraries	Copy library	23	9		**32**
	Sequence processing		9		**9**
	Traceability library			4	**4**
	Delete			7	**7**
	Other			24	**24**
Technical	RSL visualization			19	**19**
	Merge			26	**26**
	UML -> EA			35	**35**
	EA -> UML			41	**41**
	Test			22	**22**
Total		**81**	**71**	**204**	**356**

CHAPTER 4

Mapping Languages

4.1 Mapping Idea

Transformations could be treated as mappings between the source and the target models. However, not any transformation language is a mapping language. The author of the thesis believes that mapping should be defined in terms of simple relations, most probably represented by simple arrows from one element to another. Simplicity is the key. However, in traditional transformation languages it is possible to write down very complicated conditions. For example, in one sub-case A should be transformed to B, in another sub-case A should be skipped, and in a third sub-case A should be transformed to C. To describe these complicated options, all kinds of conditions spoil the simplicity of these languages.

However, in many transformation languages, especially in declarative ones, there appear some elements of mappings. A pattern with the source and the target elements separated could be considered a mapping element. One side of the diagram describes what should be transformed and the other side – what should be created. . Mapping elements in transformation languages are described in detail in Section 4.1.1.

Although in OMG RFP [119] and in the MDA guide [111] the term mappings has been used, today transformation languages are not treated as mapping languages. We may consider that in general the mapping idea in transformation languages has failed. Irrespective of that there have been attempts to create universal mapping languages. Usually these languages are incomplete. They are practically applicable only in simple cases when the relation between the source and the target is simple. To make them applicable in all transformation tasks they should possess a full power of model transformation languages. It means that they should have the same complexity as in model transformation languages. These languages are described in detail in Section 4.1.2.

An interesting approach is used in Atlas Model Weaver (AMW) [39]], proposing a universal mapping language. However, this mapping language is only a basis for defining specialized mapping languages.

To specialise this general purpose mapping language, in fact, a new mapping language should be built. This new mapping language should contain details specific to the domain processed – a feature typical of domain-specific languages. As a result we can speak about domain-specific mapping languages that could be more expressive than general purpose languages, not loosing simplicity and understandability of the language. Domain-specific mapping languages are discussed in Section 4.4.

Another view on mappings holds that they should be treated as an initial skeleton of transformations to be built. An approach of this type is proposed in [50]. Mappings build a skeleton of transformations and details are filled in the transformation language. In this case the transformation sources are generated from mapping. To describe transformation generation from mappings higher-order transformations could be used. A mapping language compilation using higher-order transformations is described in Section 7.2.

4.1.1 Transformation Languages and Mapping Languages

There is no formal generally accepted definition on considering a language either a model transformation language or a mapping language. However, in practice there is a more or less common understanding and we present our interpretation of it.

A model transformation language focuses on a precise executable transformation definition (that results in "*Turing model completeness*"). Currently, most of the transformation languages rely on the pattern-rule paradigm. A pattern specifies what fragment is to be found in the source model and a rule specifies what is to be done on the basis of this fragment (in-place update or creation in the target model). Certainly, there are big differences how the rule execution order is controlled – in a non-deterministic way aided by various guards (*NACs*, *when* and *where* conditions, etc.) or within some classic control structure.

The main paradigm of a mapping language is a direct specification of a set of correspondences between the source metamodel and the metamodel elements. The idea of correspondence is as follows – for each instance of the source metamodel element the corresponding target instance is created (or its existence is checked). An additional standard requirement claims for the language to be very easily readable; therefore frequently the correspondences are visualized as simple arrows between the metamodel elements. Other features of a mapping language depend on its use. A mapping language may simply serve as a facility for defining transformation drafts (abstractions). Then a transformation is manually created on this basis (with a possible automatic skeleton generation). Alternatively, a mapping language may serve as a precise, but still easily readable transformation specification. Then mappings are used as a source for generation of the actual transformation definition in a transformation language. To increase the expressiveness various additional features are added (filters, constraints, assignments, etc.) while trying to preserve the readability; however, completeness is not so easily reachable this way. To illustrate the main ideas behind the mapping concept a short overview of mapping specification languages is given in the next section.

An alternative way to meet both criteria (expressiveness and readability) is to narrow the application domain of a language – build a domain-specific mapping language. In this chapter we present a language exactly of that kind. Such a language will cover all typical cases of mappings in the given domain and will satisfy the readability requirement. Certainly, there is always an option to extend the generated transformation definition manually.

We conclude the section with some remarks on using the mapping ideas within some transformation languages. Thus, in *MOF QVT Relational* [128] (especially the graphical form) each relation reminds of a visual mapping in the case when both patterns are reduced to the corresponding metamodel elements. Fig. 48 presents a small transformation example in *MOF QVT Relational* [128]. The left side of the figure contains a fragment of the source model and the right side of the figure contains a fragment of the target model. Actually, *MOF QVT Relational* is bidirectional, therefore the source and the target models could be exchanged. In fact, this small example reminds of a mapping which defines that one source model fragment should be transformed to another target model fragment. However, as soon as more constraints are added the set of relations becomes significantly less readable and a transformation with complicated constraints does not remind of mappings anymore. So we can conclude that there are mapping elements in *MOF QVT Relational*.

It should be noted that the *MOF QVT Operational Mapping* sublanguage has preserved the term *mapping* for denoting an operation of creating a target model element from a source model element. However, this operation is more an elementary transformation element with various conditions and helper operations around than a relation in *MOF QVT Relational*.

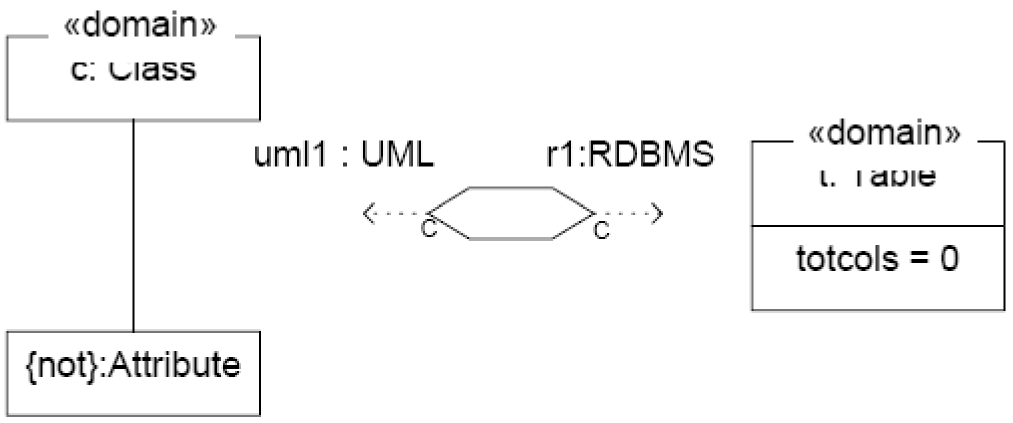


Fig. 48. MOF QVT Relational example

A similar effect appears in some other languages as well, e.g., in *ATL* [63], *AGG* [163]. A special situation is with the *TGG* [146] that has so many mapping features that sometimes is considered to be on the borderline. TGG is a graph transformation language extended with mapping elements. An intermediate model or a mapping model is used explicitly defining a transformation in TGG. In this model relations between the source and the target are directly represented. However, when the full power of patterns and *NACs* is used in TGG, it is more a traditional transformation language.

Another remark concerns bidirectionality that is an important issue for transformations, but it is out of the scope for this research since it is not so significant for our domain.

Some mapping elements could also be observed in the model transformation language MOLA (described in CHAPTER 2), although they are not as direct as in some other languages. The MOLA rule consists of a pattern and action part, although these parts are not strictly separated. The pattern part could be treated as a source of mapping and the action part – as a target of the mapping.

4.1.2 General Purpose Mapping Languages

Attempts to create universal mapping languages as a certain alternative to traditional transformation languages have been started sufficiently early.

An attempt to describe the mapping concept more precisely was made in the paper by Hausmann and Kent [51] in 2003. They used the term *mapping* to address the general understanding of connection between models and offered a graphical mapping language to specify mappings. However, the precise functionality of mappings had to be defined in OCL thus these relatively simple diagrams actually meant a complicated programming in OCL, in addition, their primary concern was bidirectionality.

In the thesis of Lopes [102] the Hausmann's and Kent's ideas have been developed much further – a mapping specification language (no special name was given to it) has been created and implemented as an *Eclipse plug-in* Lopes considered the universal approach - the specification of mappings between two arbitrary metamodels. Mapping specification (*mapping model*) has been used to generate the actual model transformation definition in ATL, more or less complete transformations could be generated if mappings were detailed by appropriate OCL expressions. In addition, the usage of abstract syntax (standard UML metamodel) has led to complicated mappings even for simple tasks.

Atlas Model Weaver (AMW) [39] provides a generic infrastructure and editor to declaratively specify weaving models between two arbitrary models. The weaving models are used to capture different kinds of links between model elements. The links have different semantics, depending on the application scenario. In fact, AMW provides a generic mapping (core) metamodel which should be extended in particular case. The *Higher-Order Transformations* (HOT) generate actual model transformations.

The most recent approach uses composite *Mapping Operators (MOps)* [198]. The basic mapping operators called *kernel MOps* provide the basic types of possible mappings (like class to class, attribute to attribute, relation to relation, etc.). Kernel MOps can be composed into more advanced mapping operators – *composite MOps*. Composite MOps can be easily reused further once defined. This approach has been implemented on the basis of AMW and also generates ATL using HOTs. All abovementioned mapping languages are general purpose ones, applicable to any domain and are based on the abstract syntax.

Another view on mapping languages is given in Guerra et al. [50], where it is proposed to use mappings as requirements specification for transformations. Mapping diagrams of *transML* (a language family for development of model transformations) are used for high-level design of model transformations; from these diagrams only transformation skeletons can be generated.

We believe that a mapping language should not be universal and complete in order to preserve the readability. If a mapping language is complete, then really it is a new transformation language. The mapping language should be used only for typical cases. There should be a close integration with a model transformation language and the rest should be written in this traditional model transformation language.

4.2 Domain-Specific Mapping Languages

As it was stated in Section 1.2.3 specialised modelling languages - *Domain-Specific Modelling Languages* – are used for specialised modelling areas. These languages are suitable for use in concrete domains. Domain-specific languages contain terms specific to the domain as language elements. Consequently, language users can operate with terms familiar to them. It raises the abstraction level and increases productivity as well.

4.2.1 Domain-Specific Model Transformations

Similarly to modelling languages there are model transformation languages suitable for certain domains. Actually, each model transformation language is more or less dedicated to a certain domain. For example, MOLA is suitable for model transformation development in MDSD. In the *Epsilon project* [93] a multi language framework has been built. This framework consists of several languages. Each of these languages is dedicated to a specialised group of transformation tasks. These languages are: *Epsilon Transformation Language* (ETL), *Epsilon Validation Language* (EVL), *Epsilon Generation Language* (EGL), *Epsilon Wizard Language* (EWL), *Epsilon Comparison Language* (ECL), *Epsilon Merging Language* (EML), *Epsilon Flock* (a language for model migration).

Model migration could be mentioned as a concrete transformation domain. Currently, there are two specialised languages for model migration: *COPE* [52] and *Epsilon Flock* [141]. In some sense these languages are mapping languages as there declarative means are used to specify relations between the source and the target models. It should be noted that specialised transformation languages perform better than languages of general purpose. In TTC 2010 the same model migration task [142] was implemented in 9 model transformation languages. The best results [140, 184] were reached by the specialised languages *COPE* [52] and *Epsilon Flock* [141].

4.2.2 Domain-Specific Mapping Languages

There are domain-specific mapping languages suitable for certain domains and based on concrete metamodels. The following are examples of such mapping languages:

- The language *R2RML* to map RDB to RDF [191] is currently under development by W3C. A draft is available [195].
- *D2RQ Mapping Language* [44] is a declarative language for describing the relation between a relational database schema and the RDFS vocabularies or the OWL ontologies.
- *D2R map* [20] is a database to the RDF mapping language.
- *Silk-LSL* (Silk Link Specification Language) [1] is provided by the Silk framework. It is a declarative language for specifying which types of RDF links should be discovered between the data sources, as well as which conditions the data items must meet in order to be interlinked.
- *RDB to OWL* [22] defines mappings to transform the RDB data to the OWL data.
- *Epsilon Flock* [141] and *COPE* [52] for model migration.

Domain-specific mapping languages may be graphical, textual or tool driven. For example, *Epsilon Flock* [141] is textual, *COPE* [52] is tool driven and *MALA4MDSD*, proposed in Section 4.3, is graphical.

There are not so many domain-specific mapping languages, therefore research on the creation of such languages is of importance. In the present thesis two mapping languages of this type are proposed. A mapping language for MDSD is described in Section 4.3 and a mapping language for the DSL tool development is described in Section 5.3.

4.3 MALA4MDSD – Mapping Language for MDSD

In this section a mapping language for MDSD - MALA4MDSD is proposed. This language is domain-specific. It is built to transform one UML model to another UML model. A typical application of such language is transformations from PIM to PSM in the MDA lifecycle. Actually, the language does not support full UML - it supports only a UML subset typically used in MDSD. More precisely, the described subset is meant for transforming only the static structure of an UML model (however, it could be easily extended to include many behaviour-related elements as well).

Unlike the mapping language approaches described in Section 4.1.2, we propose to base the mapping language on a concrete syntax of the source and the target languages. A similar idea has already been applied to transformation languages, e.g., in *AToM*^{3} [96] and [49].

The language demonstrates the cornerstones of our approach – the source and the target model structures are represented by trees. Tree nodes specify what kind of model elements appear in the given context and the mapping relations (arrows) from the source to the target tree nodes specify which kind of the target model elements are created from which source elements. Tree nodes do not correspond directly to UML metamodel classes (abstract syntax, as in [102, 39]), but to concrete syntax elements – types of nodes typically found in UML model trees in various UML tools (a sort of de-facto tree syntax of UML). This makes the tree notation significantly more readable (no large amount of abstract classes is to be shown).

4.3.1 MALA4MDSD Motivation

The usage of model transformation languages requires highly skilled specialists with deep knowledge of metamodelling. That is one of the main reasons why the industry has not yet widely accepted the MD* approaches and most of model transformation languages are used only by a small group of people closely related to language developers.

Domain-Specific Modelling (DSM) proposes to use modelling languages that use notation and concepts specific to the domain actually being modelled. It narrows the gap between languages being used to describe the problem and the solution. Similar principles may be applied to model transformation languages. Instead of using a general purpose model transformation language we propose to use domain-specific transformation languages that use elements specific to the models being transformed. Most of the model transformation languages (including the standard MOF-QVT) use abstract syntax (metamodels) to specify model transformation definitions. However, users of the modelling languages use only the concrete syntax of the language. Thus, the domain-specific model transformation language should use familiar concepts for modelling experts: the concrete syntax of the modelling language.

This should lead to the shift of roles of developers in the *Model-Driven Software Development (MDSD)* process (see Fig. 49). Metamodelling experts (highly skilled professionals) would be the developers of domain-specific modelling languages, using all the arsenal of technologies they have. The software developers (modellers) would become the actual developers and users of model transformations. Thus, the former model transformation users would become model transformation developers (and users), but the former model transformation developers would become model transformation language developers.

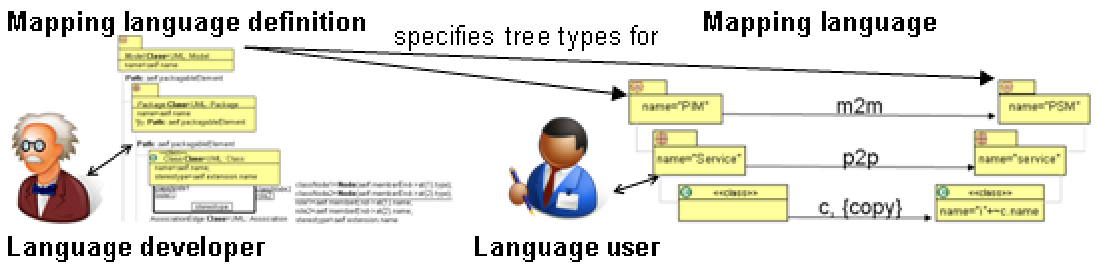


Fig. 49. Schematic roles of the mapping language family users

Another crucial aspect for a domain-specific model transformation language is the use of convenient means to represent the correspondences between the source and the target model elements in the model transformation definition. The most intuitive option to define model transformations is to use *mappings*. Mappings permit to specify transformations in a simple way, frequently by very intuitive graphics. From the very beginning of model transformation languages there has been an intention to define transformations as simple mappings. The expressive power of such general purpose mapping languages is limited; however, we demonstrate that mappings are expressive enough for transformations in specific domains.

In this section we propose an approach for building domain-specific transformation languages based on simple mappings and the concrete syntax of models being transformed so as to reach simplicity, readability and sufficient expressiveness of the language at the same time.

This section proceeds with the description of one domain-specific mapping language – MALA4MDSD. Actually, the approach proposed could be applied to a mapping language family. The mapping language described in this section is only one instance of the mapping language family. Mapping languages in the family differ by the used concrete syntax trees. The MALA4MDSD description contains occasional remarks whether the described feature is specific to MALA4MDSD or common to all mapping language family.

Section 4.4 is devoted to the description of obtaining languages of the mapping language family.

4.3.2 Basics of MALA4MDSD

The UML model structure is greatly determined by the composition relationship in general. Therefore, in practice it is sufficient to represent the UML model structure as trees. The source and the target in this domain-specific language are UML models within the same subset; consequently, both trees can contain the same kinds of nodes. For the chosen UML subset there is a predefined set of nodes to be used in a tree. It is natural to think of trees in this mapping language as UML instance tree patterns. They represent a possible structure of an instance tree in a typical UML tool containing the source or the target model. For example, it means that if a specific mapping requires that there should be a package inside a package there will be two hierarchical package nodes in our tree.

The source and the target tree nodes are connected by using mapping relations. A mapping relation means that if an instance corresponding to the source node is found in the source model then an appropriate instance should be created in the target model (here we should think of both models to be represented by their instance trees). The source tree is traversed in a top – down manner. For each valid instance of the source node the outgoing mappings are executed (i.e., target instances created). The validity of an instance is checked by using the containment relationship to the parent and the filter conditions. For the target nodes it is possible to use attribute assignment expressions to define the attribute values of the newly created instance.

A simple mapping example is presented in Fig. 50. The topmost mapping relation is executed first. It maps two UML models. In the source a UML model named “PIM” is sought for. For each such model a UML model named “PSM” is created in the target. In the real transformation context from which this example is taken there is only one model instance named “PIM” available in the source, but we do not distinguish this situation syntactically in our language. Then the second mapping is executed. The packages named “Service” in the UML model “PIM” are found. For each such package (in this case again actually only one) the corresponding package named “service” in the target UML model “PSM” is created. The third mapping relation copies all classes in the source model package “Service” to the target model package “service”. Classes with all child elements (here – attributes and operations) are copied because the *copy* modifier is used (for details see Section 4.3.5). The name of the target class is calculated using an expression. The prefix “i” is added to the source model class name; pay attention to the use of the reference “~c” to navigate the mapping named “c” from the target to the source. Thus, the expression “~c.name” gives us the name of the mapping source node (class).

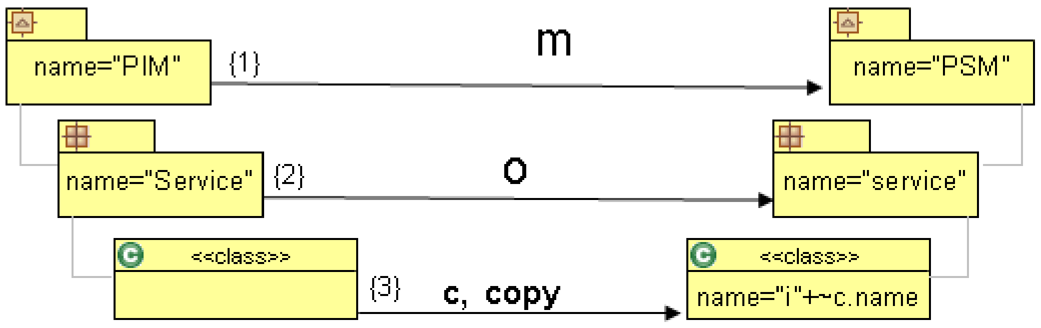


Fig. 50. MALA4MDSD example. UML model “PIM” is transformed to UML model “PSM”. Package “Service” in model “PIM” is transformed to package “service” in “PSM” model. Classes from source model package “Service” are copied to target package “service”.

4.3.3 MALA4MDSD Elements

The list of MALA4MDSD elements is given in Table 5, consisting of two parts. The first part of the table presents MALA4MDSD tree elements, defining the role of each element in the UML model, the attributes usable in MALA4MDSD and the possible child elements. In the second part the elements of the mapping language family are presented.

Table 5. List of MALA4MDSD elements

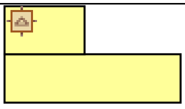
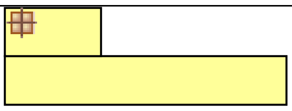
Image	Element	Description
Tree type elements		
	Model node	Corresponds to UML model. **Attributes**: name; **Child elements**: package node, recursive package node.
	Package node	Corresponds to UML package. **Attributes**: name;

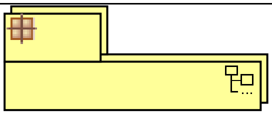
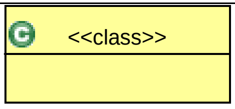
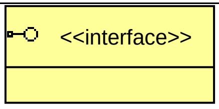
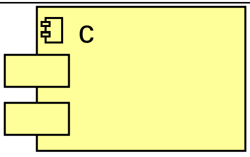
Image	Element	Description
		Child elements: package node, recursive package node, class node, interface node, component node, enumeration node, data type node, actor node, interaction node.
	Recursive package node	Describes the package hierarchy of arbitrary depth in the UML model. All elements in the hierarchy independently of depth are treated as children. **Attributes**: name; **Child elements**: class node, interface node, component node, enumeration node, data type node, actor node, interaction node; **Description**: see Section 4.3.5.
	Class node	Corresponds to UML class. **Attributes**: name, stereotype; **Child elements**: attribute node, operation node.
	Interface node	Corresponds to UML interface. **Attributes**: name; **Child elements**: attribute node, operation node.
	Component node	Corresponds to UML component. **Attributes**: name; **Child elements**: interface node.

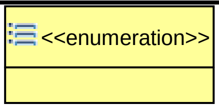
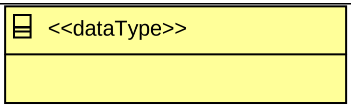
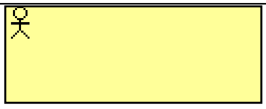
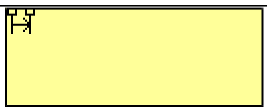
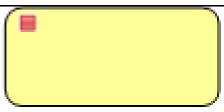
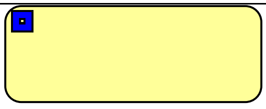
Image	Element	Description
	Enumeration node	Corresponds to UML enumeration. **Attributes**: name; **Child elements**: enumeration literals (in this use case not used explicitly).
	Data type node	Corresponds to UML data type. **Attributes**: name.
	Actor node	Corresponds to UML actor. **Attributes**: name.
	Interaction node	Corresponds to UML interaction (sequence diagram). **Attributes**: name; **Child elements:** in this use case the child elements are not used explicitly. However, all sequence diagram elements should be treated as child elements.
	Operation node	Corresponds to UML operation. **Attributes**: name, stereotype, type (primitive type name or reference to type: class node or enumeration node); **Child elements**: parameter node.
	Parameter node	Corresponds to UML operation parameter. **Attributes**: name, direction (enumeration: set of fixed values), type (primitive type

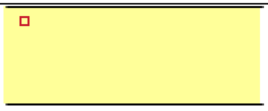
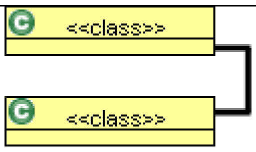
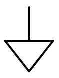
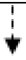
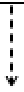
Image	Element	Description
		name or reference to type: class node or enumeration node).
	Attribute node	Corresponds to UML attribute (coded as property without association in UML model). **Attributes**: name, stereotype, type (primitive type name or reference to type: class node or enumeration node).
	Association edge	Corresponds to UML association. **Source node type:** class node; **Target node type:** class node; **Attributes:** stereotype, source role, target role.
	Generalization edge	Corresponds to UML generalization. **Source node type:** class node; **Target node type:** class node.
	Realisation edge	Corresponds to UML realisation. **Source node type:** interface node; **Target node type:** class node.
	Dependency edge	Corresponds to UML dependency. **Source node type:** class node; **Target node type:** class node.

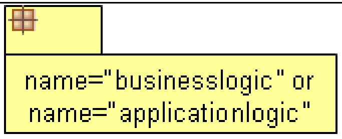
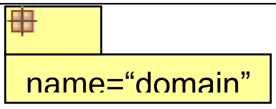
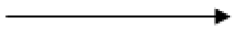
Image	Element	Description
Mapping elements		
	Constraint	In the *source tree* it is possible to define constraints in a tree element. Constraint means that only instances satisfying this constraint will be processed. Constraint language is a simplified version of OCL. Here it is possible to reference tree type elements.
	Attribute assignment	In the *target tree* it is possible to assign values to attributes defined in the tree type. Assignments are described as follows: *<attribute>=<expression>*. Expression is defined in a simplified version of OCL. It describes how the attribute value to be assigned is evaluated. Expressions are described in detail in Section 4.3.5.
	Mapping	Mapping relates the source and the target trees. It describes from which source tree element which target tree element should be created. Default mapping When mapping creates a node in the

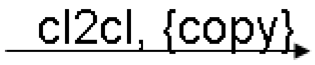
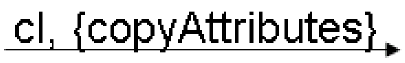
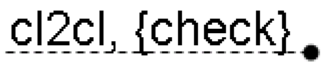
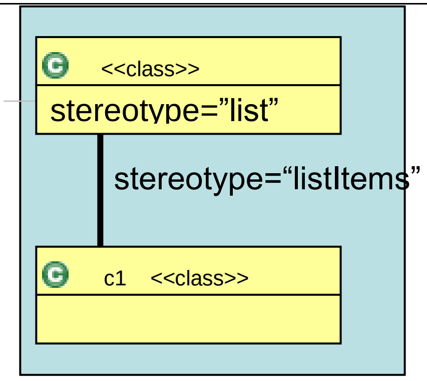
Image	Element	Description
		target model a traceability link is created. Before creation of the target instance a traceability link is used for checking whether there is a node in the target corresponding to mapping in this target context. If such an instance is found it is used and nothing is created. In case such an instance is missing a new instance is created. Mappings are ordered top down. Mappings have names that could be used in the OCL expressions. If mapping is traversed in the opposite direction the name is prefixed with the “~” symbol.
	Mapping c*opy*	*Copy* modifier means that this element and all its child elements should be copied to the target model. This modifier could be used only on mapping relating nodes of the same type. If assignment is used in the target node it rewrites the default value of the attribute obtained using *copy*. **Description**: see Section 4.3.5.
	Mapping *copyAttributes*	*CopyAttributes* modifier is used to copy the node and all its

Image	Element	Description
		attribute values. Child elements are not processed. This modifier could be used only on mapping relating nodes of the same type. If assignment is used in the target node it rewrites the default value of the attribute obtained using *copyAttributes*. **Description**: see Section 4.3.5.
	Mapping *check*	Check modifier means that a node in the target model must be found. Creating a node in the target model a traceability link corresponding to the mapping used is created. In this case the traceability link is used to find the node already created by mapping with this name. **Description**: see Section 4.3.5.

 <<class>>
stereotype="list"
stereotype="listItems"
c1 <<class>>
	Pattern	In the source tree patterns could be used to describe complicated mapping application conditions. **Description**: see Section 4.3.5.

	Custom MOLA procedure	It is possible to call custom MOLA procedures. For these procedures the first parameter should be the parent of

Image	Element	Description
		"Custom MOLA procedure" node. The type of this tree type element should correspond to the MOLA parameter. All other parameters are of the type in/out and are represented as child nodes. In this case the types should correspond again. **Description**: see Section 4.3.5.

4.3.4 MALA4MDSD UML Tree Type

To be able to define transformations it should be clear to a user what kind of elements in the source and the target trees could be used. For each tree type element the possible attributes and child elements should be defined.

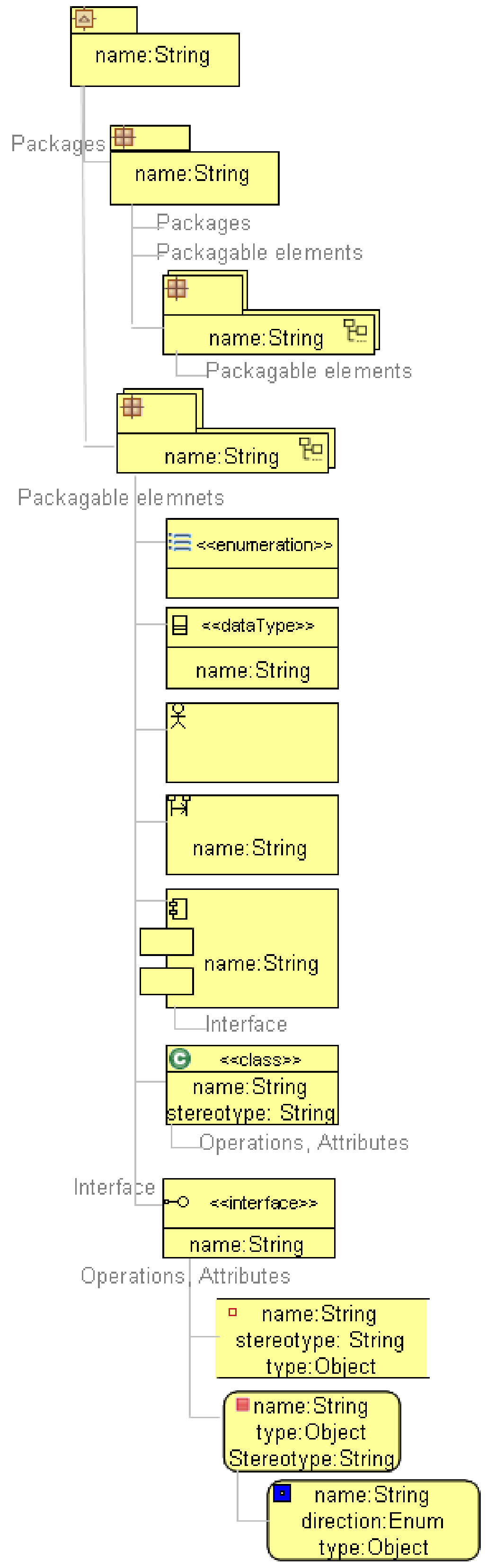
For the UML tree type used in MALA4MDSD the root node is always *Model* that can contain *Packages. Package* can contain other *Packages, Classes, Interfaces, Components, DataTypes, Actors, Interactions* and *Enumerations. Class* and *Interface* are allowed to contain *Attributes* and *Operations. Operations* contain their *Parameters.* Each of the node types has a predefined set of attributes (name, etc.).

In Table 5 all elements of MALA4MDSD tree type are listed. For each element the possible child elements and attributes are listed. However, it would be easier for a user if he/she could see these possible containments graphically and there are two alternative ways for their graphical representation. One of them is to show the tree containing the possible elements in each position. It is possible to give a name to a sub-tree and explicitly define it only once, if the sub-tree is used multiple times. Such a tree is presented in Fig. 51. This tree is very useful as a reader can easily see what kind of elements could be used as sub-elements of the given element. However, if the language has many elements this tree may get very large. Even for the UML subset used in MALA4MDSD it is hard to fit this tree on one page. An alternative option is to use the syntax similar to the *context free grammars* [149]. In this case the *non-terminal symbols* are the names attached to tree fragments. A complete tree is built by replacing the *non-*

terminal symbols with the appropriate tree fragments. This type of representing the UML tree type used in MALA4MDSD is given in Fig. 52. This syntax is more useful for large tree types as it is possible to split it in several small images. Both representations are equivalent: the first is more suitable for small tree types and the other – for larger tree types.

The source and the target trees in mapping languages are defined according to the tree type definition. The tree node type of root elements in the source and the target trees should be the same as in the tree type definition. Only the parent-child relations defined in the tree type are permitted in the source and the target trees. However, a child of the same type could be repeated multiple times in different contexts. Children of some type could be omitted if they are not needed in the defined mapping diagram (transformation). However, it is not possible to skip some intermediate elements from the tree. For example, it is not allowed to use *Parameter* directly as a child of *Class*. The parent of *Parameter* should be *Operation*.

In the source tree it is possible to add constraints to elements. It means the same tree node type could be used multiple times as a child of the same element with different conditions. In the target tree it is possible to add assignments to elements and the same element type could appear multiple times as well. It is used if different mappings describe the creation of elements in the same context.



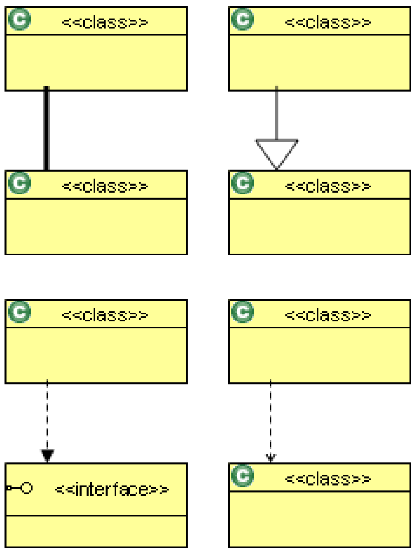


Fig. 51. MALA4MDSD UML tree type definition

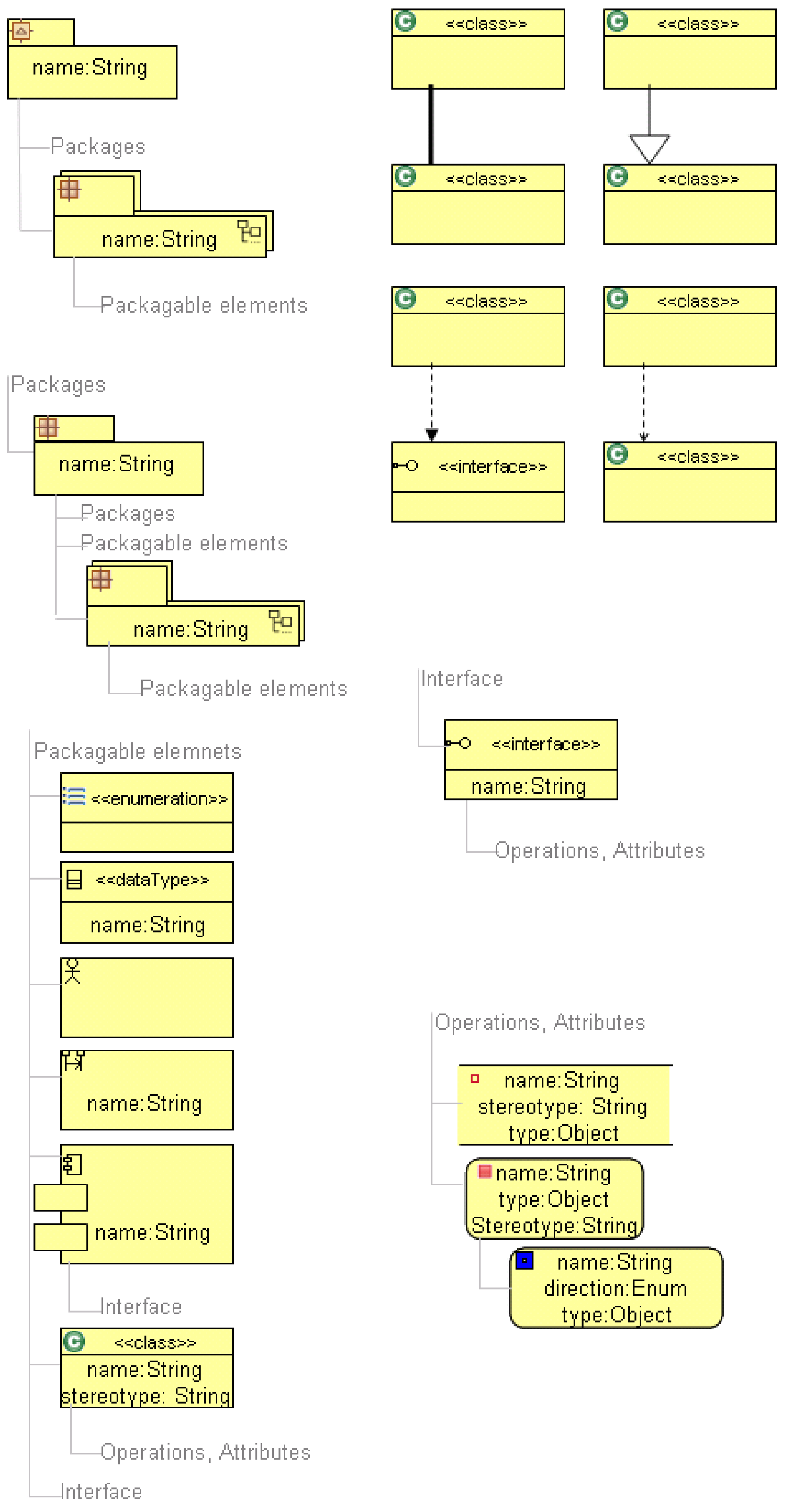


Fig. 52. Alternative tree type definition

4.3.5 More Advanced Mapping Elements

Elements described in Section 4.3.2 are the core of the proposed mapping language. To facilitate the transformation development in this mapping language some more features are introduced.

For some tasks large source and target trees with many mapping relations must be built, therefore there is a need to divide mappings into smaller sub-diagrams. One mapping program (transformation) consists of several ordered mapping diagrams. They are executed separately in the given order. The root of each tree in the mapping diagram should be the tree node of the root type in the used tree type.

Mapping Modifiers

As it was mentioned in Section 4.3.2 there are special mapping modifiers. A mapping with the *copyAttributes* modifier specifies that in the target node for each attribute an implicit assignment is performed, setting it to the value corresponding to that value in the source node.

The *copy* modifier is even more powerful. It specifies that implicit mappings are performed with the *copyAttributes* modifier for all children types of the node (at any depth, according to the tree type definition). This is a very powerful feature for copying tree fragments where nothing has to be modified. Certainly, the node types for *copy* must be the same. In Fig. 53 (p.138) the *copy* modifier is used for enumeration and class nodes. For enumeration the copied child elements are enumeration literals. For classes the child elements are attributes and operations, operations in turn are copied with their parameters – according to the type hierarchy in the tree type.

The third mapping modifier *check* means that nothing is created in the target tree, only the relevant node is found by using traces between the source and the target (another kind of arrowhead is used here). Such mappings are necessary, and as an example here may serve the location of edge endpoints in the target tree as in Fig. 53 (p.138).

Expressions

Constraint can be used for tree nodes in the source tree. Constraints are used to restrict the set of instances corresponding to the tree node. Constraints are defined by using an OCL subset. In expressions a supported OCL subset is similar to a supported

OCL subset in MOLA. In the OCL expressions tree type attributes could be used. Actually, the most popular constraint type implies adding a condition which checks that the values of attributes satisfy the conditions defined by the constraint. It is also possible to navigate the tree upwards; in this case "*.parent*" navigation is used.

Expressions are used in the target tree to define the attribute value assignments. It is possible to traverse mappings in these expressions. If mapping is traversed in the opposite direction, the name is prefixed with the "~" symbol. Mapping traversion is defined as navigation in the OCL expression. Similarly to constraint the attribute values could be used in these expressions as well.

Recursive Elements

As it was already mentioned in the previous section a UML package can contain other packages. It means there could be a package hierarchy with arbitrary depth. Sometimes we want to process this hierarchy in a generic way. Therefore, in our mapping language for packages it is possible to use a special type of node representing the whole package hierarchy (see Fig. 53, the 3rd node in the source tree). It means that the mapping applies not only to the packages in this level, but to all packages in the hierarchy. This modifier could be used in the source tree, as well as in the target tree. If the modifier is used in the source and the target trees it means that the package hierarchy must be preserved in the target as well. If the modifier is used only in the source tree it means that in the target tree the package hierarchy must be flattened. It is possible to add child elements to this package hierarchy, e.g., classes: if it is done, all classes in this hierarchy (satisfying other constraints) should be processed.

Edge Processing

So far we have considered only nodes in a UML model. However, there are also edges in UML (in the sense of diagram syntax). These edges should be processed some way as well. Therefore we add to our language edges typical of a UML model: *Association*, *Generalization*, *Implementation* and *Dependency*. These edges are represented as links between the tree nodes. Edges can be used both in the source and the target trees, edges can be mapped as well. The edge processing is done after both nodes connected by this edge are processed. In general, the edge end instances in the target are determined by maps of the corresponding line ends in the source; in more complicated cases patterns should be used.

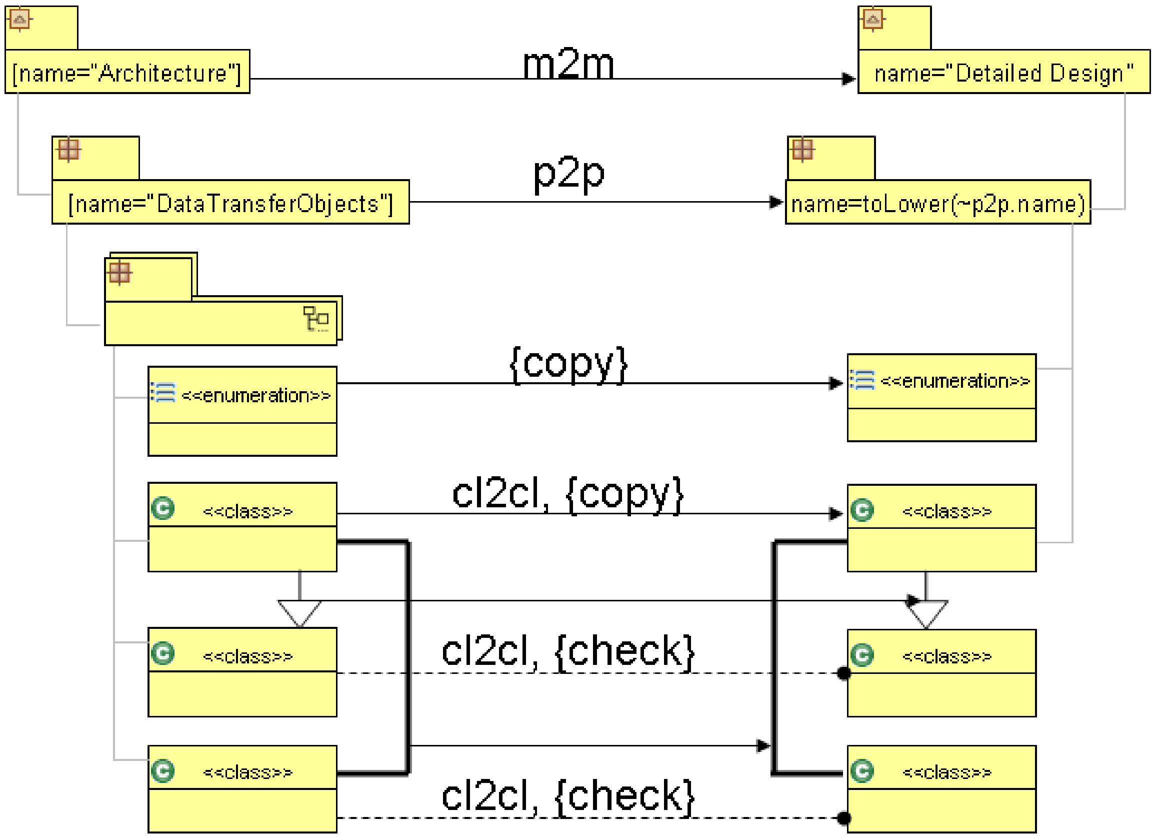


Fig. 53. Mapping example from the ReDSeeDS project. Transformation in MALA4MDSD, demonstrating the edge processing and hierarchy flattening

An edge mapping is given in Fig. 53. All *Associations* and *Generalizations* between classes in the predefined package hierarchy are copied to the target. All classes in this hierarchy have already been copied before the edge processing (by the mapping *cl2cl*). To find for an association the other end class in the target the mapping *cl2cl* is duplicated from another class node in the source (the other end of edge in the source), but this time with the *Check* modifier.

Patterns and Conditional Expressions

If the value to be assigned depends on some source element properties, *conditional assignments* (assignments included in if-then block) can be used. Of course, it is possible to code these source elements with different property values as different kinds of the source node. However, if only the attribute values in the target depend on these conditions it is not effective to introduce additional source nodes. Conditional assignment is used in 4th target node of Fig. 54 (p.145).

The same could be said about the application of constraints on mapping relations. It is possible to create different source node sets using filter conditions; however, if only something specific should be added to the target model while general mapping is the same it is not effective to add special nodes to the source tree. Adding an additional *constrained mapping relation* to a source node is a significantly more readable way.

Sometimes composition relationships alone are not sufficient to define the mapping application context. Therefore it is possible to use source *patterns*, mapping relations with application condition and conditional assignments in the target. Patterns are needed to increase the expressiveness of the mapping language – to add some of the power of pattern and rule based transformation languages. Typically patterns are used to add constraints to nodes, especially when a node should be connected to another node using some edge. However, patterns in this language are not as expressive as patterns in MOLA. The difference is that only the node types and edge types defined in the tree type may be used in a pattern, but not arbitrary domain metamodel classes and associations as in MOLA.

At least one of the nodes in a pattern should be connected with the source tree by using a parent-child relation. It is possible to give names to pattern elements, in the same way as to class elements in MOLA rules. Only one mapping from a pattern is supported. The node (or edge) used as the source of mapping is the main node in the pattern. If the mapping from a pattern is traversed in the opposite direction, then the tree element located by default is the source of mapping, however, navigation expression could be continued with the name of the pattern element. It is useful, if the attribute values of other pattern elements are required.

Integration with Custom Transformations

Although features have been introduced to raise the language expressiveness it is not possible to write an arbitrary transformation between the models in this mapping language. Therefore, it should be possible to extend the mappings defined in this language by explicit custom transformations. We have chosen MOLA as the language for custom transformations. We have introduced a special tree node type named "*custom*

MOLA procedure”. In this node it is possible to specify the MOLA procedure name to be applied when the given node is executed. The MOLA procedure can have parameters. Rules are defined how to represent these parameters in the mapping tree.

The first parameter for these procedures should be the parent of the *custom MOLA procedure* node. A type of this tree type element should correspond to the MOLA parameter. All other parameters are of the type in/out and are represented as child nodes. In this case the types again should correspond. It is possible to use the already found elements as child nodes, references to these elements are defined as a path to the tree nodes.

A study of typical application contexts of custom procedures is left for future research. This study might reveal the need to introduce new mapping modifiers to enhance the use of custom transformations.

This feature enables the possibility to apply the mapping language to transformation tasks when transformation is defined by combining simple mappings with explicit transformations for complicated fragments.

4.3.6 Mapping Language Semantics

The previous sections contained a description of the mapping language syntax. Below a description of the mapping language semantics is offered.

Multiple mapping diagrams are supported in the proposed mapping language. As already stated in Section 4.3.5 these diagrams are ordered. The mapping diagrams are executed according to the ordering.

Multiple mappings are used in the same mapping diagram. Mappings in a diagram are ordered as well. It is possible to explicitly define this ordering; in this case an explicit ordering is used. Mappings are ordered top–down according to the source end if the ordering is not defined explicitly. It should be noted that multiple mappings from the same source node are also ordered top–down.

The only exception is mappings from edges. In the ordering they are placed directly after the second node (the end node of the edge, located farther in the mapping ordering).

For target nodes without incoming mappings (nodes created with a parent), mapping is introduced. The source of the mapping is the same as the source of the parent node mapping. In the mapping ordering these mappings are inserted directly after the parent node mapping. If the multiple children of the parent node have no mappings, these newly introduced mappings are ordered top–down according to the target.

Using the principles described above it is possible to obtain a mapping ordering for any mapping diagram. Mappings in the diagram are executed according to the mapping ordering in the top–down manner.

Mapping Semantics

Although we use mappings to define a transformation from the source tree to the target tree, actually we want to transform models. These models are related to a tree type. For details see the mapping language definition facilities in Section 4.4. A transformation defined in terms of tree nodes could be translated in a transformation defined in terms of the source and the target models.

To execute a mapping we should find an instance set satisfying the mapping application conditions defined by the source tree. This condition is defined by the source tree fragment from the source node of the mapping, including all its parents, to the source tree root. Of course, conditions defined for these nodes should be included in this constraint. This could be treated as a pattern describing the application context of mapping.

When executing a transformation, the pattern defined in terms of the tree type should be transformed in the pattern defined in terms of model. It should be noted that it is possible to perform such transformation by using the tree type definition described in Section 4.4. A pattern defined in terms of model will be used to find the model instances to be transformed.

When processing the current mapping in instance level, the target instance created by the mapping should be attached to the appropriate parent instance in the target tree. It is necessary to find this parent instance corresponding to the parent tree node. The parent tree node should be related to the source tree by using some already processed mapping relation. Besides, the already processed mapping relation (from the target node parent) should go to the parent of the current mapping source node. If these conditions are not satisfied, it means the mapping diagram is semantically incorrect.

As the parent in the source tree (the source of the already processed mapping relation) is included in the pattern describing a possible application condition of the mapping, it is possible to create a pattern describing how to find an instance of the source parent node from the source node instance of the current mapping. It is also possible to define it in terms of models.

For mappings in our mapping language there is the semantics “Create, if does not exist” and for each performed mapping traceability information is created. It means that by using this traceability information from the source node instance of the already processed mapping it is possible to find a corresponding target node instance. This feature together with the previously described pattern could be used to find the parent instance for the target node instance of the current mapping.

Before execution of the current mapping it should be checked whether such mapping has not been executed before. Traceability links are saved in models, therefore, it is possible to define this check in terms of the source and the target models. Checking of the existence of such mapping is done by using the mapping name.

If nothing is found we should create an instance of the target model. This again should be done in terms of model. The target tree node type is transformed in terms of model element creation. The mapping source node again should be transformed in terms of the domain metamodel. Between the source node (defined in terms of model) and between the target node creation (defined in terms of model) the traceability creation should be defined in terms of model. Creation of the relation between the target tree node and its parent should be defined in terms of model as well.

The property values of target element are assigned according to the assignment description in the target tree. This description again is translated in terms of models.

In this way it is possible to translate the execution of mapping in terms of the source and the target models. This translation is described in detail in Section 7.2.

Mapping Modifiers

In addition to simple mappings it is possible to use mappings with mapping modifiers. In the latter case the execution semantics is modified a little.

If the *check* modifier is used, the mapping execution stops at the point of checking whether such a mapping exists. The test of the mapping existence is done for each node satisfying the mapping application conditions. If mapping does not exist for some node, then error is produced. Child elements of this element are excluded from the application context of the mapping following this mapping in the mapping ordering.

It should be noted that *copy* and the *copyAttributes* modifiers could be used only if the node types in both ends of the mapping are the same.

If the *copyAttributes* modifier is used, the mapping execution semantics is as described in the previous section. Only when transforming the target tree node to its creation in this element, the attribute value assignments are added.

If the *copy* modifier is used, the execution semantics is the same as for mapping with the *copyAttributes* modifier. The extension means that child cloning should be done as well which is performed by a call to the universal instance copier.

Edge Processing

Mappings outgoing from the edges should be processed as well. For edge mapping an application condition in the source tree is defined by the trees for both ends of the edge. It means that in the pattern describing the edge mapping application two paths to the root node are added. This pattern defined in terms of tree nodes again should be transformed in terms of the source model.

The ends of the edge in the target model should be linked with mappings to the nodes included in the pattern defining the application context of the edge mapping. This way, similarly to the location of the parent node, it is possible to locate the ends of the edge to be created in the target.

The rest is similar to the mapping processing for nodes.

Other Elements

Conditional mapping is treated as an additional constraint added to the mapping application context. The pattern adds additional constraints to the mapping application context as well. The pattern is translated in the pattern defined in terms of the domain metamodel. The rest is similar to the mapping execution semantics defined above.

Conditional assignment does not affect the mapping execution semantics. The only change is in translation of the attribute value assignments to the model terms.

4.3.7 Mapping and Transformation Comparison

In this section we will compare UML to UML transformation development in the mapping language MALA4MDSD and in a traditional transformation language. As already mentioned above a typical application of this language is transformations from PIM to PSM in the MDA lifecycle.

In the IST 6th framework project ReDSeeDS a model-driven path from the requirements to the code is investigated [3], as already described in CHAPTER 3. Two different transformation sets ("styles") from the requirements to the code have been developed. Each set contains a different structure of Platform Independent (PIM) and Platform Specific Models (PSM) and different transformations between them. These transformations have been developed in the model transformation language MOLA [76]. For a detailed description see CHAPTER 3 and [84].

We have rewritten the static structure processing of PIM to PSM transformations in the language MALA4MDSD. Table 6 contains statistics about transformations in MOLA and transformations in MALA4MDSD. For the simplest – the Basic style transformations - 19 MOLA procedures (diagrams) were needed while it was possible to write the same in MALA4MDSD with only 19 mapping links.

Table 6. Comparison of transformations from PIM to PSM, developed using the model transformation language MOLA and the mapping language MALA4MDSD

	Basic style	Keyword-based style
MOLA procedures	19	51
MOLA rules	84	137
MOLA class elements	265	418
Mapping diagrams	3	8
Mapping links	19	41
Mapping nodes	29 (source:11; target:18)	66 (source:27; target:39)

In Fig. 54 one mapping diagram from the Keyword-based style transformations is presented. In this diagram copying of Classes, Interfaces and Interface realizations from the PIM model to the appropriate place in the PSM model is presented. Classes and Interfaces in the PIM model can be located in the sub-package hierarchy under the packages "*businesslogic*" and "*applicationlogic*" (the 3rd node in the source tree represents the package hierarchy). The same sub-package hierarchy should be retained in the target model. In Fig. 54 edge processing and conditional assignment is used as well.

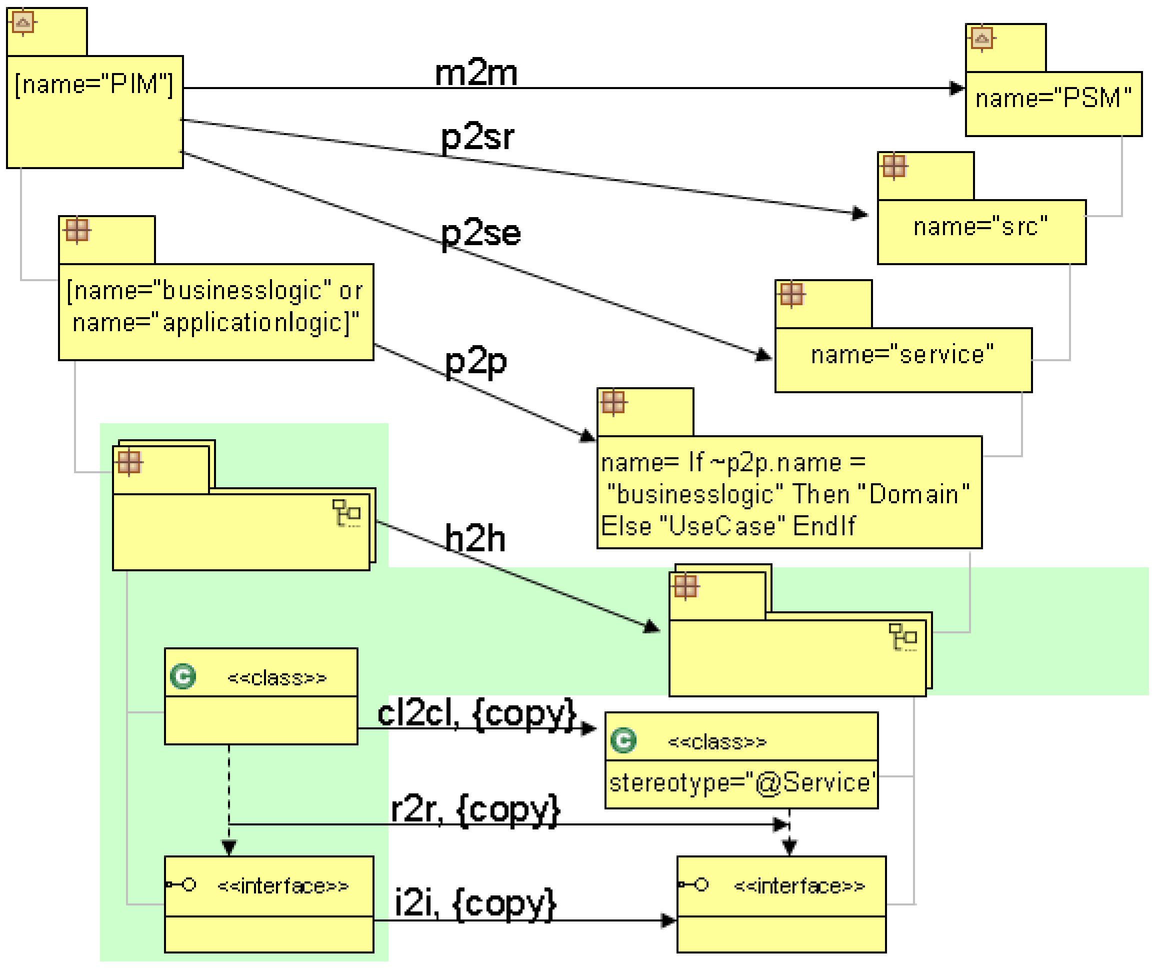
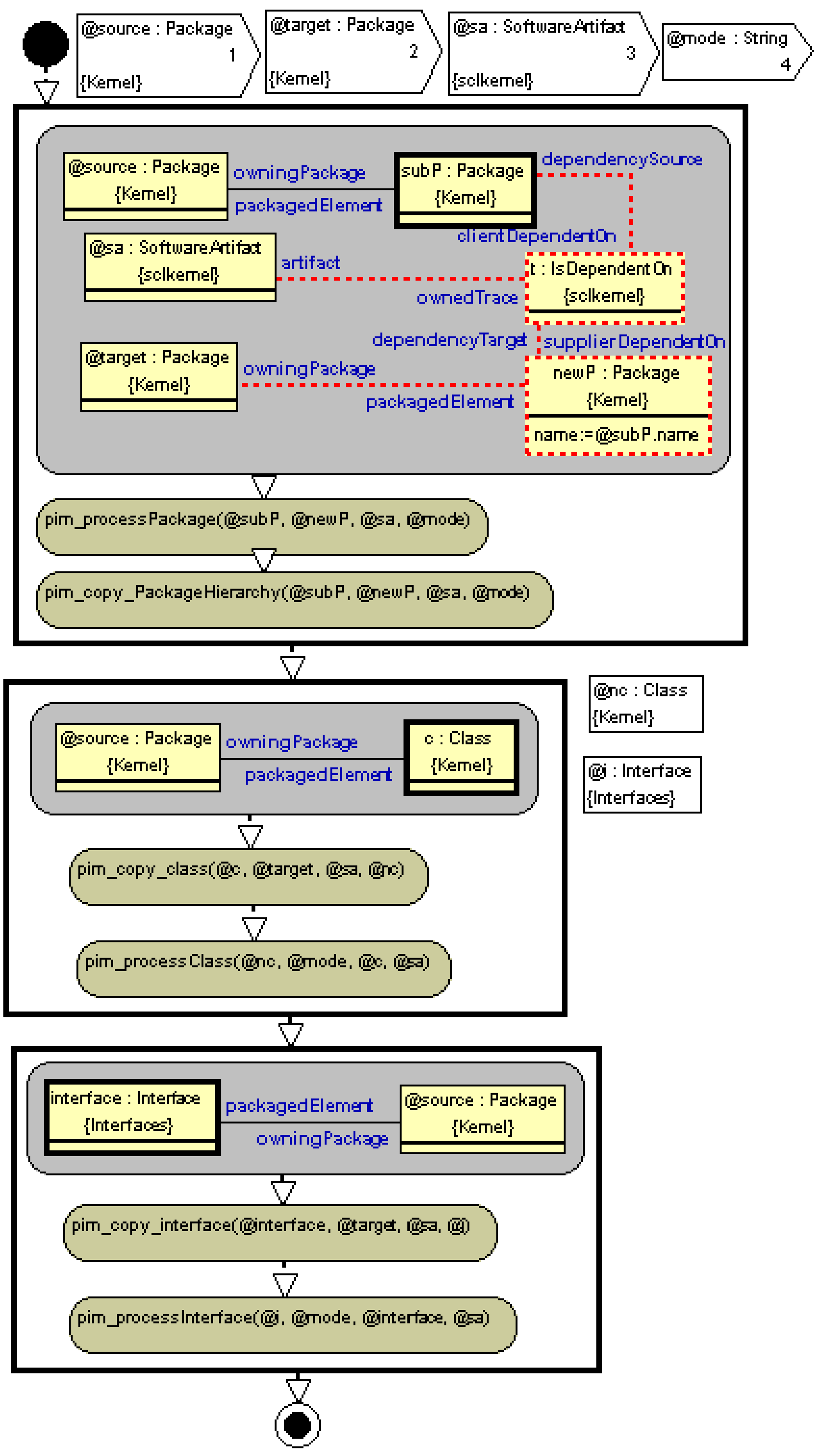


Fig. 54. Mapping example from the ReDSeeDS project. Transformation in MALA4MDSD is demonstrated. MOLA transformation for the highlighted part of the same task is presented in Fig. 55.

In Fig. 55 (p.146) a part of MOLA transformations implementing the same logic is presented. Actually, here only the package hierarchy is processed and the class and the interface copiers are invoked. It corresponds to the coloured part of MALA4MDSD diagram in Fig. 54. All the copy logic is defined directly in other MOLA procedures. This copy logic description is quite long as there has to be described that attributes, operations and operation parameters should be copied and how they should be copied in terms of UML metamodel. The mapping part above the package hierarchy symbols is described in another MOLA procedure. Interface realization processing is not presented in this MOLA transformation either.



@source : Package 1 {Kernel}
@target : Package 2 {Kernel}
@sa : SoftwareArtifact 3 {sclkernel}
@mode : String 4
@source : Package {Kernel}
owningPackage
packagedElement
subP : Package {Kernel}
dependencySource
clientDependentOn
@sa : SoftwareArtifact {sclkernel}
artifact
ownedTrace
t : IsDependentOn {sclkernel}
dependencyTarget
supplierDependentOn
@target : Package {Kernel}
owningPackage
packagedElement
newP : Package {Kernel}
name:=@subP.name
pim_processPackage(@subP, @newP, @sa, @mode)
pim_copy_PackageHierarchy(@subP, @newP, @sa, @mode)
@nc : Class {Kernel}
@i : Interface {Interfaces}
@source : Package {Kernel}
owningPackage
packagedElement
c : Class {Kernel}
pim_copy_class(@c, @target, @sa, @nc)
pim_processClass(@nc, @mode, @c, @sa)
interface : Interface {Interfaces}
packagedElement
owningPackage
@source : Package {Kernel}
pim_copy_interface(@interface, @target, @sa, @i)
pim_processInterface(@i, @mode, @interface, @sa)

Fig. 55. Transformation example from the ReDSeeDS project. The same transformation fragment in MALA4MDSD is coloured in Fig. 54.

A reader may get the impression that MOLA is not a suitable language for this task and other transformation languages would do better. However, it is not the case. Transformation languages usually deal with UML in its abstract syntax. Therefore, all processing of all classes and associations according to the UML metamodel should be precisely defined. In the mapping language UML logical elements (a sort of concrete syntax) are processed and a user should not care whether this logical element is represented with an instance of one class or with instances of two classes connected with an association (and so on) in the UML domain.

4.3.8 Related Work

All mapping languages mentioned in Section 4.1.2 are general purpose languages, applicable to any domain and they are based on an abstract syntax. Differing from the approaches described in Section 4.1.2, we propose to base the mapping language on a concrete syntax of the source and the target languages. For transformation languages a similar idea has already been applied, e.g., in *AToM*^{3} [96], and by Grønmo in [49]. A concrete syntax is used directly in model (graph) transformation rules that lead to a more familiar representation for modellers. However, this approach lacks the simplicity and power of representation of correspondences between model elements offered by the mapping languages.

There is also a similarity between our approach and a *Model Transformation by Example (MTBE)* [199] where transformation examples are specified as mappings in a concrete syntax. However, the MTBE approach requires a reasoner for transformation synthesis from examples while in our approach the defined mappings are complete transformation definitions.

Although models in the context of model-driven software development are graphs and not pure trees, we have made a brief overview on several areas where transformation languages are operating on the data represented by trees.

XML is the most popular and widespread technology. *XSLT* [194] is the transformation language used to transform data in the XML format. Although XSLT itself is an XML-based textual language, there are tools that use mappings to represent XSLT transformations, e.g., *Stylus Studio XSLT Mapper* [132] and *xsl:easy* [154]. The source and the target schemas are represented by fixed trees and all transformation logic is specified by using much more complex mapping features than it has been done in our approach.

Another field of data being trees is program rewriting. Though, the tools and languages, like *Stratego/XT* [23] or *TXL* [29], are intended for the analysis, manipulation and generation of programs, their features make them useful for transforming any structured documents.

4.4 Domain-Specific Mapping Language Definition

So far very few responsibilities of a mapping language developer have been described, namely, only to create definitions of the relevant tree type. In fact, this is only a small part of the job. The precise definition of the general mapping language execution (semantics) as far as provided in the previous sections was only from an instance tree to another instance tree. However, in real life there are only models (compliant to their metamodels) in various modelling languages and in various forms – exports from modelling tools, models in repositories, such as *Eclipse EMF* [166], a.o. So there must be facilities how to get from a model to a tree and vice versa. To make our mapping language family usable in practice a uniform solution has to be provided for these tasks.

4.4.1 MALA4MDSD Definition Issues

The previous section presented one specific mapping language for transforming the UML models. Now we want to discuss the basic principles according to which this language was defined and their possible application to similar model transformation cases.

The first issue is an appropriate selection of the model elements to be represented in the tree (source, target or both) – the nodes of the tree type. A natural hierarchical subset of the modelling language concepts has to be selected for the chosen domain. Containment has to be the most important relation in this subset since all its elements are represented in one tree. For example, the tree type defined in Section 4.3.4 described the static structure of the UML class model for typical MDSD. The chosen subset corresponds to the one represented in the model tree in most of the UML tools for a class model structure. Another selection criterion implies the elements to be represented by nodes or their subparts in the relevant diagram notation – a class diagram in the given case. The corresponding diagram notation is also the main source for the choice of elements to be represented as edges in the tree type – associations, generalisations, dependencies and realisations in the example. Lines are not shown as tree nodes, they are attached to the nodes when required (many modelling tools show also lines directly in the model tree). The notation used in our approach is a more convenient way to show how the end points of lines are defined during the mapping process (when selected in the source or created in the target).

Since the choice of the tree type elements is based on the existing diagram (or tree) notation of the model, it certainly represents the concrete syntax of the modelling notation. The concrete syntax is normally much more compact than the corresponding abstract syntax – the domain metamodel. The ratio is about 1 to 3 for the selected UML fragment. Certainly, this concrete syntax has to be unambiguously mapped to the domain metamodel (abstract syntax) since our approach to the mapping language implementation finally converts a mapping definition to transformation in MOLA working upon the domain. The traceability between the source and the target is also defined at the domain level. Such a mapping is obvious in our example, but it should be easy to define it in other cases as well.

Another feature of the language definition is the attribute list for each element in a tree type. Certainly, the attributes of the domain metamodel class mapped to the given tree element can be used in this role. However, non-containment associations navigable from the domain class (with multiplicity 1 or 0..1, playing the role of references) can also be defined as attributes – their type is the target class in the metamodel. Again, the inspiration for such attribute selection is the diagram notation – they are visualised within the main element. For example, in our mapping language, such attributes are operation type and class stereotype.

A specific mapping language is uniquely defined by its complete source and target tree type and the mapping of the tree elements to the domain metamodel. There are no domain-specific features in the mapping definition facilities and various expressions are used there. Only the mapping modifiers could be domain-specific – *copy* and *copyAttributes* are meaningful only if the source and the target trees are of the same type, otherwise some other domain-specific processing of complete sub-trees could be added.

4.4.2 Mapping Languages Definition Facilities

We propose a uniform solution for relating the models in a modelling language (such as UML) to the trees conforming to a tree type describing the selected part of the language in the form of trees extended by some edge types (e.g., the tree type *simpleUML* for MALA4MDSD). Certainly, we assume the *metamodel* of the language in MOF to be given. The solution – *domain-to-tree mapping* – is based on the tree type itself. It extends the tree type by the OCL expressions based on the metamodel and a few predefined keywords. Our mapping definition will directly show how a mapping defined in terms of tree nodes could be translated in a transformation defined in terms of models.

We will specify which metamodel class is at the basis for each *node type* (by using the *Class* keyword). In addition, a selection expression in OCL can also be provided if not all class instances qualify. Further, for each *attribute* we want to include in the node type an OCL expression describing how a relevant value from a model should be extracted. If that expression is to return a reference to another node type in our tree type, the *Node* function is used (certainly, its argument must have a type equal to a class mapped to a node type). The *Node* function is used to define the finding of association end nodes in the tree type definition, demonstrated in Fig. 56.

For each *containment* (parent-child) *relation* an OCL navigation expression specifying how child instances can be reached from the parent in a model must be provided (after the keyword *Path*). A node with a *transitive containment* (such as Package in UML) must provide a special *Path* expression (marked with an icon) within it, indicating how the next contained instance of the same type may be reached.

Similarly, the metamodel class the *edge types* are based on must be specified. Attributes are specified the same way as for nodes. A new element is the path in a model by which the relevant end node instance can be found.

It is possible to name branches of the tree type definition and to use this name as a reference to the tree type branch supported in this position, similarly as it was done for the MALA4MDSD tree type described in Section 4.3.4. Actually, the tree type description similar to the one used in Section 4.3.4 is obtained from the tree type definition throwing out the OCL expression.

A mapping language developer has to define one or two *domain-to-tree* mappings to specify the language.

Fig. 56 illustrates how the tree type *simpleUML* can be defined on the basis of the standard UML 2 metamodel. A slightly simplified version of the metamodel is assumed, e.g., such as used for the *UML 2 tool in Eclipse* [178] – just to avoid unnecessary packages, etc. All OCL expressions are assumed to be based on this metamodel. Only the top three node types: *Model*, *Package* and *Class* are visible in the fragment, but the continuation is quite similar. For all three node types the *name* attribute is defined in a natural way (the OCL *self* points to the node base class). The containment relation in all cases is defined by the same OCL navigation expression *self.packagableElement* – the UML metamodel is built this way. Only the *Association* edge is visible in the fragment. The role and stereotype attributes are defined for it (their definitions rely on the fact that only binary associations (with two ends) are used in our UML subset). Since both ends of an association are attached to classes, two similar end specifications are given.

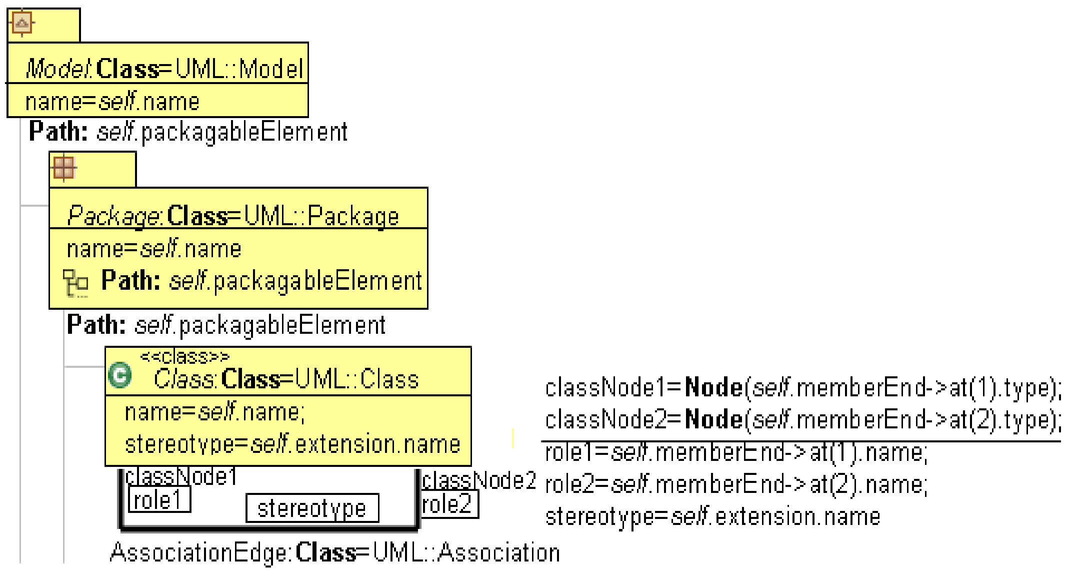

Model:Class=UML::Model
name=self.name
Path: self.packagableElement
Package:Class=UML::Package
name=self.name
Path: self.packagableElement
Path: self.packagableElement
<<class>>
Class:Class=UML::Class
name=self.name;
stereotype=self.extension.name
classNode1
role1
stereotype
classNode2
role2
classNode1=Node(self.memberEnd->at(1).type);
classNode2=Node(self.memberEnd->at(2).type);
role1=self.memberEnd->at(1).name;
role2=self.memberEnd->at(2).name;
stereotype=self.extension.name
AssociationEdge:Class=UML::Association

Fig. 56. Mapping language definition; fragment of the MALA4MDSD definition

The completion of Fig. 56 for all node and edge types is sufficient for the definition of MALA4MDSD. It should be completely clear now how it is possible to translate the transformation definition in terms of metamodel elements. The given mapping clarifies also how the node and edge typed parameters can be converted to metamodel elements (and vice versa) when a transformation language procedure is invoked from a mapping.

Another element to be defined is the “implementation” at the model level of the special *trace* edge between the trees. Since keeping the transformation traceability is of value for model management, typically a special class with associations should be added to metamodels (as it was done in the ReDSeeDS project).

This mapping definition is also sufficient for creating an implementation of a mapping language. The compiler and the editor could be generated from the tree type definition in a generic way (see Section 4.6).

To conclude some suggestions are offered for defining a specific mapping language. When an appropriate domain and a modelling language (together with the metamodel) for it have been selected, the tree type definition should include all relevant language concepts representable in a hierarchic way. Containment relations are typically based on compositions in the metamodel, but it is not mandatory (see the example in Fig. 59 p.155). Only those edge types which are relevant to the tasks to be solved in the domain should be included. The same holds true for attributes of the types.

4.4.3 Metamodel of Mapping Language Family

In this section metamodels of the mapping language family are considered. There is a core metamodel common for all mapping language family. This metamodel is presented in Fig. 58. A metamodel for the definition of the mapping language is given in Fig. 57.

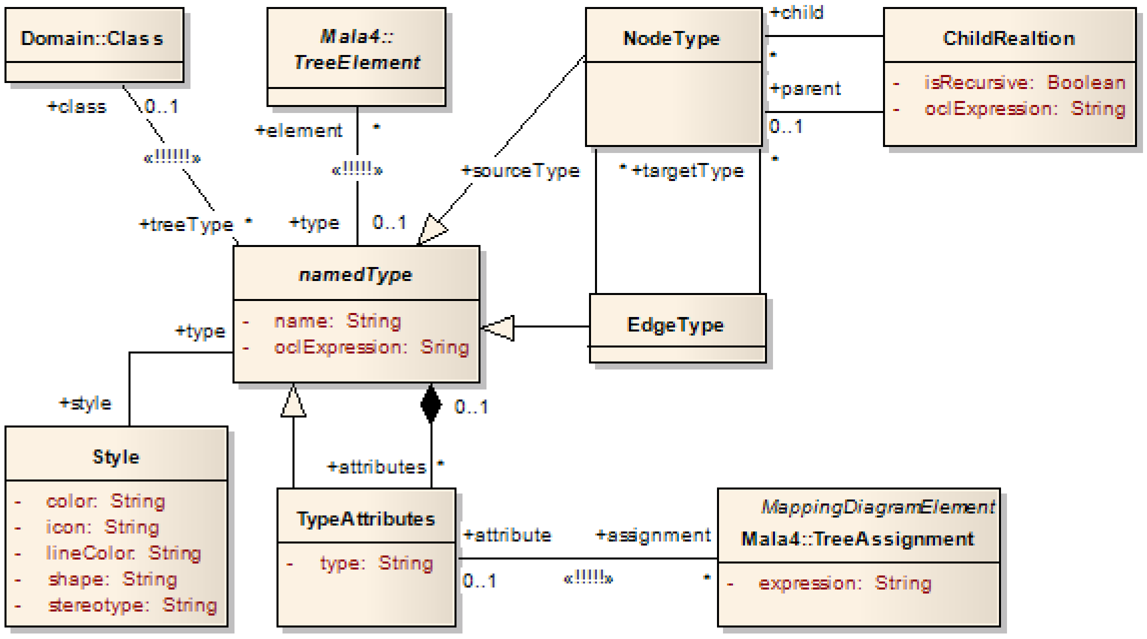


Fig. 57. Type definition for the mapping language family

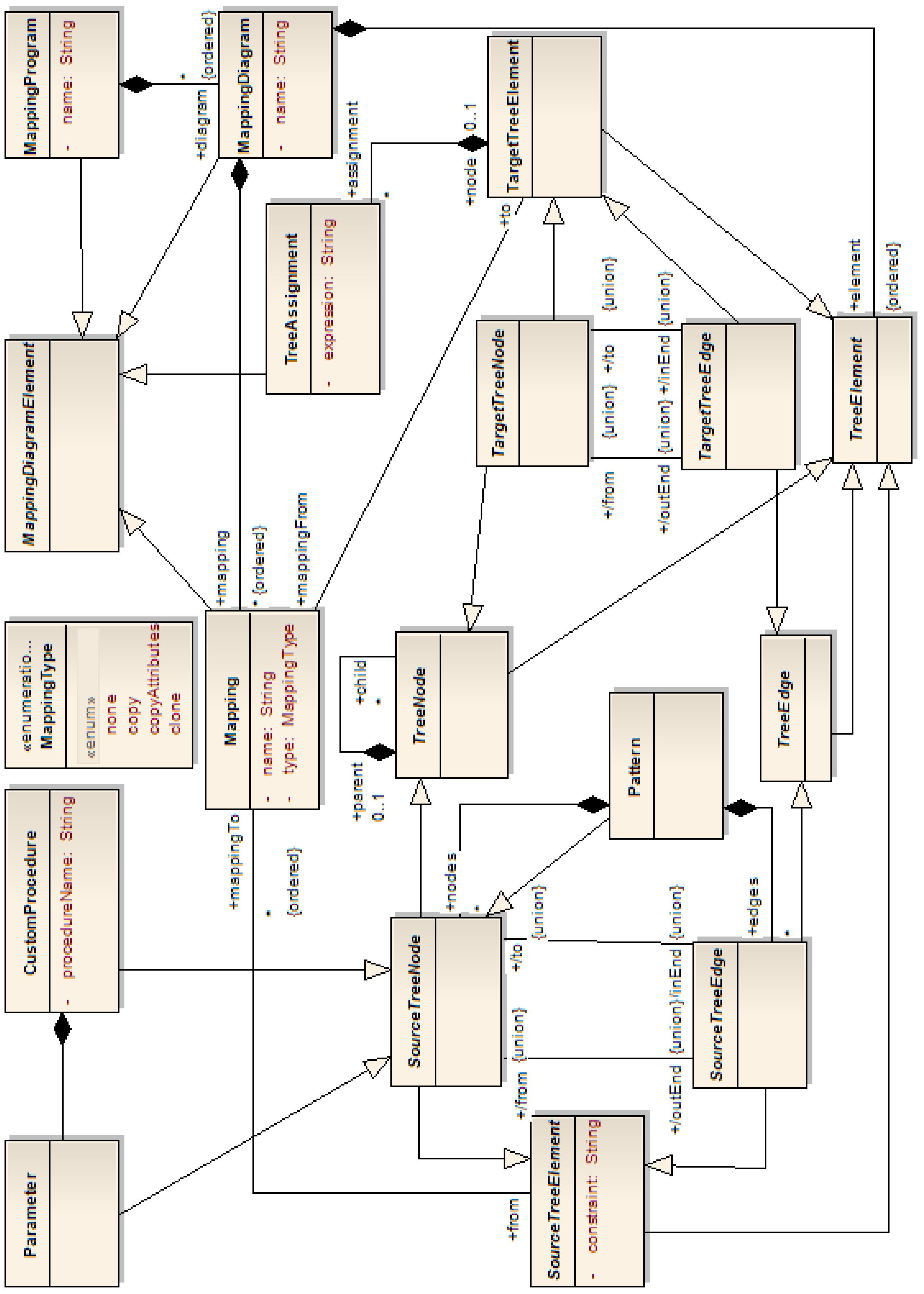


Fig. 58. Core metamodel of the mapping language family

4.5 Other Applications of the Proposed Approach

A wider application of the proposed mappings to *UML-to-UML* transformations is possible. For example, the UML subset for MALA4MDSD can be extended to include several behaviour aspects important for the MDSD tasks. The creation of interactions (sequence diagrams) in the basic cases can be described just by adding interaction and lifeline nodes and the message edge to the tree type definition (the message ordering can be emulated by the target tree element ordering). The gain with respect to explicit transformation specification of the same task is huge since the UML metamodel here is very “verbose”.

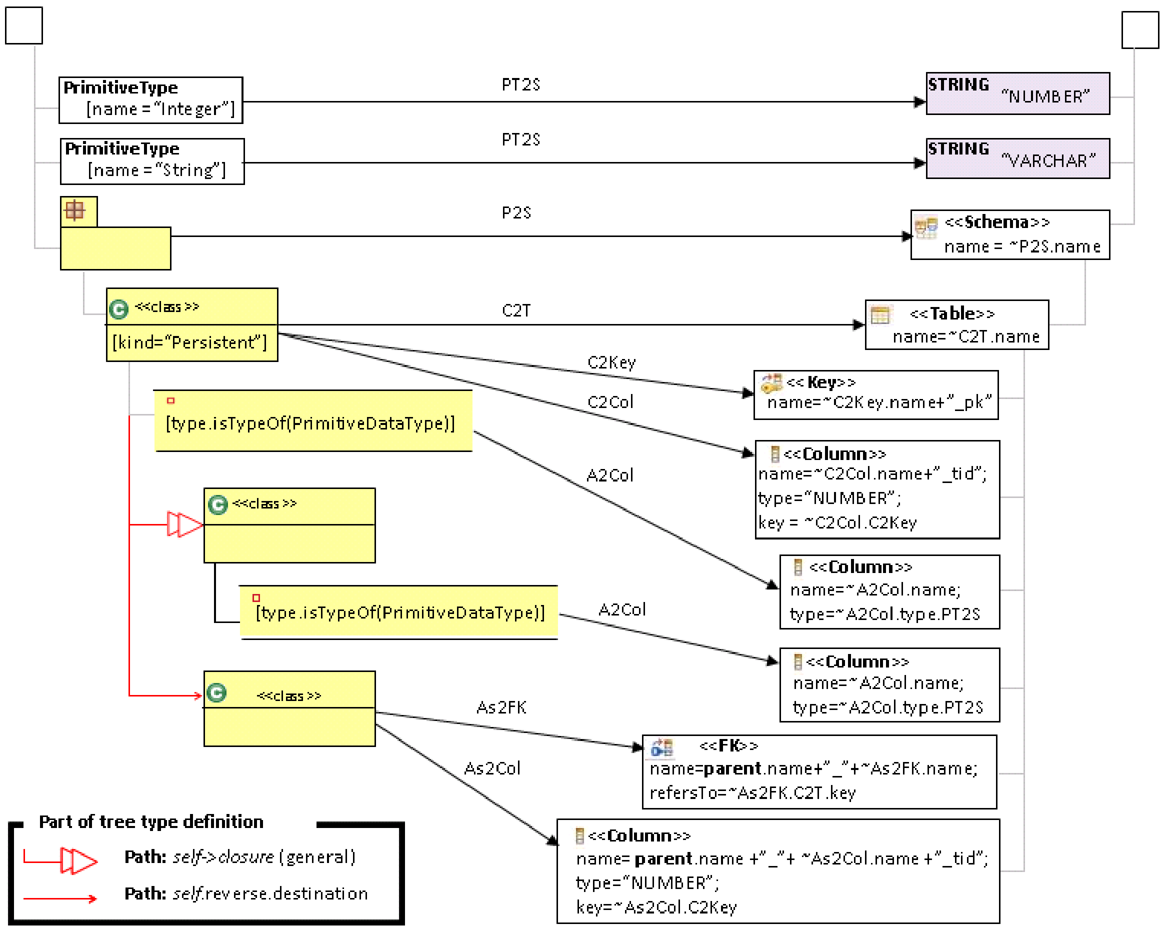
4.5.1 UML to RDB

The approach is appropriate for many other cases where UML is not involved at all or only one of the sides (source or target) is related to UML. A brief description of an example of this kind follows. It is a classical model transformation task solved almost by every model transformation language – *Class Model* to *Relational Database (RDB)*. The precise task description can be found in the appendix of the MOF-QVT standard [128], therefore we do not repeat it here in detail.

The task is to transform the persistent classes of a simplified UML model to tables of a simplified *RDB* model. A persistent class maps to a table containing a primary key and an identifying column. Primitive-typed attributes, including the inherited ones, map to columns of the table. An association between two persistent classes maps to a foreign key relationship between the corresponding tables. The only simplification of the original task is removing the recursive processing of attributes having complex data types. The solution of the task by using our approach is given in Fig. 59.

Containment relations in the source and the target trees are based mainly on the composition hierarchy in the source and the target metamodels. For example, *Table* node in the target tree may be owned by *Schema* node, but *Key* and *Column* may be owned by *Table* node. This is a natural representation and similar trees can be found in almost every database management tool.

However, we want to emphasize the flexibility of our approach – the containment relations represented by the highlighted lines in Fig. 59 are not based on composition. The first one shown as a double filled arrow represents the transitive closure of all superclasses of the given class. It can be defined by means of OCL due to the *closure* operation introduced in OCL 2.3 [126] (see tree type definition fragment in Fig. 59). The second non-composition containment relation represents the association in the simplified UML. In fact, there are many cases when the model can be represented completely as a pure tree using different containment relations depending on the needs of concrete developers. As one can see in Fig. 59 the shapes of containment relations may be adjusted according to the concrete syntax of the used modelling languages.



PrimitiveType [name = "Integer"]
PT2S
STRING "NUMBER"
PrimitiveType [name = "String"]
PT2S
STRING "VARCHAR"
P2S
<<Schema>> name = ~P2S.name
<<class>> [kind="Persistent"]
C2T
<<Table>> name=~C2T.name
C2Key
<< Key>> name=~C2Key.name+"_pk"
C2Col
[type.isTypeOf(PrimitiveDataType)]
<<Column>> name=~C2Col.name+"_tid"; type="NUMBER"; key = ~C2Col.C2Key
A2Col
<<class>>
<<Column>> name=~A2Col.name; type=~A2Col.type.PT2S
[type.isTypeOf(PrimitiveDataType)]
A2Col
<<Column>> name=~A2Col.name; type=~A2Col.type.PT2S
<<class>>
As2FK
<<FK>> name=parent.name+"_"+~As2FK.name; refersTo=~As2FK.C2T.key
As2Col
<<Column>> name= parent.name +"_"+ ~As2Col.name +"_tid"; type="NUMBER"; key=~As2Col.C2Key
Part of tree type definition
Path: self->closure (general)
Path: self.reverse.destination

Fig. 59. UML to RDB example

4.5.2 UML to XMI

There are several other transformation examples that could be very adequately specified by using the proposed mapping language approach. One such example is transformations from UML to XML. In this case the source tree could be similar to the one described in Section 4.3.4.

The XML tree could be used as a target tree. Since the XML document already has a tree structure, the target tree can be built straightforward. The root node in the XML tree should be *XML document* which contains *XML nodes* that in turn may contain other *XML nodes* and *XML attributes*. This mapping language, for example, could be used for writing a transformation from UML to WSDL. Of course, such transformation is already implemented in many UML tools and has been described in [102]. However, in our approach the mapping between the source and the target is visible. If you have a concrete WSDL file generated from some source model it is easier to understand how elements in this WSDL file have been created. You can select one XML node in the WSDL file, it is easy to find the corresponding node in the target tree as the structure of the target tree and the resulting XML file are similar. Using mapping relations it is easy to understand which UML model elements influenced the creation of such node. Consequently, this mapping definition could be useful as documentation.

Of course, UML to WSDL is not the only case when XML files from UML models are generated. Almost all UML tools have XML export. Usually XMI export is used, however, sometimes tools use their own custom formats. The export semantics could be described by mapping from UML to XML. UML models could be also used to describe the data interchanged by applications. In this case it is possible to generate XSD schemas (actually XML) describing the interchanged XML files. The same could be said about *Hibernate configuration files*, all kind of *XML data stores*, a.o.

4.5.3 Other Examples

Other examples where this approach should work could be migrating data from RDB to the existing ontologies with a similar structure (similar to the task discussed in [53]) and even for more complicated relational data transformation.

The transformation algorithm from RSL static structure to PIM static structure in Fig. 38 (p.83) has already been described by applying informal mappings which was demonstrated by means of an example. However, this example demonstrates that the source and the target models of transformation could be naturally described using trees. By replacing concrete instances from the example with tree type elements a mapping language could be obtained. The mapping language for the static structure transformation in RSL to UML could be easily created. It would be an adequate means to describe the transformations defined in the ReDSeeDS project.

However, in no way the proposed domain-specific approach is intended to replace model transformation languages in general. The pattern and rule based paradigm supported by most of the transformation languages is much better for transformation tasks which involve a complicated graph-based source model analysis. For example, tasks involving a graph structure analysis, such as finding well-structured components during the compilation of BPMN to BPEL [36], are inappropriate for the proposed mapping language.

It is likely that the mapping language would not be appropriate for defining transformations creating the behaviour part of PIM. In these transformations the pattern based analysis of the scenario sentences is widely used. Transformation languages like MOLA are more appropriate for this task.

Other limitations are related to the DSL approach in general – a certain amount of similar transformation tasks in a domain should be required to be implemented in order to outweigh the costs for the language support development.

4.6 Implementation

The main difficulty of successful adoption of a domain-specific language is the rather complex and expensive development of the language implementation. MALA4MDSD has not yet been implemented fully, however, the implementation principles are clear and the feasibility has been tested. The planned implementation scenario is the main topic of this section. We propose a universal implementation of the described mapping language family instead of implementation just for MALA4MDSD.

From the language user perspective a *graphical development environment* for transformations in this language and its *compiler/interpreter* must be created.

From the language developer perspective a tool support for the tree type definition is required. It should support the definition of a tree type on the basis of the corresponding metamodel. In the tree type definer definition facilities for the following elements are required: tree node types, tree node styles, permitted tree node containment, tree node type attributes, edge types, edge styles, edge context and finally relations between the tree type elements and the given metamodel. This involves creation of relatively simple graphical elements and property dialogs. To implement such editor, a graphical tool building framework could be used, e.g., *GMF* [172], *Microsoft DSL Tools* [28], *GRAF* [12] or *METAclipse* [86].

On the basis of the defined tree type (or a pair of them) a mapping (transformation) development tool for the defined mapping language should be created. Such a tool would embrace universal features and domain-specific ones. The universal features would include a generic support for creating a pair of trees, mappings between them and simple patterns. The domain-specific features are the specific tree node styles, edge styles, possible attributes and restrictions describing the permitted node/edge type containment. This tool could be created by using a model based DSL tool development framework. Appropriate candidates are transformation based tools *GRAF* [12] or *METAclipse* [86]. There the universal behaviour could be defined by using the tool definition facilities. Transformations describing the language specific behaviour of the tool could be generated by using higher–order transformations. In this case special languages for transformation synthesis would be useful, e.g., *Template MOLA* [69] or the extension of ATL described in [182]. Since the behaviour of METAclipse framework is defined by using the model transformations in MOLA [76] and Template MOLA is adapted to synthesise model transformations in MOLA, METAclipse + Template MOLA are selected for implementation of the mapping language editors.

Another issue is the mapping language compiler/interpreter. In this case a universal mapping interpreter/compiler could be built. The input data for the mapping interpreter/compiler will be the mapping language specification (domain-to-tree mapping based on the given metamodel) and a concrete mapping model in this language. One of the possible implementation scenarios is a compiler to model transformation language using higher-order transformations. Template MOLA could be used for this task again. However, an interpreter solution also looks feasible.

The compiler and the editor development of the mapping language family by using Template MOLA is described in detail in Section 7.2.

To conclude, appropriate means for the implementation of such a mapping language family does exist, only its implementation requires a certain technical effort.

4.7 Conclusions

In this chapter the use of domain-specific mapping languages is discussed. It is proposed to define model transformations by using simple mapping relations and tree syntax of the source and the target models. As a result it is possible to define typical model transformations in terms familiar to modellers and therefore these domain-specific mapping languages could be applied by a much wider class of users.

The proposed general principles have been applied to a family of the mapping languages where a language for a specific domain is defined by specifying the tree syntax for the source and the target. One specific mapping language – MALA4MDSD for transformations from PIM to PSM (a UML subset to a UML subset) – is discussed in greater detail. A concrete syntax similar to the model trees in UML tools is used for the source and the target models. The transformation development in this language is compared to the transformation development in a traditional model transformation language. A significant gain both in transformation size and understandability has been noticed since there is no need to deal with the technical details of the UML abstract syntax.

We propose a generic approach to the creation of domain-specific mapping languages. To define a mapping language, the tree types of the source and the target trees and their relations to models should be defined. This should be done by an expert in metamodelling and OCL. However, this should be done only once for a mapping language. Of course, the creation of a mapping language pays off only if multiple transformations in the same domain should be defined.

In no way the proposed domain-specific approach is intended to replace model transformation languages in general. For transformation tasks which involve a complicated source model analysis the pattern and rule based paradigm supported by most of transformation languages is much better. For example, tasks involving graph structure analysis, such as finding well-structured components during compilation of BPMN to BPEL [36], are inappropriate for the proposed mapping language.

CHAPTER 5

Transformations for DSML Tool Development

DSML tool development is another application area of model transformations. Transformation development for DSL tools is discussed in this section. The use of transformations and mappings in DSML tool development will be considered.

5.1 State of the Art in DSML Tool Development

The existing approaches for DSL tool development are briefly described further on.

5.1.1 Terminology Explanation

To start with, some terminology clarification is required as today different DSML development frameworks use completely inconsistent terminologies, even the terms model and metamodel are used differently depending on the context. For example, the mapping-based GMF [172] speaks only of two layers: model and metamodel, everything a tool builder creates is termed a model. We propose to combine both the transformations and the static mapping context. To avoid misunderstanding, a consistent terminology and its relations to be used in this chapter are defined in Fig. 60.

As we can see the domain metamodel is defined using MOF [120] as a meta-metamodel. A domain model is created according to the domain metamodel. It should be noted that alternative domain meta-metamodels used in some approaches in fact play the same role as MOF (and are similar to it).

The situation is not so simple with the presentation part. In every framework there is a fixed presentation type definition environment. Possibilities supported in this environment can be described with a presentation type metamodel. Presentation types for a concrete domain-specific language constitute a presentation type model defined according to the presentation type metamodel. Presentation types describe the relevant graphical element types. When data are created in this concrete DSML tool, instances of presentation model are created, but data in this model are not an instance in the presentation type model. It is an instance of the presentation metamodel describing supported graphical elements in the tool in general, e.g., line, box, label, etc. For example, in the presentation type model we can describe that we want to represent this type as a grey rounded rectangle with green lines and containing one label. In this case instances of the rounded rectangle, label and colours will be created in the presentation model with the appropriate properties set according to the presentation metamodel). After the instances have been created a user can change the colour of the rounded rectangle (if this feature is supported by the tool). In this case the presentation model is modified, but it does not affect the presentation type model. The presentation type describes only the default look of this node. Due to this reason the presentation model and the presentation type model are two separate models.

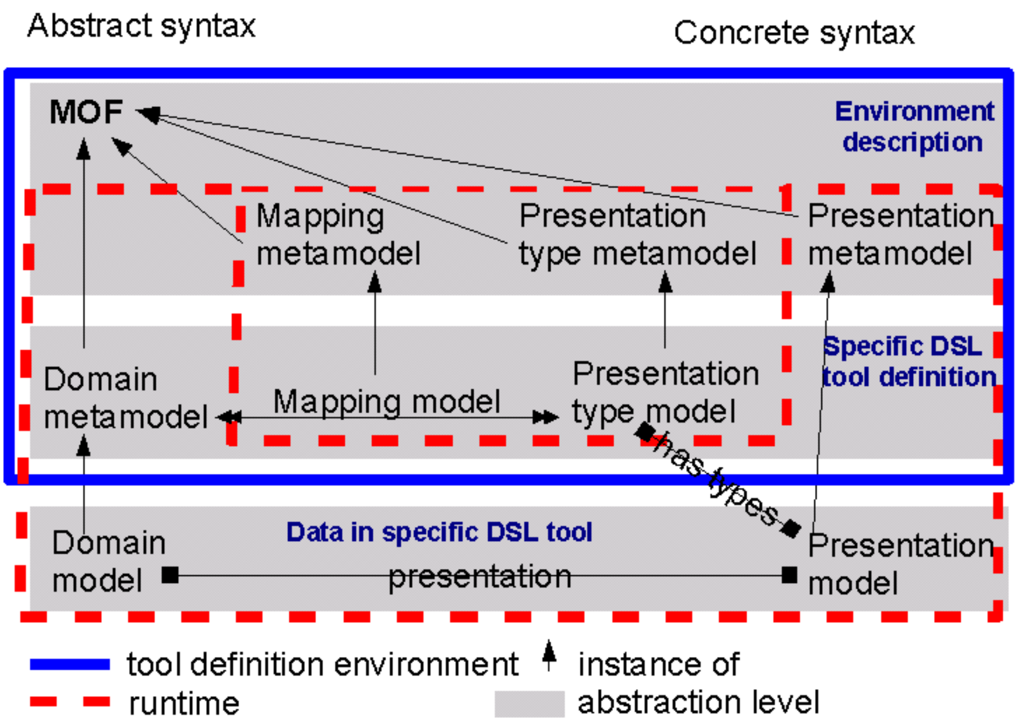


Fig. 60. Terminology definition

It is important to define a mapping model and it should be done according to the mapping metamodel. The mapping model describes the relationship between the domain metamodel and the presentation types. Mappings are not used directly at the data level.

When defining a new DSML tool in a tool definition framework, a user has to define a domain metamodel, a presentation type model and a mapping model. It should be noted that the presentation metamodel is needed directly only if mappings are defined by using model transformations. Models required at runtime for the tool created from the definition depend on whether the tool definition framework is an interpreter or a generator. If the framework is an interpreter the mapping and the presentation type models are needed to interpret them in runtime. If the framework is a generator, these models are not needed in runtime because the tool code is generated according to the data in these models.

Most of the known DSML tool definition frameworks can be correctly categorized in the framework of this terminology schema.

5.1.2 Mapping-Based Approach

A *mapping*-based approach prescribes which presentation type model element must be used to visualize each domain metamodel element. Thus, functionality of the graphical tool is basically defined by this mapping which itself can be defined as a mapping model according to the mapping metamodel. The mapping typically may be complemented by use of constraints, but only at a few selected points.

Most of the frameworks (*GMF* [172], *Microsoft DSL tools* [110], etc.) use the *generation step*, by means of which language classes are generated in the corresponding OOPL (Java, C#, etc.) from the involved models. The generated code ensures the relevant synchronization between the domain and the presentation models in runtime. If the generated functionality is insufficient, the language code can be extended manually. Actually, mapping may be used without the generation step as well – examples of it are *MetaEdit+* [109] and *Generic Modelling Tool* [26], which are model interpreters.

It must be noted that the mapping approach is easy to use. If the generated code is sufficient (or should be accompanied by a small amount of manual code), the tool definition is mainly declarative and very fast. However, when the presentation type model is dissimilar to the domain metamodel, a lot of code in OOPL must be added. To avoid this, it is a common practice for simple DSMLs to create custom domain metamodels nearly isomorphic to the corresponding presentation type metamodels (one class to one node type, etc.). However, there can be situations when it is not possible to select the domain metamodel freely, for example, if it is used for compiling, integration with other tools, etc.

Mapping definition capabilities of a framework depend on mapping design patterns supported. The most expressive static mapping language is implemented in GMF

[172]. But even this is not expressive enough. For example, every domain class mapped to a diagram node must be contained in a domain class mapped to the diagram itself (canvas in GMF). Therefore, it is impossible to implement by pure mappings standard a UML class diagram where a class is contained in a package (in the UML domain) and is visualised in several diagrams independently of its package containment.

There is also the *EuGENia* [170] framework based on GMF where the tool is defined by using the annotated *Ecore* model. The GMF models (*gmfgraph*, *gmftool*, *gmfmap*) are generated from the annotated *Ecore* model. *EuGENia* supports only a subset of GMF; however, it is possible to support full GMF modifying generated GMF models by using model transformations in *EOL* [91]. Although model transformations are used this is still a mapping-based approach as transformations are only used to compile an alternative tool language to the mapping-based approach in GMF. Transformations do not support full tool behaviour. However, if the GMF mapping definition facilities are not sufficient then extensions should be implemented in Java.

Let us consider some DSML language examples where the mapping approach is clearly insufficient. Evidently, one such group is model transformation languages. A typical example is *MOLA* [76, 59], which is a graphical language with a lot of semantic dependencies between language elements. It is important to use the native MOLA metamodel as a domain metamodel for the MOLA tool, since only this way complicated syntax checks can be performed during editing and context-sensitive lists of the valid references proposed. If the goal of the tool is to create syntactically correct models as far as it is possible, clearly it is impossible to implement this tool by using only static mappings. The same can be said about tools for other transformation languages, e.g., *MOF QVT* [122], where the native domain metamodel is even farther from the presentation. Another such group could be complicated workflow languages.

5.1.3 Model Transformation Based Approach

A complete alternative to the mapping-based approach is the *model transformation* based approach. The correspondence between the domain and the presentation is defined by *transformations* in a model transformation language, e.g., MOLA [76, 59]. These transformations define what modifications must be done in one of the models, if the other one changes (due to the user actions or other internal activities). Therefore, the correspondence between the domain metamodel and the presentation type model may be arbitrarily complicated here. In fact, transformations control the complete tool behaviour.

At first glance this approach seems more complicated for use though experience reveals that programming model element mappings in an adequate model transformation language is much easier than in a standard OOPL. The usability of the approach is also ensured by the fact that a significant part of the transformations are domain-independent and are built only once as part of the framework itself. Clearly, the transformation driven approach is more time consuming in simple cases.

The first pure transformation based project is the *Tiger project* [37]. However, a specific domain modelling notation is used there, making the domain metamodel of a language still to be close to the presentation metamodel. Standard editing actions (create, delete, etc.) are specified by graph transformations which act on the domain model, and the presentation model is updated accordingly. The main goal of the Tiger approach is to provide the building of syntactically correct diagrams only.

The most advanced transformation based framework is *METAclipse* [86] that uses the MOLA transformation language and a powerful presentation engine in Eclipse which is an extension of *GEF* [171], *GMF runtime* [172] and some other plug-ins. It is based on a presentation metamodel specially adapted for defining transformations. The current version of the *MOLA editor* [86] is built on this framework (using a bootstrapping approach). This editor provides an advanced support for ensuring the syntactical correctness of MOLA programs and a high usability. The developed editor confirms the suitability of the framework for implementing complicated DSLs.

5.1.4 Combined Approach

Usually, for some parts of the tool the correspondence from the domain to the presentation is simple (fit for mappings) while for some it is complicated (fit for transformations). The best solution would be to combine both approaches. In this case for simple one-to-one relations between the domain and the presentation the mapping-based approach could be used, but model transformations could be written for complicated parts. For example, for the abovementioned *MOLA Editor* [86] the transformation size could be reduced approximately by 50% if mappings were applicable. Simple visualisation could be defined by mappings, but transformations would still be needed for complicated consistency maintenance.

Currently there are only known a few attempts to combine both approaches in a limited way. The frameworks, using this combination to a certain extent, are the *Tiger GMF Transformation project* [162] and the *ViatraDSM framework* [133].

The *Tiger GMF Transformation project* [162] (related to the original Tiger project) proposes to extend GMF by complex editing commands. The mapping between the domain and the presentation models is defined by standard GMF facilities. But new complex model editing commands can be defined by transformations acting only on the domain model. However, this approach does not permit to define more complicated (transformation based) mappings between the domain and the presentation, which is the main goal of the approach proposed in Section 5.3.

The *ViatraDSM framework* [133] is based on the *Viatra2* [180] transformation language [31]. In this framework a mapping from the domain to the GEF-level presentation concepts has to be defined. This static mapping is interpreted by the *ViatraDSM* engine. The transformation based mapping (defined by *Viatra2* [180] rules) can be combined with the static mapping approach. The goal of *ViatraDSM* seems to be the closest to our proposal. However, a lot of principal issues are not solved there. First of all, the static mapping mechanisms support only very limited mapping possibilities – only the basic mapping patterns are supported. Mapping and transformation integration possibilities are very limited as well. Each object can be mapped using either transformations or mappings. The mapping definition for *ViatraDSM framework* has no adequate notation. Solutions to all these issues are the themes of the DSML tool development framework proposal described in Section 5.3.

We propose to use a more detailed mapping and transformation integration granularity, for example, to use transformations as pre-processors or postprocessors for mappings. A more expressive mapping language and a mapping definition notation are proposed as well.

There is one more framework *GRAF* [12] which combines both approaches to a certain extent, but in a different setting. This framework is based on an advanced tool definition (presentation type) metamodel and the corresponding *configuration tool* [157], by means of which the desired diagram structure and property dialogs are defined. The framework contains a large set of predefined transformations that implement all standard user actions related to the defined diagram type. All these predefined actions can be extended or replaced by custom transformations. The main application area for this framework is various conceptual modelling languages; consequently, there is no built-in support for domain models. If required, synchronisation with the corresponding domain can be supported by custom transformations. Complex validations and other additional options can be implemented in the model transformation language as well. It should be underlined that *GRAF* is based on the *Transformation-Driven Architecture (TDA)* [14] which is a system and tool building approach where multiple presentations and services can be linked by model transformations. Tools built by GRAF are based on TDA as well.

5.2 METAclipse

METAclipse [86] is a graphical DSL tool development framework built in the University of Latvia, Institute of Mathematics and Computer Science. The *METAclipse* framework was proposed in the PhD thesis of Oskars Vilitis [188]. This framework is suitable for DSL tool developments were verification of syntaxes and semantics is required.

The *METAclipse* framework is based on *Eclipse* [167]; it uses many Eclipse plug-ins and *GMF runtime* is one of them.

The METAclipse framework provides functionality common to all DSL tools. A concrete DSL tool is built by using model transformations that have to processes only the semantic events of the DSL tool. Other events are processed by the tool building framework. Typically model transformations for the METAclipse framework are defined in the model transformation language MOLA [76].

5.2.1 MOLA Tool

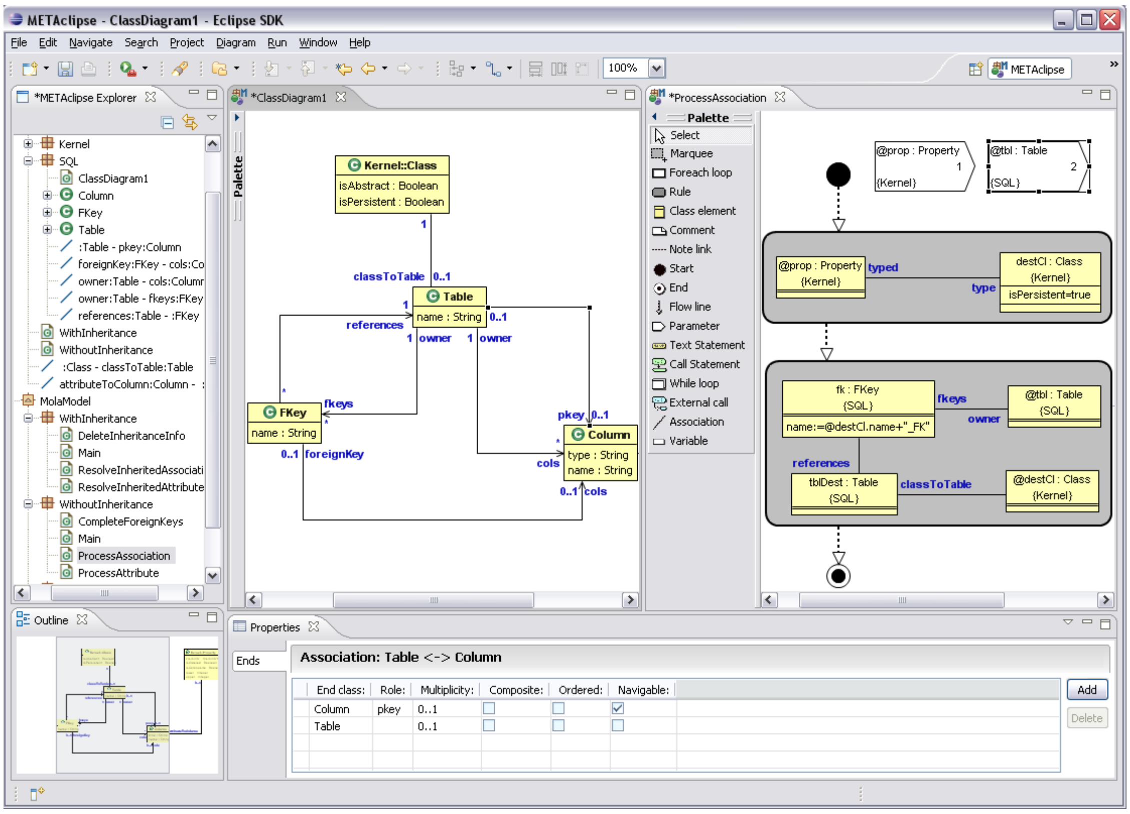
The author of the present PhD thesis has developed the first version of *MOLA 2 tool* [85] in *the METAclipse framework* [86]. The *MOLA 2 tool* was the main test-bed for the METAclipse framework, since MOLA is clearly in the DSML category for which transformation based approach is more appropriate. In MOLA there are complicated dependencies between the abstract and concrete syntaxes, therefore, it would be complicated to build the MOLA editor in a tool building framework based on mappings.

The MOLA environment has been developed in a bootstrapping manner [59] with the previous prototype editor built by using the *Generic Modelling Tool* [26] framework.

The new editor implements a lot of validity checks and a smart prompting during the diagram building.

The *MOLA 2 tool* consists of two parts – the metamodel editor and the model transformation editor. The UML class editor actually is the simplest part of the MOLA environment. The MOLA procedure editor requires much more sophisticated domain-specific logic during element building or updates. Both editors are interdependent: for example, the modification of a class name must be reflected in all class element instances in the MOLA rules that reference the given class.

In addition to the editors, the *MOLA 2 tool* contains also the *MOLA compiler* (built in a lower level transformation language *L3* [137], also developed at UL IMCS), running on the same repository. The *MOLA compiler* is described in detail in the PhD thesis of Agris Šostaks [130].



Association: Table <-> Column

End class:	Role:	Multiplicity:	Composite:	Ordered:	Navigable:
Column	pkey	0..1	☐	☐	☑
Table		0..1	☐	☐	☐

Fig. 61. MOLA editor implementation in METAclipse

Fig. 61 demonstrates the editor in action – with both a sample class and the MOLA diagrams visible. After the first version of METAclipse was completed (including about 180 domain-independent MOLA procedures), the implementation of the initial *MOLA 2 editor* required about one man-month to develop and test it (containing about

120 procedures in the domain-dependent part; there are about 30 essential classes in the domain metamodel). Adding additional services to the MOLA tool, all tool behaviour description was described by using approximately 450 MOLA procedures (including the domain independent procedures). The developed *MOLA 2 tool* was successfully applied in the European IST project ReDSeeDS [3].

However, developing model transformations for the MOLA tool required a lot of routine work. There were transformations similar to one another. Such transformations could be generated automatically from the mapping between the domain of the language and the presentation types of the language.

Still it is also necessary to describe the way the language specifies the tool behaviour. Model transformation languages are the most appropriate means for these tasks, implying that in simple cases mappings could be used, while complicated cases could be described by using transformations. The approach of this type is proposed in Section 5.3.

5.3 Mappings for METAclipse

This section focuses on the description of the way of adding mappings to a transformation based tool development framework. The *METAclipse framework* [86] and the model transformation language MOLA built by UL IMCS is chosen as the basis for the realisation of the proposed approach. The choice is based on the following – the framework is completely transformation based, it provides flexible ways of extension and it itself can be used in a bootstrapping manner for implementing the extended features.

To ensure usability of the proposed approach, mappings and transformations should be smoothly integrated. The proposed mapping language could be implemented by using an interpreter or a generator generating transformations in a model transformation language (MOLA in our case). This implementation decision affects integration possibilities. In both cases there could be used extension points where custom transformations can be added to the functionality defined by mappings. If the generator approach is used we can allow also manual modifications of the generated transformations.

The main extension mechanism should be extension points; the latter should be selected appropriately for the mechanism to suffice in the majority of cases. The extension points should permit to replace or extend the built-in mapping possibilities by custom transformations.

5.3.1 The Framework from the User Point of View

The proposed tool definition framework will be metamodel based. At the beginning the domain metamodel of a domain-specific language should be built (e.g., by the MOLA metamodel editor). The next step would be defining the presentation type model and mappings between the domain metamodel and the presentation type model. All this will be done, using graphical wizard-style dialogs in the tool development framework.

If the built-in mapping possibilities are not suitable for some task, the tool builder will be able to select/create a custom MOLA procedure (using the built-in MOLA editor). Appropriate parameters to and from this procedure should be passed to ensure integrity with the mappings. For each extension point there are predefined parameters passed to the procedures used in this extension point.

When the tool development is complete, the tool builder can press the button "*Build tool*". Thus, the tool executable in one step is obtained. Alternatively, if there is such a need the generated transformations can be edited and then compiled.

5.3.2 Mapping Definition

Mappings are based on typical mapping patterns. A large set of mapping patterns has been identified in *Generic Modelling Tool* [26] and they will be reused in the proposed approach.

The mapping definition is based on the mapping and presentation type metamodels as the abstract syntax of the "mapping language".

The visible form of this mapping language will differ from the one used for the mapping languages in CHAPTER 4. It is frequently required to define more complicated transformation logic using mappings in the DSL tool building, therefore, the tree based syntax is not appropriate. This language will show up as wizard-style dialogs that will build instances of mapping and presentation type metamodels. The appropriate tool support can be built with little effort using the *METAclipse framework*. A more detailed description is given in the following sections of this chapter. A simplified version of the mapping language from the domain to the presentation is also given in Section 7.3 where a compilator development for such languages is discussed.

The presentation definition in a graphical tool consists of several parts: property dialogs, diagrams, as well as a model tree, menus, etc. Informal mapping examples mentioned so far all have been related to mapping the domain to the diagram element types. Now we switch over to another part of the presentation – the property dialogs. It is because the proposed ideas can be easier demonstrated on this part and the corresponding metamodels are smaller. Here only an essential subset from the property dialog part of the presentation type and mapping metamodels is briefly sketched (in Fig. 63). We assume here that typical Eclipse-style dialogs are used.

When a property dialog for a domain class is to be defined, at first an appropriate property dialog type (i.e., its structure, element types and functionality) is designed, then it is mapped to the domain metamodel elements. A property dialog consists of tabs that can be either a field list (for displaying class attributes and linked class instances) or a grid (for displaying child instance properties in a tabular form). The basic element of both is a field whose type definition is the central point in the approach. It must be defined what must be shown for each field type when the corresponding class instance is selected. For many field kinds (e.g., combo box) the valid value set (e.g., a set of appropriate class instances) must be obtained and visualized. Finally, it must be defined what has to be done when the value is modified (in the Eclipse-style dialogs the model update follows immediately).

As the metamodel in Fig. 63 demonstrates, for all these situations possible typical cases are defined via mappings to the domain metamodel elements (e.g., which class attribute must be visualized in a field in the simplest case, see the fragment in Fig. 62).

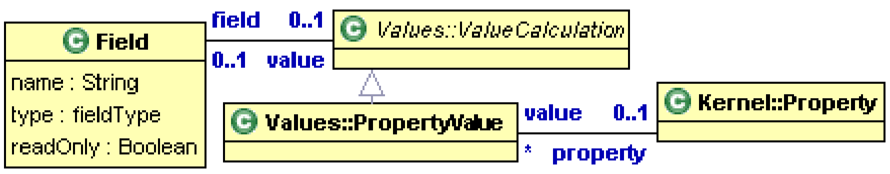


Fig. 62. Metamodel fragment, describing that the design pattern field is based directly on property

The metamodel contains also structuring elements defining various typical ways how these elementary mappings can be combined, e.g., expressions built over elementary mapped values. In all cases the corresponding mapping-based definition can be replaced by a call to a specified custom MOLA procedure. Another novel idea is using the MOLA patterns for defining custom instance set filters, e.g., for the selection of relevant child instances.

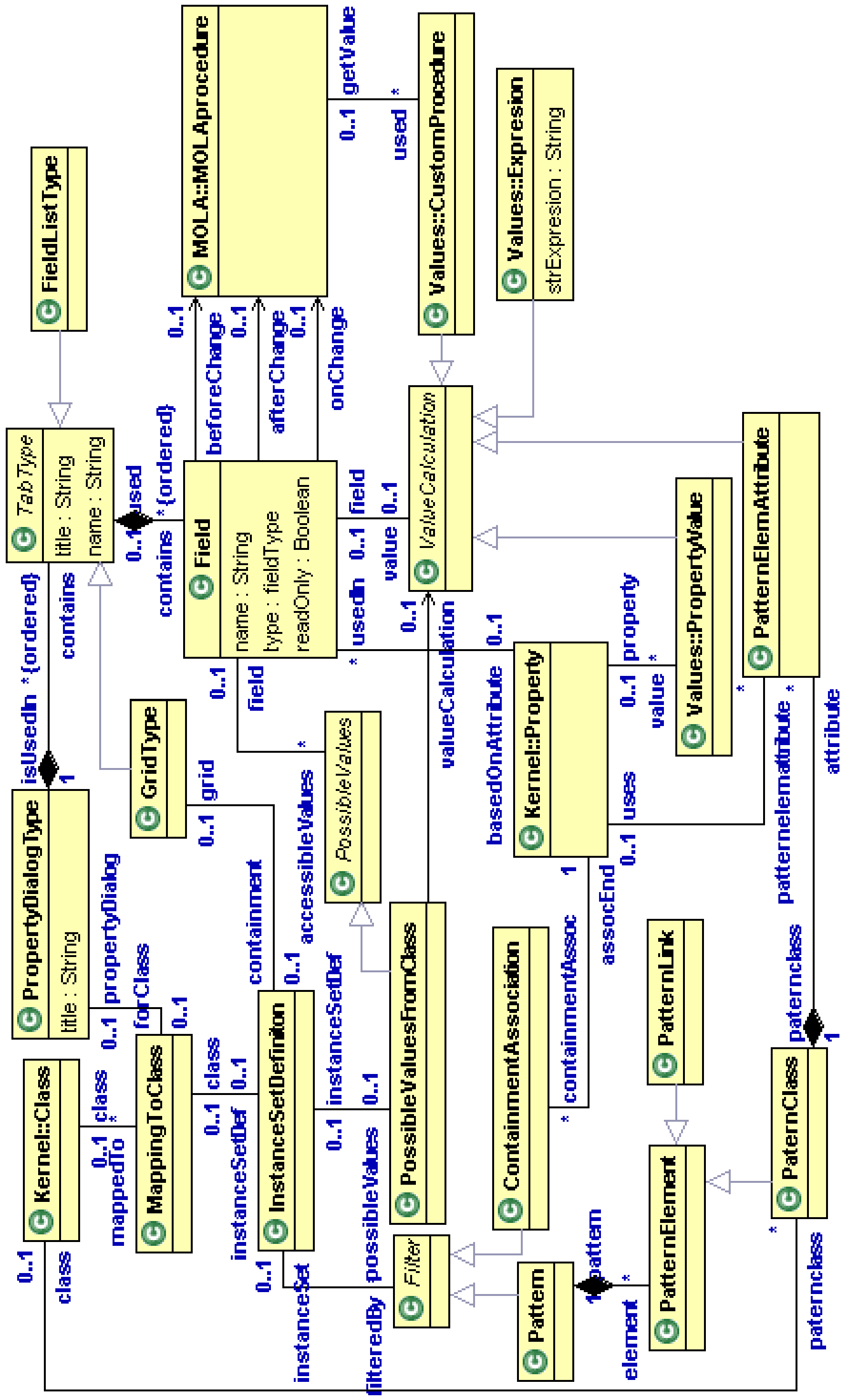
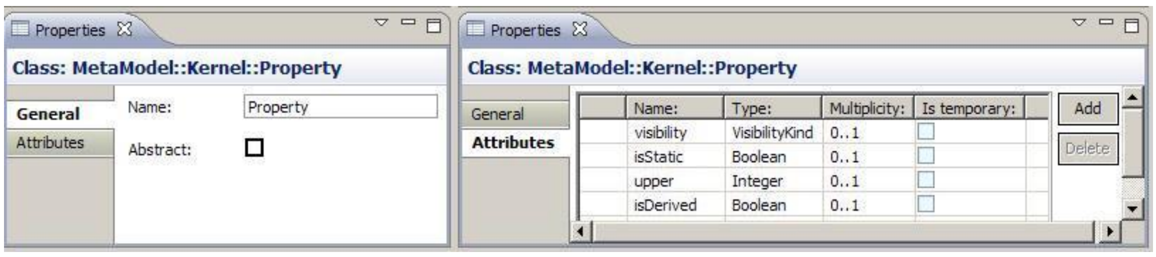


Fig. 63. Mapping and presentation type metamodel subset, describing the property dialogs

For example, we can use this mapping language to describe a property editor for the UML 2 class diagrams (based on the standard UML 2 metamodel [120]). For UML *Class* a property dialog type could be defined, consisting of two tabs. The first tab will contain a field list describing the UML Class itself. The attributes *name* and *isAbstract* are directly mapped to the fields in this tab. A uniqueness check (within a package) before the change is needed for the attribute *name*, and for this task a custom MOLA procedure can be invoked. The second tab could be a grid describing class attributes (see Fig. 64). In this case, the grid *InstanceSetDefiniton* feature is mapped to the *Property* class. The basic instance selection is via *ownedAttribute* master-detail association and additional filtering is defined by using the MOLA pattern selecting only those properties that are attributes (but not association ends).

Patterns are a very powerful tool; it allows the selected instance set to be easily specified. The use of MOLA pattern here is similar to the use of tree patterns in MALA4MDSD. Patterns are a very useful and universal tool for definition of constraints on the selected instance set.



Class: MetaModel::Kernel::Property

Name: Property

Abstract: ☐

Name:	Type:	Multiplicity:	Is temporary:
visibility	VisibilityKind	0..1	☐
isStatic	Boolean	0..1	☐
upper	Integer	0..1	☐
isDerived	Boolean	0..1	☐

Fig. 64. Class dialog example, general and attribute tab

The metamodel part for the diagram mapping and presentation types can be built the same way, only more classes would be present since it is more complicated.

5.3.3 Mapping and Transformation Integration

The most important task for the mapping metamodel is a seamless integration of mappings with custom MOLA procedures. MOLA is a procedural transformation language, therefore MOLA procedures are chosen as the integration unit. It does not restrict the integration possibilities, since any set of statements can be included in a procedure. Actually, it even allows reusing the same procedure in different contexts.

The mapping metamodel granularity and structure should be chosen so that each action could be extended or replaced by an appropriate custom MOLA procedure. The transformation based approach permits to use a more detailed mapping granularity than in the traditional mapping-based tools.

For each extension point, the set of required parameters for custom procedure is predefined. The predefined set should be compatible with the parameter set of the selected procedure.

In Fig. 65 an integration example is given. When a property dialog field is modified, a custom transformation can be executed as a pre-processor, postprocessor or instead of the action implied by the static mapping. A custom procedure can be used as well to calculate the field value to be displayed.

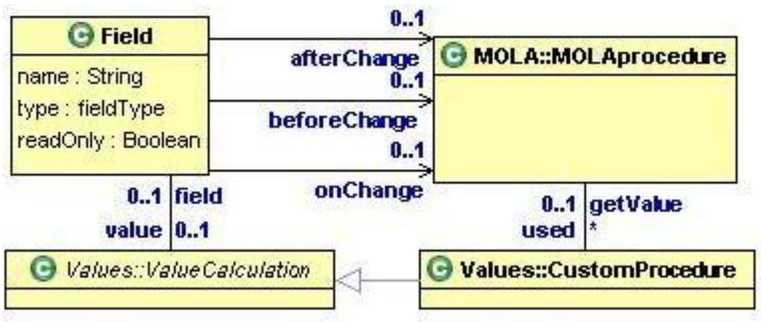


Fig. 65. Metamodel fragment describing mapping and transformation integration

The close integration of mappings and the transformation based approach is a key factor in reaching the goal when the transformations generated from mapping only need to be combined with the specified custom MOLA procedures, but require no direct manual modification.

5.3.4 Mapping Definition Language User Interface

We propose to use wizard style dialogs for the definition of presentation type model and mappings. These wizards will create instances according to the relevant metamodel. The presentation type and mapping definition will be integrated.

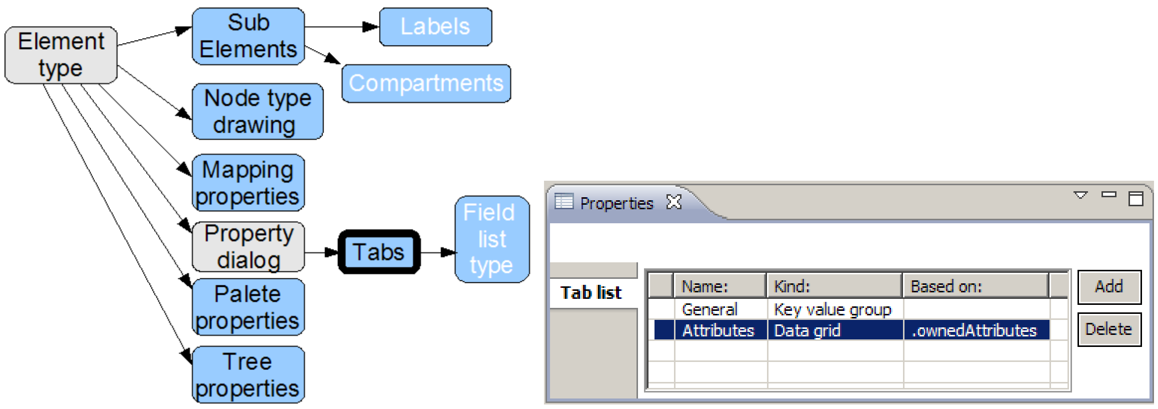
To generate presentation types and mapping for a domain class, the user will be asked to select the appropriate tool design pattern and enter additional properties of the presentation types to be created (for property dialog, diagram node type, etc.). The relevant mapping instances will be created automatically. The palette element, if needed, will be created simultaneously as well.

Wizards will be organised in several levels for the whole domain metamodel (as in GMF [172]) or on one domain class to see or modify the features related only to this class.

In addition to the presentation and mapping definition, wizards will allow for complicated cases to select custom MOLA procedures for the relevant extension points. These procedures will be created by using the built-in MOLA editor.

A natural way to implement the proposed mapping definition editor in the METAclipse framework is to build it as an extension of the existing MOLA tool [86]. Then a slightly extended metamodel definition editor can be reused for the domain metamodel creation and the MOLA editor can be used directly for creating custom procedures.

The mapping/presentation wizard itself could be implemented in several ways. A classical wizard style dialog sequence could be built, but this requires certain extensions to the METAclipse property engine. A more interesting and user friendly way could be the creation of wizard diagrams. The *dashboard in GMF* [172] could serve as a simple prototype for such diagrams. The possibilities of METAclipse permit to create dynamic wizard diagrams where each node represents some wizard dialog “page”. The dialog in such a page can be defined by using standard METAclipse property dialog facilities. The edges in such a diagram represent the order in which these pages must be visited. At the next step nodes and edges will be created and the existing ones enabled/disabled in response to the values the user has entered in the current node. A simplified sketch of a wizard diagram for a domain class mapped to a node can be seen in Fig. 66. It is assumed that the user currently defines tabs for the property dialog.



Name:	Kind:	Based on:
General	Key value group	
Attributes	Data grid	.ownedAttributes

Fig. 66. Wizard diagram example for a domain class mapped to Node

The same visual representation can be used to modify the defined mappings. After opening the appropriate wizard diagram the user can select a node and update the properties. If this modification influences dependencies to other wizard nodes, the user is asked to update these nodes as well.

We can think about other mapping visualisation possibilities, too. For example, a "mapping diagram" similar to the one in *Microsoft DSL Tools* [28] can be used with the domain metamodel on one side of the diagram and the presentation type model on the other, and with mapping lines connecting them. Actually, this mapping language would be rather similar to the mapping languages discussed in CHAPTER 4. Here mapping would be defined between the domain metamodel and the property dialogs, as well as between the property dialogs and the domain metamodel. It should be noted that these mappings would be bidirectional compared to the mappings discussed in CHAPTER 4. However, it seems that the tree is not the most appropriate representation of the domain metamodel; the class diagram representation is more appropriate. On the other hand, the tree seems a quite appropriate representation for the property dialog definition.

The domain part could be visualised by a standard class diagram. A palette element (if needed) can be given together with the presentation type. A presentation type can be visualized close to the node visualisation with this type. Instead of a label a short form of the template about the calculation of this label value can be shown. Sub-element mappings could be presented in a similar way, too.

5.4 Conclusions

In this section the graphical DSL tool development domain is discussed. The model transformation based tool METAclipse has been selected. The author of this thesis has developed transformations for the METAclipse framework and transformations for the first version of MOLA 2 tool in the METAclipse framework. To do it, in total about 450 MOLA procedures had been developed.

When analysing these transformations, it became clear that the simple part of transformations is more appropriate for mappings and the logically complicated part – for transformations.

As a result it was concluded that the tool building framework with options to combine mappings and transformation would be most appropriate for the tool development of DSLs with complicated dependencies between the domain and the presentation. Such a framework is proposed in this section.

CHAPTER 6

Template MOLA

One of the *Higher-Order Transformation (HOT)* application types in [183] is transformation synthesis. Transformation synthesis means transformation generation from various sources of information, including model mappings. A survey on HOTs [183] reveals that most of the HOTs have been written in ATL. In the case of ATL synthesis [183] the relevant ATL model is created and then extracted as a transformation text. The same task could be considered for graphical transformation languages, e.g., MOLA [76]. A MOLA transformation in abstract syntax could be created in the same way as the abstract syntax of ATL transformations. The transformation visualisation task for graphical languages is harder, but still feasible. Consequently, for graphical transformation synthesis the HOT approach is usable; however, the experience shows that usage of abstract syntax for the definition of HOT is inconvenient and time-consuming. It seems to be true for most of transformation languages, including ATL. A better template-based solution is proposed in this chapter.

There are many template-based model-to-text languages, e.g., the popular ones *JET* [174] and *mof2text* [123]. The basic application of these languages is to create a code from PSM model in the standard MDSD process. These languages typically contain facilities to navigate the given model according to its metamodel. However, the main advantage of these languages is the possibility to define the text fragment to be generated by the given rule as a textual template in the relevant concrete syntax. The variable parts in the text to be generated are specified by means of template expressions that typically contain model class attributes and variables.

An ATL transformation text could be created by using some template-based model-to-text language as well. Since MOLA is a graphical transformation language, textual template languages could not be applied here. In this chapter the problem of MOLA transformation synthesis by using template-based mechanisms is addressed.

New graphical template-based language Template MOLA for MOLA transformation synthesis is proposed. In this language elements to be created in MOLA can be defined explicitly in syntax close to the traditional MOLA statements. The generation logic in Template MOLA is described by facilities close to the standard MOLA. This part of the description is executed during the generation process. The elements to be placed in the created transformation are described in a MOLA extension consisting of template statements. The given extension is similar to the basic MOLA, but having a possibility to incorporate also template expressions that are replaced by the corresponding generation time values during the generation. Thus, the idea of textual template languages is adapted to a graphical language. The main advantages of the template approach are retained – adequate facilities to process and navigate the source model, and concrete syntax based descriptions of elements to be created as a result. The proposed solution is significantly more convenient for transformation generation than pure use of MOLA as a HOT.

All MOLA elements are retained in Template MOLA. Additionally, special template elements for easy MOLA transformation synthesis are included. They make it possible to define explicitly in a graphical syntax which MOLA elements should be created.

The Template MOLA language is an adaption of template mechanisms used for textual template languages (of the model-to-text kind) to a graphical language. Template MOLA is used for easy generation of transformations in MOLA from various input models as a substitute for the classical HOT approach.

6.1 Main Elements

In this section, the basic constructs of Template MOLA are described. The proposed Template MOLA language contains two kinds of MOLA statements: generation statements and template statements.

Generation statements are executed during the transformation generation process. They are used to define the logic of generation process on the basis of the provided input metamodel. All ordinary MOLA statements may be used as generation statements.

Template statements are meant to be “copied” to the generated “MOLA code” (in fact, a model) with template expressions replaced by the appropriate generation time values. Template statements look similar to ordinary MOLA statements but can be distinguished by their graphical style – the green colour. The most used template statements are template rule and template loop; however, other MOLA statements may be used as template statements, too.

Statements in Template MOLA are organized into procedures in the same way as in the traditional MOLA described in CHAPTER 2. A procedure may contain both generation and template statements; however, generation statements alone should constitute a valid MOLA procedure. Template statements may be interspersed between generation statements. Thus, the general idea of Template MOLA is that the "generation part" of a procedure is executed in the same way as the traditional MOLA. The only difference is that template statements to be executed in this process are copied to the resulting traditional MOLA procedures (instead of directly executing them). Certainly, there are some more complex situations to be described further, but at first glance Template MOLA means exactly that.

6.1.1 Template Rule

The most used template statement is *template rule*. In the generation time it is copied to the generated "code" (i.e., to the relevant generated MOLA procedure). Elements of the template rule may contain variable textual parts – *template expressions* (expressions enclosed in angle brackets followed (preceded) by a percent sign). These expressions are replaced by the corresponding generation time values.

An example of a template rule can be seen in Fig. 67. In this rule, the constraint in the class element *b:Class2* contains the template expression *<%@p.name%>* where *@p* is a known generation time reference (defined in the procedure containing this rule). Another kind of a variable part in a rule is a template expression specifying the class of a class element (here *c:<%@tc:Class%>*). The generation time reference *@tc* must point to an appropriate metamodel class, i.e., it must point to an instance of *Kernel::Class* (the *::Class* suffix in the syntax emphasizes that), and it must be set before the rule under discussion is to be executed. In the resulting traditional MOLA rule, this template expression is replaced by the referenced class name. Association links may also be specified by a template expression in order to adapt to a variable class element in the end. Association links are specified using *Property* at one end of the *Association*. *Property* at the other end and the *Association* is inferable from this *Property*. This template expression (*<%@prop:Property%>* in Fig. 67) must reference a property in the metamodel. The value of this reference must certainly be set correctly during the generation; in the presented example only the properties (related to *Class2*) of association linking classes *Class2* and *Class3* are valid. In the generated rule, the standard MOLA notation for association links (both role names) is used.

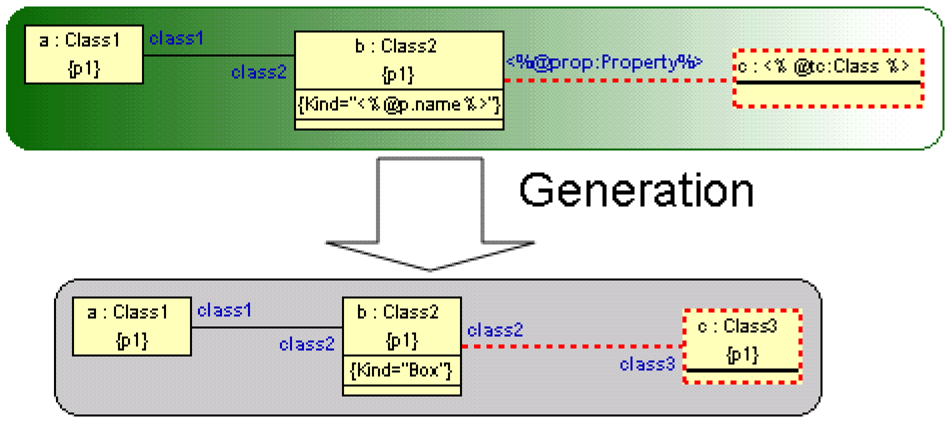


Fig. 67. An example of a template rule and the MOLA rule generated from it

The lower part of Fig. 67 shows the generated MOLA rule obtained from the template rule above. Here we assume that the reference *@p.name* has a string value “Box”, the reference *@tc* points to the class *Class3* and *@prop* to the role name *class2* of the association *class2* - *class3*.

6.1.2 Template Loop

Similarly to rules, the loop constructed in MOLA – the *foreach* loop statement – also has its template form in Template MOLA. The *template loop* is copied to the generated procedure during the generation process, including its body (which may also contain generation statements, see an example in Fig. 88, p. 226). The template loop in its *loophead* rule can use all the extensions introduced for the template rule. Fig. 68 gives an example of a template loop, a simple construct for creating copies of all instances of an arbitrary class. In the loophead of this loop, the class to be used in all class elements (including the loop variable *orig*) is defined by the template expression *<%@type:Class%>* which means that the reference *@type* must be set to the required class before the given template loop. Then a traditional MOLA loop is generated from this template loop, and the generated loop performs the instance copying for the given class. The additional class element *orig_exists* with NOT constraint is used as NAC (negative application condition) prevents a repeated copying of the copies. The example presents a very simple case of another area of a typical application of HOTs for transformation generation in [183] – building a generic transformation for a previously unknown metamodel. (This application is also discussed in Section 7.4.1.)

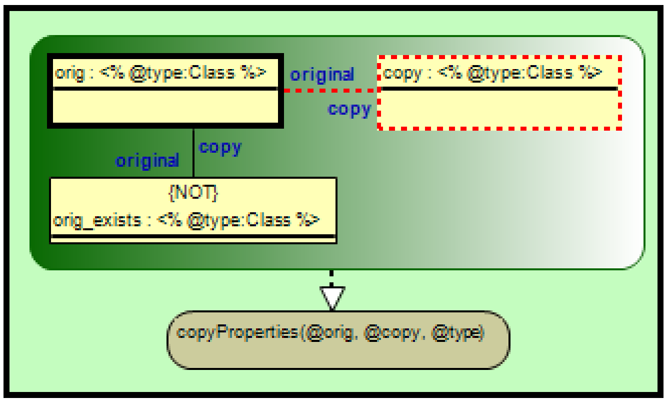


Fig. 68. An example of a template loop

6.1.3 Call Statement and Parameters

The body of the loop in Fig. 68 contains another template-related construct – a MOLA procedure call with arguments of previously unknown types (@*orig* and @*copy*). The type of these arguments is learned only during the generation process. The given procedure call contains one more argument – the reference to the type itself. This last argument is a generation-time argument which is not included in the generated invocation (it has no sense in that context). Yet for the generation of the procedure *copyProperties*, which has to perform copying of all attributes of the arbitrary class, such a parameter could be of high value for defining an appropriate generation time loop (traversing the attributes).

The exact kind of procedure parameters is visible in its declaration. There are three types of parameters that can be declared in a Template MOLA procedure – *template*, *generation* and *type* parameters. *Template parameters* are created in a generated procedure. *Generation parameters* are used in the generation time and are not created in a generated procedure. Appropriate arguments must be passed in call statements for the template and generation parameters. The *type parameters* are also used in the generation time, but they are inferred from other parameters instead of passing them explicitly. Since the types of parameters in MOLA are described by using the class *Kernel::Type*, *type* parameters may refer to the instances of *Kernel::Type* (*Class*, *PrimitiveType* or *Enumeration*) only.

6.1.4 Template Expressions

We have already given an insight into the template expressions used in Template MOLA; however, the example does not cover all possible use cases. Therefore, a short summary on template expressions follows. The most common elements where template expressions appear are class elements within a template rule. A template expression can be used to specify the class of the class element. In this case, the template expression must be a reference to *Kernel::Class* instance. If template expressions are used to specify the name of the class element, constraint or expressions in the assignment, a string expression is used for this purpose. These expressions may contain the generation time variables, parameters and attribute specifications, but no template element references. References to instances of appropriate classes can be used to specify references to objects, e.g., the attribute to be used in an assignment within a class element (a reference to *Kernel::Property*), or the source/target end of an association link (a reference to *Kernel::Property* as an end of *Kernel::Association*). Template expressions can also be used in template text statements and in call statements to specify arguments that conform to the template parameters of the called procedure.

6.1.5 Template Elements

On the whole, the idea of generating template procedures in Template MOLA and providing appropriate naming conventions for them is based on the principles similar to those in the OOP languages, such as *C++* and *Java*, also containing some template mechanisms.

A list of all Template MOLA elements is given in Table 7. The name, image and a short description are given for each element. Elements are divided in two groups: the Template MOLA elements – new elements (compared to MOLA) introduced in Template MOLA – and MOLA elements with a modified semantics, achieving modification by adding additional generation time semantics for some MOLA elements in Template MOLA. This issue is discussed in detail in Section 6.5.

Table 7. Template MOLA elements

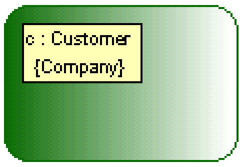
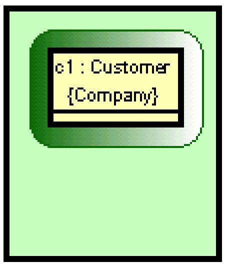
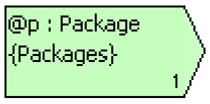
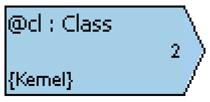
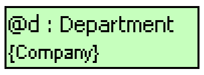
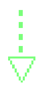
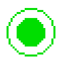
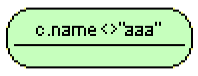
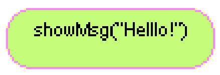
Image	**Name**	**Description**
Template MOLA Elements		
	Template rule	This element creates the MOLA rule in a synthesised transformation. The rule is created one to one. Template expressions are replaced with their generation time values (see Section 6.1.1).
	Template loop	This element creates a loop in a synthesised transformation. The Template loop may contain generation time elements, describing the algorithm for the loop body generation. Template elements executed in the loop body are generated in the loop body (see Section 6.1.2).
	Template parameter	This element indicates that a parameter should be generated for a generated MOLA procedure.
	Type parameter	This is an implicit parameter. It is used when a Template MOLA procedure may be called from a MOLA procedure. It is used to describe the type of template parameter.
	Template variable	Creates a variable in the generated MOLA procedure.

Image	Name	Description
	Template control flow	Describes generation of the control flow explicitly. May be used between the Template elements and elements of dual nature: template rule; template loop, template end symbol, template text statement, template call statement, template external call statement start symbol, end symbol, call statement (see Section 6.5.3).
	Template end symbol	Describes generation of the end symbol in a MOLA procedure (see Section 6.5.4).
	Template text statement	Text statement generations
	Template external call statement	External call statement generations

MOLA Elements with Modified Semantics

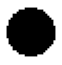
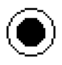
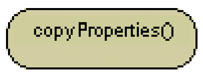
	Start symbol	Describes start of the procedure and generation of the start symbol.
	End symbol	Describes end of the procedure and generation of the end symbol, if the current control flow has no end symbol (see Section 6.5.4).
	Call statement	Executed as a call to another procedure and generation of a call statement to the generated procedure corresponding to the call. If marked as inline, the generation is omitted (see Section 6.5.2).

Image	Name	Description
⇣	Control flow	Describes the execution logic. If the template control flows are not shown explicitly, execution control flows are used to determine the template control flows to be generated (see Section 6.5.3).

6.2 Template MOLA Compared to MOLA as a HOT

A question may arise about the advantages of transformation synthesis in Template MOLA in comparison with the traditional MOLA. Writing higher-order transformations for transformation synthesis directly in MOLA requires defining of the creation of all MOLA metamodel elements explicitly (i.e., according to the abstract syntax of MOLA). To create one rule, we have to create the rule, all its class elements, all association links, all their sub-elements, and to map them to the appropriate types from the metamodel of this transformation. Fig. 69 demonstrates a transformation for the creation of one rule by using the traditional MOLA as a HOT language. Creation of the same rule in Template MOLA was demonstrated in Fig. 67 (p.182).

It is easy to see that the code for creation of this rule in Template MOLA is significantly more readable than in the traditional MOLA. First of all, the size of the rule creation pattern differs significantly. Note that in this example we considered the creation of a very simple rule. The difference is even more significant for more complicated rules. The same situation holds true for loops since they mainly consist of rules.

The same issue of complexity arises in regard to other transformation languages usable for HOT tasks.

Template MOLA allows to implement the same HOT tasks with much less effort and with a smaller amount of errors since the structure of the resulting MOLA statements is clearly visible already in the templates.

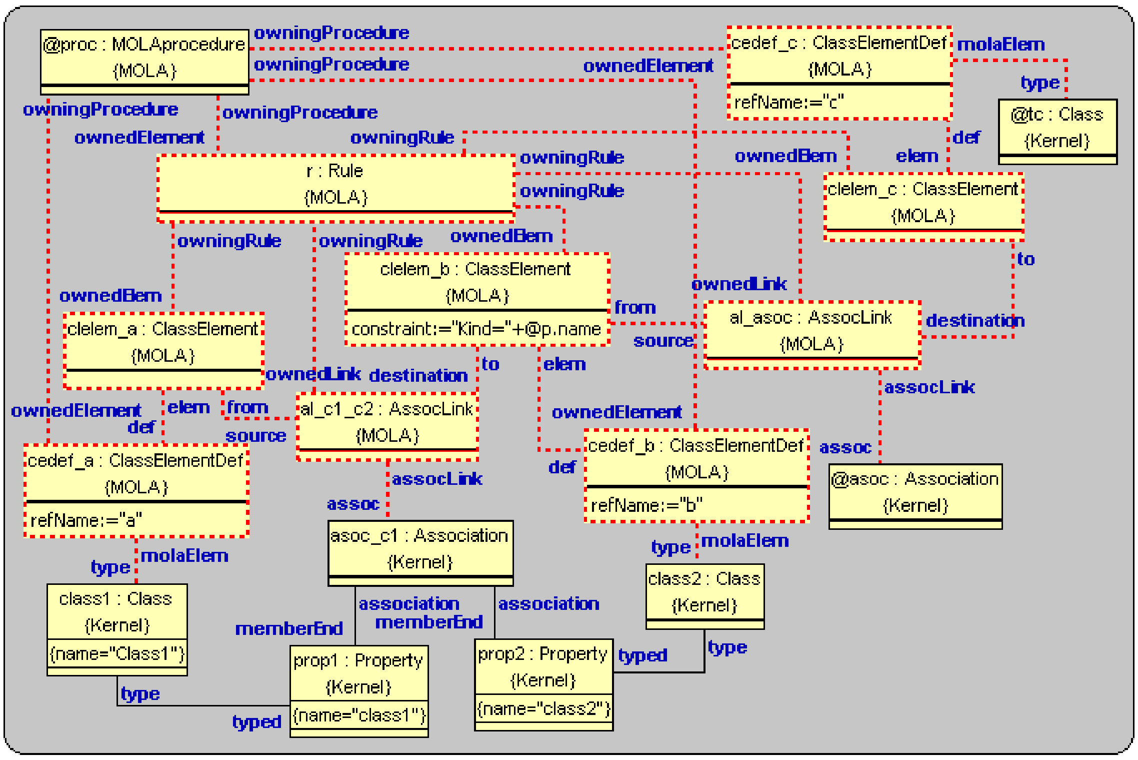


Fig. 69. Creation of the rule from Fig. 67, using MOLA as a HOT

6.3 Template MOLA Example

A simple Template MOLA example is demonstrated in Fig. 70. In this example we consider a simplified data migration from a model based repository to the *OWL/RDF* [192, 193] based repository built according to the *ODM metamodel* (see [124]). We assume that we have an UML class diagram describing the structure of the model based repository and a mapping information describing the way this model should be modified when transferring it to the OWL based repository. In particular, this mapping demonstrates which classes together with their instances should be transferred and the way the classes should be renamed. The transformation in Fig. 70 iterates through all classes mapped to OWL. For each such class it creates a rule creating an OWL class and it creates a loop copying model instances of this class to the OWL instances of this class. We can run this Template MOLA on a class/mapping model and we will obtain an efficient data migration tool just for this model. In this example the main template MOLA statements are demonstrated as well. In the template rule the value of template expression is assigned to the attribute name. This value will be determined in the generation time and then used in the generated code. The loophead also contains a class element with the template type that will also be determined in the generation time and replaced with the appropriate value. On the other hand, the types of other class elements are constant and the same in all generated transformations from this template MOLA program.

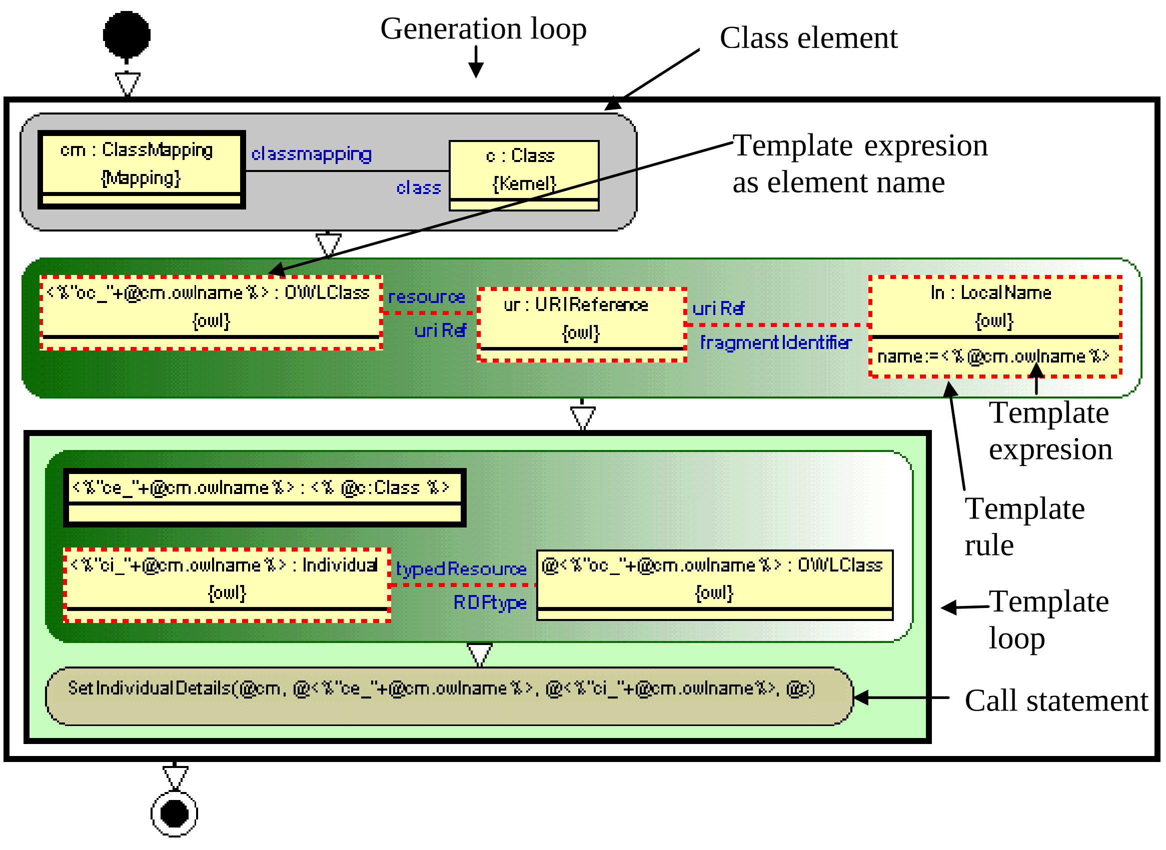
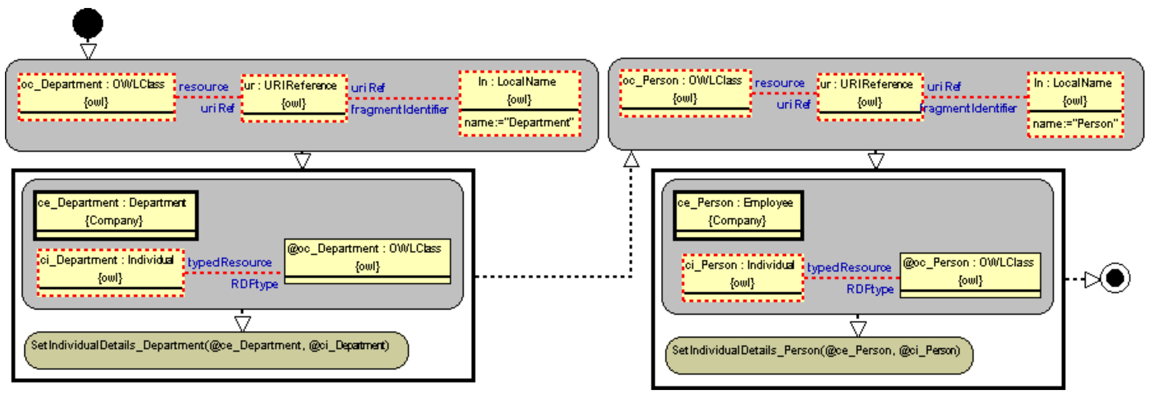

Generation loop
Class element
Template expresion as element name
cm : ClassMapping {Mapping}
classmapping
class
c : Class {Kernel}
<%"oc_"+@cm.owlname%> : OWLClass {owl}
resource
uri Ref
ur : URIReference {owl}
uri Ref
fragmentIdentifier
ln : LocalName {owl}
name:=<%@cm.owlname%>
Template expresion
Template rule
<%"ce_"+@cm.owlname%> : <%@c:Class%>
<%"ci_"+@cm.owlname%> : Individual {owl}
typedResource
RDFtype
@<%"oc_"+@cm.owlname%> : OWLClass {owl}
Template loop
SetIndividualDetails(@cm, @<%"ce_"+@cm.owlname%>, @<%"ci_"+@cm.owlname%>, @c)
Call statement

Fig. 70. Template MOLA example: Generator for copying UML class model instances to OWL instances

The transformation example in Fig. 70 contains also a *call statement*. This call statement contains two types of parameters. The parameters "@cm" and "@c" are the generation time parameters, while "@ce_..." and "@ci_..." are the template parameters. The generation time parameters are used only for transformation synthesis. The template parameters will appear in the generated code as well. It means that we will obtain a call in the generated code only with two parameters. The generation loop creates several rules and loops in one procedure. These generated rules will contain elements with different types. We need also different names for the generated elements to distinguish between them. Therefore the template expressions are used also to determine the generated element names and reference to them.

A simple example of MOLA transformations obtained by executing the Template MOLA transformation from Fig. 70 can be seen in Fig. 71. A rule and a loop is generated for each class with the mapping. In this case two classes are considered. The class *Department* together with its instances is copied without renaming to the OWL repository. The class *Employee* is transformed to the OWL class *Person*. Thus, a specific transformation has been obtained, for migrating the instance level (M0) data for these two classes to the OWL repository. We remind that the example illustrates a simplified instance level data migration, but not the general ontology migration from UML coding to OWL (as in [124] for example).



oc_Department : OWLClass {owl}
resource
uriRef
ur : URIReference {owl}
uriRef
fragmentIdentifier
ln : LocalName {owl}
name:="Department"
oc_Person : OWLClass {owl}
ur : URIReference {owl}
ln : LocalName {owl}
name:="Person"
ce_Department : Department {Company}
ci_Department : Individual {owl}
typedResource
RDFtype
@oc_Department : OWLClass {owl}
SetIndividualDetails_Department(@ce_Department, @ci_Department)
ce_Person : Employee {Company}
ci_Person : Individual {owl}
@oc_Person : OWLClass {owl}
SetIndividualDetails_Person(@ce_Person, @ci_Person)

Fig. 71. The result of transformation from Fig. 70

In these transformations as in any transformations metamodels are used. In a generated code, instances of some metamodel are transformed to the OWL metamodel instances. It means that the metamodel used in the generated code consists of two parts – the OWL metamodel and the domain metamodel. A fragment of the OWL metamodel, used in this example, is shown on the left side of Fig. 72.

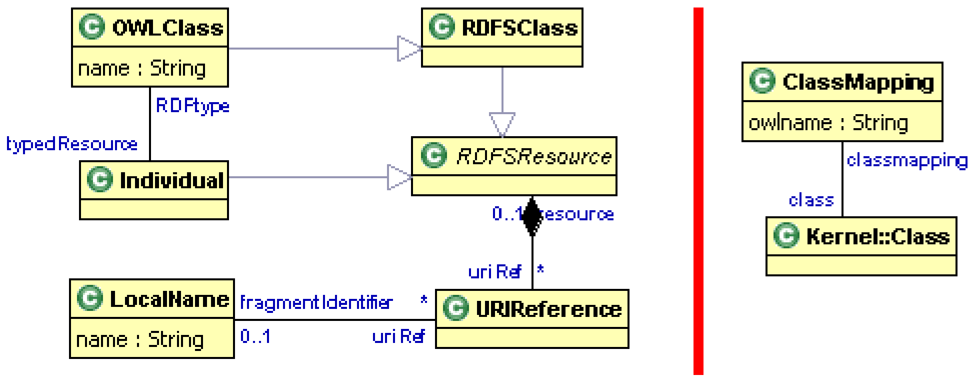


Fig. 72. A metamodel fragment used in a class model to the OWL transformation in Fig. 70

According to the task specification the domain metamodel is the UML class model describing the given repository to be transformed. In Template MOLA this domain metamodel is used as input data affecting the generated code. It means when generating transformation the domain metamodel is treated as instances of the UML metamodel. When executing the generated transformation, the domain metamodel is treated as a metamodel.

In the description of transformation logic besides the domain metamodel also *ClassMapping* (on the right of Fig. 72) is used describing how the UML classes should be transformed to the OWL classes. As a result *ClassMapping* and *Kernel::Class* (from the domain metamodel) are used in the generation time statement (rule).

An input for the Template MOLA transformation is a model defined according to the metamodel sketch shown on the right side of Fig. 72. When executing this Template MOLA transformation the result is a MOLA program. The input model consists of the domain metamodel (the repository structure description) and mappings, describing representation of this domain metamodel in OWL. From the domain metamodel description only *Kernel::Class* is shown in Fig. 72. Instances of *Kernel::Class* are classes of the processed domain metamodel. Other classes from the UML class diagram metamodel are required, for a complete definition of the transformation. Here only the class mapping is shown from the mapping metamodel. There will be other mapping classes in the complete transformation definition as well. It should be noted, that the mapping and the UML class diagram metamodels are related. This relation should be treated as a part of the mapping metamodel.

The OWL metamodel classes are used in the template rules. The pointer to the instance of *Kernel::Class* is used as well, however, here the instances of metamodel are used. Metamodelling in Template MOLA is discussed in detail in the next section.

6.4 Metamodelling Issues

As in any other transformation language, transformations in MOLA are based on the appropriate metamodel definition, frequently containing the source and the target part. The definition of a metamodel for Template MOLA is more complicated because the relevant HOT level features for defining the generation logic have to be supported. At the same time, the use of template statements requires the presence of the appropriate parts in the metamodel.

6.4.1 Use of Metamodels Defining Higher-Order Transformations in MOLA

In order to have a deeper understanding of metamodelling issues in Template MOLA, we start with the comparison to the metamodel structure required for defining a traditional HOT in MOLA for synthesis of a MOLA transformation (an example of which is in Section 6.2 above). Fig. 73 demonstrates the structure of this metamodel. The source of the HOT is the source model (a mapping definition or something similar) corresponding to the source metamodel. The HOT must create a complete MOLA transformation definition consisting of a specific metamodel for this transformation (frequently containing the source and the target parts) and the proper transformation (a set of MOLA procedures). Similarly, at the metamodel level, the definition of HOT is based on two metamodel parts that serve as a target metamodel for this HOT. Firstly, there are MOLA metamodelling facilities named MOLA MOF MM (actually, the *Kernel* package mentioned in 2.1). Secondly, the MOLA procedure metamodel (MOLA MM) is required.

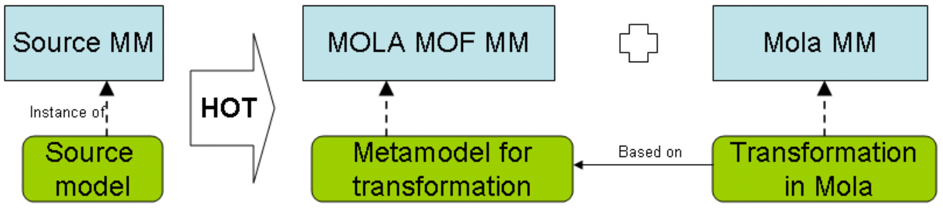


Fig. 73. Models to be used if higher-order transformations are written in MOLA

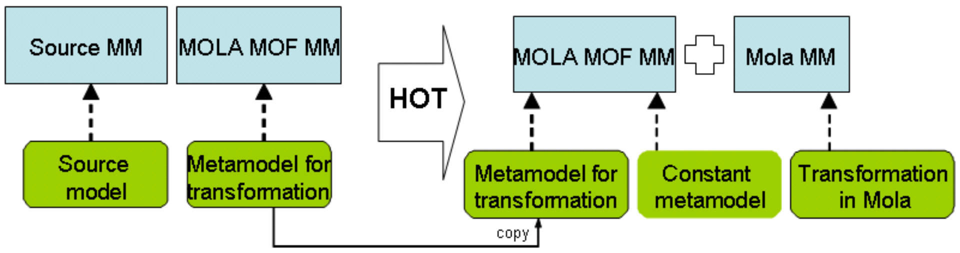


Fig. 74. Models to be used if the domain metamodel is analysed and higher-order transformations are written in MOLA

Actually, the approach presented in Fig. 73 is a simplified view on metamodels in HOTs. Very often, besides mapping the domain metamodel is analysed (as in Fig. 70) as well. This domain metamodel is used in a transformation logic description as instances of MOLA MOF, however, in the generated code it is used as a metamodel – types of class elements. Besides this domain metamodel also some constant metamodel could be used as types of class elements in the generated code. If we consider the example discussed in the previous section (actually, the generated result), the domain metamodel and the OWL metamodels were used there. The domain metamodel was the transformation source metamodel and OWL was the transformation target metamodel. In this case OWL plays the role of a constant metamodel.

It should be noted that there may be cases when one of these metamodels is empty. For example, the instance cloning, discussed in Section 7.4.1, uses only the domain metamodel. Generation of transformation between fixed metamodels may use only a constant metamodel.

6.4.2 Metamodels in Template MOLA

Now we can focus on the differences in a metamodel structure if Template MOLA is used instead of a standard HOT approach for the same tasks. Fig. 75 shows the general transformation synthesis by Template MOLA (an analogue of Fig. 73). As a rule the “runtime” metamodel for the generated transformation (more precisely, its variable part), must also be provided as an input to the Template MOLA-based HOT implementation. This situation could certainly occur in the general case of Fig. 73, but in Fig. 75 this situation is clearly syntactically visible. Such metamodel division was already introduced in Fig. 74 where MOLA was used as HOT. It is due to the necessity to use template expressions for accessing the classes of this variable metamodel part in template rules in a generic way (see Fig. 68, p.183). A typical example of such variable part is the domain metamodel (as in Fig. 70, p.189). The difference from Fig. 74 is the necessity to provide the constant part of this “runtime” metamodel for the definition of Template MOLA-based HOT. This is due to the fact that the classes of this constant part are used to define “constant” class elements in template rules. Therefore, these classes must be defined before the definition of Template MOLA rules. Although this constant part of the metamodel is clearly an instance of the MOLA MOF metamodel, in order to be referenced in “constant” Template MOLA elements, it must be provided alongside the MOLA MOF metamodel itself. Metamodel packages, included in a complete transformation definition in Template MOLA, belong to two adjacent metalevels. However, it is not confusing since the usage of their elements is clearly distinguished. Classes form different metamodels may be used in different contexts in Template MOLA. This issue is discussed in Section 6.4.4.

All different metamodel types used in Template MOLA (given in Fig. 75) are used in the example discussed in Section 6.3. The Template MOLA transformation for this example was shown in Fig. 70 (p.189) and its metamodel sketch was presented in Fig. 72 (p.190). The OWL metamodel (the left side of Fig. 72, p.190) is used as a constant metamodel. The mapping metamodel is used (in this case the class *ClassMapping*) as the source metamodel. A UML class diagram is used as a variable metamodel. In Template MOLA, the MOLA MOF metamodel is used, in the generated code its instances are used. Only *Kernel::Class* is given in Fig. 72 (p.190). However, other classes could be added as well. This metamodel is connected to the mapping metamodel (the source metamodel). The connection should be treated as a part of the source metamodel. In fact, this is a typical situation for mapping languages.

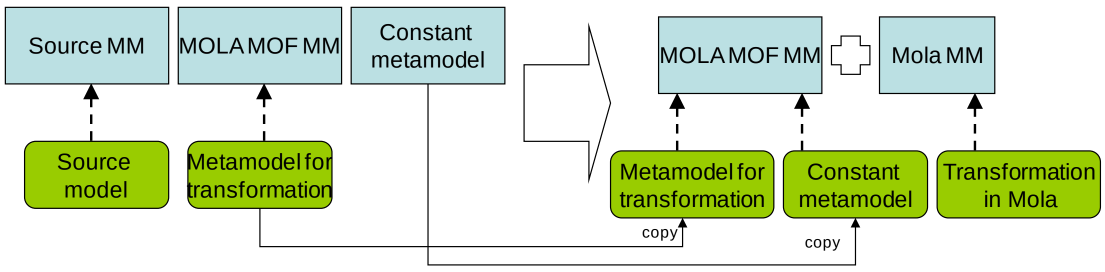


Fig. 75. Metamodels and models used for defining transformations in Template MOLA

The same way as in MOLA, in Template MOLA depending on the task specific requirements some metamodels could be omitted, as in the example of instance cloning only the domain metamodel (the metamodel for transformations) is required. In this use case the source metamodel and the constant metamodels are empty. Building a compiler for the mapping language MALA4MDSD the constant metamodel is empty. In the DSL tool building all three metamodel types are required.

6.4.3 Roles of Different Metamodels in DSML Tool Development

A typical application of HOTs in general and Template MOLA in particular is the generation of transformations from mappings for metamodel-based graphical DSL tool building. The tool building platforms, really requiring it, are *METAclipse* [86] and *ViatraDSM* [133]. However, the basic ideas can also be demonstrated in the popular *Graphical Modelling Framework (GMF)* [172] in *Eclipse* (we assume for a moment that transformations are generated in MOLA instead of *Java* for all actions). Fig. 76 illustrates the specialisation of the metamodelling situation in Fig. 73, when MOLA transformations are generated by HOT for a DSL tool – i.e., we assume that the GMF generator is implemented as a HOT instead of being written in *Java*. The source metamodel now consists of several parts with different roles. A definition of DSL normally is based on the relevant domain metamodel (abstract syntax) using, in turn, a version of MOF as a metamodel (in particular, the *MOLA MOF* could be used in such a role). Another part of the metamodel used by GMF and similar platforms is the presentation type metamodel (named graphical definition metamodel in GMF) and the mapping metamodel. Together they provide the means for graphical syntax definition of a diagram and mapping definition from the domain metamodel classes to presentation types in the diagram (by these means the instances of these classes must be visualized). The generated transformations in the runtime should use the same domain metamodel; therefore, this metamodel must be copied by the HOT to the generated transformation. There is also a constant part of the metamodel – the presentation metamodel (named notation metamodel in GMF) – which defines possible diagram elements at the runtime. This constant part should also be created by the HOT. One of the tasks the generated transformation should do in the runtime is to create a visual diagram element for a new domain class instance (according to the defined mapping). Thus, two important special features have appeared in this application: the use of the domain metamodel in two different roles (a part of the HOT source and a part of the created transformation metamodel), and the constant (independent of the source) presentation metamodel is included in the created transformation. In fact, the reuse of a part of the HOT source as a variable part of the metamodel for the created transformation is quite typical when transformations are generated by HOTs from mappings (as it was already underlined in the comments to Fig. 74).

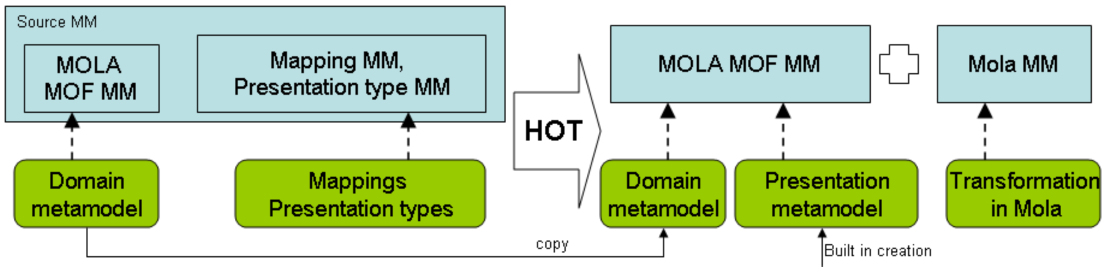


Fig. 76. Models used in case MOLA is used as a HOT for tool building

Finally, we analyse the application-to–metamodel-based tool building in Template MOLA (Fig. 77). The main difference from Fig. 76 is that the presentation metamodel plays the role of the constant part of the metamodel for transformation. Therefore, it must be provided before the definition of Template MOLA. Note that classes for mappings and presentation types can only be used in the generation (non-template) rules and loops of Template MOLA (they play the role of the source metamodel). The domain metamodel is clearly the variable part of the metamodel for transformation. An example of this kind of application is presented in Section 7.3.

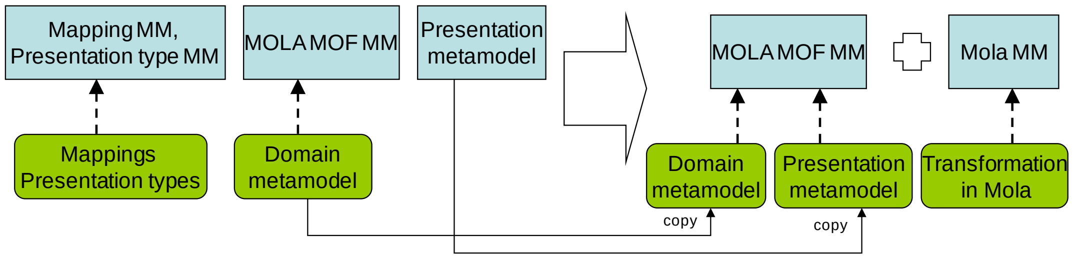


Fig. 77. Metamodels and models used to define transformations in Template MOLA for tool building

6.4.4 Use of Metamodel Elements in Template MOLA Transformations

Now, some remarks on the permitted use of metamodel elements in Template MOLA constructs. The source metamodel elements can be used directly only in the generation (non-template) statements of Template MOLA. They can also be used inside the template expressions in template statements. Elements of the variable part of the metamodel for transformation (the "runtime" metamodel) can be referenced via the corresponding classes of the MOLA MOF in the generation statements as well. The same elements can be referenced in template statements only via template expressions for the types. The elements of the constant part of the metamodel for transformation can only be used in "constant" class elements in template rules.

6.5 Elements of Dual Nature in Template MOLA

There are some elements in Template MOLA which are used on the one hand for the description of the transformation generation logic and on the other hand reflected in the generated code. Such elements are call statements, start symbols, end symbols and control flows.

The situation with start symbols is very simple. If such an element is come upon it is created in the generated code and then executed according to its semantics in the generation process.

Semantics of other elements is described in this section.

6.5.1 MOLA Procedure

The most important structuring element in Template MOLA is *template procedure*. In some sense it has a dual nature. It structures the generation algorithm into smaller parts and at the same time is reused to describe the structure of what should be generated. It should be mentioned that it is possible to generate several MOLA procedures from one Template procedure. The generated code may depend on the generation parameter values, therefore it may be required to generate one procedure for each value used (more precisely, invoked with this parameter value). In such cases we should distinguish between these procedures and give them different names. It is possible to use the default name generator or to define a template expression describing how the procedure name should be created. The default name is generated from the procedure name and the values of generation time parameters (parameter and type parameter), however, typically the custom name expressions are used. This is also the case of the example described in Section 6.3, where the *owlname* attribute from the *cm* parameter is used as a suffix in the generated procedure names. The procedure name expression is defined by using the property editor, though it is not visualised graphically.

The generated procedure name is also used to determine when a new procedure should be created and when an existing one could be reused. When a call statement is processed during the generation, the name expression of the invoked procedure is evaluated. If the value of the name expression matches the name of an existing procedure, the existing procedure will be reused. Typically, this name expression contains constants and values of the generation time parameters. The described mechanism permits to have the required control over the procedure duplication.

There can also be cases when the amount of the code generated by a template procedure is very small. So we may want to include the code generated by this procedure into the procedure it is called from (by replacing the call statement). To solve this problem we allow the "*inline*" annotation for call statements. It means that the code generated by an invoked procedure is embedded in the current one. In the generated code references to the template parameters are replaced by the values of the corresponding call parameters.

Besides optimizing the generated procedure structure, the "*inline*" annotation is vital for supporting the use of merge mechanism (see Section 6.6).

6.5.2 Call Statement and Parameters

Semantics of *call statements* is similar in this sense. They are used both for calling of procedures describing the generation logic and at the same time reused in the generated code. Unless the *inline* option is used the call statement is generated to invoke the appropriate procedure (according to the name generation expression in the called template procedure).

However, a call statement is directly related to the parameters of the called procedure. The procedure may contain the template parameters and the generation time parameters. The template parameters are kept in the generated code. The generation time parameters are used only for the description of the code to be generated by the called procedure. They are omitted in the generated call statement.

Let us consider the call statement in Fig. 70 (p.189) as an example. It has 4 parameters: 2 generation time (the first and the last) and 2 template parameters. In the generated transformation example in Fig. 71 (p.190) the generated calls have 2 parameters corresponding to the template parameters. In this case the generation time parameters are used for the description of how the body of the procedure *SetIndividualDetails* should be generated.

6.5.3 Control Flow

As already presented in Table 7, there are two types of control flows in Template MOLA: template control flow and (MOLA) control flow. Template control flows are used to explicitly define how control flows should be built in the generated code. Control flows describe the execution order of Template MOLA elements, however, frequently they are also used to infer control flows in the generated code.

A typical Template MOLA program describes synthesis of a MOLA transformation. Typically the synthesis of MOLA elements is described in a top-down manner (from the start symbol to end symbol). In this case the generation order of MOLA elements reflects also the way these elements should be connected with control flows. In this case control flows in the generated code can be easily inferred from the generation control flows; it means that only control flows describing the generation logic must be defined in simple case. This typical case is supported in Template MOLA using the heuristics described bellow. However, if more complicated control flows (e.g., branching) are required then it is necessary to use template control flows to define explicitly the control flows to be generated or even the merge mechanism to create arbitrary complicated control flow structures.

Now we will shortly describe the execution semantics of *template control flow*. Template control flows can go from one element to another element. Only the forward control flows are processed. By a forward control flow we understand a control flow whose outgoing flow end is created before the incoming flow end. If a template control flow goes from/to a template element then the element generated from this template element is used as a flow end of the control flow. If a control flow goes from/to a generation time element then this end of the control flow is moved to the next created element in the generated code. The only exception is a template control flow from foreach loop. The outgoing flow end of this control flow is the last element generated by foreach loop. If nothing is generated by foreach loop, then this control flow is skipped. A template control flow whose source end is processed, but the target end is not processed is skipped as well. It should be noted that there are rules restricting the usage of template control flows, e.g., outgoing template control flows from end statements are prohibited.

If something more specific is required, e.g., backward control flows, the merge mechanisms described in Section 6.6 should be used.

If all template control flows were defined explicitly the Template MOLA diagrams would become unnecessary complicated. Therefore, in simple cases the generated control flows are inferred from the template element execution order. It means, control flows used for the description of the generation algorithm are used also to decide what kind of control flows are to be included in the generated code. In fact, some heuristics are used there to infer how control flows should be created. In most cases the default principle described below is sufficient.

If there are two template elements in the description of the generation logic following each other and there are no explicit control flows defined, then a control flow between them is created in the generated code (more precisely, between the elements generated from these template elements). The same holds true if instead of one or both of the template elements a node with dual nature is used. Actually, this rule is more general when a new element is generated in the code, then a flow from the previously generated element to the new one is created. The same rule holds true for generation time loops as well. It means a flow between the last element generated in the previous iteration and the first element of the next iteration is created. For example, in Fig. 71 (p.190) a flow between the loop dealing with *Departments* and the rule dealing with *Persons* is created. At the beginning of the loop a flow from the previously generated element is created. After the loop a flow to the next element is created.

This automatic inference of flows simplifies transformation creation in Template MOLA. A user, creating a transformation generation procedure, does not have to define additional control flows describing the code to be generated. It should be noted that in all Template MOLA examples included in the thesis it is possible to define transformation synthesis using only (MOLA) control flows. However, if it is necessary it is possible to specify the control flows explicitly. If something even more specific is required, the merge mechanisms described in Section 6.6 should be used. By using the merge mechanism it is possible to obtain any control flow structures.

6.5.4 End Symbol

Similarly to control flows there are also a template end symbol and an end symbol. Template end symbols are used to describe the generation of end symbol: however, in simple cases the end symbol in the generated code could be inferred from the end symbol.

In these cases by executing the end symbol it is created in generated code as well. Here heuristics are used to support typical cases. It is applicable if the last template element was not an end symbol, it was not merged to the other element (see Section 6.7) and there were no explicit control flows from the last template element.

If the end symbol should be generated before the generation procedure completes its execution or multiple end symbols are required, then the template end symbol should be used. The next template statement following the template end symbol should be the element with merge (see Section 6.7) or an element with explicit incoming template control flows defined. Between them many generation time elements could be used. It should be noted that the outgoing template control flows are not allowed from the template end symbol.

6.6 Graphical Template Languages Versus Textual

Section 6.1 gives the basics of the proposed Template MOLA language for generation of MOLA programs.

In this section we want to elaborate the discussion on the principles of template-based languages for generation (both textual and graphical) and the way these principles influence the constructs chosen for Template MOLA. Textual template based languages served as a rational for introducing some more advanced constructs in Template MOLA.

We will briefly analyze the principles of those languages where the generation source is a model. These are the popular textual template languages *mof2text* [123], *MOFScript* [176], *Acceleo* [164], *Xpand* [181], *TCS* [177], a.o. The only specifically template oriented graphical generation language seems to be Template MOLA, but similar issues could appear also in languages using a concrete graphical syntax for transformation definition (*ATOM*^{3} [96], a.o.).

Template-based languages (textual or graphical) for program generation from a model consist of two parts – the model navigation part and the generation part. The generation part specifies the object which has to be created. In fact, only the generation part is fully based on the template mechanism corresponding to the given concrete syntax (textual or graphical). The navigation part is based on the control structures for traversing the source model in the order required by the generation algorithm to be implemented (the so-called visitor principle). The basic control structures always are sequence, alternative, some form of loop (iteration) and invocation of a “procedure” (or something similar). However, this basic set is not always sufficient.

Another part of languages is facilities for data extraction (query) from the model. The extracted data typically are held in some temporary data structures, including various collections. They are used for a direct substitution of the relevant variable parts of templates (variable expressions etc.) and for organizing additional generation loops. This query part may be more or less incorporated into the model navigation mechanisms or may be a more independent sublanguage. For most of the considered textual languages, the query mechanism is an independent one based on OCL or a similar language. This query mechanism as a rule supports recursion, thus, transitive closure-type queries (such as all inherited attributes of a class) can also be specified.

An essential property of textual template languages is the fact that an appropriate loop construct can surround any part of a textual template. This permits to create in a simple natural way any nested iterative structure as a result.

Another feature of textual languages is the concrete syntax for coding a reference defined in the corresponding metamodel (from a variable usage to its declaration, from a procedure call to its definition, etc.). In a textual language such a reference as a rule is coded by using a sort of a name of the referenced object (certainly, it must be unique in the given namespace). Such a reference name can be easily generated from the model by using a navigation mechanism (or query in more complicated cases). But in any case the reference can be created "in-place" from the generation algorithm point of view (no return to it is required later).

These two features determine that in most cases the above mentioned control structures are sufficient for defining the generation algorithm. The transformation algorithms are basically "single-pass" (with various distant data lookups implemented by queries). Certainly, it is true if the source metamodel contains a fragment which in a sense is isomorphic to the target object to be generated. Since textual template languages in practice are not supposed to implement arbitrary model transformations but only perform the final step of a transformation chain, this is virtually always the case.

However, for the 2D world of graphical languages the situation is not so simple even in the standard case when the source and target structures are isomorphic. First of all, it is not always so easy to enclose any part of a graphical template in a generation loop. In Template MOLA the main "regular" cases are well supported from this point of view. These include a sequence of rules or loops to be created by a generation loop. Then a template for such rule or loop is contained in the generation loop body. This case was illustrated in the example in Fig. 70 (p.189). The only issue there is the convention how flows should link the results of iteration steps. This case covers a significant part of the usage of generator loops in Template MOLA.

However, there may be other "iterative" situations, too. The first one is a number of assignments per class element dependent on some repeating element in the source model (e.g., see Fig. 79, p.205). A similar situation in textual templates creates no problems at all. But in Template MOLA it would be quite awkward to define a generation loop within a template class element.

The other big difference is referencing. The name-based referencing is used in graphical languages as well, but mainly for proper “distant” referencing – such as a procedure call to its graphical definition. However, frequently graphical edges represent a “local” reference in the concrete syntax. One such situation has already been presented. A control flow in the MOLA procedure generated by the Template MOLA example in Fig. 70 (p.189) must go from the loop generated in the previous iteration of the generation loop to the rule generated in the current iteration. This fact cannot be easily visualized in Template MOLA; it is an assumption in the generation semantics.

A similar situation can occur also with edges representing association links in a rule. There may be a necessity to create a variable number of class elements in a rule all linked by association links forming a chain (see the example in Fig. 81, p.207). It would be natural to assume that each class element is generated by one iteration of the corresponding generation loop. The corresponding association link must go from a class element generated in one iteration to the one generated in the next. No implicit assumption can be made for association links since they represent specific associations. A direct graphical notation in Template MOLA for association links, connecting two iterations of a loop, would also look quite strange.

The described situations (and other similar ones) with the necessity to relate several graphical template elements appearing in adjacent iterations of a generation loop require some generic and visually easy readable solution. The merge construct is proposed for this purpose. This construct is defined not only with Template MOLA in mind, but also other graphical template language applications.

Another issue worth mentioning relates to the generation part of Template MOLA – in fact, the normal MOLA language – which has no specific model query sublanguage. Queries are implemented by means of the standard pattern mechanism in rules. Therefore recursive queries (of the transitive closure type) require explicit recursive calls of MOLA procedures. This enforces the requirement that the merge principle should be applicable not only to generation loops, but also to recursive calls in the generation time.

6.7 Merge Mechanisms

One of the use cases where Template MOLA could be applied is transformations for generic metamodels. We may consider one simple transformation of such type –

instance cloning (instance cloning is discussed in detail in Section 7.4.1). In order to clone an instance we should create another instance and copy the values of all attributes. Fig. 78 demonstrates a transformation cloning values of all attributes of a class in Template MOLA. Functionally, this template MOLA procedure performs the required task. However, the generated code is a spaghetti code (see the right side of Fig. 78). More precisely, in a "normal" MOLA all attribute assignments should be placed in the same class element (and not a new class element generated for each one).

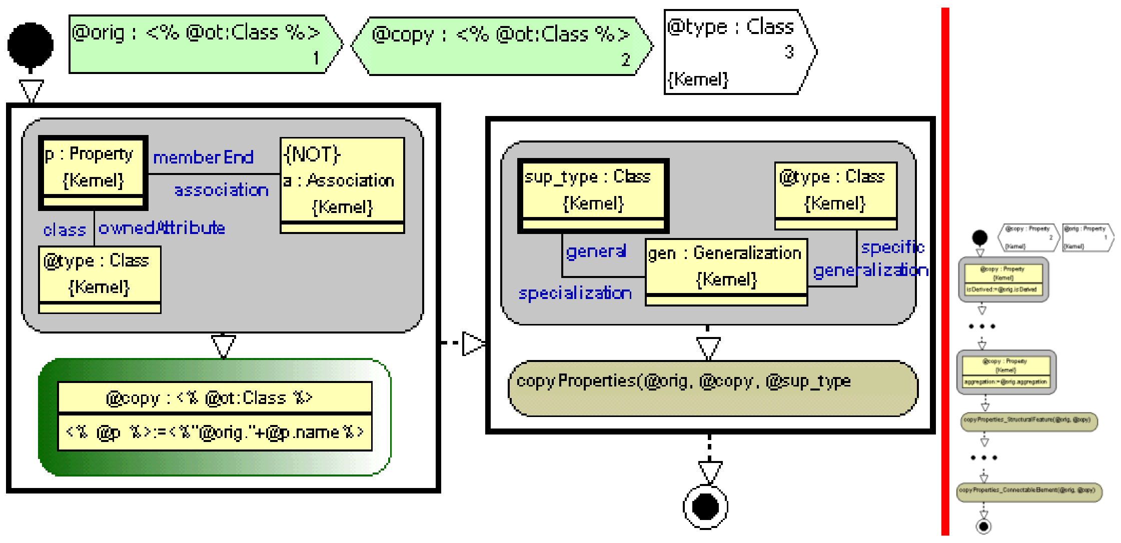


Fig. 78. The left side demonstrates the procedure for copying the property values of a class instance. On the right side there is an example of the generated transformation.

To solve this problem we introduce a merge mechanism in Template MOLA which is introduced in a generic way so that it could be applied to synthesis of code in any graphical language.

6.7.1 Merge Example

The general principle is very simple. We introduce the merge expression for all template elements. Elements are merged if the value of the merge expression is equal to the merge expression of a previously generated element (of the same kind). For elements already containing a unique identifier in a container, it is possible to use this identifier in a role of the merge expression. For template MOLA it means that the merge expression is required for the template rule and template loop. For class elements the class element name is reused in the role of the merge expression.

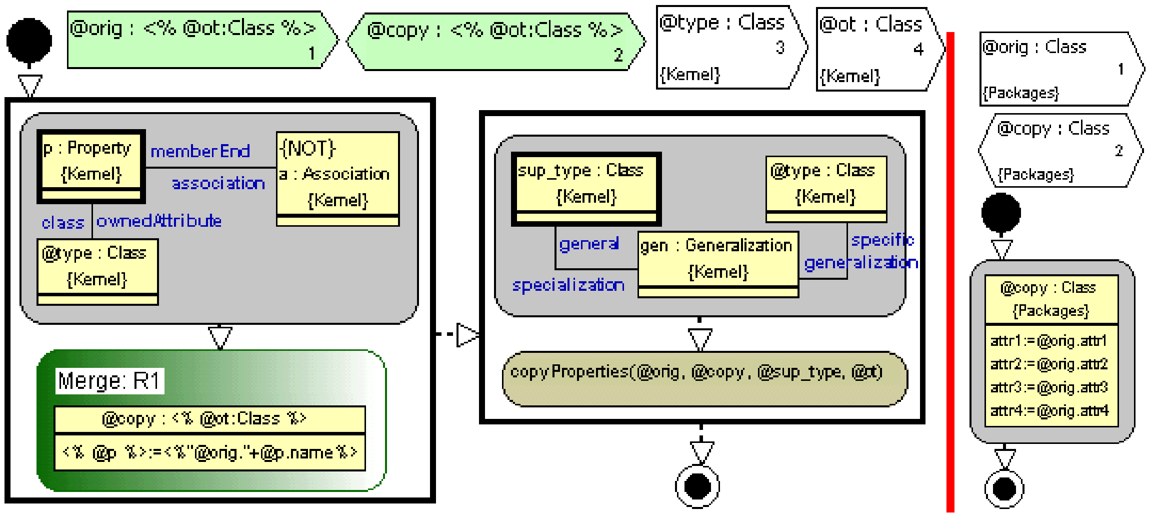
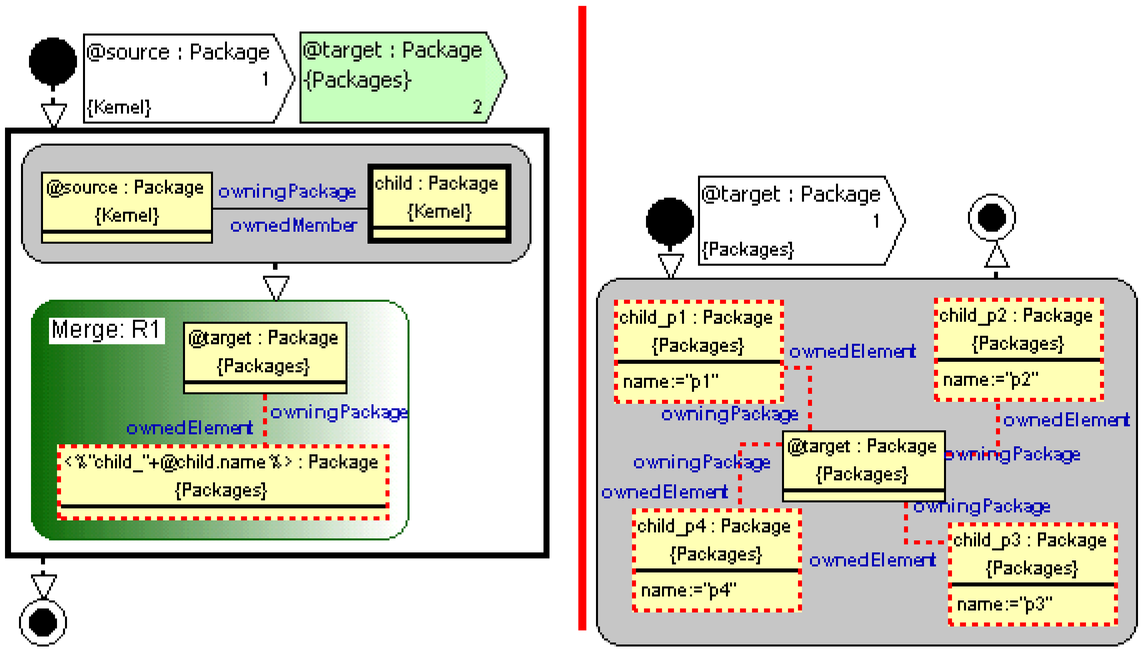


Fig. 79. The left side demonstrates the procedure for copying the property values of a class instance with a merge. On the right side there is an example of the generated transformation.

Now we can take a look at the previous example with the merge mechanism enabled. The left side of Fig. 79 demonstrates the transformation from Fig. 78 with the merge annotations. In this case the rules created in the first loop are merged, since their merge expressions all are equal. Each rule contains one class element and their class element name (which is used in the role of merge expression) is equal. Therefore the class elements are merged – all attribute assignments are placed in the same compartment. Consequently, we generate the transformation on the right side of Fig. 79. This is evident in the case when all instances of rule *R1* are generated in the same loop. However, since the template definition contains a recursive call to the same procedure, it is possible that instances are generated by a loop in another procedure instance. To ensure that instances generated in all invocations are merged together, the procedure call should be marked with *inline* annotation. It means that all elements generated by this call will be included in this procedure. Since in the generated code all elements are included in the same procedure, we can use merge mechanisms also for them. In this case the rules generated by a recursive call will have the same merge expression. It means we can merge them with the existing rules. Class elements will also be of the same type and with the same name (used as the merge expression). (Actually, this procedure has one more generation parameter when compared to the procedure in Fig. 78. This parameter is introduced to enable the element merge so that they will have the same type in all recursive calls instead of cast to a super class in recursive calls in Fig. 78.) It means we can merge also the class elements. As a result all assignments will be placed in one class element. We should also consider the assignment merge, as there can be inheritance diamonds and in such a case one attribute will have multiple assignments. Assignments are merged by the attribute, keeping the first assignment, the others are ignored.

6.7.2 Rule Merge

There are also other cases when the merge construct is useful. For example, it is the case when the set of class elements in a rule should vary depending on some condition. Such a case can occur when we have to iterate through some data and create a class element for each instance. We consider a case when we want to obtain a star-shaped rule of class elements. Fig. 80 demonstrates the way of obtaining such a rule by using the merge mechanism. We merge the rules and the class element at the centre of the star. Using a generation loop we can create as many peripheral nodes as needed in our star shaped rule. The basic semantics of the merge operation determines that all generated association links go to this merged centre node.



@source : Package 1 {Kernel}
@target : Package {Packages} 2
@source : Package {Kernel}
owningPackage
ownedMember
child : Package {Kernel}
Merge: R1
@target : Package {Packages}
ownedElement
owningPackage
<%"child_"+@child.name%> : Package {Packages}
@target : Package 1 {Packages}
child_p1 : Package {Packages} name:="p1"
child_p2 : Package {Packages} name:="p2"
child_p3 : Package {Packages} name:="p3"
child_p4 : Package {Packages} name:="p4"
@target : Package {Packages}
ownedElement
owningPackage

Fig. 80. Creation of a star shaped rule by using merge mechanisms

Similarly to the star structure described above we can also obtain an element chain in a rule. A chain example is given in Fig. 81. Combining the chain and star mechanisms we can obtain any rule structure by using merge mechanisms.

The question may arise: “Why should we create rules partially?” It is because not all elements that should be used in a rule can be created at the same time. There are no other ways to add new elements to an existing rule. Of course, we may try to split the rules in smaller ones. However, in this case we will end up with a spaghetti code. It can also affect the efficiency of the generated code. This is because each rule is matched at once in MOLA while splitting it may cause some elements to be matched repeatedly and spoil the pattern matching optimization.

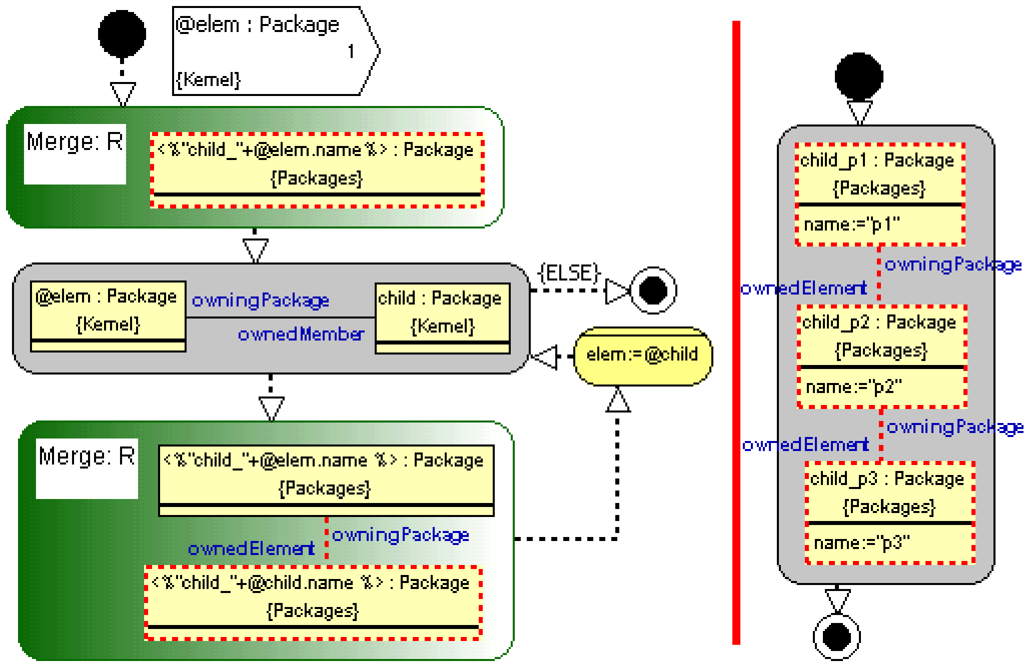


Fig. 81. Creation of a chain shaped rule by using merge mechanisms

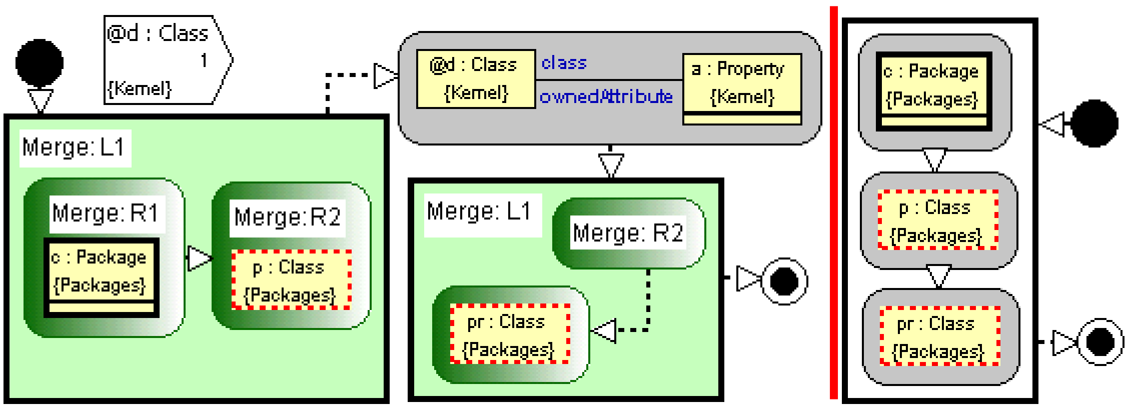


Fig. 82. Merge of loops and rules obtaining different control structures

The same mechanism, demonstrated for the rule merge to obtain different patterns, could be reused for obtaining different control structures between the loops and the rules. For this purpose it is possible to repeat an empty rule or loop only with the merge name defined. In this case it will be used to define the outgoing point of the control flow. Fig. 82 demonstrates how such construct works. In this case the default control flow semantics defined in Section 6.5.3 is redefined.

6.7.3 Merge Semantics

A brief description of the merge semantics is given in this subsection. Two rules are merged if the merge expression of a rule to be created is the same as the merge expression of an existing rule in this procedure. Rule merge means that class elements and association links to be created in the new rule will be included in the rule it is merged with according to the merge semantics of elements and links. Semantics of loop merge is similar, only instead of class elements the loop elements (rules, call statements, etc.) are treated the same way. Loop elements are created in the merged loop according to their merge semantics.

Class elements are merged if the name of the new class element is equal to the name of some existing class element in this rule. When merging class elements their types are also checked. If their types are different a merge process error message is generated and the creation of this class element and its association links is skipped. Assignments are the most important part of class element from the merge perspective. New assignments are added to the relevant existing element. If the element already contains an assignment to this attribute, the new assignment is ignored and a warning is produced. If the merged class element has a condition while the original one does not, the condition is added. If the original element already has a condition, then the condition in the merged element is ignored and a warning is produced. The same principle is applied to other features of class element. When defining a merge of class elements users should take care to avoid generation errors.

Concerning association links, it is possible only to add new association links by using the merge mechanism. If there is no link between these two class elements in the rule, then a new link is added. Otherwise the link is ignored. Association link properties are not merged.

Flow merge in a sense is similar to the association link merge. Always new flows are added. However, it is not checked whether such flow already exists.

6.8 Implementation

To implement Template MOLA, we have to consider two aspects – editing and processing of Template MOLA.

The Template MOLA editor was built as a part of the Master Thesis of Janis Iraids [60] and it has been built in a METAclipse framework using the MOLA editor as a basis. Model transformations, implementing the traditional MOLA language within a METAclipse framework, have been extended to support the desired functionality in the new editor. Since Template MOLA reuses the syntax from the traditional MOLA language, many of the MOLA procedures implementing the editing actions can be reused. The template elements can be regarded as subclasses of their related "regular" elements, thus inheriting all their required editing behaviour. A template text statement, for example, is almost equivalent to the traditional text statement from the editor's point of view. New and unique functionality can be easily included where appropriate. So even though a substantial number of new diagram elements have been introduced, the volume of the code has not grown proportionally, but much less than that. In addition, the sub-classing approach eliminates any need for non-trivial migration when converting pure MOLA transformation models to the Template MOLA transformation models.

Another aspect is the execution of Template MOLA. Several solutions were considered, including an interpreter and a Template MOLA pre-processor.

The author of the present Thesis proposes to use the pre-processor that converts Template MOLA to traditional MOLA with a later reuse of the MOLA compiler to obtain transformations for generation. This approach is similar to pre-processing of macros in C++ environments. The pre-processor replaces the Template MOLA statements with traditional MOLA rules that create corresponding instances of MOLA statements. For example, the template rule in Fig. 67 (p.182) is replaced with the MOLA rule in Fig. 69 (p.183). The newly-created MOLA transformation is compiled by using the compiler of the traditional MOLA language. Finally, the obtained transformation is used as a HOT. An experimental implementation of a pre-processor was built. The experiments confirmed that it is possible to build a pre-processor. The most complicated part was work with multiple meta-levels at the same time.

In the Master Thesis of Janis Iraids [60] the Template MOLA interpreter was considered. To create the Template MOLA interpreter, a MOLA interpreter is required.

The creation of a MOLA interpreter is the most time consuming task. Extension of a MOLA interpreter to the Template MOLA interpreter is not very labour intensive. In the MOLA interpreter the most important and also the most complicated part is the implementation of pattern matching. Currently there is only a compiler available for MOLA. The MOLA interpreter would be valuable per se, as by using an interpreter it could be possible to debug the MOLA programs.

Evaluation has revealed that the implementation of the pre-processor solution requires less effort. However, the interpreter solution is also feasible and it has other advantages.

Another issue to be considered is the readability of the MOLA sources, generated by using Template MOLA. The easiest solution is to create transformations, using only the abstract syntax of MOLA. The abstract syntax is sufficient if we want to execute these transformations without a manual extension. However, to obtain a concrete graphical syntax for the generated transformations, an abstract-to-concrete syntax transformation and an automatic diagram layout generator must be used. Some experiments have been performed in the field practice by Edgars Didrihsons, confirming that it is technically feasible to automatically create a usable concrete syntax of the generated MOLA transformations.

Note that the transformations in Template MOLA actually contain some layout information for the MOLA procedures to be generated. For example, the layout of elements in a template rule could be reused in the generated transformation. However, this issue requires further research.

6.9 Conclusions

A new graphical template-based language Template MOLA for the MOLA transformation synthesis is proposed in this section. This language leverages the advantage of template-based model-to-text languages (easy specification of the language elements to be generated) to graphical languages. These are the graphical template statements of Template MOLA – template rules and template loops that are transferred to the new transformation to be generated. Certainly, they can contain variable elements – template expressions to be replaced in the generation process which itself depends on the input model and is defined by means of the generation statements – ordinary MOLA statements included in Template MOLA. These generation statements are executed in a standard way during the generation process.

The merge mechanism for templates is proposed, enabling the possibilities to define the generation of nested graphical structures in a simple way. Even the generation of large text compartments in graphical elements (such as an attribute compartment in a class symbol) requires this mechanism in a general case. Still this mechanism has a much wider application – a graphical element has to be extended by several steps of the generation process everywhere.

It is described that it is much easier to specify a transformation synthesis task in Template MOLA than to specify the same task in the traditional HOT style (using MOLA as a HOT).

Implementation of Template MOLA is under development. The editor has already been built. For the execution of Template MOLA an interpreter is selected due to its positive side effects (e.g., the MOLA interpreter). Implementation of the interpreter is under development.

Template MOLA applications are discussed in CHAPTER 7. These applications were used to validate the applicability of Template MOLA language. The experimental usage confirmed that Template MOLA is suitable for the definition of synthesis transformations.

CHAPTER 7

Template MOLA Applications

The chapter dedicated to the discussion of Template MOLA applications focuses on the two main application areas: the mapping language compilation and the development of transformation libraries.

7.1 Mapping Language Compilation Using HOTs

In addition to the mapping language definition facilities an interpreter or a compiler is required for mapping languages. As stated in CHAPTER 4 domain-specific mapping languages could be incomplete, therefore integration with transformation languages is needed. One of the ways for achieving the integration is compilation of mapping languages to transformation languages. In this case it could be possible to extend the code generated by mapping in the transformation language.

Higher-order transformations (of synthesis type) could be used to compile mapping languages to transformation languages. Such approach was also used in AMW [39] proposing to compile mapping languages using ATL [63]. As a result it is not surprising that most of HOTs have been implemented in ATL [183], although it is possible to define HOT in any transformation language.

Thus, defining HOTs can also be done in the model transformation language MOLA, although the HOT definition using the abstract syntax of MOLA is not very suitable. The Template MOLA language defined in CHAPTER 6 is more appropriate for this task, as it was shown in Section 6.2.

Similarly, instead of standard ATL for transformation synthesis it is proposed to use ATL extension [182], by means of which the lines of code in ATL synthesis transformation could be reduced by 43.81% [182].

These specialised languages, like Template MOLA and ATL extension, are the best choices for the development of mapping language compilers. In a mapping language compiler to model transformation the mapping model should be analysed and transformations should be synthesised. We have selected to use Template MOLA for the mapping language compilation as in both studied mapping domains integration with MOLA transformations is required. Ideas for the development of mapping language compilers are described in the following sections.

7.2 Implementation of Mapping Languages for MDSD

As it was described in Section 4.6 it is planned to implement MALA4MDSD and the mapping language family by using higher-order transformations for the development of both – the editor and the compiler and each of them will contain the static part common to all languages in the family and the specific part. The latter will be generated by analyzing the language definition.

7.2.1 Editor of the Mapping Language Family

The mapping language definition will be used as an input for this higher-order transformation, generating the editor of a language. The definition will be analysed to find the graphical primitives in a language, as well as palette elements. This will be concluded from the tree type elements and their concrete syntax definition, as well as used for deciding for which elements the recursive elements could be used.

The possible child elements of the tree node will also be concluded from the tree type definition. It will be used to generate transformations for checking whether one element can be used as a child of another element.

Processing of the mapping part of the language is metamodel independent and predefined transformations will be used there. The only thing to be checked from the language definition will be whether the modifiers “*copy*” and “*copyAttributes*” are supported (whether the tree types are the same).

When creating a new mapping diagram in a mapping language, the root nodes are always included in the diagram. For each node in the diagram there are context menu points for creating child elements of the appropriate type. The list of context menu points depends on the tree type.

7.2.2 Mapping Language Family Compilation Schema

Another issue relates to the compiler development of mapping languages. Like the editor, part of the compiler could be developed in a generic way for the entire mapping language family. In addition to the specified mappings the mapping language definition will be used by these parts of compiler.

The mapping language compiler should transform the mapping defined in terms of tree types to a transformation defined in terms of the domain metamodel. The tree type definition could be used for the purpose. This definition states the way each tree type element is represented in terms of the domain metamodel. This information is widely used in the mapping language compiler.

As already stated above, the compiler defined in Template MOLA is used for the compilation of the mapping language family. The metamodels described in Section 4.4 are used as the source metamodel of Template MOLA transformation. There is used the metamodel describing the mapping language definition, as well as the mapping metamodel. Metamodels corresponding to the source and the target trees should be used as the domain metamodel.

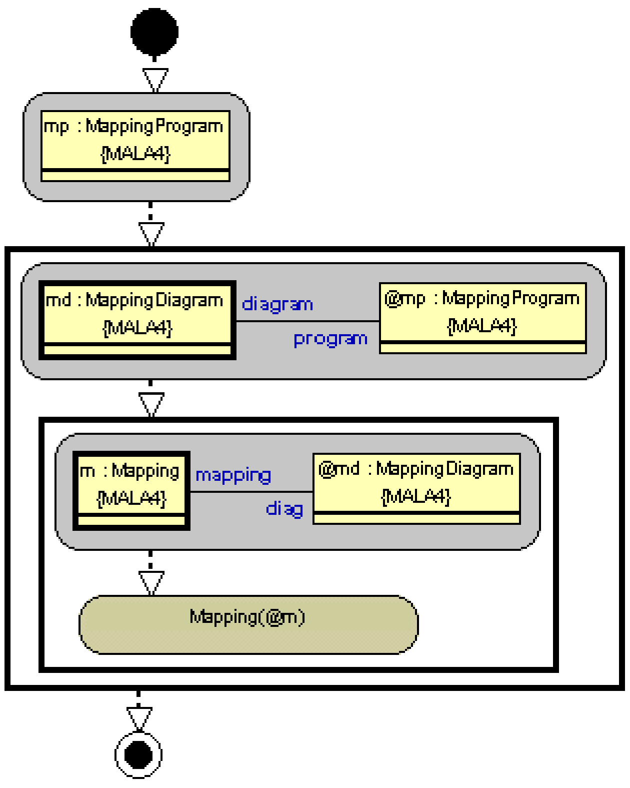

mp : MappingProgram {MALA4}
md : MappingDiagram {MALA4}
diagram
program
@mp : MappingProgram {MALA4}
m : Mapping {MALA4}
mapping
diag
@md : MappingDiagram {MALA4}
Mapping(@m)

Fig. 83. Compilation of mapping language family

There are ordered mapping diagrams in the mapping program. Diagrams should be executed according to this ordering. Each mapping diagram consists of multiple mappings that are ordered in the diagram. Mappings are executed in a top-down manner, if the ordering is not specified explicitly. When executing the mapping, all instances, corresponding to the constraints defined for the source of the mapping, are processed. It means we can process each mapping separately according to the mapping ordering. Transformation defining the mapping execution order is given in Fig. 83. It should be noted that some mappings are not defined explicitly; we assume that these mappings have already been inserted in the pre-processing step.

Compilation of a mapping is discussed in the next section with some mapping compilation aspects dwelt on in detail in the other following sections.

7.2.3 Mapping Compilation

The main ideas on mapping compilation are presented in this section. A transformation implementing the application of the mapping is created from each mapping. This transformation is generated by using higher-order transformations. We define the transformation generation algorithm in Template MOLA.

The first thing the generated transformation should do is to select an instance set for the transformation to be applied. If we think in terms of tree instances, then instances of the source node of the current mapping should be transformed. Besides, these instances should satisfy constraints defined for this node and should have as ancestors instances satisfying constraints defined for the ancestor nodes. We can treat the tree as a pattern, describing an appropriate instance set. In this pattern all nodes (and their constraints) between the root and the source node of the current mapping should be included.

As a transformation should be defined in terms of model, not tree, it means that the tree pattern should be translated in a MOLA rule defined in terms of the source metamodel elements (metamodel corresponding to the source tree). It is possible to translate a pattern defined in terms of tree in a pattern defined in terms of model by using the tree type definition. This issue is discussed in detail in Section 7.2.4.

Our mapping language has the semantics “create if does not exist”. Next we should create a rule checking whether this instance has not been processed before. In creation of this rule traceability information is used. Special attention should be paid here to mappings with the *check* modifier. This issue is discussed in detail in Section 7.2.5.

If no instance is found, then an appropriate instance in the target should be created. To create an instance in the target, it is necessary to find the appropriate parent instance. The parent node of the current mappings target node should have a mapping to some already processed source tree node. Besides, this source tree node should be in the tree between the source node of the current mapping and the root node. An instance of the source tree node could be located by using a pattern similar to the one used for the selection of instance set (or even using reference to the already found instance in this pattern). To locate the appropriate target instance again traceability could be used. The parent finding is discussed in detail in Section 7.2.6.

Finally, it is possible to implement creation of the target tree node instance. Here a rule is created by translating the target tree node creation in terms of model element creation; traceability creation should be added as well. This issue is discussed in detail in Section 7.2.7.

The last thing is processing of the *copy* or *copyAttributes* modifiers if they are used. To solve these tasks a universal instance copy library is created. In mapping compilation only a call to the library is added, if required. The library uses the tree type definition to create appropriate transformations for the tree node types.

In the following sections details regarding mapping compilation are discussed. A description is given on what should be generated in each compilation step to result in the MOLA procedure. For some steps the generation algorithm description in Template MOLA is given as well. We focus on the algorithm supporting typical cases; the other issues are only slightly touched upon.

7.2.4 Source Tree Pattern Compilation to MOLA

In this section we consider the creation of transformation that selects an appropriate instance set for the mapping application. As already stated above, this instance set should satisfy conditions defined by a tree fragment from the root node to the source node of the current mapping. The tree fragment should be translated in the MOLA program defined in terms of the source metamodel (a metamodel corresponding to the source tree) elements.

We are interested in all distinct instances of the source node of the current mapping. To process the instance set we should create foreach loop in the generated code, where a class element corresponding to the source node of the current mapping is used as a loop variable.

We assume here that there are no recursive tree nodes in the source tree, therefore it is possible to transform the whole pattern defined by the source tree in a loophead rule. Each tree node type is replaced with a class element. The type of the class element should be the domain class associated with the tree node type in the tree type definition. If the domain class in the tree node type definition is restricted by using the OCL constraint, then this OCL constraint is added to the appropriate class element.

Parent-child relations in the tree should be replaced with appropriate association links in the generated loophead rule. If classes corresponding to the parent and the child nodes are directly related by using the parent-child association, then an association link is simply added. If a longer OCL path is used, then intermediate class elements are added as well.

If expressions are used for some tree nodes, then these expressions are translated in terms of metamodel and added to appropriate class elements. It is required to translate these constraints as they were defined in terms of tree elements.

A simplified version of Template MOLA procedure processing mappings is given in Fig. 84. This procedure processes the current mapping that is received as a parameter. Here the source tree node of the current mapping is found by using the MOLA rule.

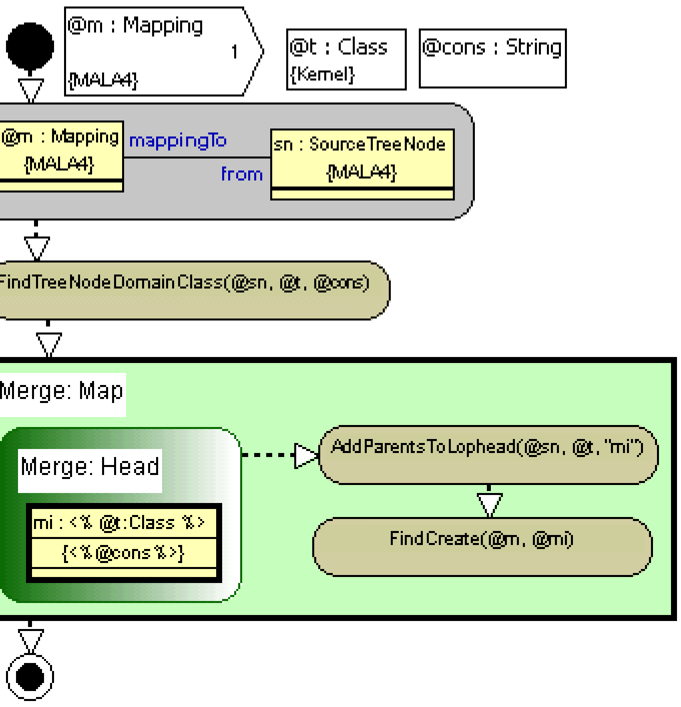


Fig. 84. Template MOLA procedure processing the current mapping

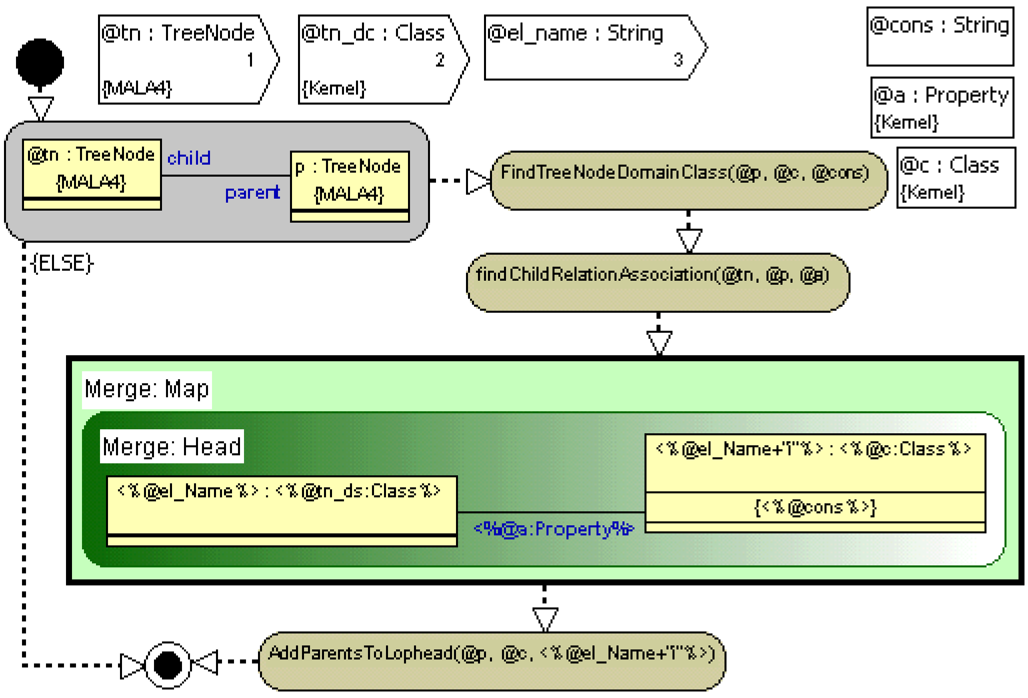
The Template MOLA procedure *FindTreeNodeDomainClass* is used to find the domain class corresponding to this tree node by using the tree type definition. This procedure has one input parameter – the source tree node – and two output parameters: the domain class corresponding to the tree node and constraints. The values of the output parameters are stored in two variables. The domain class is used as a type of the loop variable in the template loop. Constraints are used in the class element corresponding to this tree node. Constraints used in the node type definition as well as constraints used in the tree node, defined in terms of domain metamodel, are included in the returned constraint string. In the given procedure only the local constraints are supported. To support more complicated constraints, adding of additional elements to the loophead rule is required.

The next element in the procedure is the template loop. It generates a loop processing all appropriate instances of the source node of the current mapping. The generated loop will iterate through all instances of the source tree node. It means the type of the loop variable will be the domain class corresponding to the source node (the class found by using the procedure *FindTreeNodeDomainClass*). If required, then constraints are added to the loop variable as well. They are found by using the procedure *FindTreeNodeDomainClass.*

Other tree pattern elements are added to the loophead rule by using *inline* call to the procedure *AddParentsToLoophead* (given in Fig. 85). This procedure adds elements one by one to the loophead recursively processing the tree upwards. Elements to the loophead are added by using the merge mechanism. Therefore, the merge expressions for the template loop and the loophead are defined in Fig. 84. When the loophead rule defining the instance set has been created, the procedure implementing the semantics "find if does not exist" is called in the template loop in Fig. 84. It completes the processing of the current mapping.

The procedure *AddParentsToLoophead* (given in Fig. 85) is used to add the other tree elements to the loophead rule. We remind that here we still assume that there are no recursive nodes in the source tree. This procedure is recursive, it processes the parent of the current tree node and calls itself on the parent of this tree node. When the root node is reached, nothing is done.

In the first rule the parent of a tree node is found. If there is no parent (the root has been reached), the procedure completes its work. If the parent is found, the domain class and constraints corresponding to this parent are found by using the procedure *FindTreeNodeDomainClass.* After that the association relating the parent and the child tree nodes in the domain metamodel is found by using the procedure *findChildRelationAssociation*. It should be noted that only navigation expressions of length one (direct associations) are supported in the Template MOLA procedure given in Fig. 85, however, it could be easily extended to support more complicated navigation paths. In this case intermediate class elements (nodes in the path from parent to child) and multiple associations should be added to the loophead rule.



@tn : TreeNode 1 {MALA4}
@tn_dc : Class 2 {Kernel}
@el_name : String 3
@cons : String
@a : Property {Kernel}
@c : Class {Kernel}
@tn : TreeNode {MALA4}
child
parent
p : TreeNode {MALA4}
FindTreeNodeDomainClass(@p, @c, @cons)
findChildRelationAssociation(@tn, @p, @a)
{ELSE}
Merge: Map
Merge: Head
<%@el_Name%> : <%@tn_ds:Class%>
<%@el_Name+'1'%> : <%@c:Class%>
{<%@cons%>}
<%@a:Property%>
AddParentsToLophead(@p, @c, <%@el_Name+'1'%>)

Fig. 85. Procedure *AddParentsToLoophead* recursively creates the loophead rule

Finally, the parent element is added to the loophead of the template loop. This is done by using the merge mechanism. This procedure was called using the *inline* mode and the merge expression of the template loop and the loophead rule are equal to the merge expressions used in the procedure processing mappings (Fig. 84). As a result all elements appearing in the loophead in this procedure will actually appear in the loophead iterating through the source tree node instances (Fig. 84).

It should be reminded that there is also a merge of class elements where the element name is used as a merge expression. When executing the loophead given in Fig. 85, actually only one class element is added to the merged loophead as the class element corresponding to the child node has already been added previously. The element added to the loophead is connected by using the association link to the child element previously created in the loophead. This link corresponds to the parent-child relation in the domain metamodel. The association implementing the parent-child relation was found by using the procedure *findChildRelationAssociation*.

To create the loophead from the source tree pattern a merge was very appropriate as here the rule has to be created recursively.

To support recursive nodes the loophead pattern is split in several patterns and recursive calls are used. Constraints to the supported instance set are added gradually. Consequently, the Template MOLA program becomes quite complicated.

7.2.5 Implementation of "*Create if Does not Exist*"

In MALA4MDSD the semantics "create if does not exist" is used. The instance is created if it has not been created previously by the mapping with the same name. To support this feature it is required to generate a simple rule with three class elements: reference to the processed instance (a loop variable in the previous section), a class element with the type traceability class and the domain representation of the target node. For the traceability class element constraint is added checking whether the trace name is equal to the mapping name. The domain representation of the target node is obtained similarly to the way the domain representation of the source node has been obtained.

Control flows are generated from this rule. If the rule fails, then the transformation should go on with instance creation, however, prior to that the parent instance in the target model should be found. If the rule succeeds, the mapping execution should be completed. A special issue are mappings with the "check" modifier. If they fail, error is produced.

7.2.6 Finding of Parent Instance in the Target Tree

To create an instance in the target model, it is required to find the appropriate target instance to which the newly created instance should be attached.

It is done by finding a mapping from the parent node in the target model and by finding the appropriate instances of this mapping. In the MOLA pattern generated in the loophead (see Section 7.2.4) the source instance of this mapping should already be found. The source instance and traceability links are used to find the appropriate target instance. It should succeed as this mapping should already be processed according to our ordering of mappings.

7.2.7 Element Creation

Finally we are able to generate a transformation for the creation of target instance. A simplified version of this transformation is given in Fig. 86.

This procedure has two template parameters: one of them contains reference to the source node instance being processed and the other – reference to the instance to be used as the parent in the target model. At first the appropriate association is found relating the child to the parent. (Here again only simple relations are supported, similarly to the loophead creation in Section 7.2.4.)

Then the MOLA rule is created. It has four elements: reference to the source instance, traceability instance creation, target instance creation and reference to the parent instance in the target model. We assume that traceability is implemented by using the constant class *Trace* in the Template MOLA rule given in Fig. 86.

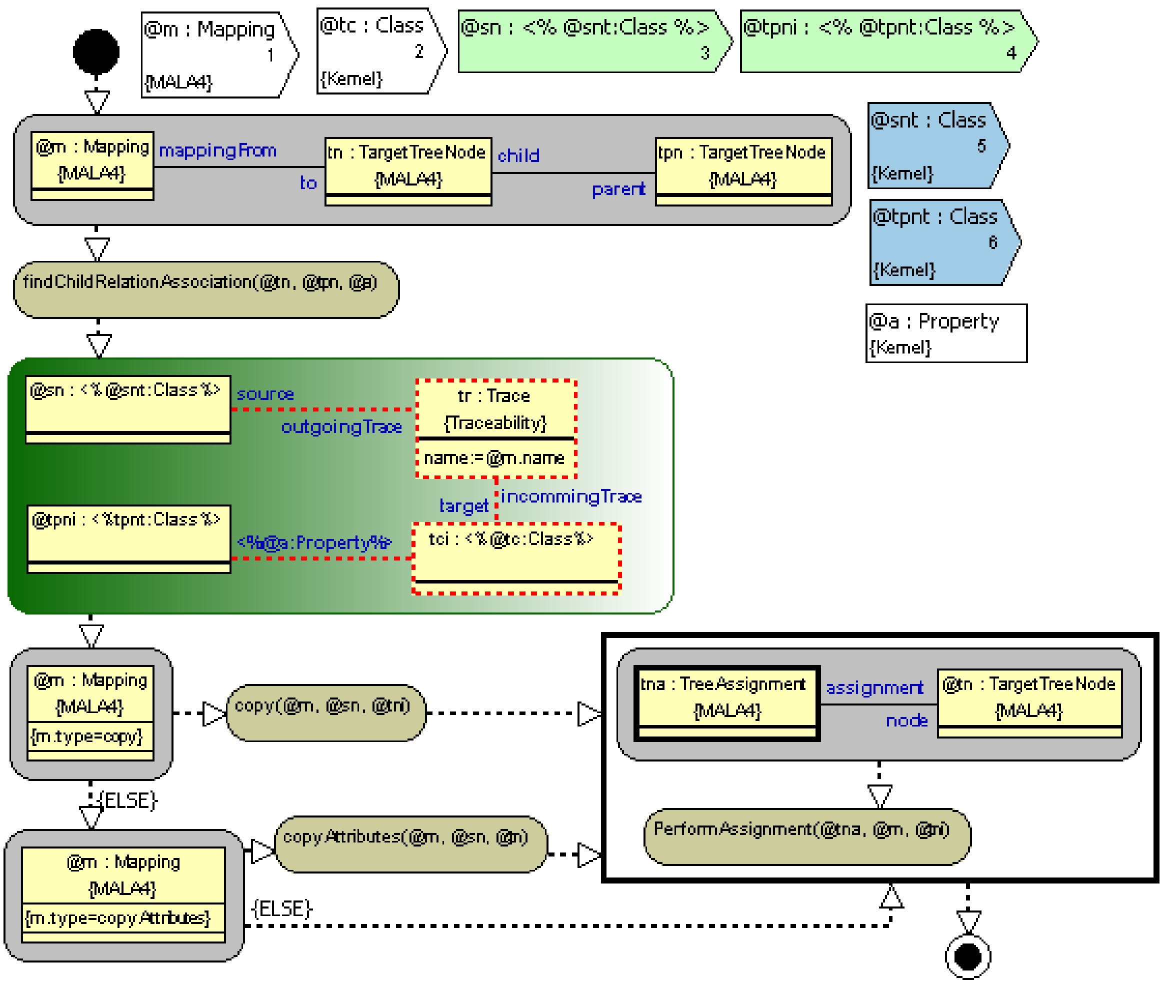


Fig. 86. Template MOLA procedure implementing the element creation

If for mapping the modifier *copy* or *copyAttributes* is used, then a call to the copy library is created, respectively copying all child elements or only the attribute values. The copy library supports copying of the tree node instances. It is implemented in a generic way, however, for each node type the appropriate copy transformation is generated by using the tree type definition. It is similar to the copy library discussed in Section 7.4.1. If only the attribute values should be processed, the procedure *copyAttributes* could be called directly.

The explicitly defined assignments are performed after the *copy* operations to replace the default values set by the *copy*. Here the assignment defined in terms of tree nodes is translated into the assignment defined in terms of metamodel elements. Each assignment is processed separately and it is done by using the Template MOLA procedure *PerformAssignemnt*.

7.2.8 Evaluation

Only the main ideas used in the compilation algorithm have been presented here. It is described what should be generated in each compilation step. The Template MOLA procedures implementing the creation of the loophead and the element creation are given as well. The use of merge is demonstrated in the loophead creation; the merge mechanism is required here as the loophead creation algorithm is recursive. On the other hand the creation of the element is very simple and is defined by using one template rule.

In the described solution, many details and exceptional cases were skipped; however, a complete compiler requires support also for these cases. Full implementation of the compiler is left for the future. Nonetheless, the experiments have confirmed that the proposed approach is technically feasible and that Template MOLA is appropriate for this task.

The overall conclusion is that Template MOLA seems appropriate for the development of a mapping language compiler. The only inconvenient issue concerns the limited OCL expression support in MOLA. It requires performing a complicated transformation of the OCL expressions to the MOLA patterns. There are two possible solutions: one is to restrict the supported OCL subset used in the mapping language (and its definition) to the subset used in MOLA; the other is to extend the MOLA constraint language with a complete coverage of OCL features.

7.3 Implementation of Mapping Language for DSL Tool Building

In this section a simplified example of tool building is presented. It is a sort of continuation of the topics discussed in Section 5.3.

As stated in Section 5.3, there are approaches combining mappings and transformations. In this case mappings are used to generate transformations. The transformation synthesis required there provides a perfect opportunity for application of Template MOLA.

We use a specific task from the tool building field as an example in this section. We assume that we have instances of some graphical DSL in the abstract syntax (a domain model), and we want to generate the corresponding visualisation (instances of the presentation metamodel). We can certainly write manually a MOLA transformation, solving the task for this concrete DSL.

In our tool building environment we have means for the domain metamodel definition, as well as for the mapping and the presentation type definition; therefore, visualisation transformation for each DSL can be created in a generic way. It means we can build a generic transformation in Template MOLA from which the transformation for visualisation creation in a concrete DSL can be generated automatically. It should be noted that here only one tool building aspect is considered. In the complete mapping language compiler the other aspects, discussed in CHAPTER 5, e.g. property dialogs, palette elements, element update, etc., should be supported as well.

To write the transformation, we need the corresponding metamodels (built according to the general schema in Fig. 77, p.196). A simplified metamodel version is used in this example. The domain metamodel is defined using a small subset of UML (see the upper left side of Fig. 87). Presentation types and a mapping metamodel are also needed. Instances of this metamodel are used as the input in the generation time. Here we present a very simple integrated mapping and presentation type metamodel where minimal information on the intended graphical form is included directly in the mapping definition (see Fig. 87, the upper right side). Instances of a domain class can be visualised as a box (*ClassToBox*) or as a line (*ClassToLine*). If the class is visualised as a box it may contain several text fields in which the values of some class properties are usually displayed (*PropertyToField*). The user syntax of this simple mapping language could be built in a way similar to the property mapping language, discussed in Section 5.3.4.

During the visualization of classes, the generated transformation has to create instances of a fixed presentation metamodel supported by the tool (see the lower part of Fig. 87). These instances appear only in the generated transformations. Therefore, the presentation metamodel is the *constant* part of the metamodel for the generated transformation (compare to Fig. 75, p.194 and Fig. 77, p.196). It describes a graph diagram with *Nodes* and *Edges*. There are *CompositeNodes* containing other *Nodes* and *Labels* for text visualization.

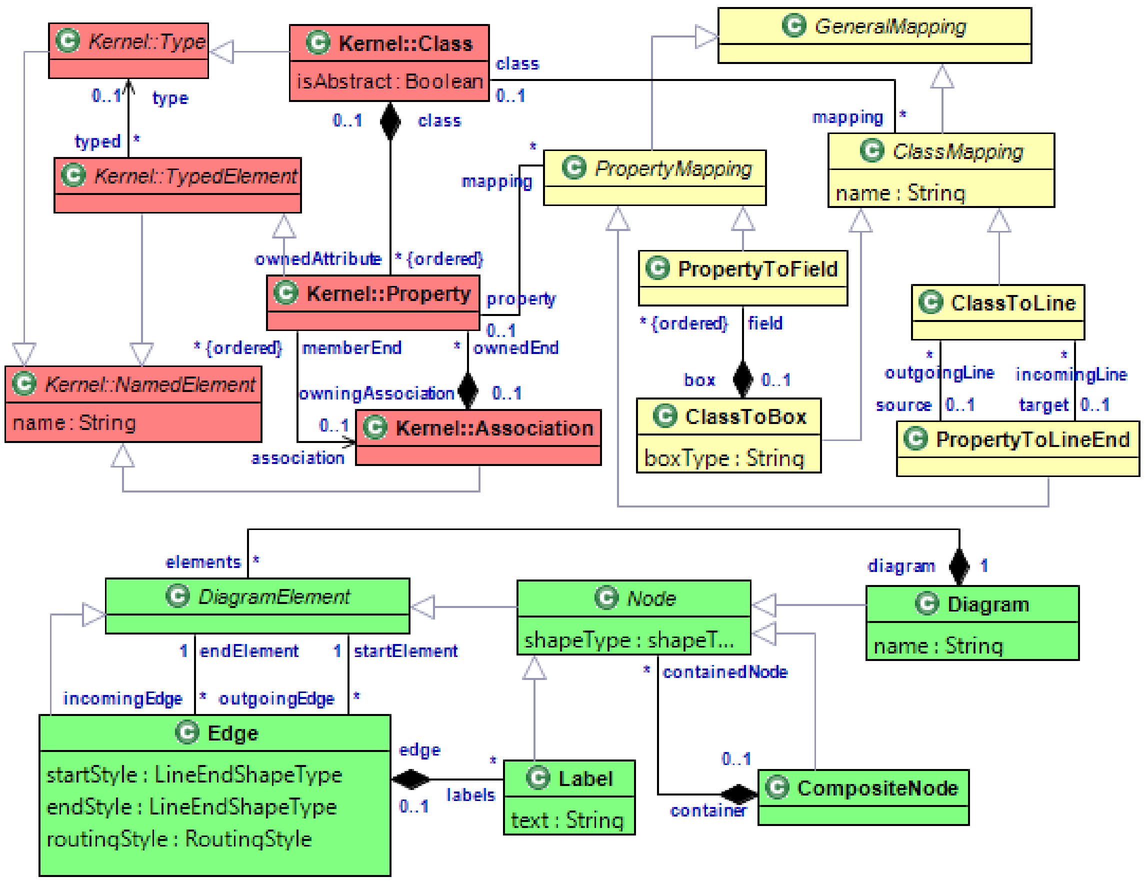


Fig. 87. A simplified domain (upper left side), mapping (upper right side) and presentation (lower part) metamodel

When metamodels and their roles are specified, we can move on to transformation definition in Template MOLA (see Fig. 88). We remind that the proper input for this generation transformation is a specific domain metamodel and a related mapping model. The transformation starts with the loop iterating through all instances of the *class to box* mapping. This loop is a generation loop and is executed in the generation time. As a result, a traditional MOLA procedure is built, containing a loop for each such mapping instance (generated from the template loop which constitutes the body of the generation time loop). The generated loops simply follow each other linked by control flows. The template loop contains the loop variable with the name being generated. The loop variable name is a concatenation of the letter "*i*" and the name of the appropriate class given by the template expression *<%@c.name%>*. The type of the loop variable is defined by the template expression *<%@c:Class%>*. In each generated loop the type (*@c*) is replaced with the concrete domain class corresponding to the mapping instance this loop is generated from. In each loop the value assigned to *shapeType* attribute is explicitly defined. This value is calculated in the generation time using the corresponding mapping data (the template expression *<%@cm.boxType%>* directly references the *boxType* attribute of the current mapping instance). Now in runtime each generated loop iterates over all instances of the corresponding domain class and creates a box for each of them.

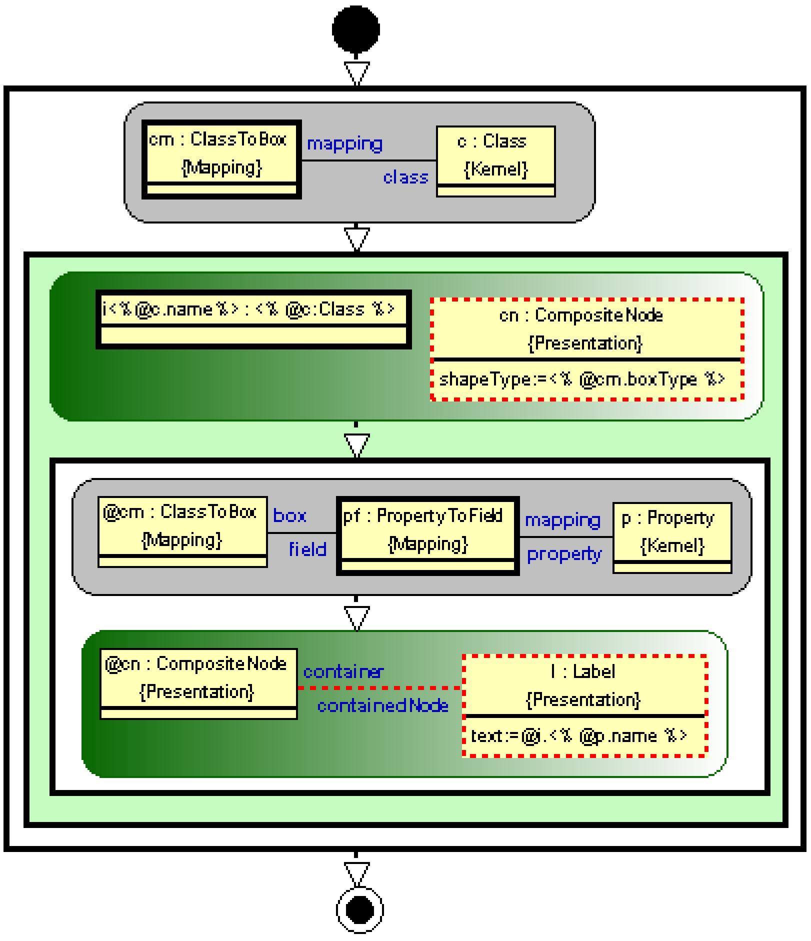


Fig. 88. Mapping implementation for tool building in Template MOLA

We must also generate transformations to create fields and set their values. Therefore, a rule for processing each field has to be generated in the loop body. To ensure this, in the template loop a generation time loop is included. This loop checks which field mappings are included into the given class mapping. A rule is created for each such field which adds a label to the box and sets its value. To set the value of the label, the relevant property value of the runtime instance should be used. To access this property, the template expression *<%@p.name%>* is used within the assignment in the template rule. During generation the generation time loop ensures that the template expression is replaced with the relevant property each time. It is not difficult to see that the generated sequence of rules will do exactly the required label creation. The structure of the generated procedure is given in Fig. 89.

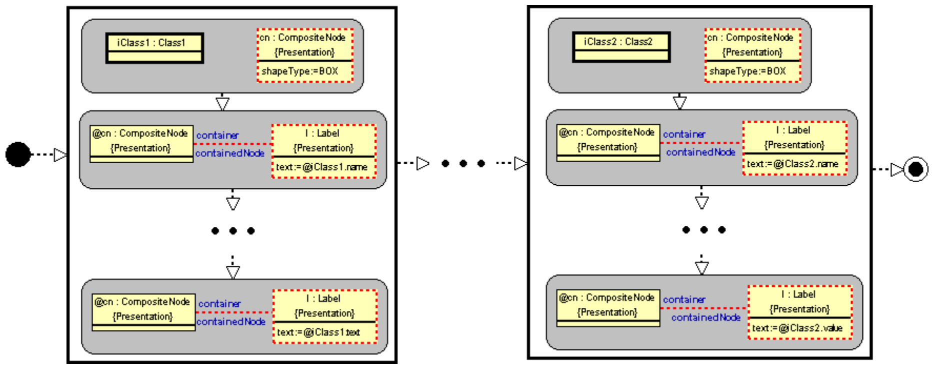


Fig. 89. A MOLA procedure generated for Fig. 88

7.4 Transformation Libraries

Another application area of synthesis transformations is the development of transformation libraries. It is important for the model transformation languages which do not support the work with multiple metalevels. In these languages model transformations are attached to the metamodel they are defined for. As a result it is not possible to define metamodel independent transformations. HOTs could be used to solve this problem. It is possible to define a transformation which reads the metamodel and creates the appropriate transformation for this metamodel. When using this approach, it is possible to create metamodel independent transformation libraries. The given HOT application is discussed in this section.

7.4.1 Transformations for Generic Metamodels

Template MOLA can be used to write transformations for generic metamodels (the metamodel is unknown at the time of writing). For example, we can write a generic instance cloning procedure. More precisely, we can write an instance cloning generator in

Template MOLA, then execute it for a concrete metamodel and run the generated traditional MOLA to clone instances of this metamodel.

Such approach can be used to create reusable transformation libraries. Model transformation reuse has been considered an important topic [34]. One of the obstacles is the complete dependency of the transformation definition on the used metamodel. Generic transformations (transformation generators) in Template MOLA could be used to create a reusable library of common metamodel independent algorithms for model processing.

This approach is less important if the transformation language contains features for work with several meta-levels at a time. However, it is useful for transformation languages like MOLA (and most of others that include the OMG standard MOF QVT [122]), which have no support for work with different meta-levels.

Generic Template MOLA procedures can be combined with the traditional MOLA. The analogy with C++ templates and Java generics is used here. For example, it is also possible to write such a template based cloning procedure in C++ (see Listing 4).

Listing 4. Template based cloning procedure in C++

```
template <class T> void Clone (T orig, T& copy) {...}.
```

In C++ this template procedure can be called with parameters of a concrete type. To process this template procedure, the pre-processor generates an instance of this procedure for every type it is called with. The same idea is used to combine MOLA with Template MOLA. This feature is required if we want to invoke reusable transformations from a transformation library.

Calls to template procedures can be used in ordinary MOLA transformations. In Fig. 90 calls to the template procedure *Clone* are demonstrated. The same pre-processor technology is applied when combining MOLA with Template MOLA as in C++ when generating procedures for each type they are called with.

Since several MOLA procedures should be generated from one template procedure, the procedure names should be generated, too (several procedures with the same name are not allowed in MOLA). Here the default name generation is used. For a template procedure, it is possible to define an expression of how the procedure name should be generated exactly, however, the default naming conventions are also provided.

One of the pre-processor tasks in combining MOLA and Template MOLA is to replace calls to the template procedure with calls to the appropriate generated procedures.

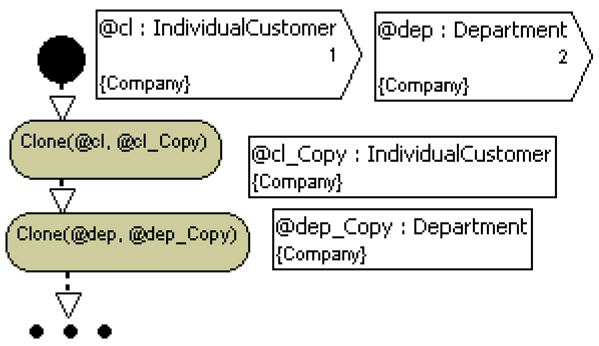


Fig. 90. An example where the traditional MOLA and Template MOLA are combined. A MOLA procedure calling the template procedure *Clone* from Fig. 91 is illustrated

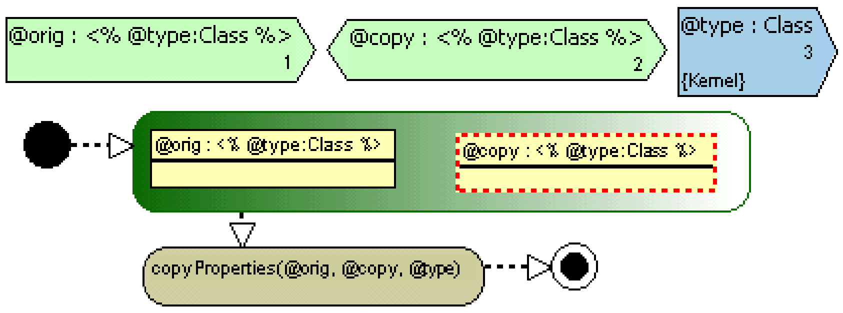

@orig : <% @type:Class %>
1
@copy : <% @type:Class %>
2
@type : Class
3
{Kernel}
@orig : <% @type:Class %>
@copy : <% @type:Class %>
copyProperties(@orig, @copy, @type)

Fig. 91. The *Clone* procedure

Fig. 91 demonstrates the content of the template procedure *Clone*. It contains two template parameters. It means that two parameters will be created in the generated procedure. Instead of the type, these parameters contain the template expression *<%@type:Class%>*. This template expression is evaluated in the generation time and replaced with the appropriate values in the generated procedures. The procedure contains one more kind of parameter – a *type parameter* (the parameter *@type*). This parameter has an analogy to C++ code, where the template parameter *T* was explicitly defined in the procedure definition. In the same way as in C++, the value of the parameter is not defined in a call, but it is inferred from other parameters. Note that the type parameter is used for this type of transformations only (transformations for generic metamodels) and is not required for typical HOT use cases. Since this template procedure is invoked from the ordinary MOLA, the referenced metamodel must be MOLA MOF itself (the *Kernel* package).

In the *Clone* procedure one rule and one call is generated. In the rule, the template expressions (which specify types of class elements) are replaced with their generation time values in the same way as in the template parameters. The call statement contains one generation time parameter and two template parameters. The template parameters are kept in the generated call. Actually, instead of a call to the template procedure, a call to the appropriate instance of the procedure generated from the template procedure is created (taking into account the name generation).

The template procedure in Fig. 92 generates the procedure to copy instance properties. It contains two template parameters and one generation time parameter. The generated procedure will have two parameters created from the template parameters. Generation time parameter is only used in the generation time.

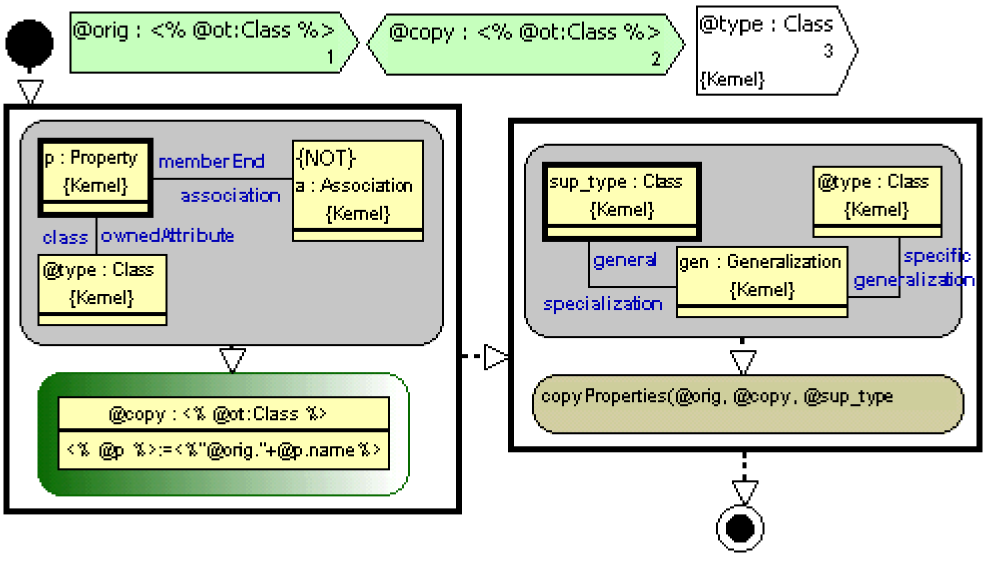


Fig. 92. The *copyProperties* procedure

The procedure *copyProperties* contains two generation time loops. The first loop (on the left in Fig. 92) iterates through all direct attributes of the class. For each attribute, it generates a rule containing a class element with an assignment in it. The value of the same attribute in the instance *orig* is assigned to this attribute. In the generated class element, all template expressions are replaced with their values. The template expressions are used for the class element type, for the attribute to be assigned and for the assigned expression. A remark on the template expression syntax: the left hand side of the assignment must be an attribute reference in MOLA. Formally, both the notation *@p* (the reference to the attribute) and *@p.name* (a string expression equal to the attribute name) could be used here. Our choice is *@p* since it expresses more directly that the left hand side is a reference (it is preferred for the implementation as well).

The second loop (on the right in Fig. 92) iterates trough all immediate super-classes of this class. For each super-class, it generates a call to a procedure that copies direct attributes of this super-class. In this way, using recursion in Template MOLA, values of all attributes are finally copied. It should be noted that the generated MOLA procedures are not recursive due to the fact that procedure names are generated when several MOLA procedures are created from one template procedure. Fig. 94 and Fig. 95 explain this situation by means of an example.

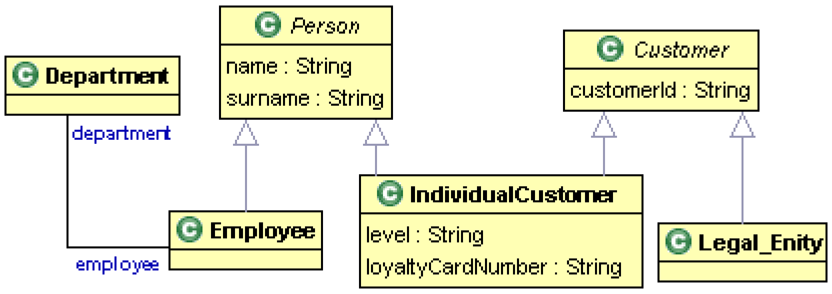


Fig. 93. A metamodel example describing information processed by a company. The class *IndividualCustomer* is used to describe the generated code in Fig. 94 and Fig. 95

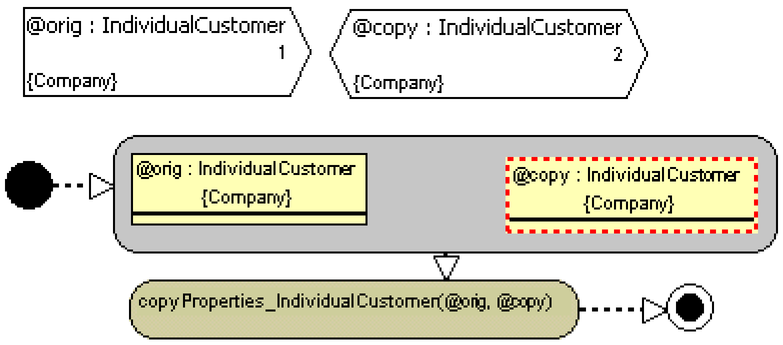

@orig : IndividualCustomer
1
{Company}
@copy : IndividualCustomer
2
{Company}
@orig : IndividualCustomer
{Company}
@copy : IndividualCustomer
{Company}
copyProperties_IndividualCustomer(@orig, @copy)

Fig. 94. MOLA procedure generated from the template procedure *Clone*

Now let us consider MOLA procedures generated from the *Clone* algorithm as described above by using Template MOLA. We will demonstrate the generated result for the first call of the procedure *Clone* in Fig. 90. The type of the instance to be cloned is *Company::IndividualCustomer*. The metamodel for this fragment is described in Fig. 93 (the package containing the fragment is assumed to be *Company*). This could be a simplified metamodel describing the information processed by a company. Fig. 94 presents the code generated form the template procedure *Clone*. The type parameter value is the type of the instance the call statement was invoked with. In this case, it is the class *Company::IndividualCustomer*. In the generated code, the type parameter *@type* is replaced with this class. The procedure call is replaced with a call to the generated procedure with appropriate types. Note that procedure names are generated in Template MOLA as well (according to the default name generation rules, which can be modified if required). The procedure name here will be appended by the class name from the type parameter. The procedure name generation is necessary because the generated procedure code depends on the type (or generation) parameter value (as shown in Fig. 95). The type parameter itself is not included in the generated code.

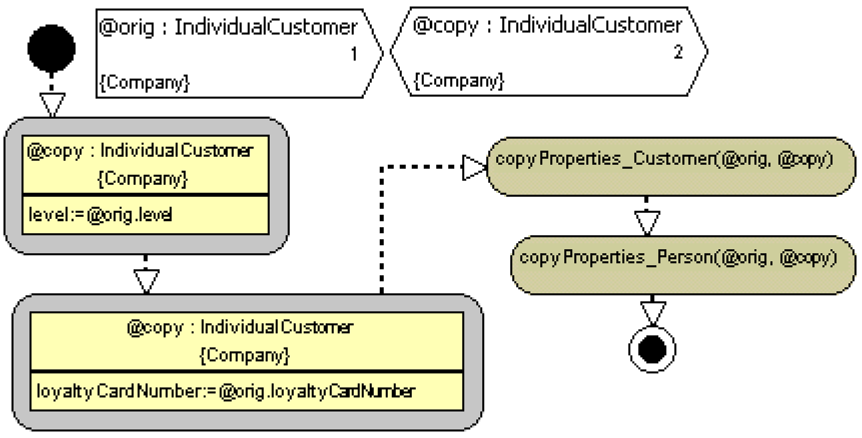

@orig : IndividualCustomer
1
{Company}
@copy : IndividualCustomer
2
{Company}
@copy : IndividualCustomer
{Company}
level:=@orig.level
copyProperties_Customer(@orig, @copy)
copyProperties_Person(@orig, @copy)
@copy : IndividualCustomer
{Company}
loyaltyCardNumber:=@orig.loyaltyCardNumber

Fig. 95. MOLA procedure generated from the template procedure *copyProperties*

Fig. 95 presents the structure of a MOLA procedure generated from the *copyProperties* procedure in Fig. 92 (p.230) when the class specified by the generation time parameter is *Company::IndividualCustomer* (i.e., it is the procedure *copyProperties_ IndividualCustomer*). The left side shows two of the generated rules for assigning direct attribute values of the *IndividualCustomer* class (to the attributes *level* and *loyaltyCardNumber*). The attribute assignments are followed by calls to the *copyProperties* procedures generated for the superclasses of *IndividualCustomer* (calls for the superclasses *Person* and *Customer* are shown). Note that the generated names of the procedures include the class name from the generation time parameter, thus there is no recursion in the generated code.

In this example the generated MOLA source is a kind of *spaghetti code*. However, it would be sufficient to have one class element containing assignments for each property. This issue could be solved using the merge mechanism described in Section 6.7. A solution of the same task using the merge mechanism is described in Section 6.7.1, the Template MOLA procedure and an example of generated code is given in Fig. 79 (p.205).

7.4.2 Transformation Design Patterns

The higher-order transformations could be used to apply transformation design patterns. It means it could be possible to generate the initial transformation code according to the transformation design pattern using HOTs, e.g., to apply some design pattern for one specific case according to the defined parameters.

It should be reminded that using HOTs it is also possible to read the transformation sources. It means it should be possible to adapt some existing transformation according to the selected design pattern.

In this way it is also possible to implement transformation refactoring and merge of several transformations.

When using higher-order transformations, we could automatically get a transitive closure according to some associations. It should be noted that there is no direct support for a transitive closure of an association in the MOLA patterns, while some other transformation languages have this feature. The TTC 2011 Reengineering challenge [54] demonstrates that there are tasks where a transitive closure could be widely used. The MOLA solution of this task used higher-order transformations to generate concrete transformations [155]. Here higher-order transformations were defined in MOLA; however, Template MOLA could be even a more adequate solution. Thus, Template MOLA can be used to add the missing language features to MOLA in a generic way.

The mapping operators proposed in [198] could also be treated as transformation design patterns with one type of the operators being the *Copy* operator, which copies the data from one model to another. *Copy* was also widely used in the mapping language MALA4MDSD. The *Copy* operation is very popular in model transformations and it would be useful to obtain a metamodel independent *copy* library. In [198] other mapping operators have been considered. However, these operators were very simple and therefore do not seem so useful in the context of this research.

On the whole the identification of reusable transformation design patterns is an interesting issue and the author of the given thesis believes that it has not been studied enough and therefore offers a perspective direction of future research.

7.5 Conclusions

There are several application areas for Template MOLA. First of all it is metamodel-based tool building for graphical DSL. More precisely, it is the generation of transformations that determine the tool behaviour according to mappings that define the tool functionality in a static way (as, for example, in GMF).

A related application could be generation of transformations from a more general kind of mappings between models. This is the area where HOTs are widely used, especially in ATL. The development of experimental mapping language compiler has confirmed that Template MOLA is applicable to solve this task. Detailed conclusions were already given in Section 7.2.

Another important application is the building of transformations for unknown metamodels. In this way reusable transformation libraries for performing typical model processing tasks could be created. Afterwards transformations from such libraries could be used in the ordinary MOLA transformations for a specific metamodel. A very simple example from this area is also provided in this chapter.

A future research direction could be an extension of Template MOLA for defining templates in other graphical languages, e.g., UML activity diagrams. Then the corresponding template statements would be defined by the graphical syntax of the generated language. Generation statements controlling the generation process would certainly remain in MOLA. This approach could be applied, e.g., for building of various process generators. This requires more research because the implementation could turn out to be more complicated than that for Template MOLA.

CHAPTER 8

Conclusions

This PhD thesis presents a research on model transformation development. Three domain-specific transformation application areas have been studied: transformations for Model-Driven Software Development, transformations for graphical DSL tool building and transformations synthesising transformations.

It is concluded that a domain-specific language is more convenient and efficient for transformation development in a specific domain. Each selected domain area confirmed this conclusion and for each domain area a domain-specific language has been built. The thesis confirms that transformation development in these specific languages is more convenient compared to transformation development in traditional model transformation languages.

The given domain-specific transformation languages should support transformation development for typical cases; however, it is not necessary to support all exceptional cases. A domain-specific language should be well integrated with general-purpose transformation languages then the processing of an exceptional case can be implemented in a transformation language. Support for processing of all exceptional cases in the domain-specific language would make this language excessively complicated.

The above given conclusions are based on research in the three selected domain areas. However, proving the general validity of these statements is a task for future research.

It should be noted that the development of domain-specific language is not free of charge. Language development pays off only at a big enough amount of transformations to be developed. In case of requiring only one small transformation the development of a new domain-specific language is very likely to be unproductive. In this case one of the existing languages should be used as the time demanded for developing a new language will be greater than the time spent developing transformations. One of directions for future research could focus on elaboration of cost-effectiveness evaluation of a new domain-specific language for a particular domain.

The transformation domains discussed in the PhD thesis are big enough to have a potential for developing many similar transformations, therefore creation of a domain-specific language will pay off.

It should be underlined that two of the specialised domain-specific transformation languages proposed in the thesis are based on mappings that are the most comprehensible means for transformation development. However, it is not possible to define transformations by using only mappings. This explains why only mapping elements are used in model transformation languages. Nevertheless, if mappings are adapted for a specific domain, then most of transformation logic could be defined by using mappings.

In fact, the approach used for MALA4MDSD could be generalised for a wider class of transformations. This approach could be used to build other similar mapping languages for other domains. Another direction of future research is studying the applicability in different domains and limitations of this approach. It would be interesting to find out whether this approach could be applied for graphical DSL tool development.

A tool for developing a mapping language compiler is also proposed in the thesis. In the given case the Template MOLA language for transformation synthesis is applicable. Mapping and the transformation integration problem is solved by using the Template MOLA for the mapping language compilation. Mappings are compiled to transformations and to accomplish the integration only calls to the appropriate MOLA procedures should be created.

The Template MOLA is suitable for MOLA transformation synthesis. Synthesis of code in other graphical languages might provide a very interesting direction of future research as it is likely that the approach similar to the one used in the Template MOLA could be used, namely, to synthesize diagram fragments in concrete syntax.

BIBLIOGRAPHY

[1] Link specification language, Available: http://www.assembla.com/wiki/show/silk/Link_Specification_Language (online, 2011.10.29)

[2] ReDSeeDS, Available: http://sourceforge.net/projects/redseeds/ (online, 2011.10.06)

[3] ReDSeeDS –Requirements-Driven Software Development System, Available: http://www.redseeds.eu (online, 2011.10.06)

[4] SHARE demo related to the paper Saying Hello World with MOLA - A Solution to the TTC 2011 Instructive Case, Available: http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_MOLA.vdi (online, 2011.10.06)

[5] Transformation Tool Contest 2011, Available: http://planet-research20.org/ttc2011 (online, 2011.10.05)

[6] The MOLA language. Reference Manual, Version 2.0 final (December 2007), Available: http://mola.mii.lu.lv/mola2fin_refmanual.pdf (online, 2011.09.27)

[7] Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Tech. Rep. ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT (2003)

[8] Ambroziewicz, A., Straszak, T., Schwarz, H., Nowakowski, W., Brogan, J.P., Kalnina, E., Celms, E., Sostaks, A., Bildhauer, D., Nick, M., Schneickert, S., Kalnins, A., Bojarski, J., Hotz, L.: Research-oriented software artefacts. Project Deliverable D5.1, ReDSeeDS Project www.redseeds.eu (2007)

[9] Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Petriu, D., Rouquette, N., Haugen, y. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6394, pp. 121–135. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/978-3-642-16145-2_9 (2010)

[10] AUTOSAR Consortium: The AUTOSAR standard, Available: http://www.autosar.org/ (online, 2011.09.01)

[11] Barzdins, J., Barzdins, G., Balodis, R., Cerans, K., Kalnins, A., Opmanis, M., Podnieks, K.: Towards semantic Latvia. In: Vasilecas, O. (ed.) Proceedings of DB&IS. pp. 203–218. Vilnius, Technika (2006)

[12] Barzdins, J., Cerans, K., Kozlovics, S., Lace, L., Liepins, R., Rencis, E., Sprogis, A., Zarins, A.: An MDE-based graphical tool building framework. In: Scientific Papers, University of Latvia (2010)

[13] Barzdins, J., Kalnins, A., Rencis, E., Rikacovs, S.: Model transformation languages and their implementation by bootstrapping method. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 130–145. Springer-Verlag, Berlin, Heidelberg (2008)

[14] Barzdins, J., Kozlovics, S., Rencis, E.: The Transformation-Driven Architecture. In: Proceedings of DSM'08 Workshop of OOPSLA 2008. pp. 60–63. Nashville, Tennessee, USA (2008)

[15] Barzdins, J., Zarins, A., Cerans, K., Kalnins, A., Rencis, E., Lace, L., Liepins, R., Sprogis, A.: GrTP: Transformation based graphical tool building platform. In: Proceedings of MDDAUI 2007 Workshop of MODELS 2007. Nashville, Tennessee, USA (2007)

[16] Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co., Greenwich, CT, USA (2006)

[17] Bezivin, J.: Broadening application area (November 2010), Available: http://modelseverywhere.wordpress.com/2010/11/05/broadening-application-area/ (online, 2011.09.15)

[18] Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: Proceedings of the 16th IEEE international conference on Automated software engineering. pp. 273–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001)

[19] Bildhauer, D., Ebert, J., Riediger, V., Krebs, T., Nick, M., Schwarz, H., Kalnins, A., Kalnina, E., Schneickert, S., Celms, E., Wolter, K., Ambroziewicz, A., Bojarski, J.: Repository selection report. Project Deliverable D4.4, ReDSeeDS Project www.redseeds.eu (2007)

[20] Bizer, C.: D2R MAP - database to RDF mapping language and processor, Available: http://www4.wiwiss.fu-berlin.de/bizer/d2rmap/d2rmap.htm (online, 2011.10.29)

[21] BLU AGE: Model2Code, Available: http://www.model2code.com/ (online, 2011.10.13)

[22] Būmans, G., Čerāns, K.: DB2OWL: Mapping relational databases into OWL ontologies - a practical approach. In: J. Barzdins, M.K. (ed.) Databases and Information Systems. Proceedings of the Ninth International Baltic Conference Baltic DB&IS 2010. pp. 393 – 408. University of Latvia, Riga, Latvia (July 2010)

[23] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72(1-2), 52–70 (2008)

[24] Brown, A.W.: Model driven architecture: Principles and practice. Software and Systems Modeling 3, 314–327 http://dx.doi.org/10.1007/s10270-004-0061-2 (2004)

[25] Budapest University of Technology and Economics, Department of Automation and Applied Informatics: Visual Modeling and Transformation System (VMTS), Available: http://www.aut.bme.hu/Portal/Vmts.aspx (online, 2011.08.26)

[26] Celms, E., Kalnins, A., Lace, L.: Diagram definition facilities based on metamodel mappings. In: Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling. pp. 23–32. University of Jyvaskyla (2003)

[27] CityRailTransit: Rail transit maps > Paris (France), Available: http://www.cityrailtransit.com/maps/paris_map.htm (online, 2011.09.13)

[28] Cook, S., Jones, G., Kent, S., Wills, A.C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley (2007)

[29] Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. 61, 190–210 (August 2006)

[30] Cordy, J.R.: Eating our own dog food. Slides of SLE 2009 Keynote (2009), Available: http://planet-sl.org/sle-conference/sle-organization/7z36rz47aezerz6er3434h/organization/sle2009/jim_cordy_sle09_keynote.pdf (online, 2011.09.27)

[31] Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual automated transformations for formal verification and validation of UML models. In: Proceedings of the 17th IEEE international conference on Automated software engineering. p. 267–270. ASE '02, IEEE Computer Society, Washington, DC, USA (2002)

[32] Cuadrado, J., Molina, J., Tortosa, M.: RubyTL: A practical, extensible transformation language. In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science, vol. 4066, pp. 158–172. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/11787044_13 (2006)

[33] Cuadrado, J.S., Guerra, E., de Lara, J.: Generic model transformations: Write Once, Reuse Everywhere. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 6707, pp. 62–77. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/978-3-642-21732-6_5 (2011)

[34] Cuadrado, J.S., Molina, J.G.: Approaches for model transformation reuse: Factorization and composition. In: Proceedings of ICMT 2008. LNCS, vol. 5063, p. 168–182. Springer, Zürich, Switzerland (2008)

[35] DSTC: Tefkat: The EMF Transformation Engine, Available: http://tefkat.sourceforge.net/ (online, 2011.08.29)

[36] Dumas, M.: Case study : BPMN to BPEL model transformation. Oryx pp. 6–9 http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2008translationcase.pdf (2008)

[37] Ermel, C., Ehrig, K., Taentzer, G., Weiss, E., Ermel, C., Ehrig, K., Taentzer, G., Weiss, E.: Object oriented and rule-based design of visual languages using Tiger. In: Proceedings of GraBaTs'06. p. 12 (2006)

[38] European Commission: ReDSeeDS: Requirements-driven software development system, Available: http://cordis.europa.eu/fetch?CALLER=PROJ_ICT&ACTION=D&DOC=40&CAT=PROJ&QUERY=01312ff3409f:cf83:56f807cb&RCN=79442 (online, 2011.10.06)

[39] del Fabro, M.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (2005)

[40] Favre, L.: Model driven architecture for reverse engineering technologies: strategic directions and system evolution. Hershey: IGI Global, New York, USA (2010)

[41] Fellbaum, C. (ed.): WordNet: An Electronic Lexical Database. MIT Press http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=8106 (1998)

[42] Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In: Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (eds.) Proceedings of TAGT. LNCS, vol. 1764, p. 296–309. Springer (1998)

[43] Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and Java. In: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations. pp. 296–309. TAGT'98, Springer-Verlag, London, UK (2000)

[44] Freie Universität Berlin: Language specification, Available: http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/#specification (online, 2011.10.29)

[45] Fujaba Tool Suite Developer Team, University of Paderborn: Fujaba tool suite, Available: http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/FUJABA/ (online, 2011.09.28)

[46] Fujaba Tool Suite Developer Team, University of Paderborn: Incremental model transformation and synchronization with Triple Graph Grammars, Available: http://www.fujaba.de/projects/triple-graph-grammars.html (online, 2011.09.28)

[47] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass. (1995)

[48] Geiß, R., Batz, G., Grund, D., Hack, S., Szalkowski, A.: Grgen: A fast spo-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) Graph Transformations, Lecture Notes in Computer Science, vol. 4178, pp. 383–397. Springer Berlin / Heidelberg (2006)

[49] Grønmo, R., Møller-Pedersen, B.: From sequence diagrams to state machines by graph transformation. In: Tratt, L., Gogolla, M. (eds.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, vol. 6142, pp. 93–107. Springer Berlin / Heidelberg (2010)

[50] Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: transML: A family of languages to model model transformations. In: Petriu, D., Rouquette, N., Haugen, y. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 6394, pp. 106–120. Springer Berlin / Heidelberg (2010)

[51] Hausmann, J.H., Kent, S.: Visualizing model mappings in UML. In: Proceedings of the 2003 ACM symposium on Software visualization. pp. 169–178. SoftVis ’03, ACM, New York, NY, USA (2003)

[52] Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009 – Object- Oriented Programming. LNCS, vol. 5653, p. 52–76. Springer, Heidelberg (2009)

[53] Hillairet, G., Bertrand, F., Lafaye, J.Y.: MDE for publishing data on the semantic web. In: Parreiras, F.S., Pan, J.Z., Aßmann, U., Henriksson, J. (eds.) Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering TWOMDE 2008, Toulouse, France, September 28, 2008. CEUR Workshop Proceedings, vol. 395, pp. 32–46. CEUR-WS.org (2008)

[54] Horn, T.: Program understanding: A reengineering case for the transformation tool contest. In: Van Gorp et al. [187]

[55] Horn, T., Ebert, J.: The GReTL transformation language. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 6707, pp. 183–197. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/978-3-642-21732-6_13 (2011)

[56] IBM: Model Transformation Framework (MTF), Available: http://www.alphaworks.ibm.com/tech/mtf (online, 2011.08.29)

[57] IKV++ Technologies AG: medini qvt project, Available: http://projects.ikv.de/qvt/ (online, 2011.10.28)

[58] Institut fur Informatik der Technischen Universitat Munchen: The bidirectional object oriented transformation language (BOTL), Available: http://botl.sourceforge.net/ (online, 2011.08.29)

[59] Institute of Mathematics and Computer Science, University of Latvia: Mola pages, Available: http://mola.mii.lu.lv (online, 2011.08.28)

[60] Iraids, J.: Template MOLA realizācija. Master’s thesis, University of Latvia (2011)

[61] Jackson, M.: Some basic tenets of description. Software and Systems Modeling 1, 5–9 http://dx.doi.org/10.1007/s10270-002-0005-7 (2002), 10.1007/s10270-002-0005-7

[62] Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS'06: Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems. LNCS, vol. 4037, p. 171–185. Springer Berlin / Heidelberg, Bologna, Italy (2006)

[63] Jouault, F., Kurtev, I.: Transforming models with the ATL. Lecture Notes in Computer Science 3844, 128–138 (October 2005)

[64] Kahle, S.: JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology (2006)

[65] Kaindl, H., Smiałek, M., Wagner, P., Svetinovic, D., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Schwarz, H., Bildhauer, D., Falb, J., Brogan, J.P., Mukasa, K.S., Wolter, K., Kavaldjian, S., Szép, A., Kalnina, E., Kalnins, A.: Requirements specification language definition. Project Deliverable D2.4.2, ReDSeeDS Project www.redseeds.eu (2009)

[66] Kaindl, H., Smiałek, M., Svetinovic, D., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Schwarz, H., Bildhauer, D., Brogan, J.P., Mukasa, K.S., Wolter, K., Krebs, T.: Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project www.redseeds.eu (2007)

[67] Kalnina, E.: DSL tool development with transformations and static mappings. In: Pretschner, A. (ed.) Models 2008, Toulouse, France, 28 September - 3 October 2008, Doctoral Symposium. ETH Zürich Technical Report, vol. 606, pp. 9–14 (September 2008)

[68] Kalnina, E., Kalnins, A.: DSL tool development with transformations and static mappings. In: Chaudron, M.R.V. (ed.) Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France. Reports and Revised Selected Papers. LNCS, Programming and Software Engineering, vol. 5421, p. 356–370. Springer (2009)

[69] Kalnina, E., Kalnins, A., Celms, E., Sostaks, A.: Graphical template language for transformation synthesis. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 244–253. Springer, Heidelberg (2010)

[70] Kalnina, E., Kalnins, A., Celms, E., Sostaks, A., Iraids, J.: Generation mechanisms in graphical template language. In: Osis, J., Nikiforova, O. (eds.) Proceedings of MDA&MTDD 2010 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development. In conjuction with ENASE 2010. pp. 43–52. SciTePress, Portugal, Athens, Greece (July 2010)

[71] Kalnina, E., Kalnins, A., Celms, E., Sostaks, A., Iraids, J.: Transformation synthesis language – Template MOLA. Scientific Papers, University of Latvia, vol. 756, p. 77–98 (2010)

[72] Kalnina, E., Kalnins, A., Iraids, J., Sostaks, A., Celms, E.: Model migration with MOLA. In: Mazanek, S., Rensink, A., Van Gorp, P. (eds.) Proceedings of Transformation Tools Contest 2010 (TTC), co-located with the International Conference on Objects, Models, Components and Patterns (TOOLS Europe). pp. 38–60. Online Proceedings http://www.ctit.utwente.nl/library/proceedings/wp10-03.pdf (2010)

[73] Kalnina, E., Kalnins, A., Sostaks, A., Celms, E., Iraids, J.: Tree based domain-specific mapping languages (2012), to appear in SOFSEM 2012 LNCS proceedings

[74] Kalnina, E., Kalnins, A., Sostaks, A., Iraids, J., Celms, E.: Saying hello world with MOLA - a solution to the TTC 2011 instructive case. In: Van Gorp et al. [187], pp. 236–251

[75] Kalnins, A., Barzdins, J., Celms, E.: Basics of model transformation language MOLA. In: Workshop on Model Driven Development (WMDD 2004) (2004)

[76] Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. In: Proceedings of MDAFA 2004. pp. 14–28. Linkoeping, Sweden (2004)

[77] Kalnins, A., Barzdins, J., Celms, E.: MOLA language: Methodology sketch. In: Second European Workshop on Model Driven Architecture (MDA), EWMDA-2 (2004)

[78] Kalnins, A., Celms, E., Kalnina, E., Sostaks, A.: Behaviour modelling notation for information system design. In: BM-MDA '09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture. pp. 1–7. ACM, New York, NY, USA (2009)

[79] Kalnins, A., Kalnina, E., Celms, E., Sostaks, A.: From requirements to code in a model driven way. In: Grundspenkis, J., Kirikova, M., Manolopoulos, Y., Novickis, L. (eds.) Advances in Databases and Information Systems. Lecture Notes in Computer Science, vol. 5968, pp. 161–168. Springer Berlin / Heidelberg (2010), 10.1007/978-3-642-12082-4_21

[80] Kalnins, A., Kalnina, E., Celms, E., Sostaks, A.: A model-driven path from requirements to code. Computer Science and Information Technologies, vol. 756, pp. 33–57. Scientific Papers, University of Latvia (2010)

[81] Kalnins, A., Kalnina, E., Celms, E., Sostaks, A., Schwarz, H., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Kavaldjian, S., Falb, J.: Reusable case transformation rule specification. Project Deliverable D3.3, ReDSeeDS Project www.redseeds.eu (2007)

[82] Kalnins, A., Smialek, M., Kalnina, E., Celms, E., Nowakowski, W., Straszak, T.: Model-Driven Domain Analysis and Software Development: Architectures and Functions, chap. Domain-Driven Reuse of Software Design Models, pp. 177–200. IGI Global, Hershey (2011)

[83] Kalnins, A., Sostaks, A., Celms, E., Kalnina, E., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Riediger, V., Schwarz, H., Bildhauer, D., Kavaldjian, S., Popp, R., Falb, J.: Reuse-oriented modelling and transformation language definition. Project Deliverable D3.2.1, ReDSeeDS Project www.redseeds.eu (2007)

[84] Kalnins, A., Sostaks, A., Celms, E., Kalnina, E., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Riediger, V., Schwarz, H., Bildhauer, D., Kavaldjian, S., Popp, R., Falb, J.: Final reuse-oriented modelling and transformation language definition. Project Deliverable D3.2.2, ReDSeeDS Project www.redseeds.eu (2009)

[85] Kalnins, A., Sostaks, A., Kalnina, E., Celms, E., Vilitis, O.: MOLA 2 Tool. In: ECMDA Tools and Services Session (2008)

[86] Kalnins, A., Vilitis, O., Celms, E., Kalnina, E., Sostaks, A., Barzdins, J.: Building tools by model transformations in Eclipse. In: Proceedings of DSM'07 Workshop of OOPSLA 2007. pp. 194–207. Jyvaskyla University Printing House, Montreal, Canada (2007)

[87] Kavaldjian, S., Kaindl, H., Mukasa, K.S., Falb., J.: Transformations between specifications of requirements and user interfaces. In: 4th Int. Workshop MDDAUI. p. 37–40 (2009)

[88] Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley (2008)

[89] Kühne, T.: Matters of (meta-) modeling. Software and Systems Modeling 5, 369–385 http://dx.doi.org/10.1007/s10270-006-0017-9 (2006)

[90] Kleppe, A.G., Warmer, J.B., Wim, B.: MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston (2003)

[91] Kolovos, D., Paige, R., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science, vol. 4066, pp. 128–142. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/11787044_11 (2006), 10.1007/11787044_11

[92] Kolovos, D., Paige, R., Polack, F.: The Epsilon transformation language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 5063, pp. 46–60. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/978-3-540-69927-9_4 (2008)

[93] Kolovos, D.: A brief history of Epsilon (2007), Available: http://epsilonblog.wordpress.com/2007/11/11/a-brief-history-of-epsilon/ (online, 2011.10.28)

[94] Krebs, T., Nowakowski, W., Kalnins, A., Kalnina, E.: Modelling and transformation language validation report. Project Deliverable D3.4, ReDSeeDS Project www.redseeds.eu (2007)

[95] Lano, K., Kolahdouz-Rahimi, S.: Model-driven development of model transformations. In: Cabot, J., Visser, E. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 6707, pp. 47–61. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/978-3-642-21732-6_4 (2011)

[96] de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering. LNCS, vol. 2306, pp. 174–188. Springer-Verlag, London, UK (2002)

[97] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, Englewood Cliffs, NJ, second edn. (2004)

[98] Lawley, M., Steel, J.: Practical declarative model transformation with Tefkat. In: Bruel, J.M. (ed.) Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science, vol. 3844, pp. 139–150. Springer Berlin / Heidelberg http://dx.doi.org/10.1007/11663430_15 (2006)

[99] Levendovszky, T., Lengyel, L., Mezei, G., Charaf, H.: A systematic approach to metamodeling environments and model transformation systems in VMTS. Electronic Notes in Theoretical Computer Science 127(1), 65 – 75 http://www.sciencedirect.com/science/article/pii/S1571066105001155 (2005)

[100] Liepiņš, R.: lQuery: A model query and transformation library. Scientific Papers, University of Latvia 770, 27–45 (2011)

[101] Šlihte, A.: The specific text analysis tasks at the beginning of MDA life cycle. In: Barzdins, G., Selavo, L. (eds.) Databases and Information Systems Doctoral

Consortium. Latvijas Universitātes raksti, vol. 757, pp. 11 – 22. University of Latvia, Latvija, Riga (July 5-7 2010)
[102] Lopes, D., Hammoudi, S., Bézivin, J., Jouault, F.: Generating transformation definition from mapping specification: Application to web service platform. In: ADVANCED INFORMATION SYSTEMS ENGINEERING. Lecture Notes in Computer Science, vol. 3520, pp. 309–325 (2005)
[103] Ludewig, J.: Models in software engineering - an introduction. Software and Systems Modeling 2, 5–14 http://dx.doi.org/10.1007/s10270-003-0020-3 (2003)
[104] Marinscu, F.: EJB Design Patterns. John Wiley (2002)
[105] Marschall, F., Braun, P.: Model transformations for the MDA with BOTL. Tech. rep., University of Twente (2003)
[106] Mazanek, S.: Hello World! An instructive case for the transformation tool contest. In: Van Gorp et al. [187]
[107] McGill University, Modelling, Simulation and Design Lab: AToM^3: A tool for multi-formalism meta-modelling, Available: http://atom3.cs.mcgill.ca/ (online, 2011.08.29)
[108] Mehta., V.P.: Pro LINQ Object Relational Mapping with C# 2008. Apress (2008)
[109] MetaCase: Metaedit+, Available: http://www.metacase.com/ (online, 2011.09.01)
[110] Microsoft: Visual studio visualization and modeling sdk, Available: http://archive.msdn.microsoft.com/vsvmsdk; http://www.microsoft.com/download/en/details.aspx?id=23025 (online, 2011.08.26)
[111] Miller, J., Mukerji, J. (eds.): MDA Guide Version 1.0.1, omg/03-06-01. Object Management Group (2003)
[112] Minsky, M.: Matter, Mind and Models. In: Proceedings of IFIP Congress 65. pp. 45–49 http://groups.csail.mit.edu/medg/people/doyle/gallery/minsky/mmm.html (Jan 1965)
[113] Modeling Languages: MDE Glossary – modeling and model-driven engineering terms, Available: http://modeling-languages.com/glossary-modeling-and-model-driven-engineering-terms/ (online, 2011.09.15)
[114] Nilsson, J.: Applying Domain-Driven Design and Patterns: With Examples in C# and .NET. Addison Wesley (2006)
[115] No Magic, Inc.: MagicDraw, Available: https://www.magicdraw.com/ (online, 2011.10.05)
[116] Object Management Group: Model Driven Architecture, Draft 3.2, omg/00-11-05 (2000)
[117] Object Management Group: Model Driven Architecture (MDA), Draft ormsc/01-07-01 (2001)
[118] Object Management Group: Meta Object Facility, V1.4, formal/2002-04-03 http://www.omg.org/spec/MOF/1.4/ (2002)
[119] Object Management Group: Request for Proposal: MOF 2.0 Query / Views / Transformations, OMG document ad/2002-04-10. (2002)
[120] Object Management Group: Meta Object Facility Core Specification, version 2.0, formal/2006-01-01 (2006)
[121] Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1, formal/07-02-05 (2007)
[122] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, formal/08-04-03 (2008)

[123] Object Management Group: MOF Model to Text Transformation Language, version 1.0, formal/2008-01-16 http://www.omg.org/docs/formal/08-01-16.pdf (2008)
[124] Object Management Group: Ontology Definition Metamodel (ODM), V1.0, formal/2009-05-01 http://www.omg.org/spec/ODM/1.0/ (2009)
[125] Object Management Group: Meta Object Facility, V2.4 - Beta 2, Beta2/2010-12-08 http://www.omg.org/spec/MOF/2.4/Beta2 (2010)
[126] Object Management Group: Object Constraint Language, V2.3 - Beta 2, ptc/2010-11-42 http://www.omg.org/spec/OCL/2.3/Beta2/ (2010)
[127] Object Management Group: Unified Modeling Language: Infrastructure, version 2.4 - Beta 2, ptc/10-11-16 (2010)
[128] Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.1, formal/11-01-01 (2011)
[129] Object Management Group: Meta Object Facility, V2.4.1, formal/2011-08-07 http://www.omg.org/spec/MOF/2.4.1/ (2011)
[130] Šostaks, A.: Implementation of model transformation languages. Ph.D. thesis, University of Latvia (2010)
[131] Oxford University Press: Model, Available: http://oxforddictionaries.com/definition/model?view=uk (online, 2011.09.07)
[132] Progress Software Corporation: XSLT Mapper, Available: http://www.stylusstudio.com/xslt_mapper.html (online, 2011.10.06)
[133] Ráth, I., Varró, D.: Challenges for advanced domain-specific modeling frameworks. In: Proceedings of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006. France (2006)
[134] Rein, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Kalnins, A., Celms, E., Kalnina, E., Bildhauer, D., Szymański, T.: Initial ReDSeeDS prototype. Project Deliverable D5.4.2, ReDSeeDS Project www.redseeds.eu (2008)
[135] Rein, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Kalnins, A., Celms, E., Kalnina, E., Bildhauer, D., Szymański, T., Mukasa, K.S., Ünalan, z.: Final ReDSeeDS prototype. Implementing the ReDSeeDS engine prototype – 2nd iteration. Project Deliverable D5.4.3, ReDSeeDS Project www.redseeds.eu (2009)
[136] Rein, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Kalnins, A., Celms, E., Kalnina, E., Szymański, T.: Initial ReDSeeDS prototype. Project Deliverable D5.4.1, ReDSeeDS Project www.redseeds.eu (2008)
[137] Rencis, E.: Model transformation languages L1, L2, L3 and their implementation. Computer Science and Information Technologies 733, Scientific Papers, University of Latvia (2008)
[138] Richardson, C.: POJOs in Action. Manning (2006)
[139] Rose, L., Guerra, E., de Lara, J., Etien, A., Kolovos, D., Paige, R.: Genericity for model management operations. Software and Systems Modeling http://dx.doi.org/10.1007/s10270-011-0203-2pp. 1–19
[140] Rose, L., Herrmannsdoerfer, M., Mazanek, S., Gorp, P.V., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schatz, B., Wimmer, M.: Graph and model transformation tools for model migration (2011), submitted to "Software and Systems Modeling" journal

[141] Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6142, pp. 184–198. Springer (2010)

[142] Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Model migration case for TTC 2010. In: TTC'10: Transformation Tool Contest (2010)

[143] Rothenberg, J.: Artificial intelligence, simulation & modeling, chap. The Nature of Modeling, pp. 75–92. John Wiley & Sons, Inc., New York, NY, USA (1989)

[144] Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: language and environment. World Scientific Publishing Co. 2, 487–550 (1999)

[145] Schürr, A.: PROGRESS: A VHL-language based on graph grammars. In: Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science. pp. 641–659. Springer-Verlag, London, UK (1991)

[146] Schürr, A.: Specification of graph translators with triple graph grammars. In: Tinhofer (ed.) WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science. LNCS, vol. 903, p. 151–163. Springer-Verlag (1994)

[147] Seidewitz, E.: What models mean. Software, IEEE 20(5), 26 – 32 (sept-oct 2003)

[148] Selic, B.: The pragmatics of model-driven development. IEEE Softw. 20, 19–25 (September 2003)

[149] Sipser, M.: Introduction to the theory of computation. Thomson Course Technology http://books.google.com/books?id=VJ1mQgAACAAJ (2006)

[150] SmartQVT: SmartQVT - a QVT implementation, Available: http://sourceforge.net/projects/smartqvt/ (online, 2011.10.28)

[151] Smiałek, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Wolter, K., Hotz, L., Mukasa, K.S., Jedlitschka, A., Bildhauer, D., Falkowski, K., Haas, J., Horn, T., Riediger, V., Schwarz, H., Kalnins, A., Kalnina, E., Sostaks, A., Celms, E., Rein, M., Drejewicz, S., Knab, S., Falb, J., Çetin, S., Tüfekçi, O., Çokkeçeci, I.: Case-driven software development. Project Deliverable D8.2.2, ReDSeeDS Project www.redseeds.eu (2009)

[152] Smiałek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T.: Complementary use case scenario representations based on domain vocabularies. Lecture Notes in Computer Science 4735, 544–558 (2007)

[153] Smiałek, M., Kalnins, A., Kalnina, E., Ambroziewicz, A., Straszak, T., Wolter, K.: Comprehensive system for systematic case-driven software reuse. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorn'y, J., Rumpe, B. (eds.) SOFSEM 2010: Theory and Practice of Computer Science. Lecture Notes in Computer Science, vol. 5901, pp. 697–708. Springer Berlin / Heidelberg (2010)

[154] SoftProject GmbH: xsl:easy 4.0, Available: http://xsl-easy.com/ (online, 2011.10.06)

[155] Sostaks, A., Kalnina, E., Kalnins, A., Celms, E., Iraids, J.: Solving the TTC 2011 reengineering case with MOLA and higher-order transformations. In: Van Gorp et al. [187], pp. 159–167

[156] Sparx Systems Pty Ltd: Sparx Systems Enterprise Architect User Guide (2007)

[157] Sprogis, A.: The Configurator in DSL Tool Building. In: Scientific Papers, University of Latvia. vol. 756, pp. 173–192 (2010)

[158] Sriganesh, R., Brose, G., Silverman., M.: Mastering Enterprise JavaBeans 3.0. Wiley Publishing (2006)

[159] Stahl, T., Voelter, M.: Model-Driven Software Development (Technology, Engeneering, Management). John Wiley & Sons (2006)
[160] Steinmüller, W.: Informationstechnologie und Gesellschaft: Einfuhrung in die angewandte Informatik. Wissenschaftliche Buchgesellschaft (1993)
[161] Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., Yusuf, L.: Patterns : Model-Driven Development Using IBM Rational Software Architect. IBM http://www.redbooks.ibm.com/abstracts/sg247105.html (2005)
[162] Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating domain-specific model editors with complex editing commands. In: Proc. AGTIVE 2007. Universität Kassel, Germany (October 2007)
[163] Taentzer, G.: AGG: A tool environment for algebraic graph transformation. In: Nagl, M., Schürr, A., Münch, M. (eds.) Proceedings of AGTIVE. LNCS, vol. 1779. Springer, 481–488 (1999)
[164] The Eclipse Foundation: Acceleo, Available: http://www.eclipse.org/acceleo/ (online, 2011.08.29)
[165] The Eclipse Foundation: ATL, Available: http://eclipse.org/atl/ (online, 2011.09.28)
[166] The Eclipse Foundation: Eclipse Modeling Framework Project (EMF), Available: http://www.eclipse.org/modeling/emf/ (online, 2011.08.29)
[167] The Eclipse Foundation: Eclipse.org, Available: http://www.eclipse.org/ (online, 2011.08.27)
[168] The Eclipse Foundation: Edapt, Available: http://www.eclipse.org/edapt/ (online, 2011.08.04)
[169] The Eclipse Foundation: Epsilon, Available: http://www.eclipse.org/gmt/epsilon/ (online, 2011.08.29)
[170] The Eclipse Foundation: EuGENia, Available: http://www.eclipse.org/gmt/epsilon/doc/eugenia/ (online, 2011.08.29)
[171] The Eclipse Foundation: Graphical Editor Framework (GEF, Eclipse Tools Subproject), Available: http://www.eclipse.org/gef (online, 2011.08.29)
[172] The Eclipse Foundation: Graphical Modeling Framework (GMF, Eclipse Modeling subproject), Available: http://www.eclipse.org/gmf (online, 2011.08.29)
[173] The Eclipse Foundation: Henshin, Available: http://www.eclipse.org/modeling/emft/henshin/ (online, 2011.08.29)
[174] The Eclipse Foundation: Jet, Available: http://www.eclipse.org/modeling/m2t/?project=jet (online, 2011.08.29)
[175] The Eclipse Foundation: Model to model (m2m), Available: http://www.eclipse.org/m2m/ (online, 2011.08.29)
[176] The Eclipse Foundation: Mofscript, Available: http://www.eclipse.org/gmt/mofscript/ (online, 2011.08.29)
[177] The Eclipse Foundation: TCS (Textual Concrete Syntax), Available: http://www.eclipse.org/gmt/tcs/ (online, 2011.08.29)
[178] The Eclipse Foundation: UML2, Available: http://www.eclipse.org/modeling/mdt/?project=uml2 (online, 2011.09.28)
[179] The Eclipse Foundation: UMLX, Available: http://www.eclipse.org/gmt/umlx/ (online, 2011.08.29)
[180] The Eclipse Foundation: Viatra2, Available: http://www.eclipse.org/gmt/VIATRA2/ (online, 2011.09.01)
[181] The Eclipse Foundation: Xpand, Available: http://www.eclipse.org/modeling/m2t/?project=xpand (online, 2011.08.29)

[182] Tisi, M., Cabot, J., Jouault, F.: Improving higher-order transformations support in ATL. In: Tratt, L., Gogolla, M. (eds.) Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6142, pp. 184–198. Springer (2010)

[183] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications. ECMDA-FA '09, vol. 5562, pp. 18–33. Springer-Verlag, Berlin, Heidelberg (2009)

[184] TTC 2010: Winners / Awards 2010 (2010), Available: http://planet-research20.org/ttc2010/index.php?option=com_content&view=article&id=103&Itemid=128 (online, 2011.10.16)

[185] Universidad de Murcia: Agile Generative Environment (AGE), Available: http://gts.inf.um.es/trac/age (online, 2011.08.29)

[186] Van Gorp, P., Mazanek, S.: SHARE: a web portal for creating and sharing executable research papers. Procedia Computer Science 4, 589–597 http://linkinghub.elsevier.com/retrieve/pii/S1877050911001207 (2011)

[187] Van Gorp, P., Mazanek, S., Rose, L. (eds.): TTC 2011: Fifth Transformation Tool Contest, Zürich, Switzerland, June 29-30 2011, Post-Proceedings. EPTCS (2011)

[188] Vilitis, O.: Metamodel-based transformation-driven graphical tool building platform. Ph.D. thesis, University of Latvia (2009)

[189] Völter, M.: MD*/DSL Best Practices update march 2011 (April, 2011), Available: http://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf (online, 2011.09.15)

[190] Völter, M., Bettin, J.: Patterns for model-driven software-development (May 2004), Available: http://www.voelter.de/data/pub/MDDPatterns.pdf (online, 2011.09.15)

[191] W3C: RDB2RDF working group, Available: http://www.w3.org/2001/sw/rdb2rdf/ (online, 2011.10.29)

[192] W3C: OWL Web Ontology Language Overview http://www.w3.org/TR/owl-features (February 2004), http://www.w3.org/TR/owl-features

[193] W3C: Resource Description Framework (RDF): Concepts and Abstract Syntax http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (February 2004)

[194] W3C: XSL Transformations (XSLT) Version 2.0 (January 2007)

[195] W3C: R2RML: RDB to RDF Mapping Language, w3c working draft edn. (September 2011)

[196] Wikimedia: File:carte métro de paris.jpg, Available: http://en.wikipedia.org/wiki/File:Carte_M%C3%A9tro_de_Paris.jpg (online, 2011.10.28)

[197] Willink, E.D.: A concrete UML-based graphical transformation syntax: The UML to RDBMS example in UMLX. In: Workshop on Metamodelling for MDA. University of York, England (2003)

[198] Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Surviving the heterogeneity jungle with composite mapping operators. In: Tratt, L., Gogolla, M. (eds.) Proceedings of the Third international conference on Theory and practice of model transformations. Lecture Notes in Computer Science, vol. 6142, pp. 260–275. Springer-Verlag, Berlin, Heidelberg (2010)

[199] Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences. pp. 285b–. HICSS ’07, IEEE Computer Society, Washington, DC, USA (2007)

APPENDIX A

List of Acronyms

A

API – Application Programming Interface

ATL – Atlas Transformation Language

AMW – Atlas Model Weaver

B

BPEL – Business Process Execution Language

BPMN – Business Process Modelling Notation

C

CIM – Computation Independent Model

CRUD – create-reade-update-delete

CTE – Code Template Framework

D

DSL – Domain-Specific Language

DSM – Domains-Specific Modelling

DSML – Domains-Specific Modelling Language

E

EA – Enterprise Architect

EOL – Epsilon Object Language

EMF – Eclipse Modeling Framework

EMOF – Essential MOF

G

GEF – Graphical Editing Framework

GMF	Graphical Modeling Framework

H

HOT	Higher-Order Transformation

J

JSP	JavaServer Pages

L

LUMII	Latvijas Univeristātes Matemātikas un informātikas institūts (Institute of Mathematics and Computer Science University of Latvia)

M

MALA4MDSD	Mapping Language for MDSD
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MD*	Model-Driven Everything
MOF	Meta Object Facility
MOLA	MOdel transformation Language
MOps	Mapping Operators
MTBE	Model Transformation By Example
MVC	Model-View-Controller

N

NAC	Negative Application Condition

O

OCL	Object Constraint Language
ODM	Ontology Definition Metamodel
OMG	Object Management Group

OOP – Object-Oriented Programming

OOPL – Object-Oriented Programming Language

ORM – Object-Relational Mapping

OWL – Web Ontology Language

P

PIM – Platform Independent Model

POJO – Plain Old Java Object

PSM – Platform Specific Model

Q

QVT – Query/View/Transformation

R

RDB – Relational DataBases

RDF – Resource Description Framework

RDFS – RDF Schema

RDP – Remote Desktop Protocol

ReDSeeDS – Requirement Driven Software Development System

RFP – Request For Proposal

RSL – Requirement Specification Language

T

TDA – Transformation-Driven Architecture

U

UI – User Interface

UL IMCS (LUMII) – Institute of Mathematics and Computer Science University of Latvia

UML – Unified Modelling Language

W

W3C	World Wide Web Consortium
WSDL	Web Services Description Language

X

XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations
XSD	XML Schema Definition