University of Latvia

AGRIS ŠOSTAKS

IMPLEMENTATION OF MODEL TRANSFORMATION LANGUAGES

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 – 2010

TABLE OF CONTENTS

INTRODUCTION

Model transformations play an important role in the Model-Driven Software Development (MDSD). The main idea of MDSD is a systematic use of **models** as primary software engineering artefacts throughout the software development lifecycle. Model-Driven Software Development refers to a range of development approaches that are based on the use of software modelling. A model expresses a particular aspect of a software system in a certain level of detail. A code of the software system is generated from models built by a system developer. The generated code varies ranging from a system skeleton to a complete product. It depends on an abstraction level of models used as a source for the generator. If the created models are at high level of abstraction, then **model transformations** are applied to create more detailed models that can be used for code generation. The model transformation is the automatic generation of a target model from a source model, according to a transformation definition [1]. Model transformation languages are used to define model transformations. Models that are used by model transformations must conform to **metamodels**. A metamodel defines a language, which specifies a model. A model transformation language uses metamodels to define the model transformation. A meta-language specifies the metamodels.

The best known Model-Driven Software Development initiative is the Object Management Group's (OMG) Model-Driven Architecture (MDA) [2], which is a registered trademark of OMG. The OMG has developed a set of standards related to MDA including the Meta-Object Facility (MOF) [3] (a meta-language), Object Constraint Language (OCL) [4], Unified Modelling Language (UML) [5] (a software modelling language) and MOF Queries/Views/Transformations (MOF-QVT) [6] (a model transformation language).

The MDA approach defines system functionality using a platform-independent model (PIM), which is written in an appropriate modelling language (for example, UML). Then, the PIM is transformed to one or more platform-specific models (PSMs), which include platform or language specific details. For example, the UML Profile for Java [7] can be used to specify the PSM. Then, the PSM is translated to the code written in the appropriate to the PSM language.

Nowadays the application area for model transformation languages is much broader. One such area is generic meta-model-based modelling tool building. The model

transformation languages can be used (and are used [8][9][10]) as a much more effective domain specific substitute for the general purpose languages which are used up to now for tool building.

The OMG was the first to state precisely the requirements what should be a model transformation language [11]. The MOF-QVT language which is an answer by OMG itself to these requirements is becoming the OMG standard for model transformations [6]. In MOF-QVT source and target meta-models conform to the MOF. There are two variants of MOF defined – the EMOF (Essential MOF) and the CMOF (Complete MOF). The MOF can be viewed as a general standard to write metamodels, but, more specifically, EMOF is used for metamodel definition in MOF-QVT. The MOF-QVT standard defines two languages for transformation development – the *Relations* and the *Operational Mappings*. The *Relations* language is at the highest level of abstraction and uses patterns and a declarative transformation definition style whenever possible. There are several realizations of the MOF-QVT language. The *Relations* textual language is implemented in the *medini QVT* [12]. The *Operational Mappings* language is implemented in the *SmartQVT* [13], several less complete implementations are also available.

There are many other model transformation languages which also satisfy the OMG requirements. There are textual model transformation languages – ATL [14], VIATRA2 [15], the Lx language family (L0-L3) [16] and also graphical model transformation languages – Fujaba [17], GReAT [18], MOLA [19]. In fact, model transformation languages existed even before the OMG coined this concept. There are several such graph transformation languages that are now being used as the model transformation languages, for example, AGG [20] and PROGRES [21].

Model transformation languages are becoming increasingly mature in recent years and range of the areas where transformation languages are being used is widening. The growing popularity of transformation languages puts stricter requirements on their efficiency. Most of the popular transformation languages are using declarative pattern definition constructs. The main implementation problem of such languages is **the pattern matching**. This problem, in fact, is the *subgraph isomorphism problem* which is known to be NP-complete [22]. However, in practice typical patterns can be matched efficiently using relatively simple methods. The use of different means of pattern definition results into different implementations of pattern matching for every language. The more

sophisticated constructs a language use, the more complicated becomes the implementation of the pattern matching.

Research carried out by the author seeks for relatively simple but efficient algorithms for pattern matching in model transformation languages used in the MDSD area. The main results of this research are algorithms which allow building efficient implementation of pattern matching for typical model transformation languages. Solutions for implementation of model transformation language MOLA demonstrate applications of these algorithms.

The most straightforward proof of the practical significance of research is the successful use of MOLA language and tool in EU 6th framework project ReDSeeDS [23] (Requirements-Driven Software Development System) which is aimed to develop methodology and supporting tools for a model-driven software development. Transformations in ReDSeeDS are specified using MOLA language and represent typical MDSD transformations.

Another main use case of MOLA language and tool is the transformation based tool building framework METAclipse [8]. Transformations are used to define the logic of a tool built by METAclipse framework. In fact, MOLA Tool itself has been built using MOLA language.

The research results presented in the thesis have achieved the desired efficiency in implementation of pattern matching for model transformation languages. Thus it has become possible to apply MDSD technology in research projects and verify these technologies in industrial cases.

Following chapters give an in-depth description of the developed pattern matching algorithms and its implementations for model transformation language MOLA:

- CHAPTER 1 briefly describes the main ideas besides MDSD and the role of model transformation languages in this process of software development. The reader is thus given the basic knowledge needed for understanding the research carried out by the author, as well as the significance of the results achieved.
- CHAPTER 2 briefly describes the model transformation language MOLA. The algorithms developed in thesis are used in the implementation of MOLA language.

- CHAPTER 3 sketches existing algorithms of pattern matching in model transformation languages. The applicability of these approaches to MOLA language is also discussed in this chapter.
- CHAPTER 4 introduces a new algorithm which uses relational database with fixed schema and translates patterns to SQL queries. The implementation of this algorithm for MOLA language is described here.
- CHAPTER 5 introduces two new algorithms of pattern matching which uses local search plan generation strategy. The first algorithm is effective for typical MDSD tasks and is based on few simple rules. Therefore the implementation of this algorithm for MOLA language is rather simple using an Lx model transformation language family. The second algorithm is based a classical local search plan generation, but introduces a new metamodel annotation mechanism which allows to enhance the efficiency of pattern matching without complicated analysis of underlying models. This chapter provides also details of MOLA implementation through L3 language.
- CHAPTER 6 demonstrates practical applications of the developed implementation of MOLA language: typical MDSD transformations in the EU 6th framework project ReDSeeDS and defining tool logic in tool building framework METAclipse.
- CHAPTER 7 lists the conclusions accumulated during the development of the thesis. Also, possible future directions of the research in implementation of model transformation languages.

CHAPTER 1

Motivation - MDSD and Model Transformation Languages

Nowadays 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 which describe different aspects of the system which is to be developed.

At first models were used as demonstrative documentation which would help to develop the system. MDSD (*Model-Driven Software Development*) is a rather new approach (emerged around the year 2000) which uses models in a broader context. This chapter explains the basic principles of MDSD and the role of metamodels, models and model transformations in this process.

1.1 Modelling, Models and Metamodels

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. There are many definitions available, but in the author's opinion the most adequate definition of modelling is the following – modelling means using something instead of something else with a definite purpose [24]. Therefore, it allows using **a model,** which is simpler, safer, and also cheaper, instead of something else that is more complicated, dangerous or more expensive.

Regarding the processes of software development the term *model* is usually applied to the abstraction of a computer system or real world in a specific context, for example, a requirements specification of the system or description of business processes of a company can be regarded as a model of the system and the company. These models let judge and draw conclusions about the system or the company. The requirements specification allows evaluating the complexity of the system and serves as the basis for software development. However, the model of business processes allows understanding the processes that take place in the company and optimizing business activities of the company. Usually we use a language as a mean for writing models, and this mean is specific for a certain type of models. It means, when we use a modelling language, it is

possible to describe different things of one type in a similar way. For example, when using the business process modelling language it is possible to describe various business processes in a number of companies.

In order to be able to process the models by using computers, it is necessary to formalize the way of model definition, which means that there must be some means (preferably universal ones) available how to define modelling languages. And these means are called **metamodels**. Generally speaking, a metamodel describes a modelling language- it is a model of a modelling language. A metamodel and a model together form two levels of metamodelling abstraction or meta-levels, where the higher meta-level describes the means which help in forming the lower level. Theoretically, there could be an unlimited number of such meta-levels, but only four are used in practice.

As it has been previously stated, a metamodel is also a model, so, in order to describe metamodels, we use a modelling language. This language is usually called a metamodelling language and it is defined by making use of a metamodel which is commonly called **a meta-metamodel**. Thus models *reside* at the first level of metamodelling or Level M1, the system that they describe, *resides* at the zero level or Level M0. The metamodel describing a model, *resides* at the second level or Level M2, but at the top-level, that is the third level or Level M3, the meta-metamodel *resides*.

At present the most popular metamodelling standard (language) is MOF *(Meta-Object Facility),* developed by the international standards organisation OMG, which describes four meta-levels (see Fig. 1). Currently the actual MOF version is 2.0 [3].

In practice many models are described by using one of versions of the modelling language UML [5], developed by OMG, (in Fig. 1 UML language is used to illustrate the MOF standard). Naturally, UML metamodel is defined by using MOF metamodelling language. It must be noted that MOF does not define the visible part of the language (*concrete syntax*), but it defines its abstract syntax. Of course, this is not the only metamodelling language. There exist other ways of defining metamodels, such as *KM3* [25] and EMF *Ecore* [26] - the metamodelling languages which are compatible with *EMOF* [3], a subset of MOF. In order to define a modelling language, one can use also ontology [27].

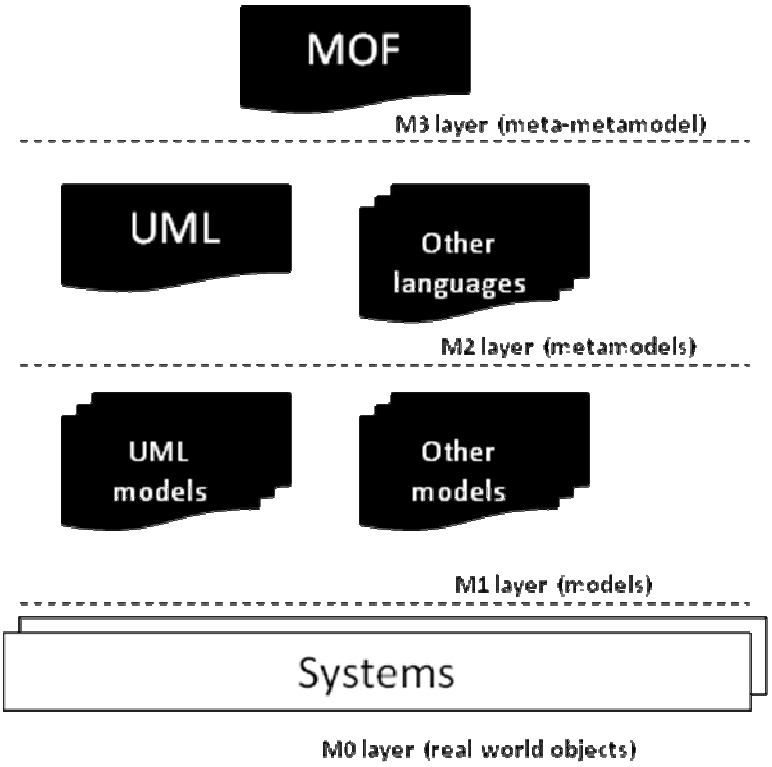


Fig. 1. Example of OMG MOF meta-level hierarchy.

There exist not only graphical, but also many textual modelling languages (actually, any OO programming language can be considered to be a modelling language).

In order to illustrate what models can be encountered during the software development process there are some typical examples of the models:

- UML class diagrams- the system analysis model,
- UML activity diagrams- the business process model,
- UML use case and activity diagrams- the system requirements specification,
- UML class diagrams where J2EE stereotypes are used- the detailed design model,
- BPMN diagrams [28] - workflow definition.

1.2 Model Driven Software Development

Until 2000, when OMG launched a new initiative *Model Driven Architecture* (MDA), many MDA ideas were already being used in practice. However, it was happening more intuitively rather than systematically. In 2001 OMG published the first version of MDA manual [2] which described basis and applications of MDA.

The essential MDA idea is the following: in order to develop complicated software, it is necessary to exploit various metamodelling principles systematically. An important conclusion followed that models had different roles during the development process of software. The following three roles of models were offered:

- *Computation Independent Model* – **CIM**, a model which describes what system must do (requirements) and in what environment the system must work (for example, business processes), but it does not imply any information about implementation of the system.
- *Platform independent model* – **PIM**, a model which describes the architecture of the system, but does not imply any details about the platform, in which system is going to be built (for example, .NET, EJB, CORBA specific details).
- *Platform Specific Model* – **PSM,** a model which contains specific details for the platform.

These models are used **successively**, that is, at first CIM model is made, and then it is supplemented or transformed, so that PIM is obtained, after that PIM is supplemented with specific details for the platform, and finally the software code is obtained from PSM. In practice similar models were already used, but MDA offered to **automate** this process, that is, to perform automatic **model transformations**. In this way the models become an essential part of the software development process. Software developers are able to operate at a higher level of abstraction, which has a radical influence on quality and speed of development of complicated systems. It should be noted that this process does not require an absolute automation, and it is hardly possible here. Each model is updated manually and then it is changed by means of model transformation.

Thus, a model transformation is an automatic process when the source model, which corresponds to a fixed metamodel, is transformed into a target model, which corresponds to another (or the same) fixed metamodel (see Fig. 2). It must be noted that the model transformation itself is defined by using source and target metamodels.

In the classical MDA approach the software is developed in such a manner that there exist one PIM model and one or more PSM models from which a code for different platforms is generated, depending on needs of the developer. MDA allows using only UML language for model description.

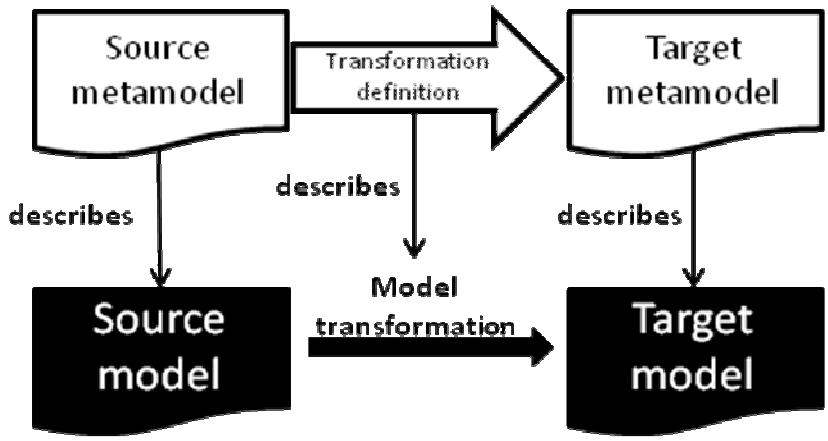


Fig. 2. An implementation scheme for model transformations.

However, MDSD (*Model Driven Software Development*) views this issue in a broader context. The development process does not fix the usable modelling languages and allows applying also arbitrary formalized means of metamodelling. However, the majority of metamodels is set by means of MOF or compatible metamodelling language. MDSD does not strictly regard the roles of models and views the development process as a successive development of models by taking advantage of model transformations. Thus one can consider that MDA is a specific case of MDSD that is worth mentioning because it is the basis of all these ideas. It must be noted that the specialized modelling languages-DSL (*Domain Specific Language*), have become exceedingly popular. They are used for modelling specific fields, for example, a language for automobile servicing software development (AUTOSAR [29]), mobile telephone software development [30], and many other. These languages increase efficiency of software development in these fields. Also models and model transformations are increasingly used in implementation of the DSL languages.

1.3 Model Transformation Languages

The previous chapter concluded that automatic model transformations are one of the most essential parts in the process of model driven software development (MDSD). Model transformation turns one model into another in accordance with a specific definition of model transformation (see Fig. 2). The definition of model transformation can be stated as a program which is written by using one of the existing software languages, however, operating with models, which are described by means of metamodels, creates specific requirements for this language. It turned out that in practice

the existing software languages are not really suitable for defining model transformations. Therefore, in 2002 OMG announced a request for proposals about development of a standard for a new type of software language, **model transformation language**- *QVT (Queries/ Views/ Transformations)* [11].

What and how can we describe with a model transformation language? The essential requirement for the model transformation language is the ability to process the models, which are set by means of the metamodels (in concept of OMG- only with MOF), that is, by means of this language one must be able to work with a set of instances of the metamodels (classes), as well as recognise and change them. It is also essential that definitions of model transformations must be *understandable* for both the human and computer- they must be as declarative as possible. Of course, there must be an appropriate tool support available for a successful application of the language.

As a part of OMG request for proposals there were submitted several language standard projects. However, over the years they have merged, and now there is one standard project left- MOF QVT. In the development of this project 16 institutions participate, including *IBM, Sun* and four universities. Although according to the plan the standard language had to be ready by March 2005, the first version of the standard *MOF QVT 1.0* was issued only in April 2008. At the moment the actual version is *MOF QVT 1.1 Beta 2* [6], which has been issued in December 2009.

Simultaneously with MOF QVT, a number of model transformation languages are being developed, not directly related to the OMG request for proposal - *MOLA* [19], *Lx* [16], *GReAT* [18], *UMLX* [31], *ATL* [14], *Tefkat* [32], *MTF* [33], *ATOM*^{3} [34], *VMTS* [35], *BOTL* [36], *Fujaba* [17], *RubyTL* [37], *Epsilon* [38], *Henshin* [39]. Also graph transformation languages turned out to be suitable for solving MDSD tasks, therefore, such languages as *AGG* [20], *PROGRES* [21], *TGG* [40], *GrGen* [41], *VIATRA2* [15] were used for defining model transformations. In Chapter 2 of the thesis one can find out about the model transformation language MOLA. In this research MOLA is particularly emphasised, because the author of the thesis has participated in the development process of this language.

The significant number of various model transformation languages might seem surprising, however, there must be regarded several conditions, which initiated the development of these languages. Firstly, lots of tasks emerged that were easier to solve by means of model transformations. Therefore, each of the previously mentioned

transformation languages is suitable for solving a particular class of tasks. For example, MOLA is suitable for MDSD tasks, but VIATRA2- for development of model driven simulation software. Secondly, the model transformation standard MOF-QVT does not have a completely developed implementation. Now *MOF-QVT Operational* is supported by *SmartQVT* tool [13] and *Eclipse M2M QVT Operational* project [42]. But *MOF-QVT Relational* is partly implemented by *MediniQVT* [12] tool. Therefore, the standard is mostly used as documentation, but in practice other model transformation languages are being used.

One of the most popular means, which is used in model transformation languages, is a model **pattern**. The pattern is a declarative means. It helps in setting the metamodel fragment, to which a corresponding model fragment must be found. The located model fragment is supplemented, corrected or deleted according to the proper transformation algorithm. The pattern and the executable operations together form the **rule** of transformation. Consequently, the definition of model transformation is made by a set of rules written in the model transformation language. Patterns are used by many transformation languages, such as MOF-QVT, MOLA, GReAT, ATOM^3, Fujaba, AGG, PROGRES, VIATRA2, and GrGen. However, the means that are used in them to provide the order of execution of rules is the essential factor that differentiates languages and states their suitability for solving different tasks.

Pattern matching is a process in the result of which a fragment of a model (a set of instances) is found which corresponds to the particular pattern. In general it is an NP-complete problem [22]; therefore an efficient implementation of pattern matching is an essential (even the most essential) precondition for an efficient implementation of the model transformation language.

CHAPTER 2

MOLA Language

Model transformation language MOLA is described in this chapter. The author of this thesis has actively taken part in the development of MOLA language and its implementation. Pattern matching algorithms developed by the author have been used in the implementation of MOLA. More about MOLA language can be found in [19], [43], [44] and in the web page of MOLA project [45].

MOLA is a graphical model transformation language, which is being developed by the Institute of Mathematics and Computer Science, University of Latvia, since 2003. Metamodels have been already used by IMCS [46], [47]; however the request of OMG for model transformation language proposal (QVT RFP [11]) was the determinant to start the development of a new language. The goal of MOLA project is to provide a simple and easy readable (therefore graphical) model transformation language, which would cover the typical transformation applications in Model Driven Software Development (MDSD). The declarative rules are commonly used in MOLA transformations together with simple procedural control structures governing the order in which rules are applied to the model.

2.1 MOLA Overview

MOLA is a graphical model transformation language, which is used for transforming an instance of a source metamodel (the source model) into an instance of the target metamodel (the target model). A transformation definition in MOLA consists of the source and target metamodel definitions and one or more MOLA procedures.

Source and target metamodels are jointly defined in the MOLA metamodelling language, which is quite close to the OMG EMOF specification [3]. These metamodels are defined by means of one or more class diagrams, packages may be used in a standard way to group the metamodel classes. Actually, the division into source and target parts of the metamodel is quite semantic, as they are not separated syntactically (the complete metamodel may be used in transformation procedures in a uniform way). Typically, additional mapping associations link the corresponding classes from source and target metamodels; they facilitate the building of natural transformation procedures and document the performed transformations. The source and target metamodel may be the same – that is the case for in-place model update transformations. The MOLA metamodelling language is defined formally in the *Kernel* package of the MOLA metamodel (see Fig. 3).

MOLA procedures form the executable part of a MOLA transformation. One of these procedures is the main one, which starts the whole transformation. MOLA

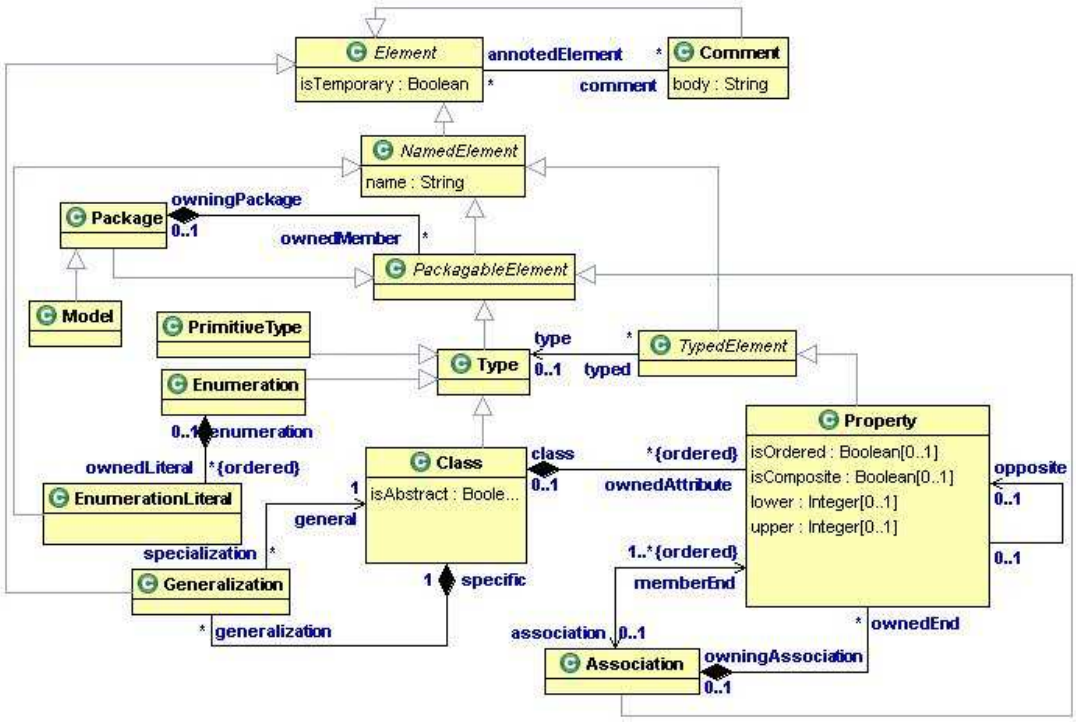


Fig. 3. The metamodel of the MOLA metamodelling language

procedure is built as a traditional structured program, but in a graphical form. Similarly to UML activity diagrams (and conventional flowcharts), control flow arrows determine the order of execution of MOLA statements. Call statements are used to invoke sub-procedures. However, the basic language statement of MOLA procedures is specific to the model transformation domain – it is the **rule**. Rules embody the **pattern matching** paradigm, which is typical of model transformation languages. Each rule in MOLA has the pattern and the action part. Both are defined by means of **class-elements** and **association-links**. A class-element is a metamodel class, prefixed by the element (*role*) name (graphically shown in a way similar to UML instance). An association-link connecting two class-elements corresponds to an association linking the respective classes in the metamodel. A pattern is a set of class-elements and -links which are compatible to the metamodel for this transformation. A pattern may simply be a metamodel fragment, but a more complicated situation is also possible – several class-elements may reference the same metamodel class – certainly, their element names must differ (these elements play different roles in the pattern, e.g., the start and end node of an edge). A class-element may also contain a constraint – a Boolean expression in a simplified subset of OCL. The main semantics of a rule is in its pattern match –a model fragment must be found, where an instance of the appropriate class is allocated to each class-element so that all required links are present in this fragment and all constraints evaluate to true. If such a match is found, the action part of the rule is executed. The action part also consists of class-elements and links, but typically these are create-actions – the relevant instances and links must be created. An end of a create-link may also be attached to a class-element included in pattern. Assignments in class-elements may be used to set the attribute values of the instances. Instances may also be deleted and modified in the action part. Thus a rule in MOLA typically is used to locate some fragment in the source model and build a required corresponding fragment in the target model. If several model fragments satisfy the rule pattern, the rule is executed only once (on an arbitrarily chosen match). Such a situation should be addressed by another related construct in MOLA – the loop construct. In addition, the reference mechanism (a class-element may be a reference to an already matched or created instance in a previous rule) is used to restrict the available match set. Thus, rules are typically used in MOLA in situations where at most one match is possible. Certainly, there may be a situation when no match exists – then the rule is not executed at all. To distinguish this situation, a rule may have a special *ELSE*-exit (a control flow

labelled *ELSE*), which is traversed namely in this situation. Thus, a rule plays in MOLA the role of an *if-then-else* construct as well.

Another essential construct in MOLA is the **loop** (more concretely, foreach loop). The loop is a rectangular frame, which contains one special rule – the **loophead**. The loophead is a rule which contains one specially marked (by a bold border) element – the **loop variable**. The semantics of a foreach loop is that it is executed for all possible matches for the loophead, which differ by instances allocated to the loop variable (possible variations for other loophead elements are not taken into account). In fact, a foreach loop is an iterator which iterates through all possible instances of the loop variable class that satisfy the constraint imposed by the pattern in the loophead. With respect to other elements of the pattern in the loophead, the *existential semantics* is in use – there must be a match for these elements, but it does not matter whether there are one or several such matches. Thus a foreach loop is the main MOLA construct, which is used to code a situation: "*for each instance of ... which satisfies ... perform the following transformation ...*". Namely such situations in informal descriptions of model transformations are frequently called transformation rules, but in MOLA they must be formalised as foreach loops. In addition to the loophead, a loop typically contains the loop body – other MOLA statements whose execution order is organised by control flows. The loop body is executed for each iteration of the loop. Since the loophead is a rule, it may also contain create actions, thus simple transformations of source model elements may be coded in MOLA by loops consisting of the loophead only. For nested loops the main organising feature is the possibility to reference the loop variable (and other elements) of the main loop in the pattern of the nested loophead, thus specifying an iteration over all related instances (to the current instance in the main loop).

There also are other available constructs in MOLA procedures. Procedures may have **parameters** (of type of a metamodel class or a primitive type) and local **variables** (also of both types). These elements may be used in MOLA rules. In addition, **text-statements** (consisting of a constraint and assignments) may be used to process these elements more directly. For primitive-typed variables the text statement is the only option. A text statement containing a constraint (a Boolean expression) may also have an *ELSE*-exit and serve as an *if-then-else* construct (in addition to rule). 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). It should be noted that

MOLA has no built-in UI support (MOLA is oriented towards behind-the-scenes transformations), therefore diagnostic messages and similar situations should be addressed via a library of external procedures. All MOLA procedure elements are defined formally in the *MOLA* package of the MOLA metamodel (see Fig. 4).

The execution of a MOLA transformation on a source model starts from the main procedure. A loop is executed while there are instances to iterate over. Then the next construct according to the control flow is executed. If a rule without a valid match is to be executed, and this rule 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 (within a loop body). When the main procedure reaches its end, the transformation is completed.

2.2 Simple MOLA Example

In order to illustrate the basic MOLA concepts, briefly listed in the previous section, a simple MOLA transformation example is provided. It is the *classical* example from an *abstract* MDA area – simplified UML class diagram to simplified database schema definition.

Let us assume that we have to build an initial part of the database schema definition – tables and columns from a class diagram. The source model (simple class diagrams) is described by a significantly simplified fragment of *Classes* package in the UML 2 metamodel (see Fig. 5). Though only the very basic elements in this source metamodel are retained, still it has the feature that a class attribute is represented by the `Property` metamodel class, and so are the association ends. Therefore each `Property` has to be analyzed, whether it really represents an attribute. All metamodel classes in this fragment are placed in the `Kernel` package. The `Class` metamodel class has one additional tag – the Boolean `isPersistent`, which is treated in this example as a *normal* attribute.

The target metamodel is even simpler – it contains only two classes `Table` and `Column`, both in the SQL package (see Fig. 5). The association `cols` expresses the ownership of `Column` by a `Table`, the association `pkey` – that the corresponding `Column` is a primary key for the `Table`.

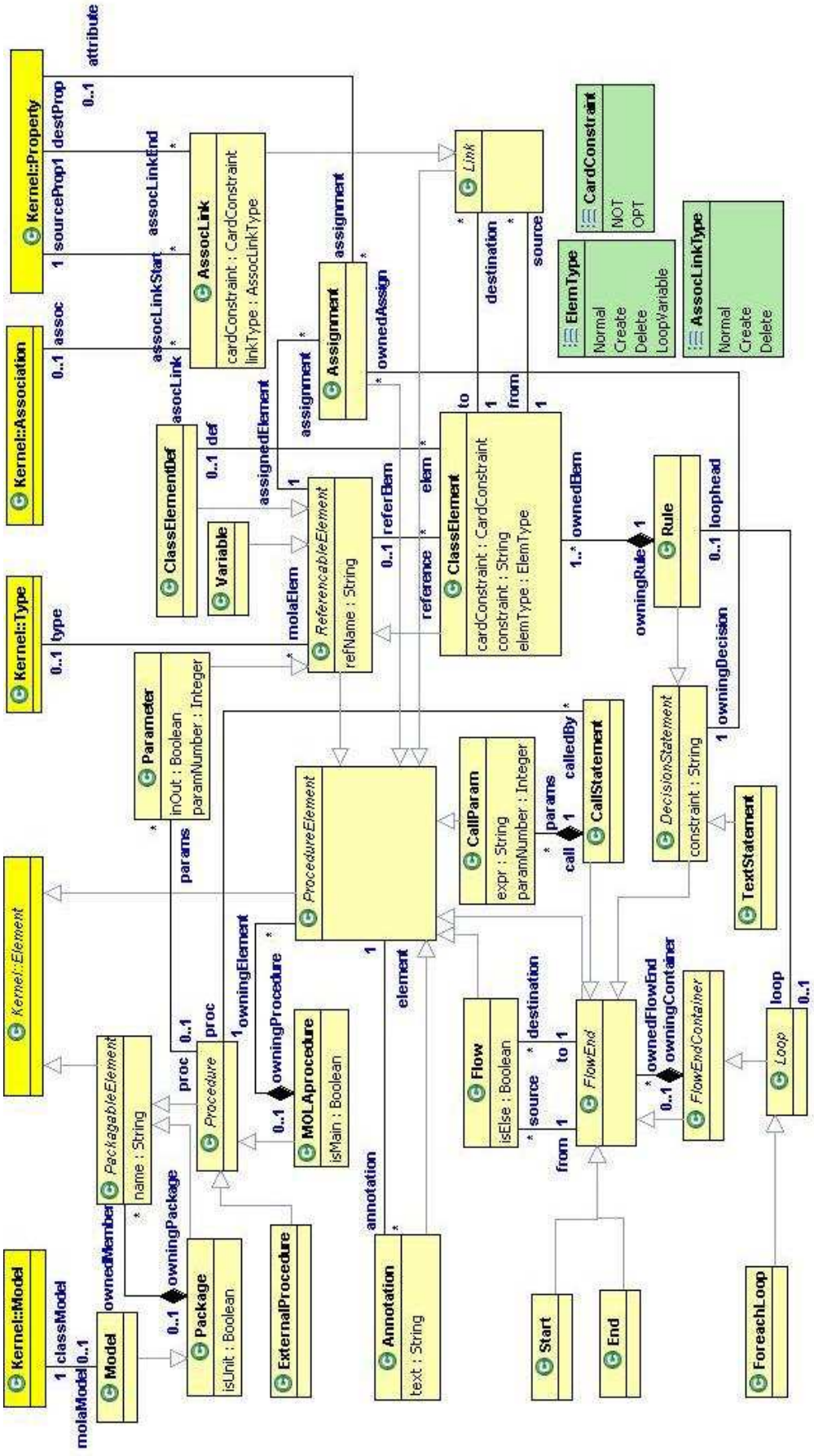


Fig. 4. The metamodel of the MOLA procedure elements

Two mapping associations link the source and target metamodels – `classToTable` goes from `Class` to `Table` and `attributeToColumn` from `Property`.

The transformation to be specified is the following – for each persistent class (i.e., `Class` instance) we have to build a `Table` and its primary key `Column` (with a specifically defined name and type String). For each attribute of such a class, whose type is a primitive one, we have to build a `Column` in the corresponding `Table` with the same type, but for an attribute with an `Enumeration` type – a `Column` with type String. The `Column` name coincides with the attribute name. Associations in this oversimplified example are not taken into account.

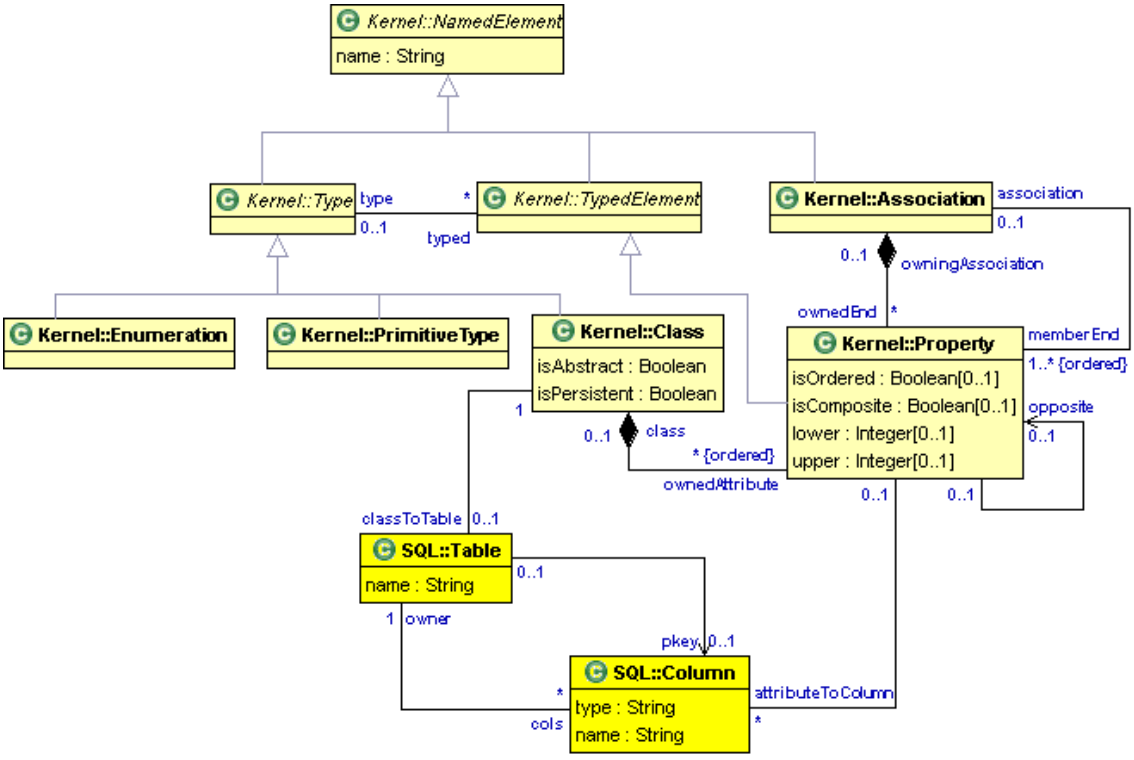


Fig. 5. The metamodel of the example

The metamodel example shows that metamodels are defined in MOLA in a standard way, by class diagrams, but only EMOF level facilities are permitted. Generalization is used in a standard way.

The transformation itself consists of two MOLA procedures – `Main` (which is really the main one) and `ProcessAttribute`, which is invoked by `Main`. Fig. 6 shows the procedure `Main`.

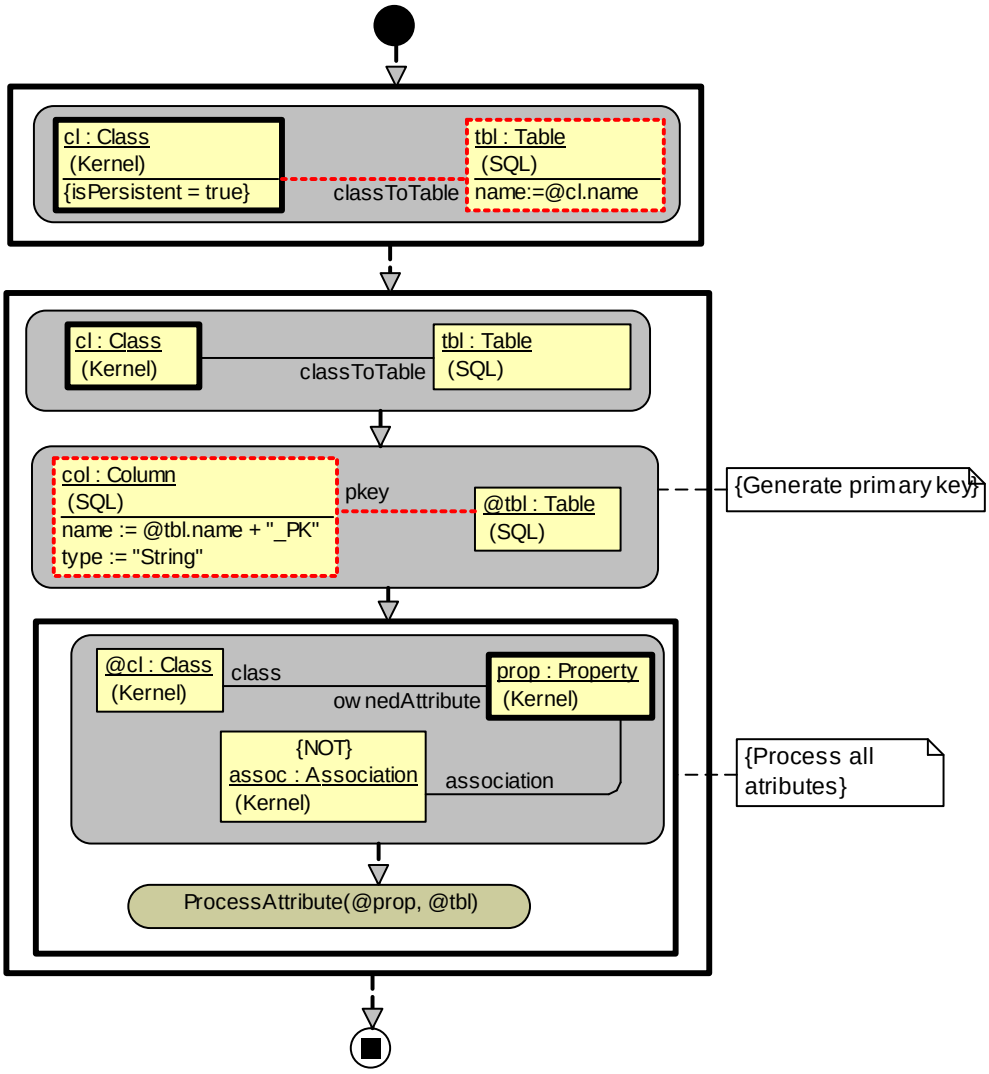


Fig. 6. The MOLA procedure *Main*

The start and end symbols of a MOLA procedure are represented in the same way as in UML activity diagrams. Control flows are drawn by dashed lines. The first element to be executed in this procedure is a foreach loop (a rectangle with bold lines). This loop consists of the sole loophead rule (a rule is visualized by a grey rounded rectangle). The pattern part of this rule (elements with black borders) contains only one class element – the loop variable `cl` corresponding to the metamodel class `Class` (loop variables are distinguished from ordinary elements by bold borders). This class element contains also a constraint specifying that the attribute `isPersistent` must have the value `true`. Thus, the semantics of this simple loop (and included pattern) is – the loop is executed for every instance of `Class` in the source model, where `isPersistent` has the value true.

The action part of the rule contains one class element `tbl:Table` and one link. The class element is of create type (red dashed borders), and it contains one assignment – the value `@cl.name` (the value of the attribute `name` in the matched element `cl`) must be assigned to the attribute name (of the `Table` instance to be created). The sole link in the rule is of create type too (a red dashed line) and corresponds to the mapping association in the metamodel (between the `Class` and `Table` classes). The correspondence between links in MOLA rules and associations in the metamodel visually is shown via role names, at least one of the role names must be present for a link and UML syntax rules for classes guarantee that a unique specification is possible (the MOLA reference shows that internally a link is directly related to an association). Thus, the first loop is iterated over all persistent `Class` instances in the source model and for each such instance a new instance of `Table` is created and its `name` attribute is set to the same String value as the name of the class. In addition, these two instances are linked by the `classToTable` link.

This first loop is a typical *design pattern* for simple transformations in MOLA – loop through the instances of a class in the source model and for each valid instance build something in the target model.

The control flow from the first loop leads to the next foreach loop, which again iterates over all classes in the source model (the loop variable is based on `Class`). However, this time the pattern is more interesting – it contains one more class element (`tbl:Table`) and one link connecting these elements. The semantics is very natural – only these instances of `Class`, which have a `classToTable` link to a `Table` instance, qualify as valid for iteration. Since this loophead has no actions, for each iteration immediately the first construct of the loop body – the next rule is executed. It should be noted, that actually the second loop is iterated over literally the same instances as the first loop (persistent classes), since namely for these instances the first loop has built the `Table` instance and the required link. Therefore in an *optimized program* for this example both loops could be merged in one. The two loops are retained in this example for demo reasons (to demonstrate a pattern for a loop) and because in a more realistic version (where associations also need to be transformed) namely this *two pass* approach can provide a solution.

The next rule in the loop body builds a `Column` instance (the primary key column), assigns the required values to its attributes and links this new instance to the `Table` instance located by the loophead. Note the use of **element reference** - `@tbl:Table` in the rule. The reference construct (an element notation prefixed by the "@" character) says that namely the instance found by a previous rule (here the loophead) must be used. *The previous rule* means the last (according to the execution order) rule, where the referenced element (without the "@" character) was matched in the rule pattern, or created in the action part. If a reference is used in a pattern, it means that no matching is done for this element, simply the known instance is used to build a constraint for other pattern elements, or the instance is used as an end point for the link to be built (this is the given case). The use of the reference as a qualifier for an attribute in an expression has the natural meaning – the attribute value of this instance is taken.

The next construct to be executed in the loop body is a nested loop. It uses the `Property` class for its loop variable and is meant to loop over the attributes of the current `Class` instance. The loophead contains a pattern, where the reference `@cl:Class` says that only the `Property` instances linked to this known instance must be iterated upon, in addition there must be no `Association` instance linked to a valid `Property` (by the `association` link). The **cardinality constraint NOT** is used in a pattern element to specify that an appropriately linked instance must not exist at all in the model (a NOT-constraint is available also on links in MOLA, but there it says only that a link must not exist). Let us remind that the NOT-constraint is required here to filter these `Property` instances, which are association ends. The initial part of the loop pattern – the loop variable linked to a reference from the owning loop pattern – is very typical to nested loops in MOLA.

The nested loop in its body has only one construct – the call of the subprocedure `ProcessAttribute` (which builds the required columns), using references to the known instances `prop` and `tbl` as parameters. Certainly, the types (classes) of these parameters must match the parameter definitions in the invoked procedure. Here the classes coincide, but subclass instances may also be supplied (as in OO programming).

This concludes the definition of the `Main` procedure. When all the relevant iterations are completed, this simple transformation has built the required tables and columns.

It remains to give some comments on the subprocedure `ProcessAttribute`, which is shown in Fig. 7.

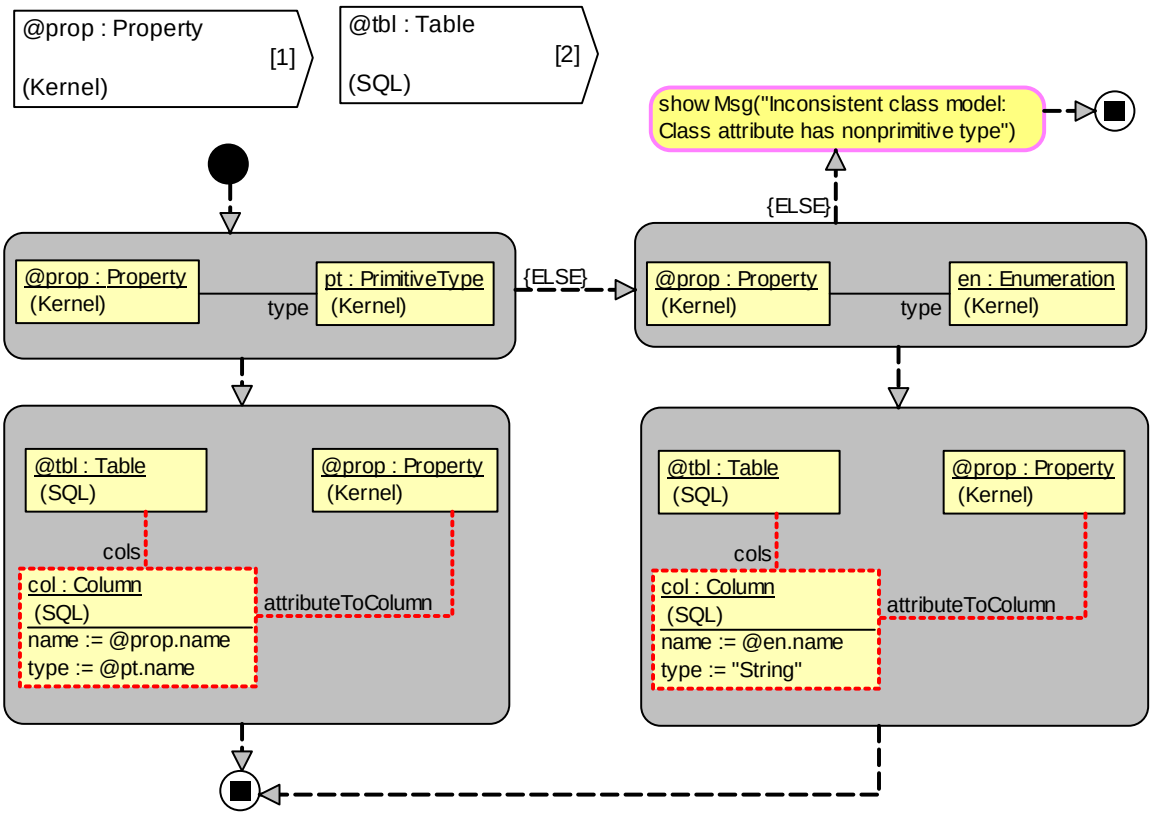


Fig. 7. The subprocedure `ProcessAttribute`

The two top symbols in the diagram are parameter definitions (their positions must be numbered, since calls use the positional notation). Parameters can be freely used in patterns, as element references would be.

This MOLA procedure has no loops, since the parameters already provide the exactly required instances. The first rule serves as a typical **if-condition** in an *if-then-else* construct. It is used to distinguish whether the attribute type is primitive or an enumeration. If the rule pattern matches (the type is primitive), the next rule (followed to via the unlabelled flow) builds the `Column` instance and sets its attributes. Note that in this rule the reference `@pt` is legal, since the previous rule has matched and located this instance (it would not be legal to use this reference in the other branch).

If the first pattern fails, the alternative rule (accessed via *ELSE-flow*) is executed. If its pattern matches, the alternative building rule for the enumeration case is executed. If the second condition fails too (e.g., the attribute type is another class), the external

procedure `showMsg` is invoked. This external procedure is built-in in MOLA environment and it is used to display a simple message box with the provided text.

CHAPTER 3

Pattern Matching in Model Transformation Languages

Besides MOLA there are many model and graph transformation languages which use declarative patterns to specify transformation rules. The language specific pattern features of several popular languages are described in this chapter. An overview of the most popular approaches for pattern matching implementation is also given in this chapter.

3.1 Patterns in Model Transformation Languages

The closest *relative* to MOLA in the world of model transformation languages is Fujaba Story Diagrams from Fujaba Tool Suite [17]. Fujaba is a graphical model transformation language which uses imperative control structures and declarative patterns. The specification of patterns in Fujaba is almost identical to MOLA. There is a restriction on patterns in Fujaba - the pattern must contain at least one bound (previously matched) element. The graphical syntax, of course, differs for both languages, but that is obvious for independently developed languages. The most significant difference between the two is the foreach loop. Fujaba does not specify the loop variable and loops are executed through all of the possible matches of the pattern. In MOLA only the distinct instances that correspond to the loop variable are iterated over. MOLA foreach loop is more readable and easier to use, because of the loop variable.

A different programming paradigm is used in the graph transformation language AGG [20], which is a typical example of a declarative transformation language. AGG does not have any imperative control structures, and rules that describe patterns are being executed independently. The only way to affect the execution order is to use layering. Each rule in AGG includes a pattern which is specified by LHS graph and NACs. NACs are used by declarative transformation languages mainly to distinguish already processed model elements. Negative patterns are used differently in MOLA because of the specific loop construct. MOLA also has negative pattern elements, but they are used to express a *logical* negative condition.

The graph transformation language PROGRES [21] is a textual graph transformation language where patterns (graph queries) are specified graphically. Patterns allow using similar and even richer options than previously noted transformation languages. The ordering of statements is managed by algebraic structures and PROGRES follows declarative PROLOG-like execution semantics.

Graph transformation language VTCL (Viatra Textual Command Language), which is part of the VIATRA2 framework [15], defines patterns using textual syntax. VIATRA offers broad possibilities for the pattern definition: negative patterns may be at arbitrary deep level; the call of a pattern from another pattern and even recursive patterns are allowed; the language may work both with model and metamodel. The execution order of rules is managed by ASM (Abstract State Machine) language constructs which are purely imperative. VIATRA has a rudimentary graphical syntax of patterns, however it seems that whole expressiveness of the language may not be available there.

Another textual graph transformation language, which has appeared in recent years, is GrGen [41]. The expressiveness of patterns in this transformation language is close to VIATRA. Transformation rules are combined using similar algebraic constructs to PROGRES (except the PROLOG-like execution semantics).

3.2 Related Pattern Matching Implementations

The authors of the graph transformation language PROGRES already in 1998 [48] were the first ones who examined the pattern matching issue in the context of transformations. Since then this issue has been solved in several graph and model transformation languages. Let us look at the most popular ways how pattern matching is being implemented in different transformation languages.

One of the most popular ways of implementation of pattern matching is by generating the **local search plans**. The basic idea of this approach is the following: in the optimal way finding a fragment, which corresponds to the pattern, by using the *basic* lookup operations (such as to find the first instance of a certain class; to find the instance of a certain class when navigating the link; to check the attribute value of a certain instance, etc., that actually is executed in almost constant time). By means of the basic operations a model fragment corresponding to the pattern is *built*. Usually the process starts from a potentially suitable class instance, and gradually the fragment of the model

is supplemented in correspondence with the pattern, that is, the rest of the instances are chosen so that they form a suitable component of the fragment wanted. If it is impossible to find a suitable instance, backtracking takes place. The search continues until a suitable fragment is found, or all potential fragments are checked, but none of them is suitable. The local search plan (LSP) is the order in which the basic operations are applied. The aim of LSP generation is to find such an optimal order which uses the basic operations as few as possible in order to find a model fragment corresponding to the pattern.

So, to find the best LSP, typically different *heuristics* are used which help to choose the optimal implementation order of the operations. The most typical version is to use *cardinalities (multiplicities*) of a metamodel element, usually an association, for example, the instances matching the pattern are *navigated* in such a way that mostly *navigation* takes place along the link towards the end of the association with a cardinality `0` or `0..1`. In this way the set of instances that should be checked is radically diminished. Implementation of the graph transformation language PROGRES [48] is based exactly on this principle. However, the cardinalities of the metamodel elements do not depict in full the real cardinalities in a specific model. For example, the cardinality * of the association end indicates that there can be more than one link to match, but it does not provide more precise information. It is possible to obtain more detailed evaluation of the cardinalities of certain model elements by analysing typical models where transformations with given patterns are used. This type of analyses can be performed in VIATRA language implementation [49]. This approach is suitable when a proper amount of corresponding models is available. However, in practice there are frequent situations when transformations must be built before any model is available. It is possible to obtain more precise values of the certain cardinalities exactly before the execution of the transformation, by examining the model which is going to be changed. In this case this information must be provided by the model repository, but it is not always done. In this case also the search plan must be generated during the execution process that can diminish the efficiency of the method and make the implementation more complicated. This method is used in the implementation of the transformation language *GrGen* [50].

The transformation language Fujaba uses a simpler LSP generation strategy. Pattern matching always starts from an instance corresponding to the bound pattern element (it exists always). Searching continues along the links in accordance with the

pattern [51]. Despite this approach being simple, it works almost as well as the already mentioned approaches [52].

Also in MOLA implementation a similar approach is used [53], described in detail in Chapter 5 of the thesis. MOLA uses also a more complicated LSP generation algorithm which employs the cardinalities of the metamodel elements and the mechanism of the metamodel annotations which lets the transformation writer use his knowledge about the real cardinalities in the models [53]. Also this approach is discussed in detail in Chapter 5 of the thesis.

LSP generation is not the only way of solving the pattern matching problem. In order to solve this problem it is possible to use other popular technologies and methods. One of these technologies is the **relational databases**. The basic idea of the method is to save the model in the relational database in accordance with some database scheme and carry out pattern matching by means of SQL queries. In this way the optimization mechanisms of query execution are exploited which are accessible in all well-known relational database management systems. Implementation of this method is rather simple, as it is possible to build an SQL query correspondent to the pattern or a chain of queries. Its execution, that is, the most complicated part, can be left to the query optimization algorithms. This approach is used in one of implementations of the transformation language VIATRA [54]. The model is saved in the relational database whose schema corresponds to the metamodel which describes this model. Thus the schema of the database is generated corresponding to each metamodel. For each pattern several SQL views are generated which correspond to the pattern and negative conditions. Pattern matching reduces to execution of SQL queries corresponding to the views. Relational database is used also in implementation of MOLA language [55], which is discussed in detail in Chapter 4 of the thesis. Unlike the previously mentioned implementation of VIATRA language, in this case the fixed database schema is used and exactly one SQL query for each pattern.

It is possible to **reduce** pattern matching **to CSP** (*Constraint Satisfaction Problem*). CSP has ready-made solutions which make solution of pattern matching possible. CSP is defined as a set of variables which must find a state, satisfactory for number of constraints. The typical examples are game *Sudoku* [56] and map colouring problem [57]. The search of such condition is called variable ordering and this process is rather similar to generation of the search plan in LSP methodology. Thus pattern elements

receive the corresponding CSP variables and a set of constraints, and if they are solved, also the corresponding pattern matching problem is solved. This solution is used in implementation of AGG language [58].

The previously described solutions are trying to find the corresponding model fragment in time, which depends on the size of the model (number of instances) and on the size of the pattern (number of pattern elements). **Incremental pattern matching** allows finding the corresponding model fragment for a pattern in constant time. The basic idea of this method is *cache* the fragments corresponding to the pattern, and when model changes, update this information. But *cache* requires additional memory resources. In this case changing the model is inefficient, because in case of any change, the information about the model fragments corresponding to the pattern must be updated. The typical MDSD transformation model is being changed constantly. There must be created the corresponding element in the target model practically for each element of the source model. It must be noted that before the execution of the transformation, when loading the model into the memory, the *cache* process must be performed and it needs a definite time of execution. Because of these reasons incremental pattern matching is not suitable for MOLA language. This approach is implemented in VIATRA language [59] and it works very successfully in solving tasks when the number of transformations is small and local. VIATRA incremental pattern matcher is built by using *RETE* networks [60].

The authors of VIATRA offer also **hybrid pattern matching** [61] which is able to combine different approaches, for example, LSP generation and incremental pattern matching. This approach offers to choose which method to use for a specific pattern. The choice can be made during transformation development or execution. It is based on the statistics of the available memory.

Patterns in the popular model transformation language ATL [14] are *hidden* within Boolean expressions of OCL language and helper functions widely used by ATL. ATL and MOF QVT [6] are not addressed here, because to our knowledge no pattern matching implementation details are available for them.

CHAPTER 4

Implementation of MOLA using Relational Databases and SQL

The pattern matching algorithm which uses relational database is described in this chapter. The implementation of this algorithm for model transformation language MOLA is one of the main results of these thesis. The results have been published in [55] and MOLA Tool has been presented in the Tool Session [62] of the *European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2005).*

This version of MOLA tool has been developed with mainly academic goals – to test the MOLA usability, teach the use of MDSD for software system development and perform some real life experiments. This has influenced some of the language design requirements, though with easy usability as one of the goals and sufficient efficiency for research purposes as the second.

4.1 Overview of Architecture

Similarly to many model transformation environments, MOLA environment consists of two major parts: MOLA **Transformation Definition Environment (TDE)** and **Transformation Execution Environment (TEE)**. TDE is completely related to the metalevel *M2* according to MOF terminology, while TEE is at *M1* level. TDE is used by expert users, who define new model transformations in MOLA for the adopted MDSD technology or modify the existing ones from a transformation library to better suit the needs of a specific project. Since MOLA is a graphical language, TDE is a set of graphical editors built on the basis of Generic Modelling Tool [46] (a generic metamodel based modeling framework (GMF[1]), developed by University of Latvia, IMCS together with the Exigen Company). The execution environment (related to *M1* level) is intended for use by system developers, who according to the selected MDSD methodology perform the automated development steps and obtain the relevant target models. Two forms of TEE are available. The form closer to an industrial use is an Eclipse plug-in,

[1] Do not confuse with Eclipse Graphical Modeling Framework [63]

which can be used as a transformation plug-in for UML 2.0 modeling tools, including the commercial IBM Rational tool RSA [64]. Another form is a more experimental one. It is based on Generic Modelling Tool as a generic modeling environment and is intended for various domain specific modeling and design notations.

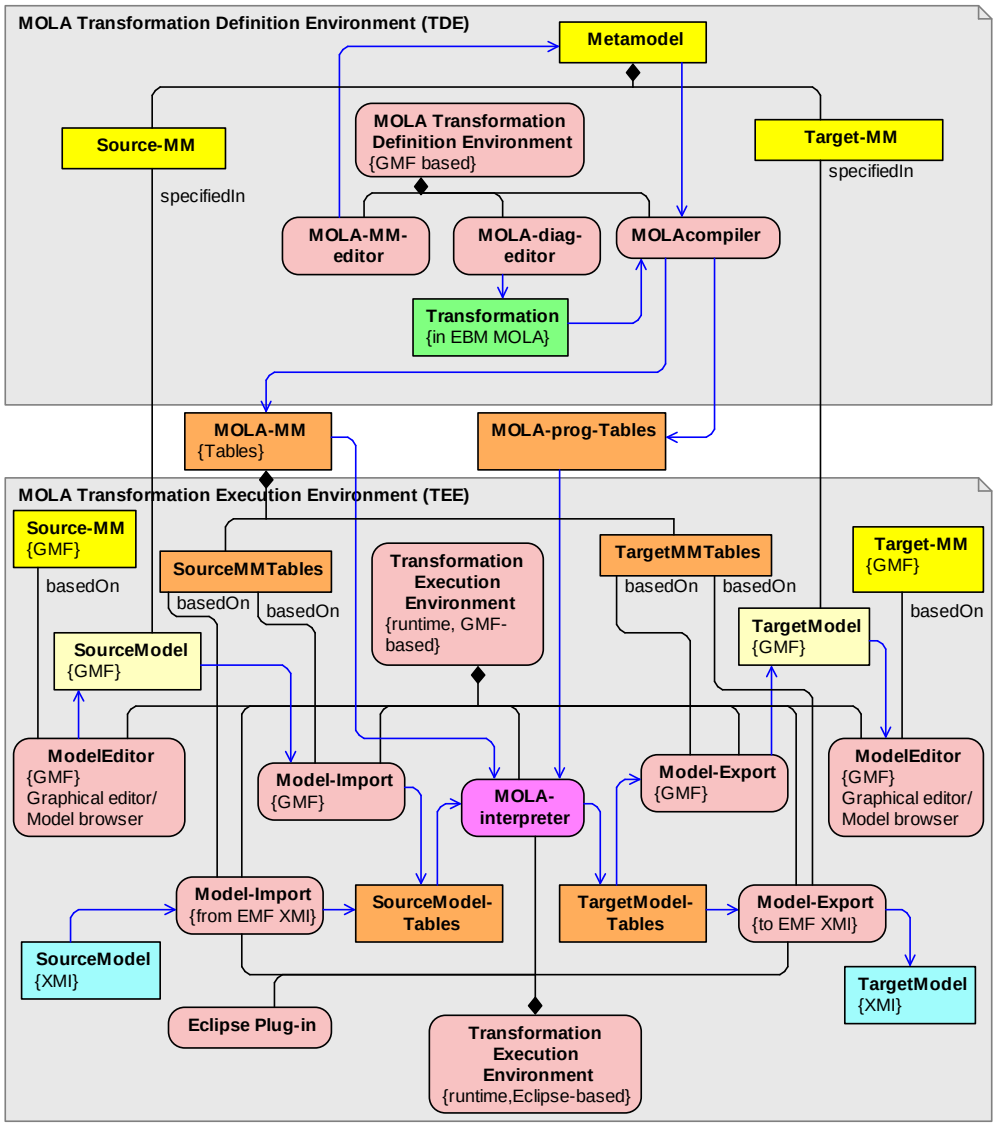


Fig. 8. MOLA Tool environment architecture.

Fig. 8 shows both the components of the MOLA tool (rounded rectangles) and the used data objects (rectangles). Besides the traditional class diagram notation, arrows represent the possible data flows. Data objects in MOLA runtime repository are annotated as tables because it is SQL based. Now some more comments on the MOLA TDE. It contains graphical editors for class diagrams (EMOF level) and MOLA diagrams. Both

the source and target metamodels are shown in the same class diagram, together with possible mapping associations. A transformation is typically described by several MOLA diagrams, one of which is the main. Since the graphical editors are implemented on the basis of Generic Modelling Tool, they have professional diagramming quality, including automatic layout of elements. In addition to editors, TDE contains the MOLA compiler which performs the syntax check and converts both the combined metamodel and MOLA diagrams from the Generic Modelling Tool repository format to the MOLA runtime repository format.

The main component of **MOLA TEE** is the **MOLA Virtual machine (VM)** (interpreter), which actually performs the transformation of the source model to the target model. As it was already mentioned, the goal of this implementation is to provide a simple and sufficiently efficient implementation of MOLA. The key factor in reaching this goal is an appropriate implementation of MOLA VM, since the implementation cost and efficiency of all the service components is nearly the same for all considered solutions to MOLA VM. And in turn, a crucial point of MOLA VM implementation is an appropriate repository and execution environment for pattern matching. This is due to the fact that the implementation of control structures and executable actions in MOLA (due to their procedural nature) is very straightforward in all cases. It should be noted that the choice of repository and execution environment are closely linked ones, thus the rest of the section actually will be devoted to these issues.

Typically model transformation languages are implemented on metamodel based repositories, the most typical of which is *Eclipse EMF* [26]. Several model transformation tools have been built using EMF as a repository [14], [38], [39]. The EMF API in Java provides the most basic actions for building a pattern matcher. The next version of MOLA implementation is also implemented on such repositories- MIIREP [65], JGraLab [66] and also the mentioned EMF.

It has been already shown [67] that a very efficient MOLA pattern matching implementation is possible on such a basis. However, the available low level operations in these APIs (even lower level than analyzed in [67]) make the implementation sufficiently complicated. Therefore another solution was considered – to what degree an SQL database can be used as a repository for pattern matching. On the one hand, the repository structure must match closely enough to EMOF – similarly as EMF does. On the other hand, the desire was to use the powerful capabilities of SQL for a simple high

level implementation of pattern matching. Such a solution was found, which is described in the next section. The only remaining concern was performance issues – whether the query optimization in SQL databases can at least be not very far from the optimal performance described in [67].

4.2 Implementing Patterns by Natural SQL Queries

MOLA VM operates with models – MOF level M1. However, for each model element its metaclass must be known – for pattern matching or any other MOLA action. Therefore MOLA VM has to know the complete metamodel (M2 level) for the transformation. As it was described in CHAPTER 2 the metamodelling facilities in MOLA are approximately those of EMOF. The most natural way is to store the metamodel in tables which correspond to EMOF metamodel classes. However, due to efficiency reasons, the *plain old class metamodel* containing `Classes`, `Associations` and `Attributes` (but not `Properties` as association ends) occurred to be more convenient to be coded by the corresponding SQL tables (see the left column of Fig. 9). It can be easily seen, that in fact it is equivalent to EMOF, therefore MOLA compiler can easily store the metamodel in these tables. In addition, there are tables for identifying metamodels and models themselves.

The storage of model elements – instances of metamodel classes, associations and attributes is completely straightforward in the corresponding three tables (see the right column of Fig. 9). The MOLA program is also naturally stored in tables according to the MOLA metamodel, but since we here are mainly concerned with pattern matching, this coding is not so important. The only fact to be mentioned here is that the MOLA compiler for each program element (loop, rule, pattern class element, pattern link etc.) generates a unique identifier. This fixed database schema is much easier to implement than the metamodel-specific one used in [54].

Let's find out how a MOLA pattern can be naturally mapped to an *SQL Select* statement. The idea is that each class element in the pattern corresponds to an occurrence of the table `class_inst` (actually an alias of it) in the `From` clause. Similarly, each pattern link corresponds to an alias of the `asoc_inst` table in the `From` clause. Next the `Where` clause is formed. Firstly, each pattern element (i.e., the corresponding alias of `class_inst`) must mandatory have the specified class, i.e., its `meta_class_id`

column must have the given value (metamodel elements are fixed during MOLA execution). Similarly it is for links (association instances) in the pattern.

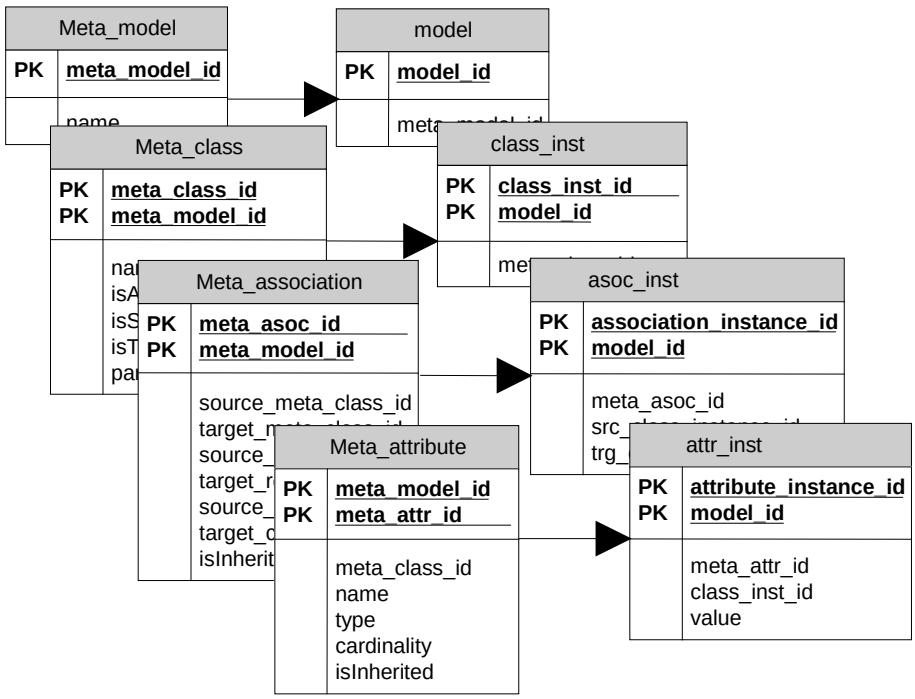


Fig. 9. SQL Tables for storing metamodels and models.

A more non-trivial part of the `Where` clause must specify that each link does link the relevant instances, i.e., `src_class_inst_id` is equal to the `class_inst_id` of the corresponding (association source) alias of class_`inst`, similarly for the `trg_class_inst_id`. For reference elements (`@p:Package` in Fig. 10) it must be specified, that their `class_inst_id` has the given value (reference elements always correspond to a fixed instance in MOLA). The most complicated part in the `Where` clause are the attribute constraints, which already are Boolean expressions. However, the simple attribute names used in MOLA constraints must be substituted by additional aliases of `attr_inst` in the `From` clause, in addition, the transformed expression must be added to the `Where` clause.

Fig. 10 illustrates the generation of an SQL query from a pattern. The pattern is a very simple one – a *foreach* loop head containing the loop variable (of type `Class,` with a constraint) and a reference (to the instance of `Package`) linked by the `package` link. Lines illustrate the described above mapping graphically, the color coding (or levels of gray in the black-and-white version) shows which parts of the query were obtained from

one pattern element. The alias names are generated from the pattern element identifiers built by the MOLA compiler and therefore are unreadable.

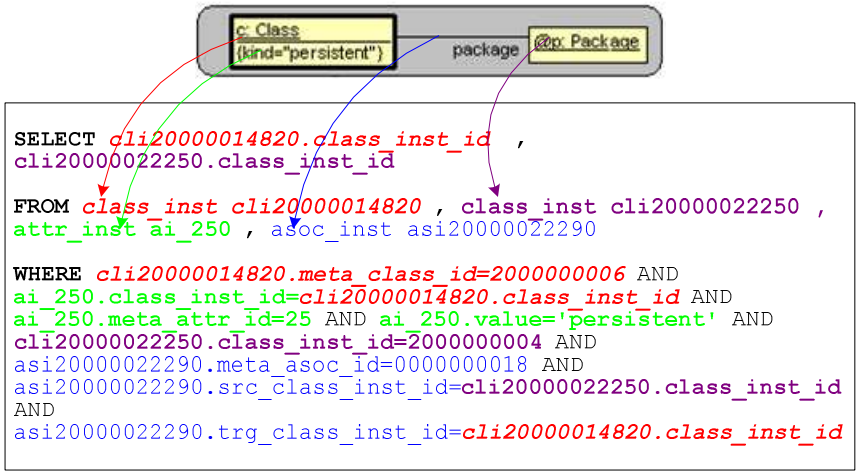


Fig. 10. Generation of an SQL query from a pattern.

The result of the query (a virtual table) is defined in such a way that each row represents (identifiers of) class instances forming a valid match.

Now it can be easily seen, that the built SQL query indeed expresses the pattern match semantics, which for the given example asserts that instances of the metaclass `Class` must be sought, which have the link `package` to the fixed instance of `Package` and which have the given value of the attribute `kind`. Since the pattern is inside a *foreach* loop, all such instances (all matches returned by the query in this simple case) must be processed. A similar argument applies to any MOLA pattern.

Thus the simplicity of the pattern mapping to SQL query has been shown, it remains to show that this SQL Select can easily be built by the MOLA VM (actually it is a sort of *JIT-compiling*). It is being done in several steps. First, the class elements of the pattern are picked up and for each of them an element in the `Select` list and in the `From` list (the table `class_inst` with a new alias) is added, with the MOLA compiler-generated unique element identifier used as the alias name. In addition, a term in the `Where` condition is added, which specifies that the instance must be of the relevant class (or that the instance is the given one for reference elements). Then in a similar manner each link of the pattern is processed. Here the term added to the `Where` part is more complicated, it has to state both that the link's association is the relevant one and that the endpoints are the corresponding class instances. The latter fact is easily to state due to the fact that the MOLA compiler has documented this via references to the relevant element

identifiers and namely these identifiers are used as aliases for the element selection. Then pattern constraints are processed, each adding to the `From` part (the required attribute instance) and to the `Where` part (the expression itself). Simple OCL expressions having a direct counterpart in SQL and some simple OCL set expressions are supported.

Finally, some remarks on the negative patterns. A negative part can be added as a `NOT EXISTS` subquery to the `Where` condition. In the case of a NOT-element, the subquery has just one alias of the `class_inst` in the `From` list plus aliases for the links connecting the element with the positive part of the pattern. The `Where` part of the subquery is generated similarly as for positive patterns.

4.3 Database Performance Issues

In this section the performance of the generated queries in several databases, which are relevant for MOLA tool, is analyzed. A query generated from a pattern is somewhat special in the sense that it is a so-called **self-join** – aliases of the tables `class_inst` and `asoc_inst` are repeated in the `From` clause as many times as there are elements and links in the pattern respectively. Large self-join queries are non-typical for standard database applications and therefore may be processed by some engines not so optimally.

The first natural choice for an experimental tool was the open source database MySQL, the version 5.0.12 [68]. The first intuitive performance evaluations were also encouraging, but it was clear that a more thorough analysis of query optimization is required.

Since it has been shown [67] that pattern matching in MOLA can be performed very efficiently as a sequence of small queries on a reasonable model repository (and the database schema described in previous section is such), it is clear that potentially the generated *large* queries can also be executed efficiently. Since the performance of a join type SQL query is mostly dependent on the join order of tables in `WHERE` part [69], the right order in which the tables in a complicated self-join are joined must be found that is equivalent to the sequence of small queries.

Let us explain the situation in detail on an example (Fig. 11). This example is a fragment of the MOLA transformation transforming a class model to OWL notation [70] (used as a benchmark in Section 4.4), namely, the *foreach* loophead is shown, which

generates an OWL object property for each UML association instance (for classes the corresponding OWL Classes are already built). It was shown in [67], that for cases such as in Fig. 11, the optimal order is to start from the loop variable (the element `as:BinaryAssociation`, all instances of which must be tested anyway), and to proceed along the paths leading away from the loop variable. In the example there are two such paths – one leading via the link `targetEnd` to `objEnd:Property` and further, and another one starting with the link `sourceEnd`. Even without seeing the metamodel, it is clear that in a valid class model this is an optimal order – a UML binary association has just one `targetEnd` (i.e., just one row in the table `asoc_inst`, where the join condition is true), which in turn is followed by just one `objEnd` (one row in `class_inst`)and so on. Fig. 11 illustrates this order by numeric tags. The generated query corresponding to this pattern is shown in Fig. 12.

Certainly, there are other optimal orders – any of the paths could be traversed first, and the paths can be traversed *intermittently*. Similar *easy-to-be-explained* optimal join orders exist for more complicated patterns, where paths may have *cross-links* and where reference (fixed) elements exist (see more in [67]).

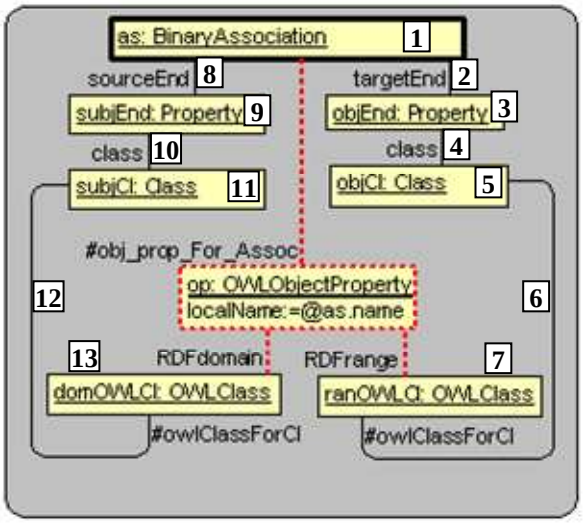


Fig. 11. Optimal pattern matching order

Further, it was to be found, how close the MySQL query execution plans are to an optimum, and at what expenses such a plan is found. Fortunately, MySQL has the `Explain` statement [71], which reveals some details of the execution plan. Fig. 13 shows the join order of query shown in Fig. 12, exposed by the `Explain` statement. Actually, two experiments are merged there – one with order tags in squares has been

performed on a small source model (29 rows in `class_inst`, 39 rows in `asoc_inst`).

Another one has been performed on a large source model (725 rows in `class_inst`, 975 rows in `asoc_inst`), the join order (where different from the first one) is shown in circles. For the large model the join order is equivalent to the optimal one, only another starting point has been selected, and paths are traversed intermittently. For the small one the deviation is larger, but also not critical.

However, if the number of elements and links in a pattern is increased, the query execution time also increases. The query (discussed above) having a pattern with 7 elements and 6 links executes in 200ms on a model with 3000 class instances and 4000 links, a query with 8 elements and 7 links in 600 ms on the same model, 9 elements and 8 links in 3200ms, but 10 elements and 9 links in 43000ms that is a significant jump. There are only few papers on MySQL optimization [72], [73], and they do not explain the optimization of the specific self-join queries used in MOLA pattern matching. Another observation should be mentioned – the `Explain` statement execution itself requires nearly as much time as the query execution, so we can assert that MySQL query optimization in case of large self-join queries is not optimal – it itself is too time consuming.

Thus we have to rely on our *black box* experiments, which say that MySQL optimization is acceptable when there are limits on the pattern size (no more than 8 elements), but the query execution time increases too much for larger patterns, to make sense in using this RDBMS for pattern matching.

Thus the current version of MySQL can be used for MOLA runtime repository, but with restrictions on MOLA transformation patterns. The hope is for versions to come (the current version performs better than those tested earlier), but next versions could only raise the limit for pattern size – not remove this restriction completely.

Due to the mentioned above problem other alternatives were sought. Possible alternatives are MSDE 2000 [74] – the free *small* version of MS SQL 2000 server, PostgreSQL [75] – another popular open source RDBMS, MSSQL Server 2005 Express [76] – the free *small* version of MS SQL 2005 server.

```
SELECT cli20000020780.class_inst_id cli20000020970.class_inst_id ,
  cli20000021040.class_inst_id , cli20000021110.class_inst_id ,
  cli20000021180.class_inst_id , cli20000021260.class_inst_id , li20000021330.class_inst_id
FROM class_inst cli20000020780 , class_inst cli20000020970 , class_inst cli20000021040 ,
  class_inst cli20000021110 , class_inst cli20000021180 , class_inst cli20000021260 ,
  class_inst cli20000021330 , asoc_inst asi20000021080 , asoc_inst asi20000021150 ,
  asoc_inst asi20000021300 , asoc_inst asi20000021400 , asoc_inst asi20000021700 ,
  asoc_inst asi20000021760
  WHERE cli20000020780.meta_class_id=2000001847 AND
  cli20000020780.meta_model_id=0000000000 AND cli20000020780.model_id=0 AND
  cli20000020970.meta_class_id=2000001790 AND
  cli20000020970.meta_model_id=0000000000 AND cli20000020970.model_id=0 AND
  cli20000021040.meta_class_id=2000001721 AND
  cli20000021040.meta_model_id=0000000000 AND cli20000021040.model_id=0 AND
  cli20000021110.meta_class_id=2000001723 AND
  cli20000021110.meta_model_id=0000000000 AND cli20000021110.model_id=0 AND
  cli20000021180.meta_class_id=2000001790 AND
  cli20000021180.meta_model_id=0000000000 AND cli20000021180.model_id=0 AND
  cli20000021260.meta_class_id=2000001721 AND
  cli20000021260.meta_model_id=0000000000 AND cli20000021260.model_id=0 AND
  cli20000021330.meta_class_id=2000001723 AND
  cli20000021330.meta_model_id=0000000000 AND cli20000021330.model_id=0 AND
  asi20000021080.meta_asoc_id=2000001835 AND
  asi20000021080.meta_model_id=0000000000 AND
  asi20000021080.src_class_inst_id=cli20000021040.class_inst_id AND
  asi20000021080.trg_class_inst_id=cli20000020970.class_inst_id AND
  asi20000021080.model_id=0 AND asi20000021150.meta_asoc_id=2000001725 AND
  asi20000021150.meta_model_id=0000000000 AND
  asi20000021150.src_class_inst_id=cli20000021040.class_inst_id AND
  asi20000021150.trg_class_inst_id=cli20000021110.class_inst_id AND
  asi20000021150.model_id=0 AND asi20000021300.meta_asoc_id=2000001835 AND
  asi20000021300.meta_model_id=0000000000 AND
  asi20000021300.src_class_inst_id=cli20000021260.class_inst_id AND
  asi20000021300.trg_class_inst_id=cli20000021180.class_inst_id AND
  asi20000021300.model_id=0 AND asi20000021400.meta_asoc_id=2000001725 AND
  asi20000021400.meta_model_id=0000000000 AND
  asi20000021400.src_class_inst_id=cli20000021260.class_inst_id AND
  asi20000021400.trg_class_inst_id=cli20000021330.class_inst_id AND
  asi20000021400.model_id=0 AND asi20000021700.meta_asoc_id=2000001858 AND
  asi20000021700.meta_model_id=0000000000 AND
  asi20000021700.src_class_inst_id=cli20000020780.class_inst_id AND
  asi20000021700.trg_class_inst_id=cli20000020970.class_inst_id AND
  asi20000021700.model_id=0 AND asi20000021760.meta_asoc_id=2000001852 AND
  asi20000021760.meta_model_id=0000000000 AND
  asi20000021760.src_class_inst_id=cli20000020780.class_inst_id AND
  asi20000021760.trg_class_inst_id=cli20000021180.class_inst_id AND
  asi20000021760.model_id=0
```

Fig. 12. Generated query example

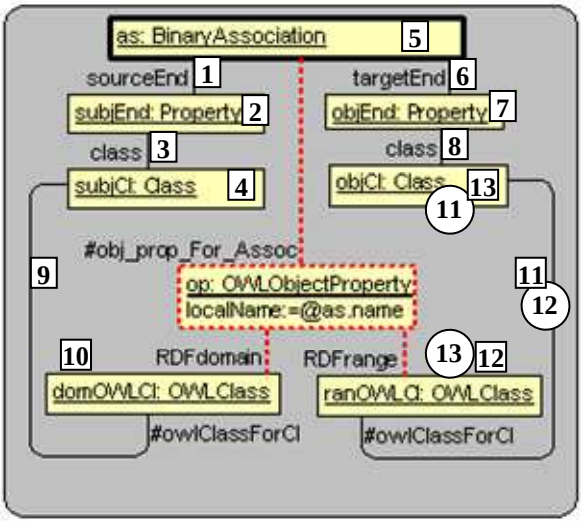


Fig. 13. MySQL query plan (table join order).

Similar performance experiments on large queries have been performed with these engines too. Single pattern query execution times for these alternatives were significantly better (*Microsoft* products) or similar (PostgreSQL). The join order was nearly optimal. It can be concluded from available references [77] that both MS SQL and MSDE use instance data for query optimization in a more sophisticated way. However, experiments show that execution of a complete transformation is much slower than by using MySQL. MySQL was faster by an order of magnitude. It seems that MSDE 2000 and MSSQL Server 2005 Express engines have major problems with completing large sequences of SQL queries, because of built-in features such as workload governor [78] in MSDE 2000, which decreases the server performance.

Thus, MySQL is a satisfactory implementation for MOLA runtime repository if the pattern size does not exceed 8-9 elements (actually, only the *free* pattern elements count – those which are class elements, but not references or parameters, in Fig. 13 all pattern elements are free). The existing experience of using MOLA tool on some nearly real life examples has confirmed this. The transformation execution times in these examples testify that apparently *close-to-optimal* join order was used by MySQL in most cases. Nearly all patterns in these examples were below the size limit. In practice it is also possible to bypass the limit by decomposing a pattern into several smaller ones (actually, even without sacrificing the transformation readability).

An alternative approach would be to enforce the optimal join order manually, since MySQL has such possibilities. Unfortunately, these features are vendor-specific

extensions of SQL. In addition, finding of this order during query generation is a significant part of implementing the pattern via *small queries* and therefore much more complicated.

4.4 Benchmark Results

The previous section demonstrated that usage of MySQL database server as model repository and pattern matching engine has proven to be sufficient. To estimate MOLA Tool performance the experiments have been done.

A simple task and appropriate model transformation tool for comparison have been chosen. The choice – AGG [20] is a popular graph transformation language that uses pattern constructs similar to MOLA, only explicit NAC's (negative application conditions) must be added. AGG rules have no explicit control structures, but in simple cases MOLA control structures can be adequately emulated by AGG rule layering. AGG has already been used for benchmark testing [79], thus allowing ensuring certain correctness of the experiment. The transformation was executed on both MOLA Tool and AGG for models of various size and complete execution times were measured. Both MOLA Tool and AGG were used with configurations recommended by developers. The example transforms *simplified UML class diagram* to *simplified OWL diagram.* Metamodels are shown in Fig. 14.

The transformation creates an `OWLClass` instance for every `Class` instance and `OWLDataTypeProperty` for every `Property` which is an owned attribute of the `Class`. This task is done using nested loops. The first *foreach* loop iterates through all `Class` instances and the nested *foreach* loop iterates through appropriate `Property` instances. The third *foreach* loop creates `OWLDataTypeProperty` for each `BinaryAssociation` (Fig. 15). Though this transformation is very simple it is a typical representative of MDSD tasks where frequently a model has to be transformed to a semantically equivalent one in another notation.

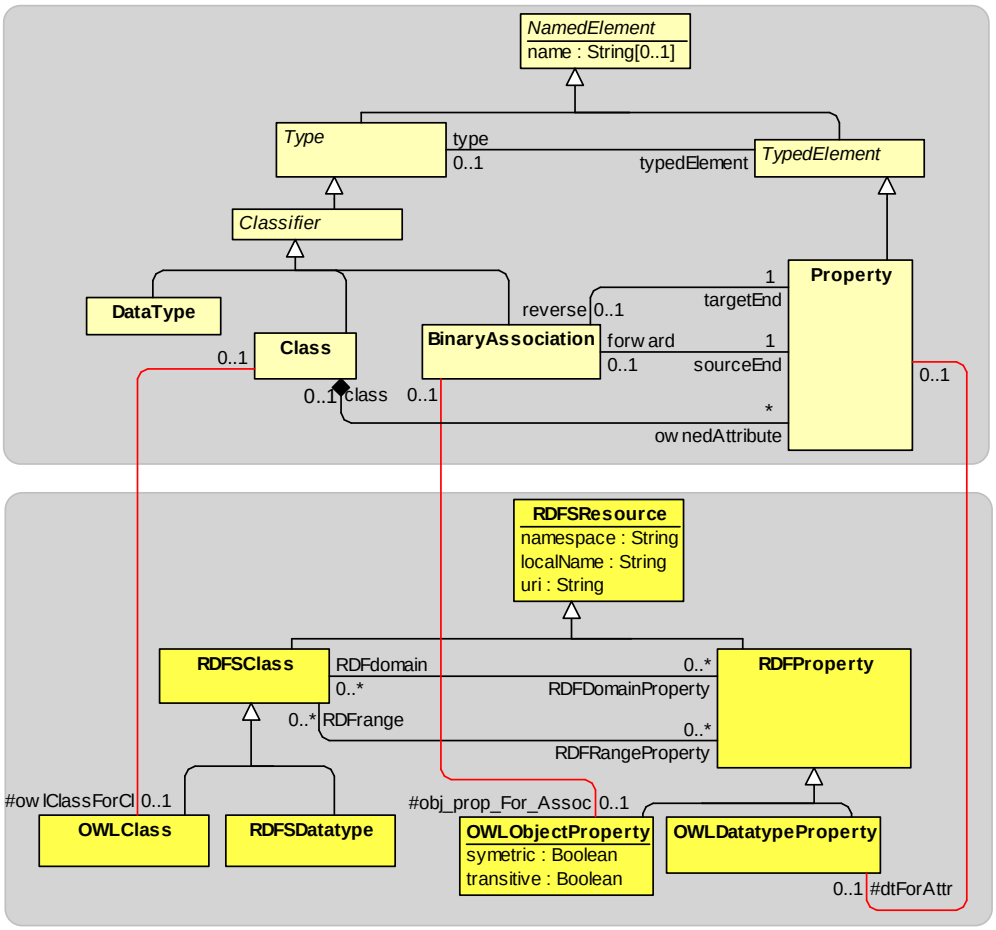


Fig. 14. Metamodels of UML Class Diagram and OWL Diagram

The transformation was executed on a hyper-threaded Intel Pentium4 3GHz, 1 GB RAM Windows XP workstation. No additional performance tuning was done to MySQL database server or operating system configuration. Identical models of various sizes were prepared for MOLA Tool and AGG. The first column of Table 1 contains model data size N – the number of class instances in the model. Second and third columns contain complete transformation time for MOLA and AGG measured in seconds.

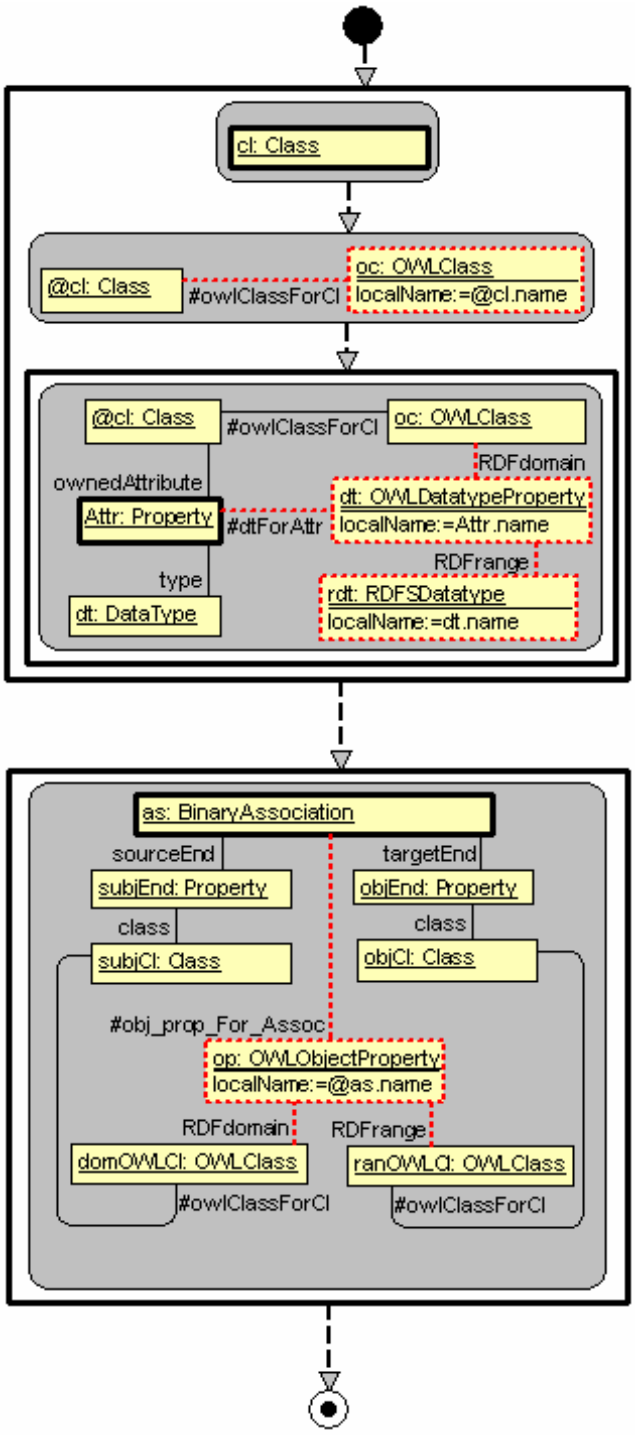


Fig. 15. Transformation UML Class Diagram to OWL Diagram

Both MOLA Tool and AGG showed sufficient performance on models with size below N=175. MOLA Tool execution time grows nearly linearly up to model size N=3500, but starts to grow faster above this value. Thus the current MOLA Tool implementation performs well in this range, but real examples could be also larger – there are ontologies containing more than 5000 *OWL Classes*. Real transformations are also

more complicated. AGG has problems similar to MOLA Tool, but both tools are usable for tasks they are designed for.

The main relational database engine feature, which enables fast search, is table indexing [77]. The MOLA Tool uses table indexes in the most appropriate way; apparently this ensures the nearly linear time growth for queries.

The reason for faster complete transformation time growth for large N lies in the fact that the model size grows while transformation is being executed.

A proportional to N number of insert and update operations must be done in this MOLA program and each operation time grows due to the need of refreshing indexes (but indexes are crucial for fast pattern matching). A similar problem is the main reason for AGG slowdown, even to a larger degree, as it is shown in [79].

Table 1. Benchmark Results

	Transformation ExecutionTime (s)	
Model size (N)	**MOLA**	**AGG**
42	1	4
56	1	6
70	2	9
84	3	14
175	5	62
400	10	334
1050	19	8280
1750	36	
3500	65	
17500	1781	

For real MDSD tasks it is typical that a new model must be built of size proportional to the source model. Thus not only the pattern match time influences the performance, but still it seems to be the key factor.

4.5 Summary

Both simple and sufficiently efficient implementation of pattern matching via SQL queries has been built. Thus this is a viable solution at least for an experimental tool (what this version of MOLA tool is). Several model transformations supporting real MDSD style development (automated use of Hibernate persistence framework in Java – a plug-in for the RSA tool, conversion of UML activity diagrams to BPMN notation and

other) have been built and tested on examples of realistic size [80], [81]. In none of these examples the *natural* pattern size in MOLA programs exceeded 8 – the critical value up to which the given MOLA implementation is efficient. These experiments and benchmark tests described in the paper have shown that the implemented MOLA VM performs satisfactorily and MOLA is a suitable transformation language for typical MDSD tasks – transforming a UML model to another one closer to the system implementation. However, for an industrial usage of MOLA a special in-memory repository and a compiler/interpreter that implements the principles described in [67] could be required. The main reason could be the desire to get rid of any limits on pattern size; also the general performance for large models is expected to be better. Such a solution is discussed in the next Chapter.

Certainly, these results obtained for MOLA implementation have value also for other transformation languages, where the pattern match semantics is similar.

CHAPTER 5

Implementation of MOLA Using L3 Language

The pattern matching algorithm which uses L3 language and local search plan generation is described in this chapter. The implementation of this algorithm for model transformation language MOLA is one of the main results of these thesis. The results have been published in [82] [53] and MOLA Tool has been presented in the Tool Session [83] of the *European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2008).*

The most critical part of the implementation of a pattern-based transformation language is the implementation of the pattern matching. It has been already shown [67] that an efficient MOLA pattern matching implementation is possible. In fact, some kind of local search plans are generated and executed by this approach. It is based on only few *basic lookup operations* needed to iterate over a model. They are:

- `getNext(Class Cl)` - returns the next instance of a class `Cl` upon each call. There is also an initialization for it - `initializeGetNext(Class Cl)`
- `getNextByLink(Association as, Cl1 inst, Class Cl2)` - returns one by one instances of a class `Cl2` that can be reached by links corresponding to association `as` from a fixed instance `inst`. There is also an initialization for it, with similar parameters - `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)`
- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` - checks whether a link of required type is between instances
- `eval(Cl inst, Expr exp)` - evaluates a local constraint on attributes

Thus, the target language of the MOLA compiler or the API of a repository that is used for implementation of the MOLA interpreter (Virtual Machine) must contain similar operations. This approach requires the implementation of the pattern matching algorithm using such low-level constructs. That is a sufficiently complicated task.

The Lx language family [16] (L0, L0`, L1, L2, L3) is an appropriate target for MOLA compiler. Each next language of Lx family has been built extending the previous (see Section 5.3). L0 language as well as MOLA has such concepts as *procedure*, *parameter*, *variable*, *sub-procedure call*. These concepts can be mapped directly from MOLA to L0 language. These basic features along with basic lookup operations are included in the L0 language, but commands introduced in the following languages L0`-L3 (imperative pattern matching, looping and branching commands) allow much easier implementation of the MOLA compiler than API of repositories. That is possible because these commands are at an abstraction layer much closer to MOLA concepts, such as foreach loop and rule, than lower level languages or API of metamodel based repositories. Thus L3 language provides all necessary features that allow us to build an efficient MOLA compiler.

5.1 Architecture of MOLA Compiler

An efficient compiler has been already built [16] for the Lx language family. Actually, an efficient implementation of the L0 language has been built and a compiler for each next language is built using the bootstrapping method [84]. It means that the previous language in the family is used to build the compiler for the next one (L0 for L0' compiler, L0' for L1 compiler and so on).

Several metamodel-based in-memory repositories MIIREP [65], EMF [26] and JGraLab [66] have been chosen to store metamodel and its instances for the implementation of L0 language. These repositories have appropriate low-level API's implemented as a C++ (MIIREP) or Java (EMF and JGraLab) function libraries. Therefore an intermediate result of the L0 compilation is a C++ or Java program. The final result of the L0 compilation is a dynamic link library (DLL file) or JAR file that can be executed over a repository instance which contains the appropriate metamodel and model.

The bootstrapping method used to build compilers for the rest of the Lx family languages requires that programs written in L0' to L3 must be stored in the repository that is used by L0 language. Thus the metamodel of these languages is required. All languages of the Lx family are described by the same metamodel because each next language is

derived from the previous one by adding some new features. Therefore the metamodel of the last language in the chain (L3) describes also all the previous languages.

The first step in the compilation of a L3 program is to obtain a model - an instance of the L3 metamodel. It is a representation of the L3 program in the metamodel-based repository. This step is a separate step in the whole process of the compilation which requires parsing of the text file and building a model. It is implemented using a traditional programming language (C++). Obtained lexemes [85] are stored in the repository as a very simple lexeme model [86]. Next, the transformation language L0 is used to obtain the L3 program model from the lexeme model.

When a program model has been built the actual compilation is being performed. The L3 (also L2, L1, L0') compiler actually is a model transformation. In this case, an in-place transformation is used – the L3 program model is overwritten by a semantically equivalent L2 program model (also L2 by L1, etc.). The final result of the chain of compilation steps is an L0 program model which is semantically equivalent to the initial L3 program given as the input file. The chain of compilation steps (from L3 to L0) can be treated as one step (the corresponding transformations are invoked one after another).

The last step in the compilation process is the code generation (a model to text transformation). An L0 language text file is generated. Also this step is done using the L0 language extended with native functions for file handling written in C++. Actually, only one write to file function is needed.

Since the whole L3 compilation process has been divided into three separate steps, there is a possibility to start with any step if the appropriate model has been prepared. This fact is used by MOLA to L3 compiler – MOLA program is being compiled directly to an L3 model. This allows decreasing significantly the complexity of the implementation of MOLA to L3 compiler. Actually, it allows using transformation language L3 to build MOLA to L3 compiler.

The first MOLA Transformation Definition Environment (MOLA Editor) [87] was built on the basis of Generic Modelling Tool [46] – a domain specific modelling framework, developed by UL IMCS together with the Exigen Company. The models (MOLA program and metamodel) were stored in a compatible format to the repository used by the L0 language. Thus the input for the MOLA to L3 compiler, a model of a MOLA transformation, already could be obtained. In fact, no other natural representation of a MOLA program than a model can be obtained, because MOLA is a graphical

transformation language. The most appropriate way to implement MOLA compiler to any suitable language is using model transformations. Thus, the first MOLA compiler was implemented using L3 language.

Since the MOLA Editor required more sophisticated features than the Generic Modelling Tool domain specific modelling framework could offer, the next MOLA Editor- MOLA2 Tool, has been built. MOLA2 Tool uses the METAclipse framework [8], which is based on Eclipse platform [88] and model transformations. It should be noted that METAclipse uses the same repository as the L0 implementation. Therefore it was possible to develop transformations for MOLA2 Tool using MOLA itself and the first MOLA compiler. The second version of MOLA to L3 compiler has been built for MOLA2 Tool, using L3 language too.

Although there are two implementations of MOLA to L3 compiler, there are no significant differences in the architecture and general ideas of implementations of both compilers. The main difference between these two implementations is the MOLA metamodel. The MOLA metamodel for MOLA2 Tool was improved by eliminating metamodel restrictions enforced by Generic Modelling Tool and by making it more suitable for compilation. The experience and a significant part of the code from the first version of MOLA to L3 compiler is reused in the second version. This work is based on the second version of MOLA to L3 compiler.

Compilation of a MOLA transformation is divided into four steps. Each of them is performed by a separate component – compiler. These components are:

- MOLA to L3 compiler
- L3 to L0 compiler
- L0 to C++ or Java compiler
- C++ or Java to executable file compiler

The general architecture of MOLA compiler is shown in Fig. 16. There may be a question – why such a large number of compilers are used? Why do not use direct compilation from MOLA to repository API? The answer is in the low complexity and reusability of the each step. Each compiler transforms a higher-level language to a lower-level language. It is much easier to build compiler to a language that is at a closer abstraction level to the source language. Especially it is so if the general concepts of both

languages are similar. This is the reason why L3 (and not L0) is used as the target language for MOLA.

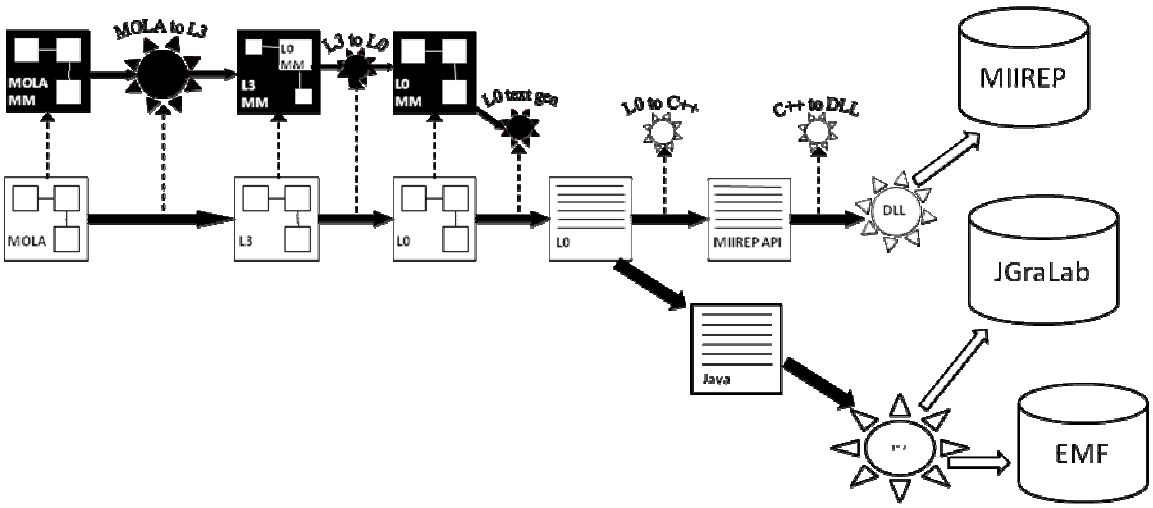


Fig. 16. The general architecture of MOLA compiler

Another issue is the reusability. The compiler of L3 language had been already built and this implementation was efficient. The efficiency of the generated code does not suffer if MOLA compiler is built on top of the compiler chain. It has allowed implementing MOLA on other EMOF compatible repositories, EMF [26] or JGraLab [66], and then only L0 compiler must be rewritten. Even less, only the actual code generator in L0 compiler must be rewritten – lexical and syntax analyzers can be reused. The last compiler (L0 to code) is dependent of the API of the model repository.

The only disadvantage of a long compiler chain is a longer compilation time. To deal with this issue a program has to be structured. The most common approach is to use code units. Each unit is compiled to a separate object. Next, a linker is used to obtain a single executable. A similar idea is used also in the MOLA2 Tool. Packages are used to structure a MOLA program. A package may be defined as a MOLA unit. That means that all MOLA procedures that are contained by the unit are compiled to a separate L0 unit. This allows using L0 compiler as a linker that assembles all L0 units into one C++ or Java project. Thus model transformations (MOLA and L3-L0'compilers) can work with smaller models that helps to improve the overall performance of the compilation process.

5.2 Model-Driven Compiling

The usage of models and transformation languages in the process of compilation is not new. The ATL model transformation language [14] has already been used to compile CPL to SPL [89] and FIACRE to LOTOS [90]. The ATL language itself is also compiled using a domain specific only for this purpose created language - ACG (ATL Code Generation language) [91]. All these are textual languages and the model-to-model transformation is used for actual compilation similarly to the way it was used in the example of the L3 to L0 compilation [86]. A similar idea is used also in the SmartQVT [13] implementation. The QVT code is parsed to obtain the model representation of a QVT transformation and the actual compilation to the Java file is performed using this model.

A similar pattern of the compilation is used in all examples. Three basic steps are performed:

- parse an input program and obtain the model of it
- compile the model of the input program to a model of an output program
- generate the code of the output program from the model

This approach may be called **model-driven compiling** – models are used as core elements of the compilation process (see Fig. 17).

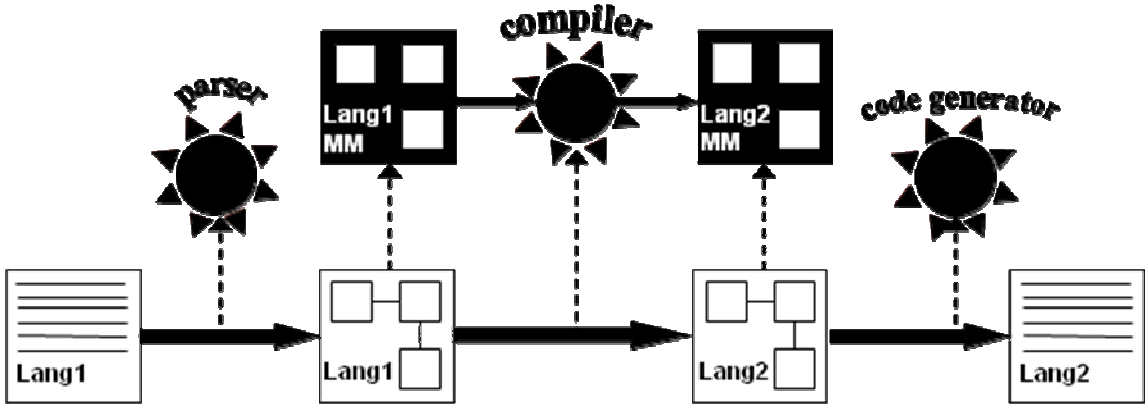


Fig. 17. Model-driven compiling

These steps are similar to phases of a compilation in the traditional compilation technique [85]. The lexical and syntax analysis are performed by the parser. The semantic analysis, intermediate code generation (target program model) and optimization are performed by compiler (model transformation). The code generation is done in the last

step. A model of a source program is stored according to the language metamodel. Actually, the parse trees used in traditional compilation technique can be treated as a sort of models. Thus, the similarity is obvious.

All three steps of the model-driven compiling require appropriate metamodels already built for both input and output languages and a transformation written using a model transformation language suitable for the compilation tasks. Actually, text-to-model (T2M), model-to-model (M2M) and model-to-text (M2T) languages are needed. An exporter or importer written in a general purpose programming language can be used instead of the T2M and M2T transformations. Certainly, the choice of the programming language depends on the repository used to store models.

The model-driven compiling is even more appropriate for graphical languages such as MOLA. Since programs of graphical languages are stored as models, the first step can be omitted – the model-to-model transformation that implements a compiler can be applied directly.

The main gains of using model-driven compiling are:

- The higher level of abstraction that is provided by model transformation languages allows reducing the complexity of compiler implementation.
- This is the most appropriate way to compile graphical languages, because they are mostly implemented using some metamodel [26] or graph-based [66] repository. Actually, programs (diagrams) of such languages are models and the usage of a model transformation language is the most natural approach.
- If the concrete syntax of the input language is based on some general *coding* language, like XML [92], then model transformations can be applied to obtain a model of the program from its *coding*. In this case, a standard parser can be used to obtain the model of the *coding*. Next, the model transformation can be used to obtain the model conforming to the input language metamodel. A similar approach is applicable also for the output language.
- Since attribute grammars have been used to specify the semantics of programming languages [93], a precise definition of the model

transformation between source language and target languages can be used to define the semantics of the source language even in more readable way.

The first experience using **model-driven compiling** was quite promising. The MOLA to L3 and L3 to L0 [86] compilers have been developed. The implementation of both compilers has shown that using transformation language for compilation tasks reduces the complexity of the implementation. However, the best practice of model-driven compiling has yet to be developed and a comparison to the traditional compilation techniques [85] must be done.

5.3 L3 from Lx Language Family

The Lx language family as any other model transformation language uses some sort of metamodelling language. It is quite close to the OMG EMOF specifications. The main difference is that there are no packages in this metamodelling language. The metamodel of this language is shown in Fig. 18.

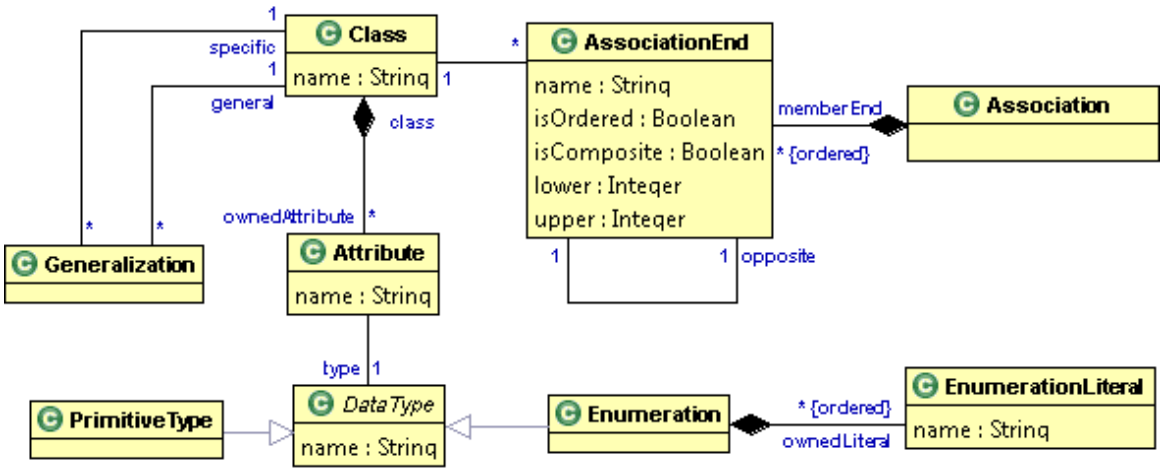


Fig. 18. The metamodel of Lx metamodelling language

Classes and binary associations are core elements of this language. Classes can have attributes which can be primitive or enumeration-typed. There are four pre-defined primitive types – *String*, *Integer*, *Boolean* and *Real*. There are no possibilities to define new ones.

The basic commands (constructs for a textual definition of a metamodel) of the Lx family metamodelling language are the following:

- **class** <className>**;** - defines class with a given name.
- **attr** <className>**.**<attrName>**:**<ElementaryTypeName>**;** - defines attribute with a given name and type.
- **assoc** <className>**.** [**{ordered}**]<cardinality> <roleName>/<roleName><cardinality> [**{ordered}**] **.** <className>**;** - defines association with corresponding properties.
- **compos** <compositeClassName>**.** [**{ordered}**] <card><roleName> / <roleName><card> [**{ordered}**] **.**<partClassName>**;** - defines compositions with corresponding properties.
- **rel** <subClassName>**.subClassOf.**<superClassName>**;** - defines a generalization relationship between given classes.
- **enum** <enumName>**:{** <enumLiteral1>**,**< enumLiteral2>**,** … **};** - defines enumeration with given elements.

An elementary unit of L0 transformation is a **command** (an imperative statement). L0 transformation contains several parts:

- global variable definition part
- native subprogram (function or procedure) declaration part (used C++ or Java library function headers)
- L0 subprogram definition part. Exactly one subprogram in this part is the **main**. The main subprogram defines the entry point of the transformation. An L0 subprogram definition also consists of several parts:
  - Subprogram header
    - **procedure** <procName>**(**<paramList>**);** Subprogram header, the (formal) parameter list can be empty. Parameter list consists of formal parameter definitions separated by ",". A parameter definition consists of its name, the parameter type (the type can be an elementary type or a class from the metamodel), and the passing method (parameters can be passed by reference or by value). If the parameter is passed by reference, its type name is preceded by the **&** character.
    - **function** funcName>**(**<paramList>**):** <returnType>**;** - return type name can be an elementary type name or class name.
  - Local variable definitions

- **pointer** <pointerName> **:** <className>**;** - defines a pointer to objects of class <className>.
- **var** <varName> **:** <ElementaryTypeName>**;** - defines a variable of elementary type. <ElementaryTypeName> is one of elementary types.

- Keyword **begin -** starts subprogram body definition
- Subprogram body definition
- Keyword **end -** ends subprogram body definition.

The subprogram body definition may contain the following commands:

1. **return;** - returns execution control to caller procedure or function.
2. **call** <subProgName>**(**<actPrmList>**);** – calls a subprogram. Actual parameters list can be empty. Actual parameter list consists of binary expressions separated by ",".
3. **label** <labelName>**;** - defines a label with the given name.
4. **goto** <labelName>**;** - unconditionally transfers control to label <labelName>. The label <labelName> should be located in the current subprogram.
5. **first** <pointer> **:** <className> **else** <label>**;** - positions <pointer> to an arbitrary object of class <className>. Typically, this command in combination with the **next** command is used to traverse all objects of the given class (including subclass objects). If the class does not have objects, <pointer> becomes **null**, and execution control is transferred to the <label>. The class in this command must be the same as (or a subclass of) the class used in pointer definition. If it is a subclass, then the pointer value set is narrowed (for the subsequent executions of **next**).
6. **first** <pointer1> **:** <className> **from** <pointer2> **by** <roleName> **else** <label>**;** - similar to the previous command. The difference is that it positions <pointer1> to an arbitrary class object, which is reachable from <pointer2> by the link <roleName>. Similarly, this command in combination with the **next** command is used to traverse all objects linked to an object by the given link type.
7. **next** <pointer> **else** <label>**;** - gets the next object, which satisfies conditions, formulated during the execution of the corresponding **first** and which has not been visited (iterated) with this variable yet. If there is no such object, the <pointer> becomes **null**, and execution control is transferred to <label>.
8. **addObj** <pointer>**:**<className>**;** - creates a new object of the class <className>.

9. **addLink** <pointer1>**.**<roleName>**.**<pointer2>**;** - creates a new link (of type specified by <roleName>) between the objects pointed to by the <pointer1> and <pointer2> , respectively.
10. **deleteObj** <pointer>**;** - deletes the object, which is pointed to by <pointer>.
11. **deleteLink** <pointer1>**.**<roleName>**.**<pointer2>**;** - deletes link, whose type is specified by <roleName>, between objects pointed to by <pointer1> and <pointer2>, respectively.
12. **setPointer** <pointer1>=<pointer2>**;** - sets <pointer1> to the object, which is pointed to by <pointer2>. In place of <pointer2> the *null* constant can be used.
13. **setVar** <variable> **=** <binExpr>**;** - sets <variable> to <binExpr> value. <binExpr> is a *binary* expression consisting of the following elements: *elementary variables, subprogram parameters* (of elementary types), *literals, object attributes* and *standard operators* (+,-,*,/,&&,||,*!*).
14. **setAttr** <pointer>**.**<attrName>=<binExpr>**;** - sets the value of attribute <attrName> (of the object, pointed to by <pointer>) to the <binExpr> value.
15. **type** <pointer> **==** <className> **else** <label>**;** - if the type of the pointed object is identical to the class <className>, then control is transferred to the next command, else control is transferred to <label>. In place of the equality symbol == an inequality symbol != can be used. This command is used for determining the exact class of an object.
16. **var** <variable>==<binExpr> **else** <label>**;** - if the condition is *true* , then control is transferred to the next command, else control is transferred to <label>. In place of equality symbol other (<, <=, >, >=, !=) relational operators compatible with argument types can be used.
17. **attr** <pointer>**.**<attrName> **==** <binExpr> **else** <label>**;** - if the condition is *true* then control is transferred to the next command, else control is transferred to <label>. Other relational operators (<, <=, >, >=, !=) can be used too.
18. **link** <pointer1>**.**<roleName>**.**<pointer2> **else** <label>**;** - checks whether there is a link (with the type specified by <roleName>) between the objects pointed to by <pointer1> and <pointer2>, respectively.
19. **pointer** <pointer1>==<pointer2> **else** <label>**;** - checks whether the objects pointed to by <pointer1> and <pointer2> are the same. Instead of <pointer2> *null* constant can be used. The inequality symbol (!=) can be used too.

It is easy to see that the language L0 contains only the very basic facilities for defining transformations [94].

Language L0' - model transformation language L0' is based on the language L0. The new feature of L0' is the possibility to make long arithmetic expressions (in L0, only unary and binary expressions were allowed).

Language L1 - is supplemented with an imperative pattern matching feature (*suchthat* block), so that it is possible to search for instance that match some condition. The *suchthat* block may be used with **first** and **next** commands. The *suchthat* block can contain conditions on values of variables or attributes, links between instances and other. In fact, all L1 commands can be used to specify pattern condition, including the nested **first** commands.

The textual syntax for the pattern (*such-that* block) is as follows:

```
(first | next) <pointerName1> : <className> [ from
<pointerName2> by <roleName> ] [
suchthat
begin
<L1Commands>
end; ]
```

The condition holds if it is possible to successfully [86] reach the end of the block (i.e., successfully execute its last command). If the condition fails then the next instance is examined. The *conditional* commands in L0 (commands that have an *else* branch) may be used without the *else* branch in the *suchthat* block. If in such a command the undefined *else* branch is to be executed then the condition defined by the pattern fails.

Language L2 - has the possibility to make loops. A special command exists in L2 with which it is possible either to visit all instances of the specified class or just those instances of the class that match the given pattern. The textual syntax for the loop is as follows:

```
foreach <pointerName1> : <className> [ from <pointerName2> by <roleName>
] [ suchthat
  begin
      <L2Commands>
  end ]
  do
  begin
      <L2Commands>
  end;
```

Language L3 - has the branching command – a standard *if-then-else* construct can be used. The textual syntax of the branching command is as follows:

```
  if
```

```
   begin
     <L3Commands>
   end
then
   begin
     <L3Commands>
   end
[ else
   begin
     <L3Commands>
   end ];
```

The L3 metamodel (the Lx language family metamodel) is shown in Fig. 19.

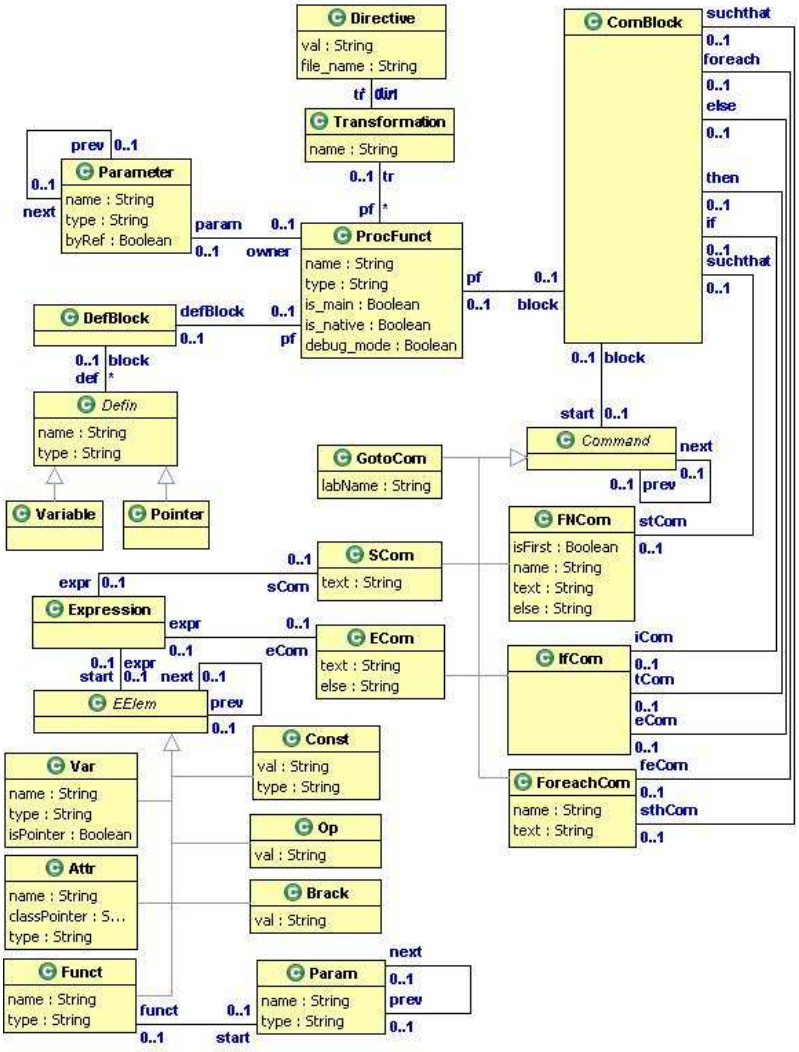


Fig. 19. The metamodel of L3 language

It has already been shown [67] that MOLA language can be implemented efficiently using a set of low-level operations for patterns. There is a direct mapping from the required operations to the commands of Lx model transformation family.

- `initializeGetNext(Class Cl)` and `getNext(Class Cl)` operations can be mapped to **first** *c:Cl* and **next** *c* commands. These commands return all instances of a given class. In the beginning the **first** *c:Cl* command must be called to initialize the iteration through required instances and afterwards the **next** *c* must be called to iterate through
- `initializeGetNextByLink(Association as, Cl1 inst, Class Cl2)` and `getNextByLink(Association as, Cl1 inst, Class Cl2)` operations can be mapped to the **first** *c:Cl2* **from** *inst* **by** *as* and **next** *c* commands. These commands return all instances of a given meta-class navigable by links of the given type from a fixed instance. The iteration must be done similarly as in the previous case

In fact, the **first ... suchthat** command can be used instead of pair of **first** and **next**. Actually the **first ... suchthat** is compiled to these commands. Thus, MOLA compiler can use a closer construct to pattern as a target.

- `checkLink(Cl1 inst1, Cl2 inst2, Association as)` operation can be mapped to the **link** *inst1.as_rolename.inst2* command. The semantics of this command is the same as the semantics of this operation – check the existence of a link of the given type between two fixed instances.
- `eval(Cl inst, Expr exp)` operation is an expression interpreter and the MOLA realization to L3 must implement a generator of sequences of L3 commands that interprets the given expression. The core elements of such expressions are attribute or variable value checks. These operations can be mapped to **attr** *inst.*<*attrname*><*relation*><*expression*> and **var** <*varname*><*relation*><*expression*> commands accordingly. Arithmetic expressions can be mapped to expressions introduced by the L0' language. Constraints that are complex (Boolean) expressions where conjunction, disjunction and negation are used can be mapped to a sequence of commands which interprets the given expression.

5.4 Mapping from MOLA to L3

This section contains a detailed description of the mapping from MOLA to L3. That includes a mapping of metamodelling language constructs and a mapping of MOLA procedure and its elements to constructs of the L3 language.

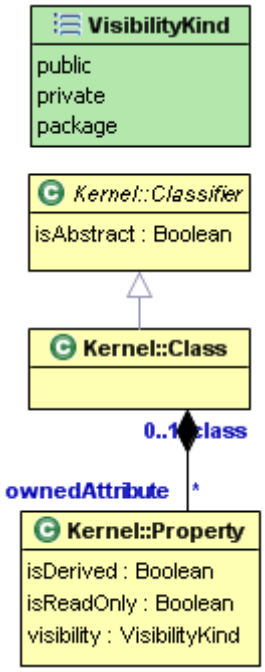
5.4.1 Mapping of Metamodelling Languages

Both MOLA metamodelling language and the Lx family metamodelling language are based on EMOF. So the mapping is straightforward. For describing this mapping we will use the meta-class names from MOLA and Lx family metamodelling language metamodels shown in Fig. 3 and Fig. 18. The MOLA related meta-class names are prefixed by the *Kernel* prefix, but the Lx related meta-class names are prefixed by the *Lx* prefix.

- Each *Kernel::Class* instance is transformed to *Lx::Class* with the same name, but since there are no packages in Lx, the *Lx::Class* name is prefixed by all parent package names.
- Both languages have pre-defined primitive types. All primitive types that are in MOLA - *String*, *Integer*, *Boolean* – are also in Lx.
- Each *Kernel::Enumeration* instance is transformed to *Lx::Enumeration* instance and each *Kernel::EnumerationLiteral* instance is transformed to *Lx::EnumerationLiteral* instance owned by the appropriate enumeration.
- Each *Kernel::Generalization* instance is transformed to *Lx::Generalization* instance. Of course, *general* and *specific* links are set to the appropriate classes.
- Each *Kernel::Association* instance is transformed to *Lx::Association* and appropriate association ends that are represented as *Kernel::Property* instances linked by *memberEnd* link to the association are transformed to *Lx::AssociationEnd* instances. They are linked to the appropriate class instances. Multiplicity, ordering and composition information of association ends are also transformed directly to Lx.

- Each *Kernel::Property* instance that is an attribute is transformed to an *Lx::Attribute* instance. Since MOLA allows only primitive or enumeration-typed attributes the correspondence is direct.

An example of the transformation is given in Fig. 20.




```
class Kernel::Classifier;

class Kernel::Class;

class Kernel::Property;

enum VisibilityKind : {public,private,package};

compos Kernel::Class.[0..1]class/ownedAttribute[*].Kernel::Property;

attr Kernel::Classifier.isAbstract:Boolean;

attr Kernel::Property.isDerived:Boolean;

attr Kernel::Property.isReadOnly:Boolean;

attr Kernel::Property.AggregationKind:AggregationKind;

attr Kernel::Property.VisibilityKind:VisibilityKind;

rel Kernel::Class.subClassOf.Kernel::Classifier;
```

Fig. 20. An example of MOLA and Lx metamodelling languages.

5.4.2 Mapping of the Procedure Headers

MOLA procedures form the executable part of a MOLA transformation. The L3 language also has procedures. Both MOLA and L3 procedures may have parameters that may be *in* (passed by value) or *in-out* (passed by reference). Both languages may have variables declared. In L3 the class-typed variables and parameters are called *pointers* and have a different syntax, so compiler must distinguish class-typed variables from enumeration and primitive-typed variables. Each non-reference class element that is used in rules in a MOLA procedure is transformed to a pointer declaration. Actually, the transformation of procedure header is straightforward and does not need a detailed description. An example of the transformation of a MOLA procedure header is shown in

Fig. 21 (the L3 code in all examples is used to better illustrate the result of compilation. Actually, the compiler produces instances of the model of an L3 program)

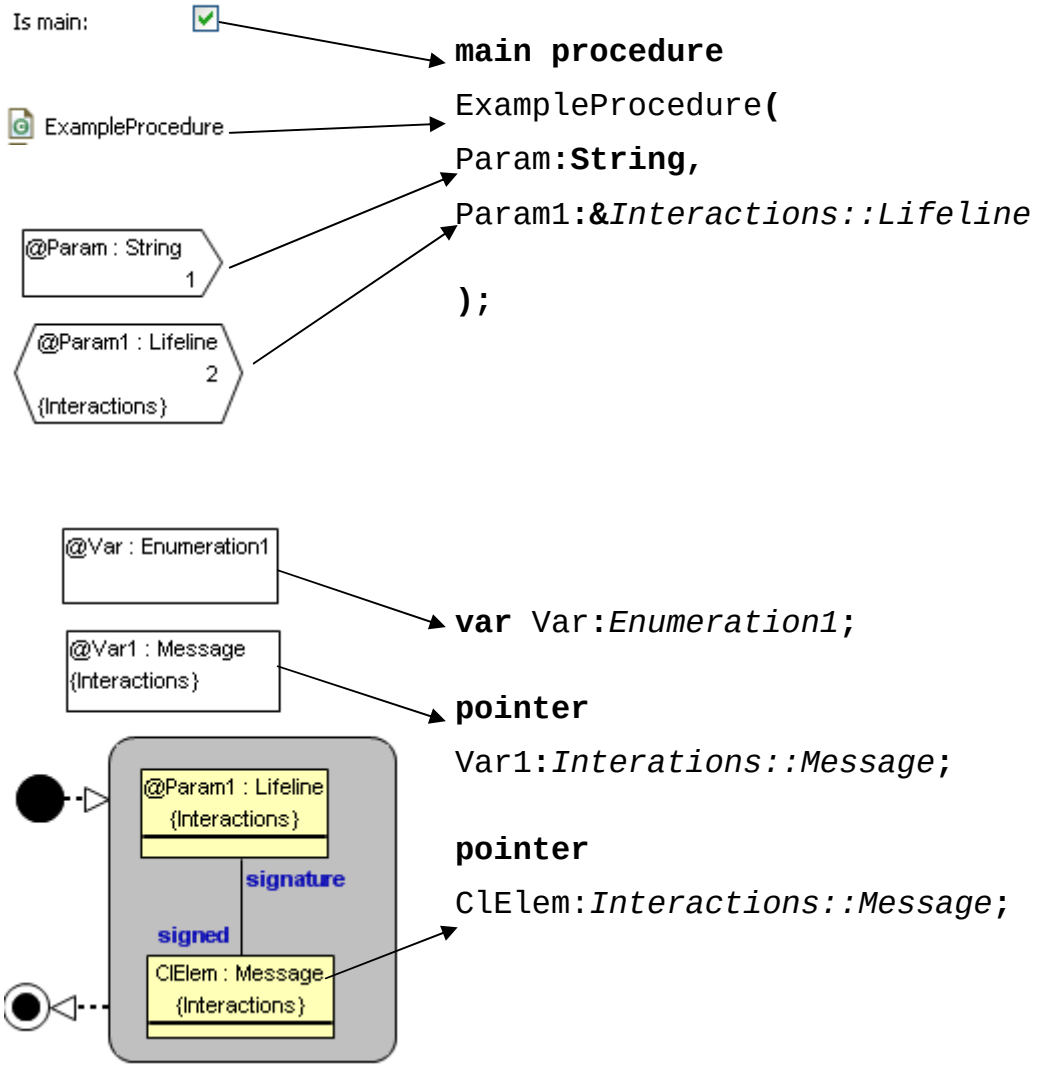


Fig. 21. An Example of MOLA Procedure header transformation to L3

5.4.3 Mapping of the Execution Control Flows

The basic statements of MOLA are rule and foreach loop. There are also other MOLA statements - text-statement, call-statement, etc. Control flows are used to determine the order of execution of MOLA statements within one MOLA procedure.

There is exactly one start symbol in a MOLA procedure. It defines the entry point of the MOLA procedure. Other statements may pass the execution control to another statement or terminate the execution of the procedure. End symbols are used to terminate the execution of the procedure. They define the exit points of the MOLA procedure. The

execution of the procedure may be terminated also by a text-statement or a rule, if the corresponding control flow is not present. Actually, a text-statement and a rule are used as traditional branching constructs (they may have two outgoing control flows, one of them labelled *ELSE*). A foreach loop contains nested MOLA statements (loop body) that are executed in each iteration. It has a special statement - loophead (rule-based loophead), which defines the entry point to the loop-body. There may be any other MOLA statement in the loop (except start-statement) – nested loops are also allowed. A statement that has no outgoing control flow terminates the current iteration of the loop. A branching statement also may terminate the current iteration of the loop, if one of outgoing control flows is not present. Other statements (call-statement, etc.) just pass the execution control to the next statement. Control flows in MOLA procedure may connect statements in an almost arbitrary way, there are only few restrictions. Incoming control flows are not allowed to the start symbol and loophead. Outgoing control flows are not allowed from end symbol. Also it is not allowed to *jump* into a loop from an outside statement (it is allowed to *jump* out).

Control flows and MOLA statements form a directed graph, where some nodes (loops) may contain a nested graph. This graph is the control flow graph (CFG) of a MOLA procedure. The control flow graph is a data structure used by traditional compilers for analysis and optimization of a program execution [85].

The most natural way to code a control flow graph in a textual language is to use a labelled block of code for every node and a *jump* command for every edge. Thus each node of the MOLA control flow graph will compile to a block of L3 code. The block of code starts with a **label** command that unambiguously identifies the block. The execution control is passed to another code block using a **goto** command. If the execution of a MOLA procedure must be terminated, then a **return** command is used.

According to the different types of statements described above we can distinguish five types of nodes in the control flow graph of a MOLA procedure and define the mapping to L3 language for these types:

- Entry node (start symbol) is a unique and mandatory node. Here we do a little optimization – no L3 code block is created for start-statement. The outgoing control flow determines the first MOLA statement that in turn determines the first code block of the procedure.

- Exit node (end symbol) is compiled to the following code block (in what follows, a simple template language is used – L3 keywords are bolded, other parts of code are shown in angular braces containing an intuitive description):

```
label <label name>;
return;
```

- Simple node (e.g. call statement) haven't an outgoing *ELSE* control flow. It is compiled to a simple code block – a sequence of commands depending on the actual type of MOLA statement and the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow.

```
label <label name>;
<sequence of commands>;
goto <next label name>;
```

- Branching node (e.g. rule) may have two outgoing control flows, where one of them may be an *ELSE* control flow. It is compiled to an **if-then-else** command. The *if-block* contains the condition, *then-block* contains the action part of the MOLA rule or text-statement and *else-block* contains a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing *ELSE* control flow. The last command in the main code block is the **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the other (*non-ELSE*) outgoing control flow.

```
label <label name>;
if
begin
<condition commands>;
end
then
begin
<action commands>;
end
else
begin
goto <next else label name>;
end;
goto <next label name>;
```

- Loop node (e.g. foreach loop) contains a nested control flow graph. Since a loop and its loophead cannot be used separately, a common L3 code block is created for both nodes. A loop is compiled to a **foreach** command. The *suchthat* block contains the condition, the *do* block contains the action part of the loophead. The *do* block contains also a **goto** command to the **label** command of the code block

that is created from the MOLA statement connected by the outgoing from the loophead control flow. The last command in the *do* block is a **label** command. This label is used to receive back the execution control from the code blocks that terminate an iteration of the loop. Thus a MOLA statement which terminates the execution of the current iteration of the loop passes the execution control to this **label** command instead of terminating the execution of the whole procedure. In fact, the execution control is passed away from the *do* block of a **foreach** command, but it is received back just at the end of an iteration. Thus, the code blocks that are created from MOLA statements within the loop body are included in the corresponding L3 loop body indirectly - using **goto** and **label** commands. The last command in the main code block is a **goto** command to the **label** command of the code block that is created from the MOLA statement connected by the outgoing control flow of the loop.

```
label <label name>;
foreach <loop variable name> suchthat
begin
<loophead condition commands>;
end
do
begin
label <loophead label name>;
<loophead action commands>;
goto <loophead next label name>;
label <loop iteration end label name>;
end
goto <next label name>;
```

The complete code of the procedure is assembled using code blocks obtained in the way just described. The first code block is determined by the start-statement. All other code blocks may be added to the procedure in an arbitrary order, because the order of execution is determined only by **label** and **goto** commands – not by the order in which command blocks are added to the procedure.

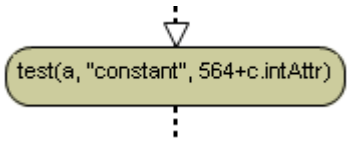
The result will be likely a sort of *spaghetti code* [95], but this causes no danger because the L3 code is just an intermediate code which is compiled further. This code is not read by a transformation developer. The wide usage of the **goto** commands does not cause any loss in the overall performance.

5.4.4 Mapping of MOLA Statements

The control structure aspect of the mapping of MOLA statements to L3 commands has already been described in the previous section. This section contains a detailed description of the mapping for each MOLA statement including data processing and pattern matching aspects.

The mapping for start and end statements has already been described. The start-statement is used to determine the first MOLA statement and end-statement is transformed to the **return** command.

The **call statement** is transformed to the **call** command. Since the mapping from a MOLA procedure to L3 procedure is one-to-one, the called L3 procedure is the same that is mapped from the MOLA procedure called by the MOLA call-statement. The L3 language allows only binary expressions to be used as actual parameters of the **call** command. MOLA allows arbitrary expressions (of appropriate type) to be used as actual parameters (the same problem is for functions in an expression). Our solution is to use temporary variables or pointers (depending on the actual type of a parameter) and **setVar** or **setPointer** commands to calculate the values of expressions. These commands must be executed before the **call** command. If the actual parameter is a MOLA variable, parameter or class element identifier, then a temporary variable is not used. An example of the compilation is shown in Fig. 22.




```
var temp_var1:String;
var temp_var2:Integer;
begin
…
label id_lab1;
setVar temp_var1=”constant”;
setVar
temp_var2=564+c.intAttr:Integer;
call test(a,temp_var1,temp_var2);
goto id_labx;
…
```

Fig. 22. Compilation of call statement

As it was described before, the **text statement** is transformed to the **if-then-else** command. MOLA text-statement has two main parts – a condition (constraint) which is expressed using OCL-style expression and a list of assignments. The condition holds if the expression evaluates to *true*. The condition is compiled to the *if* block of the **if-then-else** command. Assignments are compiled to the *then* block of the **if-then-else** command.

Assignments are used in the text statement to assign values to elementary variables and pointers. The L3 commands that are used for this task are **setVar** and **setPointer**. In MOLA the value that is being assigned is expressed using a *simple expression* of an appropriate type. A simple expression of *Integer* type may contain *Integer-typed* variable, parameter or attribute specifications, *Integer* constants, pre-defined functions (*size, indexOf, toInteger*) and arithmetic operations (addition, subtraction, multiplication). A simple expression of *String* type may contain *String-typed* variable, parameter or attribute specifications, *String* constants, pre-defined functions (*toLower, toUpper, substring,* and *toString*) and a concatenation operation. A simple expression of *Boolean* type may contain *Boolean-typed* variable, parameter or attribute specifications, *Boolean* constants (*true* and *false*) or pre-defined function (*isTypeOf, isKindOf, toBoolean*). A simple expression of *enumeration* type may contain *enumeration-typed* variable, parameter or attribute specification, *enumeration* literals or a pre-defined function *toEnum*. A simple expression of *class* type may contain a *class-typed* variable or parameter specification (pointer), *null* constant or typecast.

In L3 similar expressions are allowed, but there are few differences. They are: there is no direct typecast of a pointer, actual parameters in a function call may be only a binary expression of an appropriate type. The list of pre-defined functions in L3 also does not match all the pre-defined functions of MOLA language. The solutions of these problems are rather simple. In addition, some kinds of expressions in L3 allow more features than in MOLA, but these features are not relevant for MOLA compiler.

Table 2. Correspondence of elements used in expressions in MOLA and L3

MOLA	L3
String, Integer, Boolean, enumeration-typed constants, NULL constant	+
elementary variables, pointers	+
attribute specification	+
+,-,*,concatenation	+
direct typecast (class-typed)	temporary variable and extra **setPointer** command used
function call	temporary variables and extra **setVar** commands for complex parameters used

MOLA pre-defined functions	L3 extended library of native functions used
toEnum, toInteger, toString, toBoolean	+
indexOf, toLower, toUpper	extended library used
size, substring	+
isTypeOf, isKindOf	temporary variable and **type** command used

The complete table of correspondence is shown in Table 2. The left column describes features used in MOLA expressions and the right column shows the correspondence in L3. The plus sign means that the mapping is direct. If there is no direct mapping the basic principles of a solution are shown. It may be the usage of a temporary variable (typecast and function call) or the usage of an extended library of native functions (*indexOf, toLower, toUpper* functions).

Though L3 expressions allow Boolean operations, they cannot be used with relational operators (<, >, etc.). Relational operators may be used only in **var** and **pointer** commands. That makes the compilation of *Boolean* expressions used in MOLA more difficult.

In MOLA the simplest condition is a simple expression of the *Boolean* type (no relational operators, no Boolean operations). Then it is compiled using a temporary variable and a **var** command in the following way:

Condition:

<simple boolean expression>

```
if
begin
[<extra commands>]
setVar temp_var=<simple boolean
expression>;
var temp_var==true;
end
…
```

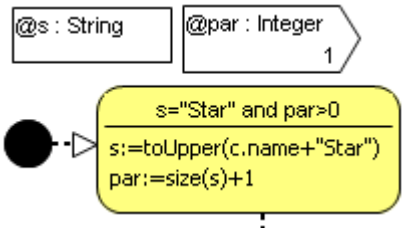
The extra commands may be needed when the extra calculations are needed, e.g. to compute argument values for Boolean-typed function call.

Usually a condition contains also a relation (>, <, >=, <=, =, <> operators can be used). Since the left and the right operands may be arbitrary expressions of the same type, the value of each expression is computed and stored in a temporary variable. Then these variables are compared using a **var** or **pointer** command depending on the type of expressions.

Condition:	**if** **begin**
<expression1><relation>	[**<**extra commands**>**] **setVar/setPointer** temp_var1=<expression1>;
<expression2>	[**<**extra commands**>**] **setVar/setPointer** temp_var2=<expression2>; **var/pointer** temp_var1**<**relation**>**temp_var2; **end** ...

A condition in MOLA may contain also Boolean operations - conjunction (**and**), disjunction (**or**) and negation (**not**) – together with relational operators. The L3 has no such features, but it is shown [16] that it is possible to construct L3 code that implements the Boolean operations. The algorithm implemented in MOLA to L3 compiler uses the same principles.

An example of the compilation of a MOLA text statement is shown in Fig. 23.



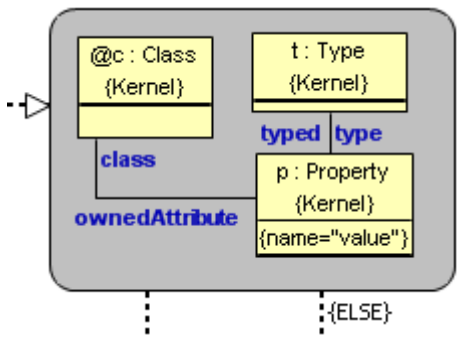

```
if begin
  setVar _mvar_6=false;
  setVar _mvar_9=s;
  setVar _mvar_10="Star";
  var _mvar_9==_mvar_10 else _mlabel_8;
  setVar _mvar_6=true;
  label _mlabel_8;
  setVar _mvar_7=false;
  setVar _mvar_12=par;
  setVar _mvar_13=0;
  var _mvar_12>_mvar_13 else _mlabel_11;
  setVar _mvar_7=true;
  label _mlabel_11;
  setVar _mvar_4=false;
  var _mvar_6==true else _mlabel_5;
  var _mvar_7==true else _mlabel_5;
  setVar _mvar_4=true;
  label _mlabel_5;
  var _mvar_4==true;
end then begin
  setVar _mvar_14= c.name:String+"Star";
  setVar s= toUpper(_mvar_14);
  setVar par= Length(s)+1;
end else begin
  return;
end;
```

Fig. 23. Compilation of text statement.

Another and the most important decision statement in MOLA is a **rule**. It is also compiled to the **if-then-else** command. The condition of the rule is expressed using a pattern. The implementation of pattern matching typically is the most demanding component to implement and also the key factor determining the implementation efficiency.

The most obvious way to compile a MOLA pattern to L3 commands is to start from one (chosen by some algorithm) class element and traverse the pattern graph. The result of such compilation is a *first* command created for the initial class element and nested *first* commands for other class elements. It is obvious that the same pattern can be matched in different ways using the basic L3 commands. Finding the most efficient way (the optimal search plan) is the main task for pattern matching. The pattern matching implementation in details is discussed in next sections.

An example of the compilation of a pattern is given in Fig. 25.




```
if
begin
  first p:Kernel::Property from c
by ownedAttribute suchthat
  begin
    setVar _mvar3=p.name:String;
    setVar _mvar4="value";
    var _mvar3==_mvar4;
    first t:Kernel::Type from p
by type;
  end;
end
then
…
```

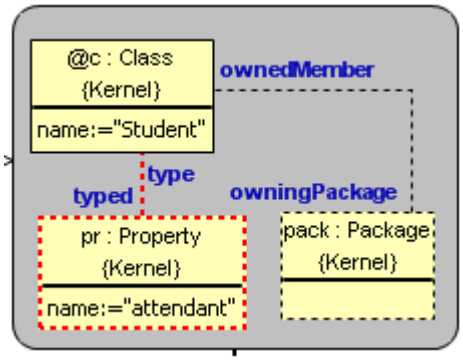
Fig. 24. Compilation of MOLA rule - pattern.

The action part of a rule consists of class elements, association links and assignments that are included in class elements. C*reate* and *delete* class elements are used to create and delete particular instances. *Create* and *delete* association links are used to create and delete links. The assignment is used to assign the value of an attribute of a particular instance. The value is specified using expressions that have been already described in previous sections. The correspondence between MOLA and L3 constructs is shown in Table 3.

Table 3. Correspondence of constructions used in action part of the rule.

MOLA	L3
create, delete class-elements	**addObj**, **deleteObj** commands
create, delete association-links	**addLink**, **deleteLink** commands
attribute value assignments	**setAttr** commands

The L3 code that is created for the action part of the rule is placed in the *then* block of the **if-then-else** command. An example of the compilation of the action part of a rule is shown in Fig. 25.




```
if begin …end
then begin
  addObj pr:Kernel::Property;
  addLink pr.type.c;
  setAttr c.name="Student";
  setAttr pr.name="attendant";
  deleteLink c.owningPackage.pack;
  deleteObj pack;
end else
…
```

Fig. 25. The compilation of the rule – action part.

The last MOLA statement described in this section is the **foreach loop**. The implementation of a loop is one of the crucial issues in the realization of the MOLA compiler. An incorrectly chosen search structure may cause serious efficiency problems. The condition of a loop is expressed using the pattern of the loophead, which contains a special class-element – the *loop variable*. The iteration is performed over all instances that correspond to the loop variable.

The loop is compiled to the **foreach** command. The condition of the loop is compiled to the *suchthat* block of the **foreach** command. The compilation of the loophead pattern is similar to the compilation of the rule pattern and is also discussed in next sections. The action part of the loophead is being compiled in the same way as the action part of a rule. The created code is added to the *do* block of the **foreach** command.

For example, it is possible to compile the loop, depicted in Fig. 11, in the following way:

```
foreach as:BinaryAssociation suchthat
  first subjEnd:Property from as by sourceEnd suchthat
  first subjCl:Class from subjEnd by class suchthat
```

```
  first domOWLCl:OWLClass from subjCl by #owlClassForCl suchthat
  first objEnd:Property from as by targetEnd suchthat
  first objCl:Class from objEnd by class suchthat
  first ranOWLCl:OWLClass from objCl by #owlClassForCl
do
  addObj op:OWLObjectProperty;
  addLink as.#obj_prop_For_Assoc.op;
  addLink op.RDFdomain.domOWLCl;
  addLink op.RDFrange.ranOWLCl;
  setAttr op.localName:=as.name;
```

As we see, **foreach** loop is naturally compiled to command *foreach … suchthat* command. The *first ... suchthat* commands are nested in each other according to the *navigation* order of the elements corresponding to the pattern- searching begins from the loop variable, continues along the both branches of the pattern, that consist from the class elements and links. The commands *first from by* are included in the block of the prior *suchthat* command. If any of these commands is not executed, that is, the corresponding *first* instance is not found, then *backtracking* takes place – the next instance, which corresponds to the previous operation, is taken. Accordingly, the main task of MOLA compiler is to arrange the *first … suchthat* commands in the order that makes pattern matching the most efficient.

5.5 The Simple Pattern Matching Strategy

Implementation of pattern matching for MOLA uses the local search plan generation strategy. This is one of the most popular strategies, however typically it requires a sophisticated analysis of pattern or even underlying model to choose the best search plan. A simple algorithm (in the sense of how complex is the implementation) is proposed which is efficient for the typical MOLA patterns used in MDSD-related tasks (it is efficient also for others if appropriate constructs are used). The simple algorithm uses the following principles:

- if the pattern contains a reference class element, then the pattern matching starts from the reference (if there are more than one, then an arbitrary is chosen).
- otherwise the pattern matching starts from the loop variable in a loophead or from arbitrary chosen element in a normal rule.
- pattern matching is continued with class elements accessible from already traversed class elements by association links.

If rule pattern contains several independent pattern fragments, then these fragments are processed independently by the same principles – such fragments can be treated as separate patterns.

Pattern matching in a regular rule is started from the reference class element, if such class element exists in the pattern. Though MOLA does not require the presence of a reference class element in the pattern, the practical usage of MOLA has shown that most of the regular rules contain it. It is because the usage of imperative control structures causes reuse of the previously matched instances, which are represented by the reference class elements in MOLA. This is one of the main reasons why such simple optimization technique works almost as well as more sophisticated approaches.

Use of reference class elements is natural also in loopheads. It is common to have a loop over, for example, all properties of a given class. This task can be easily described, using a single MOLA loop, where the pattern in the loophead is given using the reference class element and the loop variable. See the loophead of the inner loop in Fig. 26 for the typical case. In this case the pattern matching is started from the reference element (*@pack*) reducing the search space dramatically. Of course, the path from the reference class element to the loop variable may be longer. The only restriction is that cardinalities of associations along the path (except one directly before the loop variable) should be "1" or "0..1".

For foreach loop statements without a reference in the loophead, pattern matching is started from the loop variable in the loophead. Practical usage of MOLA has shown that typical tasks are naturally programmed using patterns, where cardinalities of association links leading from the loop variable are "1" or "0..1". This causes the execution of the loop to work in a linear time dependant on the number of the instances corresponding to the loop variable. Of course, this does not apply for every example, but if an appropriate metamodelling (UML-like, using composition hierarchy) and imperative algorithms are used, then this condition holds for most cases.

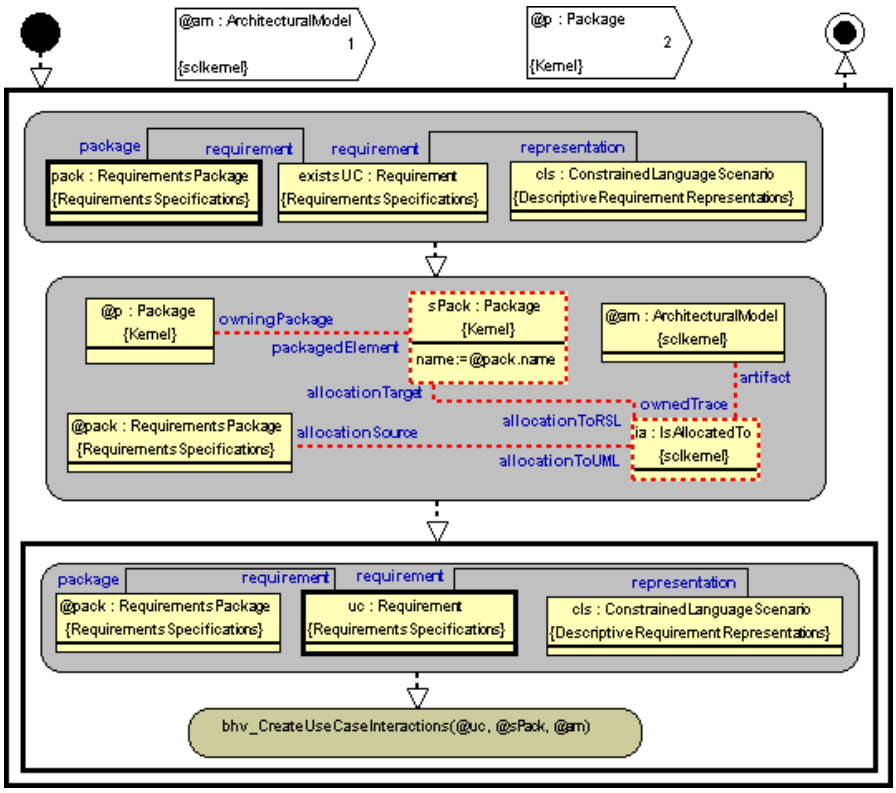


Fig. 26. Transformation example - MOLA procedure building package hierarchy.

Note the loophead of the outer loop in Fig. 26. Though cardinalities of association links leading from the loop variable are "0..*", the pattern matching started from loop variable is still efficient. Since class elements other than the loop variable provide the "existence semantics" (find first valid match), in practice this loop works also in linear time because almost all requirements are described using scenarios. In fact, this additional constraint is used to filter out those few cases where requirements are described using different means.

Note that this strategy does not even require the analysis of the cardinalities of metamodel elements at the same time remaining efficient in the practical usage. A similar pattern matching strategy is used also by Fujaba. The bound variable (reference class element in terms of MOLA), is even required by the pattern in Fujaba. However, the benchmark tests [52] have shown that this strategy performs as well as more sophisticated strategies. The same tests also have shown that an appropriate usage of the language constructs (improvement of Fujaba transformation) causes a significant positive impact on the performance. The same holds also for MOLA, however the feature which distinguishes both languages is the loop variable in the MOLA foreach loop. First of all, the transformation becomes more readable for human reader; secondly, it gives slight

advantage in the performance of the pattern matching. It allows iterating through the instances corresponding to the loop variable only, while other patterns elements are checked just for the existence. On the contrary, Fujaba is forced to examine corresponding instances to all pattern elements in the foreach loop.

5.6 Benchmark Results

The simple pattern matching strategy has been implemented in the MOLA Tool for MOLA language. The benchmark tests for this implementation have been carried out. The example described in the Section 4.4 has been reused. The same tests have been repeated for MOLA implementations for MIIREP, JGraLab and EMF repositories.

Table 4. Benchmark results of MOLA implementation for different repositories.

	Transformation execution time (ms)		
Model size (N)	**MIIREP**	**EMF**	**JGraLab**
1750	134	78	277
3500	266	106	388
17500	1349	378	1366
35000	2856	659	2601
87500	6872	1926	6288
175000	15222	3221	11609
350000	27614	7348	23420

The benchmark results are shown in Table 4. Since the transformation which is shown in Fig. 15 has been tested, similar measures are used. The first column depicts the size of model used for tests. The model size (N) is a total number of class instances in a source model. Transformation execution times for MOLA implementation have been shown in the next three columns. The times have been measured in milliseconds rather seconds as it was done in the previous test (see Table 1). It should be noted that the performance has been much better than for previous implementation. For example, the models of size N=3500 have been processed in less than one second in the new implementation, while the old (SQL-based) implementation executes the same transformation in 65 seconds (see Table 1).

The MOLA implementation through Lx language family and simple pattern matching strategy perform in less than 1 second for models of size N≤10000 which is a typical size of model used in MDSD. Since the example used in the benchmark is a typical MDSD transformation (all instances in a model of tree-like structure are processed), benchmark tests show that MOLA implementation is efficient for MDSD-related tasks.

It is interesting to compare also the performance of MOLA on different model repositories. For all repositories the execution times grow almost linearly against the size of a model. The EMF repository has shown the best results. Two other repositories (MIIREP and JGraLab) perform equally strong. MIIREP is better for small models, but JGraLab is better for larger models (the execution times grow slower for JGraLab). However, the difference between results is quite narrow. It should be noted, that all implementations have been tested on large source models (N=350000). They have been processed in less than a half minute. Note that in the example every source model element must be processed and target element created.

It should be noted that the performance of a repository has a great impact on overall performance of transformation technology. For example, the loading and saving EMF-based models are quite inefficient compared to the execution of transformations. For a model of size N=350000 the loading data took ~16 seconds and saving data took more than 10 minutes, while execution of transformation took just ~7 seconds. JGraLab has much better results – loading model took ~1 second and saving model after transformation took ~3 seconds. However, for all repositories the saving time of model increases non-linearly. This problem should be taken into account, but typically MDSD-related transformations are used within some modeling tool and model is saved only when a work with the tool has been ended.

5.7 Local Search Planning Using Annotated Metamodels

MOLA language can be used not only in the MDSD-like domains, where patterns are similar to those described in the previous section, but also in others. A more advanced pattern matching technology should be used to support efficient matching of these patterns. The classical local search planning approach is used in MOLA for these cases. This algorithm uses similar principles as the implementations of the languages

PROGRES [48], VIATRA [49] and GrGen [50]. It should be noted that this algorithm hasn't been fully implemented in the MOLA Tool yet. At first a **search graph** (*host graph*) is built corresponding to the pattern. By using the association cardinalities, existing in the metamodel and additional annotations, the *weights* are placed on the edges of the search graph. The weight of the edge reflects the priority with which the operation, corresponding to this edge, is chosen in LSP. The way, how the weights of the edges of the search graph are chosen, is the essential difference among all implementations of LSP generation algorithms. Subsequently in the search graph the *minimal spanning tree* is located, from which LSP is read in the final step.

5.7.1 Local Search Plan Generation

The search graph is built for a pattern in the following way (see Fig. 27):

- One vertex is added to the search graph for each class element in the pattern.
- Two oriented edges, which connect the corresponding vertices, are added to the search graph for each association link in the pattern. These edges represent a possible navigation options from class instances which correspond to class elements in the pattern. The first option is to check the existence of corresponding link using L0 command *link*. It can be done in a constant time and it requires that both instances at the ends of the corresponding association link have been matched. The second option is to match a class instance using L0 command *first from by*. In this case only an instance corresponding to source vertex in the search graph (class element in the pattern) has to be known.
- A special vertex – a *root* vertex – is added to the search graph. Edges are added outgoing from the root vertex to every other vertex. They represent a possibility to match a class instance corresponding to a class element using *first* command.

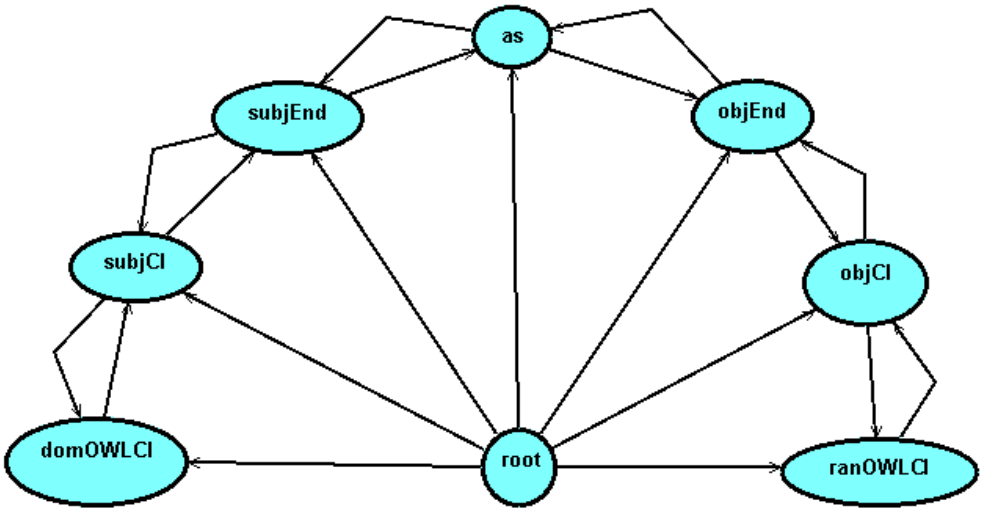


Fig. 27. Search graph without weights for the pattern in Fig. 11

A local search plan corresponds to a spanning tree in a search graph. The root of the spanning tree is the root vertex. Every edge in the spanning tree corresponds to a *first suchthat* command. Those pairs of edges (corresponding to the same association link) which are out of spanning tree are compiled to *link* commands. The *first suchthat* commands are nested accordingly to a traversal order of the spanning tree.

There are many ways to construct a spanning tree in a search graph. Consequently, there are many local search plans which implement the pattern matching for the given pattern. For example, one can take all edges from the root vertex and it will be a spanning tree. However, this search plan can be hardly called efficient. Every set of instances which corresponds to the class elements in the pattern should be examined in the worst case. A local search plan is more efficient if class instances are matched using links from already found instances. It implies checking of less model fragments which means less execution of backtracking step of *first suchthat* commands. Thus, the best search plan is one which requires the smallest number of basic lookup operations executed – the smallest number of backtracking steps of *first suchthat* commands in the case of MOLA. Let's call the number of basic lookup operations performed during the execution of a local search plan the *cost* of the search plan.

A pattern matching algorithm has to find out how *expensive* are each *first* command for every edge in a search graph. Basically it means to find out how many instances in the worst case should be examined to find a valid one. The nature of patterns

in model transformation languages is such that pattern elements (class elements in MOLA) represents instance of a given class. A *first suchthat* command also iterates through instances of a particular class, therefore an appropriate measure to estimate the potential number of instances to be checked is the total number of instances of the given class. A *first from by* command reduces the number of possible checks to the number of connected class instances by links of particular association. If one can provide the number of instances needed to be checked by operations corresponding to edges in the search graph (cost of operation), then these numbers can be put on the corresponding edges as weights. Now in the weighted search graph we can try to find the cost of particular search plan.

Since for every search plan there is a spanning tree representing it in the search graph, let us assume that in the spanning tree there are edges with weights $c_1, c_2 \dots c_n$, where n is a number of class elements in a pattern. These weights correspond to the largest possible number of operations, which are executed in order to find a corresponding instance. As the commands are executed successively and backtracking takes place, then in the worst case the cost of a local search plan is $C_i = c_{i1} + c_{i1}c_{i2} + \dots + c_{i1}c_{i2}\dots c_{in}$.

The *best* search plan is a plan with the lowest cost – the lowest C_i. We must take into notice that for every search plan $C_i \leq n c_{i1}c_{i2}\dots c_{in}$, therefore to find the best search plan means to find a search plan having the smallest $c_{i1}c_{i2}\dots c_{in}$. It means that we must find a spanning tree in the search graph which has the lowest product of all weights of corresponding edges. It can be found by using, for example, the efficient *Chu-Liu/Edmonds* algorithm [96], which finds the minimal spanning tree in the directed graph. We must note that this algorithm is searching a spanning tree with the smallest sum. Since all weights in the search graph are positive (they are number of instances), they can be replaced with their logarithms. In such way the *Chu-Liu/Edmonds* algorithm can be used to find *minimum product spanning tree* (because $\lg(ab)=\lg(a)+\lg(b)$). When the search plan is found, the appropriate L3 commands must be created which is a quite simple task.

As it was mentioned, similar algorithms have been implemented in several model transformation languages [48], [49], [50]. The main difference is in the way the costs of operations (weights of edges) are determined. In [50] the runtime analysis of a model is

performed before every execution of a pattern. In [49] the analysis of models is performed in the design time which works if there are models available. In [48] the information from pattern and metamodel is used (basically the cardinalities of association ends).

In MOLA we are using only information which is available at the design (compile) time. In fact, pattern and metamodel is available only. So, what useful information about number of instances can be obtained from a pattern specification? Patterns in MOLA may contain a reference – in a previous rule already found instance. Such instance is not searched at all – it has been already found! The corresponding edges in the search graph can have weight 1 – this instance can be found in a constant time. No other information about operation costs is in pattern. However, a metamodel shows cardinalities of associations corresponding to association links in a pattern. When navigating from an already located instance, the number of the class instances to be checked, depends on the cardinalities of the corresponding associations. If the cardinality is `0..1` or `1`, the navigation takes place in constant time, therefore the weight of the corresponding edge is 1. If the cardinality is `1..*` or `*`, then in the worst case all instances of the certain class must be reread. However, the practice shows that the real models are rarely full graphs and the majority of the real association cardinalities are less by a number of times compared to the total number of the class instances. Since there is no more information on actual cardinalities in a model, the cost estimation for operations navigating by * or 1..* associations can be based on these assumptions only. Therefore in MOLA a simple cost model can be used:

- For an edge to a vertex representing a reference $c_i=1$
- For an edge from the root vertex $c_i=1000$. Of course, it is not a precise number, but all other weights (in fact, a weight for edges representing * associations) can be adjusted accordingly to represent a proportion of instances in typical models
- For an edge if it corresponds to the end of MOLA association with cardinality * or 1..* $c_i=100$.
- For an edge if it corresponds to the end of MOLA association with cardinality 1 or 0..1 $c_i=1$

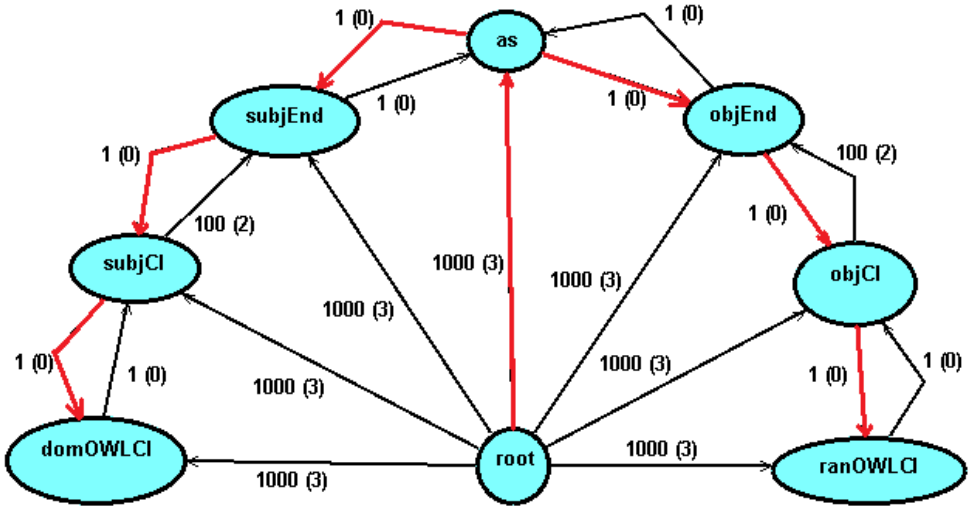


Fig. 28. Search graph with weights and minimum spanning tree depicted for the pattern in Fig. 11

See Fig. 28 where weights are added to the search graph for the pattern (see Fig. 11). The red edges denote the minimum product spanning tree – the *best* local search plan. In the parenthesis the logarithms of weights are shown which are actually used by the *Chu-Liu/Edmonds* algorithm. In this case, there are several equally efficient search plans (it is possible to start pattern matching also from *subjEnd* or *objEnd* nodes).

It should be noted, that the simple pattern matching algorithm described in the previous sections generates the same local search plan as just presented. It shows that the simple pattern matching algorithm works as efficiently as more sophisticated algorithm for such MDSD-related task. However, the simple algorithm has been designed taking into account the specifics of MDSD-related tasks. Of course the algorithm described in this section will perform better (or at least as well) for other tasks. But the main *value* of this algorithm is the possibility to integrate it with the annotation mechanism which allows using domain knowledge in the pattern matching in a simple and elegant way. The annotation mechanism is described in the next section.

5.7.2 Annotation Mechanism

The search algorithm described above optimizes the search plan selection using only data from the metamodel and pattern specification. Other approaches that are based on the statistical analysis of the model collect actual cardinalities for classes and associations (the number of instances of the given class in the model) give very efficient

results, however there are situations where such analysis cannot be made (e.g. the runtime repository does not support the required statistics for runtime analysis or there are no models created yet in the case of offline analysis). Therefore we propose an approach which allows using developer's knowledge of model constraints that otherwise could be obtained only by analysis of existing models. A part of actual cardinalities can be already predicted at the design time of a transformation. Development of a transformation requires a good knowledge of the corresponding domain. Therefore, the transformation developer should be able to predict prospective cardinalities. Of course, the precise number of the instances cannot be predicted, except for singleton classes. However, the proportion of instances for different classes is frequently known. For example, the number of properties in UML model is several times greater than the number of classes. Since neither the metamodelling standard MOF, nor UML class diagrams provide convenient means for the specification of the prospective cardinalities, we propose to annotate the metamodel and patterns in MOLA. Our goal is to have a simple, handy annotation mechanism that helps to select an efficient search plan for the pattern matching.

We allow annotating classes and association ends in the metamodel and class elements and association link ends in patterns. An annotation predicts the number of instances for classes and the number of instances reachable by links for association ends. Pattern matching algorithm takes into account the annotations, and edge weights in the search graph are adjusted accordingly. In fact, an annotation sets the priority on the pattern element. The lower the predicted number of instances is for the pattern element, the higher priority it gets for the pattern matching. Annotations made in the metamodel affect the pattern matching algorithm in every rule where pattern elements of the corresponding type are used. Annotations made in the pattern affect the pattern matching algorithm only in the scope of the rule. The developer annotates metamodel elements during the development process of the metamodel. Since metamodelling requires the knowledge of the modeled domain, typically there are no problems to resolve actual cardinalities. It should be noted that annotations are optional - they are additional means to improve the efficiency of transformations. The following annotations can be used:

SINGLE - denotes that the class (or navigation result) has at most one instance. Such instances and links as well as references are preferred for the pattern matching.

FEW - denotes that the class (or navigation result) has a nearly constant number of instances, or it is relatively low compared to the total number of instances in the model. For example, we can expect that in a UML class diagram a typical class will have about 5-10 properties, and this number is independent of the model size. Such links will be preferred over links that are not annotated for the pattern matching.

MANY - denotes that the class (or navigation result) has a relatively large number of instances, and this number grows together with the size of the model. For example, in a UML class diagram the number of typed elements for every type grows as the size of the class diagram increases. Links that are not annotated will be preferred over links with the *MANY* annotation for the pattern matching.

As annotations do not show a precise number of instances, but only the number of the corresponding class (or the result of navigation) instances against the total number of instances in the model, then in the *cost model* we choose weights, which correspond to a probable number of instances in the underlying models:

- For the edge from the *root* vertex if it
  - is to SINGLE annotated vertex or to a vertex corresponding to the reference, then its weight is $c_i=1$,
  - is to the vertex without annotations, then $c_i=1000$. Let us assume that this is a typical number of instances in the model, and the rest of weights we choose proportionate to this weight,
  - is on *FEW* annotated vertex, then $c_i=100$,
  - is on *MANY* annotated vertex, then $c_i=10000$.
- For the edge if it corresponds to the end of MOLA association, which
  - is with a cardinality `0..1` or *SINGLE* annotated, then $c_i=1$.
  - is without annotation with cardinality `*` or `1..*`, then $c_i=100$.
  - is *FEW* annotated, then $c_i=10$,
  - is *MANY* annotated, then $c_i=1000$.

Therefore, by using only information from the metamodel, which is supplemented with the corresponding annotations, the real cardinalities of the model elements are taken into notice. Although they are not denoted absolutely precisely, it is enough that there is information available about the proportion of number of instances in a model. The chosen weights seem to be appropriate.

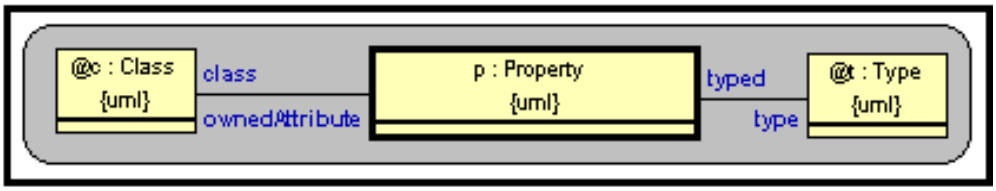


Fig. 29. Pattern example - annotation use case.

Fig. 29 shows a pattern in a loophead where annotations help to find the best search plan. This loop iterates trough every property (*p*) of the given class (*@c*) having the given type (*@t*). The problem is that associations *ownedAttribute* and *typed* both have cardinality "*" and without additional information both are treated equally (un)efficient for pattern matching. However, in practice the average number of owned attributes for a class is by magnitude less than typed properties for a type. Therefore, adding annotations *FEW* and *MANY* to *ownedAttribute* and *typed* association ends accordingly gives the desired result (see Fig. 30). The pattern matching is started from the reference *@c* and continued with the loop variable.

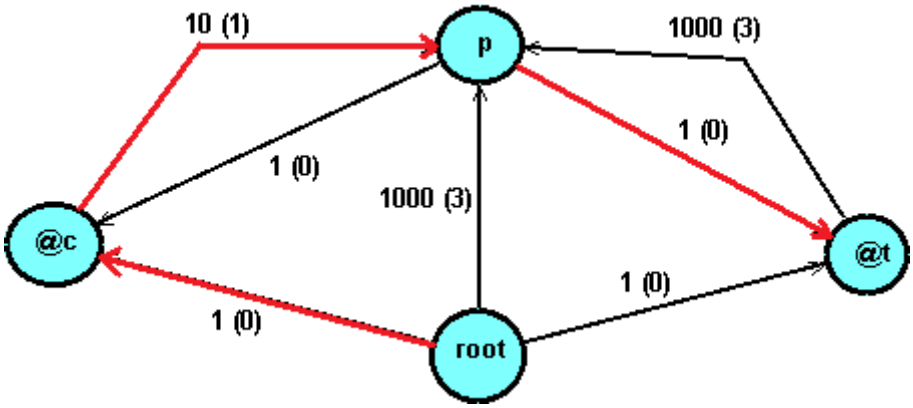


Fig. 30. Search graph with minimum spanning tree depicted for pattern in Fig. 29

CHAPTER 6

Use Cases of MOLA

MOLA language and tool have been used practically in several projects. This chapter describes two main use cases of MOLA – the typical MDSD tasks in the ReDSeeDS project [23] and specification of functionality for tools built with the METAclipse framework [8].

6.1 ReDSeeDS

MOLA language has been used in the EU 6th framework project ReDSeeDS (Requirements-Driven Software Development System). The goal of ReDSeeDS project is to create framework (languages and tools) for MDSD based development. ReDSeeDS framework includes the basic reuse approach. This approach is case-based, where a reusable case is a complete set of closely linked (through traceability links created by model transformations) software development technical artefacts - models and code.

ReDSeeDS project took place between years 2006 and 2009. Universities from Germany, Poland and Latvia, as well as, industrial partners from Poland, Germany, Lithuania and Turkey were participants of the project. Author of this thesis has actively participated in activities of ReDSeeDS project related to MDSD.

The ReDSeeDS approach covers a complete chain of models for MDSD – from requirements to code. Each transition in this chain is to a great degree assisted by formal model transformations. Requirements are specified in the requirement specification language RSL [97], which has been developed as part of the ReDSeeDS project. A significant part of RSL is the specification of requirements for system behavior in a controlled natural language. The next models in the model chain are obtained using model transformations which are specified using MOLA language. Transformed models are described using a ReDSeeDS-specific subset of UML. This subset together with RSL forms the ReDSeeDS Software Development Specification Language (SDSL). Updates after every transformation step can be made also manually. A UML modeling tool *Sparx Enterprise Architect (EA)* [98] is used within ReDSeeDS project. It is a commercial tool

which allows creating and updating UML models. The interoperability between ReDSeeDS engine and EA is implemented also using model transformations.

During the ReDSeeDS project two model-based methods [97], [99] have been proposed and the corresponding sets of transformations in MOLA developed. Both methods use the RSL and SDSL (UML) to specify models and ReDSeeDS engine to store and process them. However, the essential difference is the set of design patterns (*architecture style*) used by both methods.

6.1.1 Description of Keyword-Based Approach

The keyword-based approach [99] has been developed by IMCS, University of Latvia. Starting from requirements, a chain of models (see Fig. 31) for a MDSD of the software system is used. To a great degree, this chain is inspired by the classical MDA approach. However, the specific structure and construction principles of models in this approach are determined by the chosen architecture style, which includes the set of selected design patterns. All the models are built in UML using an appropriate profile.

Specific keywords are preserved by the keyword-based approach. If the pre-defined keywords (e.g. *select, show)* are used in the requirement specification, then they become the specific constructs in the target model (e.g. selection from a list, calling appropriate user interface method).

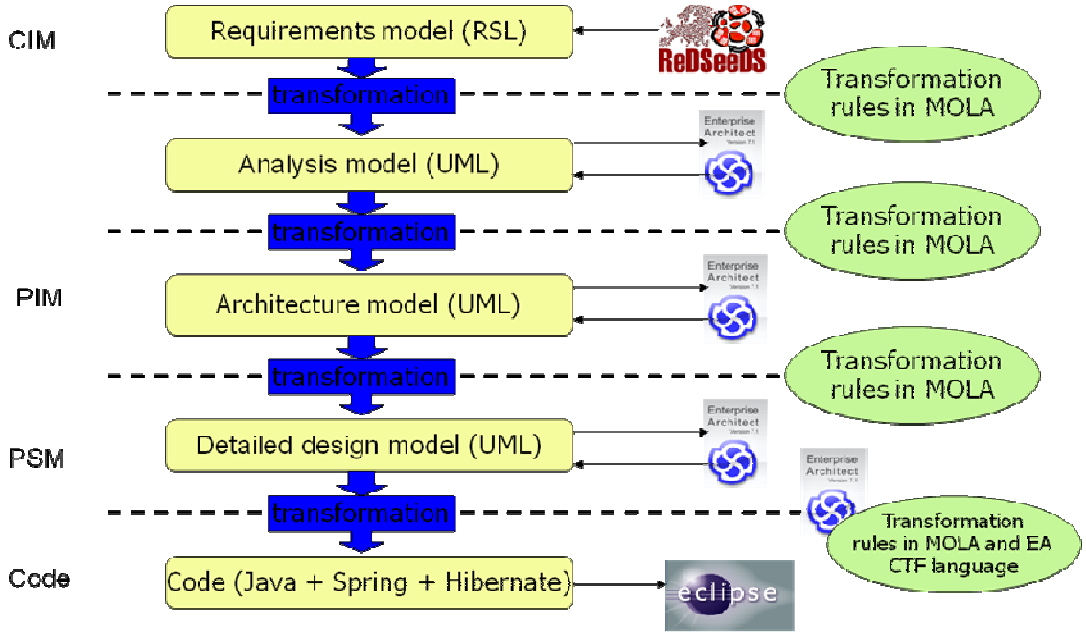


Fig. 31. Model chain for keyword-based approach.

Initially the Analysis model is extracted by transformations from requirements. This model has no direct counterpart in the classical MDA chain. In the Analysis Model the most important part is a class diagram describing the main concepts of the software system to be created. Stereotypes are used to distinguish different types of concepts according to the Analysis Profile.

The next model in this chain is the Architecture model. In this model, the implementation structure is represented according to the behavior extracted from use case scenarios. This model is platform-independent and could be used as a basis for 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 requirements permit to generate a non-trivial part of solution behavior.

The final model in the chain is the Detailed Design model. From this model code fragments for the selected platform can be generated. Currently the chosen platform is Java in the Spring/Hibernate framework [100], [101]. In this model stereotypes corresponding to Spring-specific annotations are used. In the final step the data from this model are transformed to Java code with Spring/Hibernate annotations.

6.1.2 Description of ReDSeeDS Basic Approach

The ReDSeeDS Basic approach [97] has been developed by Warsaw University of Technology. Just like in the keyword-based approach a chain of models (see Fig. 32) for a MDSD of the software system is used. The ReDSeeDS Basic approach includes three transformations steps.

The first transformation step creates the architecture model from the requirements written using RSL. This approach concentrates on automatically generating the components of the system and interactions between various parts of the system and user. A set of sequence diagrams is generated in the architecture model. Methods are added to the appropriate interfaces for each call in the sequence diagrams.

After the architecture is ready (generated from requirements and enhanced by an architect), it can be transformed into the detailed design. The transformation process uses only the information contained in the architectural model, assuming that transformation from requirements to architecture extracted all the possible information for generating the detailed design model. The specific rules to a large degree are based on the chosen design patterns (e.g. DTO, DAO and Factory). The rules assume a Java and ORM facility (e.g.

Hibernate) to be used as the basis of the platform, but no specific details of the platform appear in the rules. Therefore they could be applicable to other kinds of platforms as well.

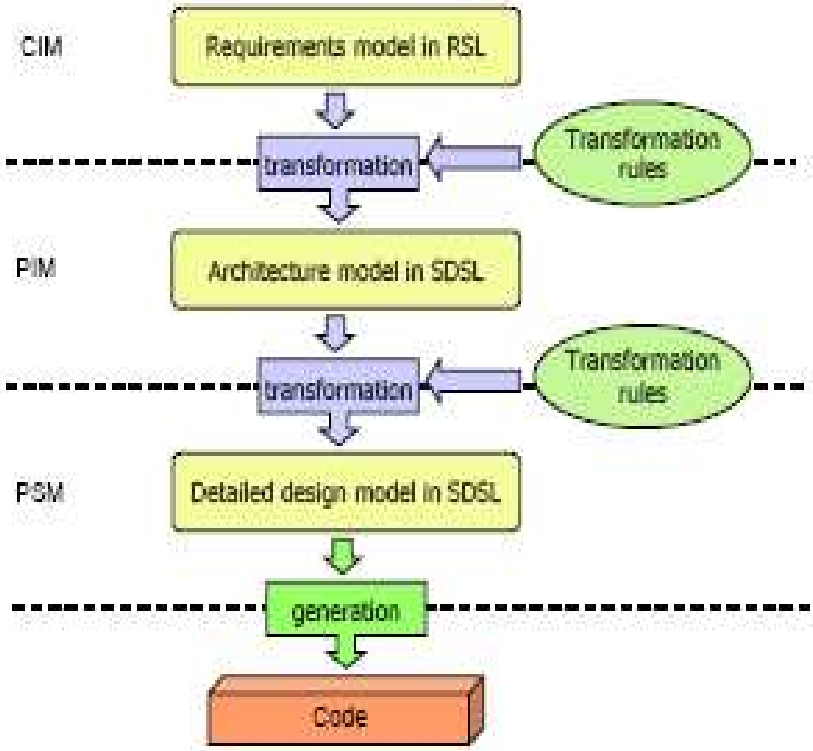


Fig. 32. ReDSeeDS basic approach.

Detailed design of a software system is the lowest level of its specification. It contains all the logical elements - classes and relations between them for each component in the architectural specification. The detailed design model is the basis for implementation in a specific programming language (e.g. Java, C#).

The EA code generation templates are applied to detailed design model in the last step. The package hierarchy, declarations for all classes (DAO, DTO, etc.) and methods are included in the generated code. Bodies of obtained methods should be filled in manually, since the detailed design model in this style in fact contains no behavior.

6.1.3 Empirical Study of Pattern Matching Cases in ReDSeeDS

In this section the analysis of typical patterns in the ReDSeeDS project is done. As it was mentioned before, one of the goals of the project is MDSD using RSL and UML

languages. The main idea is to obtain a part of the software system automatically from requirement specification using model transformations.

To approximately estimate the volume of the transformations written during the ReDSeeDS project we are giving some statistics. The model-based methodologies used in the project cover quite a large subset of UML being generated - UML class, activity, component and sequence diagrams are being generated. Both methodologies include several transformation steps. The first step for both methodologies is the transformation of requirements. The next steps are generating new UML models adding more specific details. ~350 MOLA procedures have been developed during the ReDSeeDS project. They include ~200 loops and ~800 rules that give ~1000 pattern specifications. We have investigated the structure of patterns used in the project and most of them are fit to the simple pattern matching strategy used by MOLA.

Fig. 26 refers to the typical usage of loops in ReDSeeDS project - the MDSD tasks are *compilation*-like jobs where every element of the source model is processed and corresponding elements in the target model are created. Since RSL and UML model elements form a tree-based hierarchy, the transformation algorithms traverse model elements in the top-down style starting from the top elements of the hierarchy. Therefore, the most natural way to describe such traversing is by using nested foreach loops referencing the previous loop variables. The pattern may contain additional class elements for collection of all necessary neighborhood instances or specifying additional constraints on the existence of appropriate nearby instances.

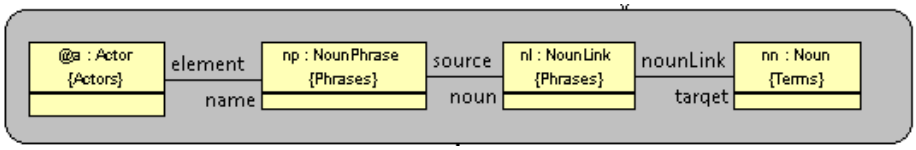


Fig. 33. Pattern example - collecting nearby instances.

Another typical pattern used in the ReDSeeDS project is depicted in Fig. 33. This pattern finds the name of an actor (names are coded as noun phrases in RSL). Note, that all associations leading from the Actor class to the Noun have cardinality "1" or "0..1" - each actor has exactly one name (represented by noun phrase), there is only one noun link for each noun phrase and every noun link is connected to exactly one noun. Therefore this pattern is matched in constant time when the simple pattern matching strategy is applied. This is a typical case where MOLA rule is used to collect the nearby instances.

A variation of the previous pattern is shown in Fig. 34. This pattern describes the collecting of nearby elements of a UML interaction. The owning classifier and the component corresponding to the lifeline named "UIComponent" should be matched. Unlike in the previous example there is an association with cardinality "*" leading from the referenced element (to Lifeline). However, as we see in practice, typically there is only one model element in the model satisfying the given constraint and the *suspicious* association has low cardinality in practice. In this case there are no more than 5-10 lifelines per interaction. Thus this pattern matches in linear time with regard to the number of lifelines in the given interaction, which is relatively low.

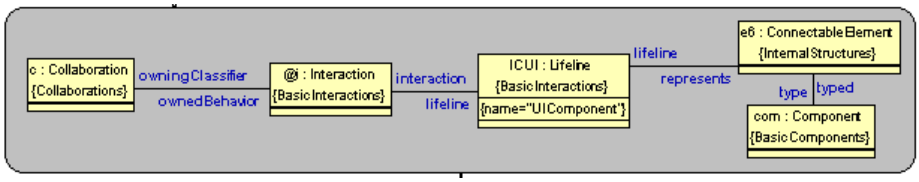


Fig. 34. Pattern example - collecting nearby instances using additional constraints

We have tested the transformations on several sufficiently large software cases developed within the ReDSeeDS project. The patterns described above are the most typical patterns used in MOLA transformations for the ReDSeeDS project. The total amount of such patterns is about 95% of all patterns. Some specific sub-tasks require non-typical patterns which theoretically may cause insufficient pattern matching performance, however in practice they are performed on elements which are relatively low in number compared to the number of constrained language sentences. Thus, they do not affect the overall performance of pattern matching.

There was made a conjecture that a transformation program in MOLA written in an appropriate style becomes efficient at the same time [67]. Our empirical analysis of typical patterns in the ReDSeeDS project confirms that this holds also in praxis and MOLA is a suitable model transformation language for MDSD-related tasks. In this case the simple pattern matching algorithm gives efficient results.

The ReDSeeDS basic approach has been implemented in MOLA and executed on various requirements specifications. For example, the requirement specification containing 8 scenarios, 42 constrained language sentences has been transformed to architecture model in ~6 seconds. The target model has 662 UML elements in total including 24 packages, 10 components, 31 classes, 17 interfaces, 71 methods, 8 sequence diagrams. The detailed design model has been generated in ~5 seconds from the

architecture model. The target model contains 451 UML elements including 16 packages, 44 classes, 17 interfaces, 169 methods.

This approach has been tried also for a real-life example. The requirements specification of simple internet banking system containing 19 scenarios, 102 constrained language sentences has been transformed to architecture model in ~10 seconds. The target model has 2114 UML elements including 27 packages, 12 components, 72 classes, 59 interfaces, 218 methods, 19 sequence diagrams. The detailed design model has been generated in ~16 seconds from architecture model. The target model has 1425 UML elements including 18 packages, 116 classes, 59 interfaces, 507 methods.

It should be noted that total time of transformations execution turns out to be almost linear with regard to the total number of constrained language sentences in the requirement scenarios specified in the RSL for the case. The total transformation execution time seems to be reasonable for such a real-life example, because these experiments considered regeneration of the whole model. In fact, the importing and exporting models from and to the UML modeling tool (EA) have executed significantly longer then transformations itself. It is also possible to specify transformations regenerating just a part of the model which requires to be updated accordingly to changes made in source models.

6.2 ReDSeeDS Integration with Sparx Enterprise Architect

As it has been already mentioned in the previous section, *Sparx Enterprise Architect (EA)* is a UML modelling tool, which was used in ReDSeeDS project. It is a popular modelling tool (also in Latvia), which allows creating UML models and generating a code for many programming languages (for example, Java, C#, C++). In ReDSeeDS tool (engine) UML models are stored in JGraLab [66] model repository, but the tool itself does not provide possibility of editing and graphical viewing. Therefore it was necessary to provide a model transfer between EA and ReDSeeDS tool. For this purpose the model transformation language MOLA is used.

In order to provide the exchange of models between the mentioned tools, the format, in which EA stores UML models, was investigated. It was described by using metamodelling means, that is, a metamodel was built, which directly reflected the inner structure of the EA models. Thus, in a simple way a universal tool is built, which

transfers UML models from the EA database to JGraLab, and vice versa. The whole logically complicated work- model transformation between EA and ReDSeeDS formats (metamodels) in this case is possible to execute by means of the model transformations, which is a more suitable manner of model processing than the programs written in programming languages are.

The general scheme of model exchange between the mentioned tools can be seen in Fig. 35. A similar manner can be used also in other cases when the model exchange between different modelling tools is necessary.

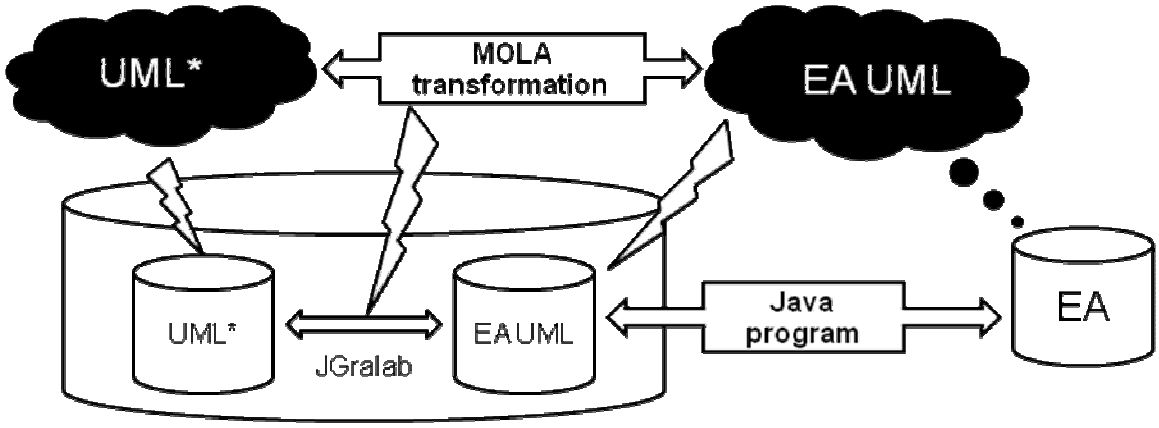


Fig. 35. EA and JGraLab model exchange schema.

Also in this use case of MOLA the use of pattern matching was sufficiently efficient. The task of model exchange is rather similar to the typical MDSD tasks that require processing of all elements of one corresponding type and creating appropriate elements in the target model. Thus the suitability of the chosen algorithms for this type of tasks was shown once again. It must be noted that in the tool integration tasks also high performance is important. However, it depends not only on efficiency of model transformations, but on the efficiency of underlying tool, the EA in this case. The API of EA has been used to import and export models to and from the tool, however it causes the major slowdown of overall performance. Unfortunately there are no better ways to collaborate with EA, but this approach seems to be more efficient for tools with more efficient implementation.

6.3 Tool Building in METAclipse

MOLA Tool has been built on the basis of METAclipse tool building framework [8], which also has been developed by the University of Latvia, IMCS. METAclipse is a metamodel and transformation based tool building platform, which is specially fit for the support of complicated graphical domain specific languages, and MOLA is such a language. From the technical point of view, METAclipse is a set of Eclipse plug-ins which extends the functionality of standard Eclipse components EMF, GEF and partially, GMF [26] [102] [63]. It contains advanced presentation engines, which support graphical diagram building, property editing and all other diagram and model related facilities. More precisely, the engines perform all the various visualization and user interaction related tasks in a standard way typical to Eclipse environment, they do these jobs on the basis of a fixed presentation metamodel. However, the main functionality of a tool based on METAclipse is defined by transformations, which link the domain and presentation (visualization) models in the tool, fill up property dialogs, and process the updated property values. In METAclipse framework these tool-specific transformations are built in MOLA language. Architecture of METAclipse framework is shown in Fig. 36.

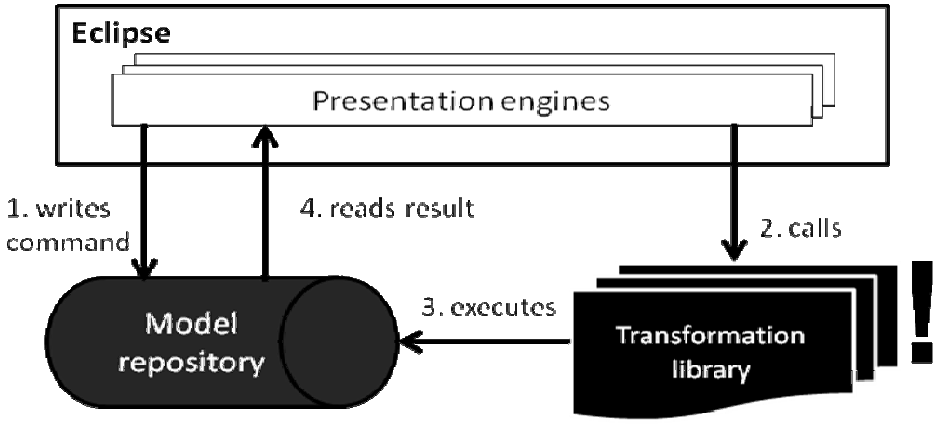


Fig. 36. Architecture of the *METAclipse* platform.

Each of the *METAclipse* engines exposes its functionality to transformations through a strictly defined metamodel that serves as an interface through the transformations and engines. The set of commands that can occur in the given engine as a result of user actions is also part of the metamodel of the engine. Commands are used to invoke the transformations. Each instance of a command represents an atomic user action and is the smallest piece of an action in *METAclipse* platform. All actions that require

purely graphical changes are processed directly in the *METAclipse* engines. Only the semantic actions (the ones that trigger changes in the domain model or changes in the presentation model that are unique to a specific tool) are triggering commands and passed to the transformations for processing. *METAclipse* platform immediately filters out the commands that do not require the invocation of transformations and invokes the mechanisms of the corresponding engines in order to make the changes in the models. Therefore, listeners that do not require the invocation of transformations are implemented already in the platform. Command listeners for processing of semantic actions have to be implemented in the transformation library as branches of the main model transformation with branching conditions that depend on the passed command.

The latest version of MOLA Tool has been built using the MOLA Tool itself. Initially source procedures of MOLA tool have been built using the previous version of MOLA Tool. Currently the source procedures of the *MOLA* editor have been completely transferred from *Generic Modeling Tool* environment to the MOLA editor implemented with the *METAclipse* platform. The current functionality of the editor is defined by ~450 *MOLA* procedures.

The efficiency of model transformations is even more important in the context of tool building than in MDSD-related tasks. Transformations are executed reacting on actions performed by user. Response must be as fast as possible. Practical usage of MOLA Tool has been shown that transformations are being executed efficiently. To verify it, the transformation execution time is measured for user actions which require significant effort of transformation compared to the effort of METAclipse framework itself. For example, a class name is shown on every class element in a MOLA program. If the name of a class is changed (user changes it) then every occurrence of this class in class elements must be updated. To test the performance of transformation implementing such action we created a MOLA project having 800 class elements corresponding to the same class. Changing the name of the class took less than a second.

Similar results are shown in the transformation project implementing the ReDSeeDS methodologies. The total number of class elements in the model transformations for ReDSeeDS is greater than in the previously described project- ~2700. But in the same time there are at most 180 class elements having the same type and it is much less than in the previous example. The same class renaming action has been executed also in ReDSeeDS transformations project. It took less than a second too.

Similar results are observed also for other user actions which rely mainly on the model transformations written in MOLA. It proves that the MOLA Tool (in fact, model transformations used in METAclipse framework) scales well also for larger projects and is usable practically. It should be mentioned, that the model saving problem described in section 5.6 is actual also in MOLA Tool (METAclipse framework). For larger transformation projects (like ReDSeeDS) the saving takes a significant amount of time (~20 seconds). Although it is inconvenient, it affects just the frequency a transformation developer uses the save button. However, this issue should be solved in the future.

A screenshot of MOLA Tool is shown in Fig. 37.

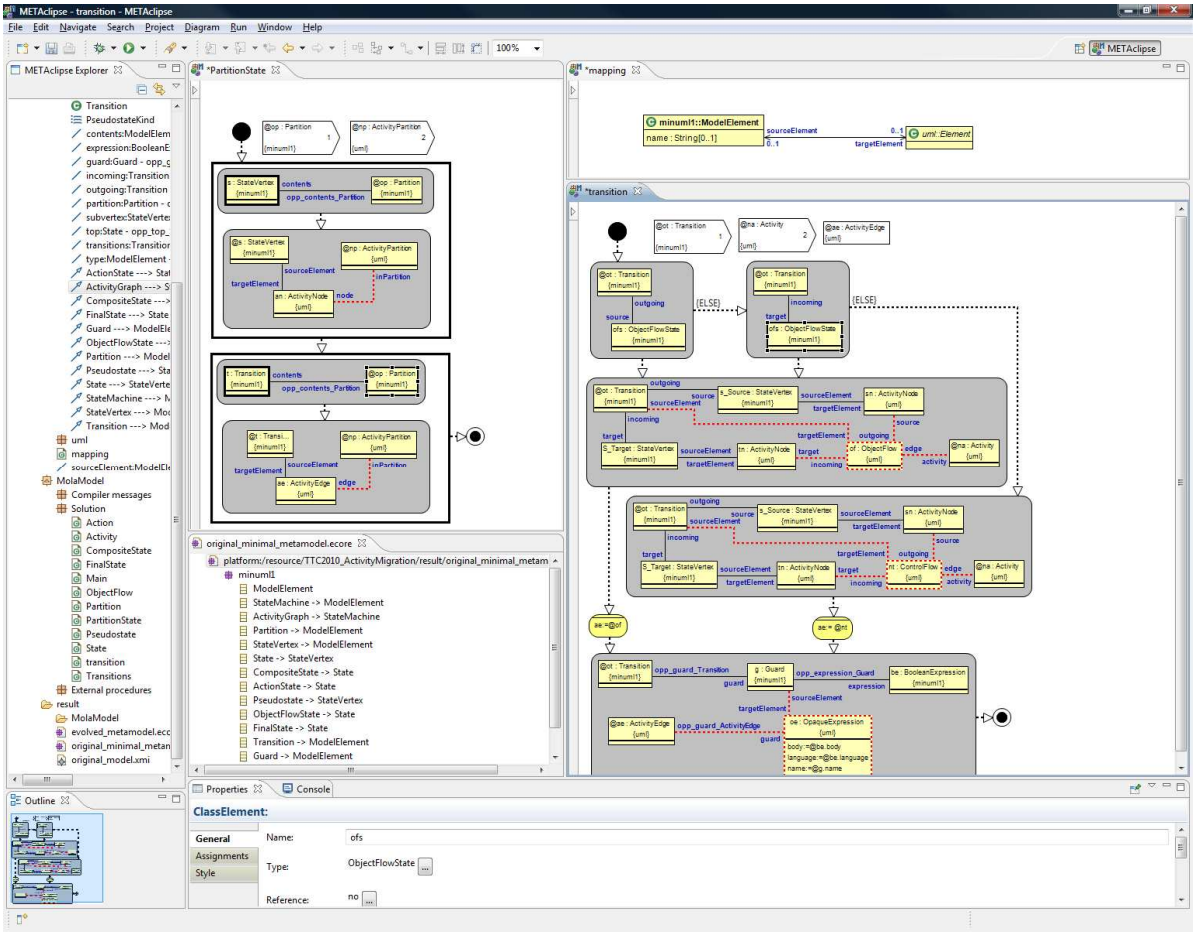


Fig. 37. MOLA2 Tool.

CHAPTER 7

Conclusions

The main goal of the research was to develop a simple and efficient implementation of pattern matching for model transformation languages. In order to achieve this goal, the following tasks are accomplished:

- A new pattern matching algorithm has been developed for model transformation languages. The algorithm uses relational database and SQL queries. The main advantage of the algorithm is the simple mapping from pattern to single SQL query. The implementation of this mapping is easy thus enabling fast development of an efficient model transformation language prototype. In this case the hardest part, the actual pattern matching, is done by query optimization features of a relational database management system.
- The developed algorithm has been practically implemented for model transformation language MOLA. An interpreter for MOLA has been built which works on most popular relational database management systems. The MOLA interpreter has been used for academic and research goals. How MOLA language is fit for MDSD has been tested using the interpreter.
- A new simple pattern matching algorithm which uses local search plan generation has been developed. It works on metamodel-based repositories which are commonly used to store models in popular modelling environments. The algorithm is efficient for MDSD transformations, which are typically dealing with models of tree-like structure where every element of source model should be processed and an element in a target model should be built.
- The developed algorithm has been practically implemented for MOLA language. A MOLA compiler has been built to lower-level model transformation language L3. MOLA transformations can process models stored in several metamodel-based repositories, including EMF, JGraLab and MIIREP. The compiler is part of MOLA Tool which has been successfully used in the EU 6th framework project ReDSeeDS for development of MDSD transformations.

- An efficient algorithm has been developed which is more universal (it is efficient not for MDSD-related tasks only). It is based on classical local search plan generation strategy and together with a new metamodel annotation mechanism allows building efficient model transformations without any complicated runtime model analysis. Comparing to other implementations it allows utilize knowledge of particular domain to build efficient transformations.

A review of pattern matching mechanisms for the most popular model transformation languages has been presented in this thesis. There are several pattern matching approaches, but the most popular is the local search planning. In fact, it is the most universal strategy - it gives efficient results for different types of patterns. However, implementations of more advanced approaches are rather complex, although simpler strategies (like in case of MOLA and Fujaba) frequently give similar results. Of course, that holds not for every use case, but mostly for the domain the transformation language is designed for. For example, MOLA is efficient for MDSD-related tasks, as the empirical analysis of typical MOLA patterns in the ReDSeeDS project has shown. Other languages are efficient in other domains, e.g. VIATRA in the simulation of complex systems or Fujaba in the program refactoring domain.

A great role for efficient pattern matching is played also by the constructs of the pattern used in the language. MOLA offers very natural means for describing MDSD-related tasks, the foreach loops combined with reference mechanism. At the same time even the simple pattern matching algorithm which has been implemented for MOLA works efficiently in these cases. Thus, for the compiler-like tasks, where every element of a structured model (like UML) should be processed, MOLA can be used with a high efficiency, but with very simple implementation of pattern matching. Of course, the certain *design patterns* briefly discussed in the thesis should be ensured in MOLA programs, but they are very natural and easy to use.

MOLA is used not only for MDSD-related tasks (though it is designed for that). Therefore more universal pattern matching strategy based on analysis of the pattern and underlying metamodel have been developed. It hasn't been fully implemented in the MOLA Tool yet. So the benchmark tests haven't been done for this algorithm. We have introduced the metamodel annotation mechanism, which captures the domain knowledge of actual cardinalities in the metamodel. It permits to make pattern matching more

efficient, that otherwise could be achieved only by runtime analysis of models which may itself be costly at runtime or not available at design time.

The future work is to identify model transformation domains - the areas where typical patterns are used. The most appropriate pattern matching approaches should be addressed for each domain. Since most of the model transformation language developers provide information on pattern matching implementation for their languages; that would make the choice of the most appropriate model transformation language easier for a concrete task. Of course, the pattern matching implementation is not the only condition helping to make the decision. However, usually, if the language constructs are fit for the task, then it is a great chance that pattern matching will be also appropriate. We believe that practically the appropriate pattern matching algorithms can be developed for specific tasks (domains) despite pattern matching being an NP-complete problem in general.

A domain specific annotation language may be developed to use other knowledge of domain than cardinalities. In fact, it means extending metamodelling languages with special features which capture information crucial for pattern matching.

Currently there is an ongoing work on implementation of algorithm described in the section 5.7. The implementation of the algorithm will allow using MOLA efficiently also for other kinds of tasks not just for MDSD. It is an important aspect also in the context of integration of MOLA transformations into the Eclipse ecosystem. Eclipse EMF has become a *de facto* standard of model repository in the modelling community. A significant part of models are stored in EMF. There are also lots of metamodels written in EMF Ecore metamodelling language. One of the problems the EMF-based model transformation implementations are dealing with is the association navigability - associations in EMF (references) are navigable in one direction only. The simple pattern matching algorithm described in section 5.5 requires that associations are navigable in both directions. Therefore the model pre-processing step is performed before transformation runtime. In the pre-processing step a model is transformed to an intermediate model containing *missing* references. The algorithm described in the section 5.7 can solve the navigability problem without any additional model pre-processing steps in the same time maintaining sufficient efficiency. Solving this problem would allow direct integration of model transformations in a wide range of Eclipse (EMF) -based modelling tools.

There are no doubts that an efficient implementation of model transformation language offers many new possible directions of research. Model transformation languages are used in the software development (MDSD) or tool building (METAclipse framework). These research fields offer still unanswered questions. Models, metamodels and model transformations can be used in many other areas of research. For example, model transformations may be used for complex data processing in frameworks for classical information systems. The task is to find appropriate use cases where the usage of model transformations (and model transformation languages) fits at most.

The great potential of models in the field of software development is not realized yet; however a significant leap is expected in the near future.

REFERENCES

1. Kleppe, A.G, Warmer, J.B., and Bast, W. ***MDA explained: The model driven architecture: Practice and Promise***, Boston: Addison-Wesley, 2003
2. Object Management Group. ***MDA Guide Version 1.0.1.***, 2001, (on-line, 04.06.2010) http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf
3. Object Management Group. ***Meta Object Facility Core Specification, version 2.0***, 2006. (on-line, 04.06.2010) http://www.omg.org/spec/MOF/2.0/PDF/
4. Object Management Group. ***Object Constraint Languege (OCL) Version 2.2.*** (on-line, 20.06.2010) http://www.omg.org/spec/OCL/2.2/PDF
5. Object Management Group. ***Unified Modeling Language (UML) 2.3: Infrastructure and Superstructure***, 2010 (on-line, 04.06.2010) http://www.omg.org/spec/UML/2.3/
6. Object Management Group. ***Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). Version 1.1 - Beta 2***, 2009. (on-line, 04.06.2010) http://www.omg.org/spec/QVT/1.1/Beta2/
7. Object Management Group. ***Metamodel and UML Profile for Java and EJB Specification Version 1.0.*** (on-line, 20.06.2010) http://www.omg.org/cgi-bin/doc?formal/2004-02-02
8. Kalnins A., Vilitis O., Celms E., Kalnina E., Sostaks A., Barzdins J. ***Building Tools by Model Transformations in Eclipse.*** Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.), Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland, 2007. pp. 194. - 207.
9. Rath, I., Varro, D. ***Challenges for advanced domain-specific modelling frameworks.*** Proc. of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006, France
10. Ermel, C., Ehrig, K., Taentzer, G., Weiss, E. ***Object Oriented and Rule-based Design of Visual Languages using Tiger.*** Proceedings of GraBaTs'06, 2006, pp. 12
11. Object Management Group. ***Request for Proposal: MOF 2.0 Query / Views / Transformations,*** 2002, OMG document ad/2002-04-10.
12. IKV++ Technologies AG. ***medini QVT Project.*** (on-line, 04.06.2010) http://projects.ikv.de/qvt/
13. SmartQVT: ***SmartQVT - A QVT implementation***, (on-line, 07.06.2010) http://sourceforge.net/projects/smartqvt/
14. Jouault, F., Kurtev, I.: ***Transforming Models with ATL.*** In Bruel, J.M., ed.: Proceedings of MoDELS. Volume 3844 of LNCS., Springer (2006) 128–138*Rīks: ATL Project.* (on-line, 04.06.2010) http://www.eclipse.org/m2m/atl/
15. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D. ***VIATRA - visual automated transformations for formal verification and validation of UML models.*** In: Proceedings of 17th IEEE International Conference on Automated Software Engineering, IEEE Comput. Soc (2002) 267–270 Rīks: The Eclipse Foundation. VIATRA2 Home page. (on-line, 04.06.2010) http://www.eclipse.org/gmt/VIATRA2/
16. 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. Volume 4800 of LNCS., Springer (2008) 130–145. *Rīks: IMCS, The Lx transformation language set home page*. (on-line, 05.06.2010), http://lx.mii.lu.lv/

17. 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.J., Rozenberg, G., eds.: Proceedings of TAGT. Volume 1764 of LNCS., Springer (1998) 296–309 *Rīks: University of Paderborn. Fujaba Tool Suite*. (on-line, 04.06.2010) http://www.fujaba.de
18. Agrawal A., Karsai G, Shi F. ***Graph Transformations on Domain-Specific Models.*** Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003. *Rīks: Vanderbilt University. GReAT.* (on-line, 04.06.2010) http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT
19. Kalnins, A., Barzdins, J., Celms, E.: ***Model Transformation Language MOLA.*** In Aßmann, U., Aksit, M., Rensink, A., eds.: Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers. Volume 3599 of LNCS., Springer (2004) 62–76. *Rīks: IMCS. MOLA pages.* (on-line, 04.06.2010) http://mola.mii.lu.lv
20. Taentzer, G. ***AGG: A Tool Environment for Algebraic Graph Transformation.*** In Nagl, M., Schürr, A., Münch, M., eds.: Proceedings of AGTIVE. Volume 1779 of LNCS., Springer (1999) 481–488 *Rīks: TU Berlin. The <AGG> Homepage.* (on-line, 04.06.2010) http://user.cs.tu-berlin.de/~gragra/agg/
21. Schürr, A., Winter, A.J., Zündorf, A. ***The PROGRES approach: language and environment.*** Volume 2. World Scientific Publishing Co. (1999) 487–550
22. Cook, S. A. ***The complexity of theorem-proving procedures***. Proc. 3rd Ann. ACM Symp. on Theory of Computing 151–-158 (1971).
23. The European IST 6th framework project ***ReDSeeDS – Requirements-Driven Software Development System.*** (on-line, 04.06.2010) http://www.redseeds.eu
24. Rothenberg, J. ***The Nature of Modeling.*** In L. Widman et al., eds., Artificial Intelligence,Simulation, and Modeling, Wiley, New York, 1989.
25. Jouault, F., Bezivin, 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, Bologna, Italy. Volume 4037 of Lecture Notes in Computer Science., Springer (2006) 171–185
26. The Eclipse Foundation. ***Eclipse Modeling Framework (EMF)***. (on-line, 04.06.2010) http://www.eclipse.org/emf/
27. Parreiras, F.S.; Staab, S.; Winter, A. ***On Marrying Ontological and Metamodeling Technical Spaces.*** In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7. ACM Press.
28. Object Management Group. ***Business Process Modeling Notation.*** (on-line, 04.06.2010) http://www.bpmn.org/
29. AUTOSAR Consortium. ***The AUTOSAR Standard.*** (on-line, 04.06.2010) http://www.autosar.org/.
30. Kelly, S., Tolvanen,J-P. ***Domain-Specific Modeling.*** Wiley, 2008.
31. Willink E.D. ***A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX.*** Workshop on Metamodelling for MDA, University of York, England, 24-25 November, 2003. *Rīks: The Eclipse Foundation. UMLX Subproject.* (on-line, 05.06.2010), http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/UMLX/index.html

32. Lawley, M.J., Steel, J. ***Practical Declarative Model Transformation With Tefkat*** In Satellite Events at the MoDELS 2005 Conference, LNCS Vol. 3844. Jamaica, October 2-7, 2005. *Rīks: DSTC. Tefkat: The EMF Transformation Engine.*(on-line, 05.06.2010) http://tefkat.sourceforge.net/
33. IBM. ***Model Transformation Framework (MTF).*** (on-line, 05.06.2010), http://www.alphaworks.ibm.com/tech/mtf
34. De Lara, J., Vangheluwe, H. ***AToM3: A tool for multi-formalism and metamodelling.*** In R.-D. Kutsche and H. Weber (eds.), 5th Intern. Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April8-12, 2002, Proceedings, vol. 2306 of LNCS, pp. 174-188. Springer, 2002. *Rīks: McGill University, Modelling, Simulation and Design Lab. ATOM3 A Tool for Multi-formalism Meta-Modelling.* (on-line, 05.06.2010) http://atom3.cs.mcgill.ca
35. Levendovszky T., Lengyel L., Mezei G., Charaf H. ***A Systematic Approach toMetamodeling Environments and Model Transformation Systems in VMTS***, 2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT 2004, Rome, Italy, 2004. *Rīks: Budapest University of Technology and Economics, Department of Automation and Applied Informatics. Visual Modeling and Transformation System (VMTS)* (on-line, 05.06.2010) http://www.aut.bme.hu/Portal/Vmts.aspx
36. Marschall, F., Braun,P. ***Model Transformations for the MDA with BOTL***, Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, Enschede, The Netherlands (2003), pp. 25–36. *Rīks: Institut fur Informatik der Technischen Universitat Munchen.The Bidirectional Object Oriented Transformation Language (BOTL).* (on-line, 05.06.2010) http://botl.sourceforge.net/
37. Cuadrado, J.S., Molina, J.G., Tortosa, M.M. ***RubyTL: A Practical, Extensible Transformation Language***. Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. LNCS 4066, Springer 2006. *Rīks: Universidad de Murcia. Agile Generative Environment (AGE).* (on-line, 05.06.2010), http://gts.inf.um.es/trac/age
38. Kolovos, D.S., Paige, R.F., Polack, F.A.C. ***The epsilon transformation language.*** In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp.46–60. Springer, Heidelberg (2008) *Rīks: The Eclipse Foundation. Epsilon.* (on-line, 04.06.2010) http://www.eclipse.org/gmt/epsilon/
39. The Eclipse Foundation, ***Henshin***. (on-line, 05.06.2010) http://www.eclipse.org/modeling/emft/henshin/
40. Schurr, A. ***Specification of graph translators with triple graph grammars***. In Tinhofer, editor, Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science, number 903 in LNCS, pages 151–163. Springer-Verlag, 1994.
41. Geiß, R., Batz, G.V., 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.: Proceedings of ICGT. Volume 4178 of LNCS., Springer (2006) 383–397 *Rīks: IDP Goos. GrGen.NET.* (on-line, 04.06.2010) http://www.info.uni-karlsruhe.de/software/grgen/
42. The Eclipse Foundation. ***Model To Model (M2M)***, (on-line, 05.06.2010) http://www.eclipse.org/m2m/
43. Kalnins, A., Barzdins, J., Celms, E. ***Basics of Model Transformation Language MOLA.*** ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA) , Oslo, Norway, June 14-18, 2004

44. Kalnins, A., Barzdins, J., Celms, E. ***MOLA Language: Methodology Sketch.*** Proceedings of EWMDA-2, Canterbury, England, September 7-8, 2004, pp.194-203.
45. Institute of Mathematics and Computer Science, University of Latvia ***MOLA pages***, (on-line, 05.06.2010) http://mola.mii.lu.lv
46. Celms, E., Kalnins, A., Lace, L. ***Diagram definition facilities based on metamodel mappings.*** Proceedings of the 3rd OOPSLA (Workshop on Domain-Specific Modeling) , University of Jyvaskyla, 2003, pp.23-32.
47. Kalnins, A., Kalnina, D., Kalis, A. ***Comparison of Tools and Languages for Business Process Reengeneering***. Proceedings of the Third International Baltic Workshop on Databases and Information Systems, Riga, 1998, pp. 24-38
48. Zündorf, A. ***Graph Pattern Matching in PROGRES.*** In Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G., eds.: Proceedings of ICGT. Volume 1073 of LNCS., Springer (1994) 454–468
49. Varro, G., Friedl, K., Varro, D. ***Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans.*** Electronic Notes in Theoretical Computer Science 152 (2006) 191–205
50. Batz, G.V., Kroll, M., Geiß, R. ***A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching.*** In Schürr, A., Nagl, M., Zündorf, A., eds.: Proceedings of AGTIVE. Volume 5088 of LNCS., Springer (2008) 471–486
51. Fischer, T., Niere, J., Torunski, L. ***Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML.*** Master thesis, University of Padeborn (1998)
52. Geiß, R., Kroll, M. ***On Improvements of the Varro Benchmark for Graph Transformation Tools***, Technical Report, 2007
53. Sostaks A. ***Pattern Matching in MOLA***. Proceedings of the 9th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2010), Riga, Latvia, July 5-7, 2010, University of Latvia Press, Riga, Latvia, 2010, pp. 309-324.
54. Varró, G., Friedl, K., Varró, D. ***Implementing a Graph Transformation Engine in Relational Databases.*** Software & Systems Modeling 5(3) (2006) 313–341
55. Kalnins A., Celms E., Sostaks A. ***Simple and Efficient Implementation of Pattern Matching in MOLA Tool***. Proceedings of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, Lithuania, July 3-6, 2006, pp. 159-167.
56. Simonis, H. ***Sudoku as a constraint problem.*** In CP Workshop on Modeling and Reformulating, Constraint Satisfaction Problems, 2005, pages 13
57. Kumar, V. ***Algorithms for Constraint Satisfaction Problems: A Survey***, AI Magazine 13(1): 32-44,1992
58. Rudolf, M.: ***Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching***. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proceedings of TAGT. Volume 1764 of LNCS., Springer (1998) 238–251
59. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G. ***Incremental pattern matching in the viatra model transformation system.*** In: Proceedings of GraMoT, ACM (2008) 25–32
60. C. L. Forgy. ***Rete: A fast algorithm for the many pattern/many object pattern match problem.*** Artificial Intelligence, 19(1):17–37, September 1982.
61. Bergmann, G., Horváth, A., Ráth, I., Varró, D. ***Efficient Model Transformations by Combining Pattern Matching Strategies***. In Paige, R.F., ed.: Proceedings of ICMT. Volume 5563 of LNCS., Springer (2009) 20–34
62. Kalnins, A., Celms, E., Sostaks, A. ***MOLA Tool.*** ECMDA Tools Session, 2005.

63. The Eclipse Foundation. ***Eclipse Graphical Modeling Framework (GMF)***. (on-line, 20.06.2010) http://www.eclipse.org/gmf/
64. IBM. ***Rational Software Architect (RSA).*** (on-line, 05.06.2010) http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html
65. 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, Vilnius, Technika (2006) 203–218
66. Kahle, S. ***JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen.*** Master thesis, University of Koblenz-Landau, Institute for Software Technology (2006)
67. Kalnins, A., Barzdins, J., Celms, E. ***Efficiency Problems in MOLA Implementation.*** 19th International Conference, OOPSLA'2004 (Workshop "Best Practices for Model-Driven Software Development") , Vancouver, Canada, October 2004, p. 14.
68. Oracle Corporation. ***MySQL – The world's most popular open source database***. (on-line, 05.06.2010) http://www.mysql.com/
69. Date, C. J. ***An Introduction to Database Systems*****, Chapter 17, Optimisation**, Addison-Wesley, 7th Edition, 2000
70. W3C. **Web Ontology Language (OWL)**. (on-line, 20.06.2010) http://www.w3.org/2004/OWL/
71. Oracle Corporation. ***MySQL Reference Manual***. (on-line, 05.06.2010) http://dev.mysql.com/doc/mysql/en/index.html
72. Katchaounov, T. ***An Overview of the MySQL 5.0 Query Optimizer***. The MySQL Users Conference, 2005.
73. Dubois, P. ***MySQL, Chapter 4, Query Optimization.*** Sams, 3rd Edition, 2005.
74. Microsoft. ***Microsoft SQL Server 2000 Desktop Engine (MSDE 2000).*** (on-line, 05.06.2010) http://www.microsoft.com/downloads/details.aspx?familyid=413744d1-a0bc-479f-bafa-e4b278eb9147&displaylang=en
75. PostgreSQL Global Development Group. ***PostgreSQL - Open Source Database Server.*** (on-line, 05.06.2010) http://www.postgresql.org/
76. Microsoft. ***Microsoft SQL Server 2005 Express Edition*** (on-line, 05.06.2010) http://www.microsoft.com/sqlserver/2005/en/us/express.aspx
77. Elmasri, R., Navathe, R. ***Fundamentals of Database Systems, Chapter 18, Query Processing and Optimisation,*** Addison-Wesley, 3rd Edition, 2000.
78. Microsoft. ***The SQL Server 2000 Workload Governor.*** (on-line, 20.06.2010) http://msdn.microsoft.com/library/default.asp?url=/library/enus/architec/8_ar_sa2_0ciq.asp
79. Varro, G., Schurr, A., Varro, D. ***Benchmarking for Graph Transformation***, Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing 2005 (VL/HCC 05), Dallas, Texas, USA, September 2005, IEEE Press, pp 79-88.
80. Kalnins A., Celms E., Sostaks A. ***Model Transformation Approach Based on MOLA.*** ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML '2005). (MoDELS/UML'05 Workshop: Model Transformations in Practice (MTIP)) , Montego Bay, Jamaica, October 2 -7, 2005
81. Kalnins, A., Vitolins, V. ***Use of UML and Model Transformations for Workflow Process Definitions.*** Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006). , Vilnius, Lithuania, July 3-6, 2006, pp. 3-14.

82. Sostaks A., Kalnins A. ***The Implementation of MOLA to L3 Compiler***. Articles of the University of Latvia, "Computer Science and Information Technologies", Riga, Latvia, 2008, pp. 140-178.
83. Kalnins, A., Sostaks, A., Kalnina, E., Celms, E., Vilitis, O.: ***MOLA 2 Tool.*** ECMDA Tools and Services Session, 2008.
84. Efron, B., Tibshirani, R.J. ***An Introduction to the Bootstrap***, Chapman & Hall/CRC, 1994, 436 p.
85. Aho, A., Sethi, R., Ullman, J. ***Compilers: Principles, Techniques, and Tools.*** Bell Laboratories, 1986
86. Rencis, E. ***Model Transformation Languages L1, L2, L3 and their Implementation.*** Scientific Papers, University of Latvia, Computer Science and Information Technologies, 2008, pp. 103-139.
87. Kalnins A., Celms E., Sostaks A. ***Tool support for MOLA.*** Proceedings of International Workshop on Graph and Model Transformation (GraMoT), Tallin, Estonia, September 2005. p.12
88. The Eclipse Foundation, ***Eclipse.org,*** (on-line, 05.06.2010) http://www.eclipse.org/
89. Jouault, F., Bezivin, J., Consel, C., Kurtev, I., Latry, F. ***Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages***. In: Proceedings of the 1st ECOOPWorkshop on Domain-Specific Program Development (DSPD), July 3rd, Nantes, France. (2006)
90. The Eclipse Foundation. ***ATL Use Case - Compiling a new formal verification language to LOTOS (ISO 8807)*** http://www.eclipse.org/m2m/atl/usecases/FIACRE2LOTOS/
91. Jouault, F., Allilaire, F. ***An introduction to the ATL Virtual MachineV1.0 draft*** (on-line, 20.06.2010) http://www.eclipse.org/m2m/atl/doc/ATL_VM_Presentation_%5B1.0%5D.pdf
92. W3C. ***Extensible Markup Language (XML) 1.1 (Second Edition).*** (on-line, 20.06.2010) http://www.w3.org/TR/xml11/
93. Slonneger, K., B. Kurtz. ***Formal Syntax and Semantics of Programming Languages. A Laboratory Based Approach***, Addison-Wesley Publishing Company, 1995.
94. Rikacovs, S. ***The Base Transformation Language L0+ and Its Implementation.*** Papers, University of Latvia, Computer Science and Information Technologies, 2008, pp. 75-102
95. Dijkstra, E. W. ***GOTO Statement Considered Harmful***, Letter of the Editor, Communications of the ACM, March 1968, pp. 147-148.
96. Edmonds, J. ***Optimum Branchings.*** Journal of Research of the National Bureau of Standards 71B (1967), 233–240.
97. Smialek, M., Bojarski, J., Nowakowski, W., Ambroziewicz, A., Straszak, T. ***Complementary use case scenario representations based on domain vocabularies.*** In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: Proceedings of MoDELS. Volume 4735 of LNCS., Berlin, Heidelberg, Springer (2007) 544–558
98. Sparx Systems. ***UML tools for software development and modelling – Enterprise Architect UML modeling tool***, (on-line, 05.06.2010) http://www.sparxsystems.com/
99. Smialek, M., Kalnins, A., Kalnina, E., Ambroziewicz, A., Straszak, T., Wolter, K. ***Comprehensive System for Systematic Case-Driven Software Reuse.*** In: J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorny, B. Rumpe: In Proceedings of SOFSEM 2010: Theory and Practice of Computer Science, Vol 5901, LNCS, Springer, Berlin/Heidelberg, 2010, pp. 697-708.

100. ***SpringSource.org***, (on-line, 05.06.2010) http://www.springsource.org/
101. JBoss Comunity. ***Hibernate***. (on-line, 05.06.2010)http://www.hibernate.org
102. The Eclipse Foundation. ***Eclipse Graphical Editing Framework (GEF)***. (on-line, 20.06.2010) http://www.eclipse.org/gef/