

Editor Definition Language and its Implementation

Audris Kalnins, Karlis Podnieks, Andris Zarins, Edgars Celms, Janis Barzdins

Institute of Mathematics and Computer Science,
University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{audris, podnieks, azarins, edgarsc, jbarzdin}@mii.lu.lv

Abstract. Universal graphical editor definition language based on logical metamodel extended by presentation classes is proposed. Implementation principles of this language, based on Graphical Diagramming Engine are described.

1 Introduction

Universal programming languages currently have become more or less stable, the main effort in this area is oriented towards improving programming environments and programming methodology. However, the development of specialised programming languages for specific areas is still going on (most frequently, this type of languages is no more called programming languages, but specification or definition languages). One of such specific areas is the definition of graphical editors. The need for various graphical editors and similar tools based on graphical representation of data increases all the time, because due to increased computer speeds and size of monitors it is possible to build graphical representations for wider subject areas. In this paper the **Editor Definition Language (EdDL)** for a simple and convenient definition of wide spectrum of graphical editors is proposed, and the basic implementation principles of EdDL are described.

Let us mention some earlier research in this area. Perhaps, the first similar approach has been by Metaedit [1], but its editor definition facilities are fairly limited. The most flexible definition facilities (and some time ago, also the most popular in practice) seem to be the Toolbuilder by Lincoln Software. Being a typical meta-CASE of early nineties, the approach is based on ER model for describing the repository contents and special extensions to ER notation for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. The diagram itself is being defined by means of a frame describing graphical objects in it and the supported operations. A more academic approach is that proposed by Kogge [2], with a very flexible, but very complicated and mainly procedural editor definition language. Another similar approaches are proposed by DOME [8] and Moses [9] projects, with fairly limited definition languages. Several commercial modelling tools (STP by Aonix, ARIS by IDS prof. Scheer etc) use a similar approach internally, for easy

customisation of their products, but their definition languages have never been made explicit.

Our approach in a sense is a further development of the above-mentioned approaches. We develop the customisation language into a relatively independent editor definition language (EdDL), which, on the other hand, is sufficiently rich and easy to use, and, on the other hand, is sufficiently easy to understand. At the same time it can be implemented efficiently, by means of the universal Editor Engine. Partly the described approach has been developed within the EU ESPRIT project ADDE [3], see [4] for a preliminary report.

2 Editor definition language. Basic ideas

The proposed editor definition language consists of two parts:

- the language for defining the logical structure of objects which are to be represented graphically
- the language for defining the concrete graphical representation of the selected logical structure.

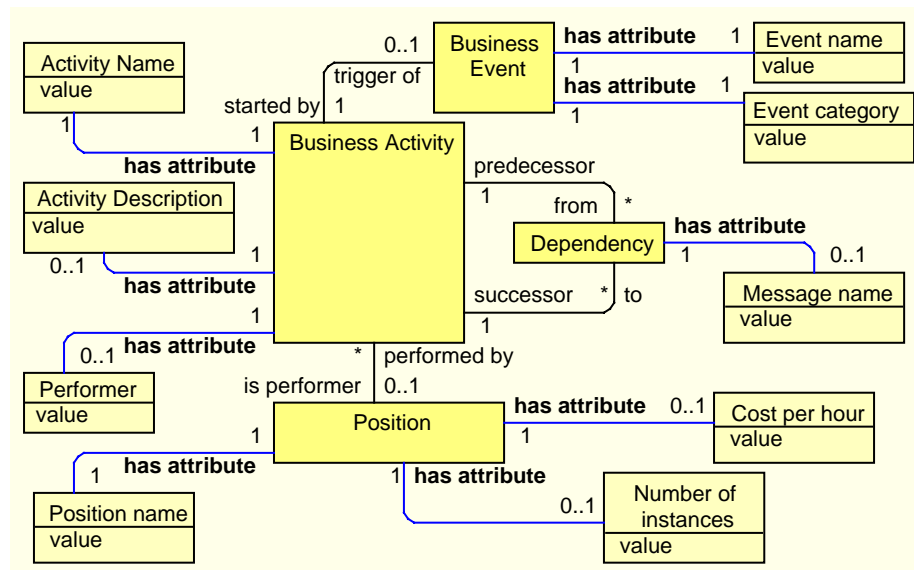


Fig. 1. Logical metamodel example

The separation of these two parts is the basis of our approach. For describing the logical structure there exists a generally adapted notation – by UML class diagrams [5], which typically is called the **logical metamodel** in this context. Fig.1 shows a

simple example of a logical metamodel for business activities. *Business activities* are assumed to be parts of a larger process. There may be a *dependency* between business activities (this dependency may be a *message* passed from one activity to another, but also something more general). An activity may be triggered by an (external) *business event*. Business activity may have a *performer* – a *position* within a company. This example will be used in the paper to demonstrate the editor definition features. Fig.1 needs one technical remark to be given. Attributes of a class there are extracted as separate classes, linked via an association with the predefined role name **has attribute** to the parent class. This attribute extraction is technically convenient for defining the presentation language part. Otherwise the logical metamodel is an arbitrary UML class diagram.

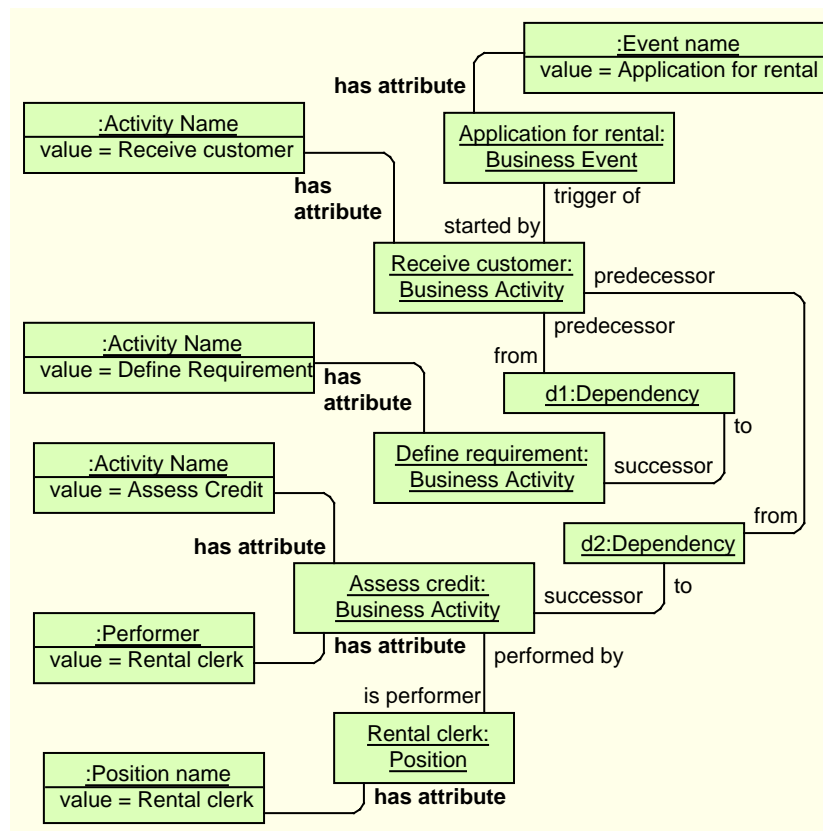


Fig. 2. Example of instance diagram

Fig.2 shows an example of an instance diagram (object diagram in UML terms) corresponding to the logical metamodel in Fig.1. Our goal is to define the corresponding editor, which in this case could be able to present the instance diagram as a highly readable graphical diagram in Fig.3 (where the traditional rendering of dependencies by oriented lines is used). A special remark should be given with

respect to *Position*. It is not explicitly represented in diagram in Fig. 3, but double-clicking on the performer name is assumed to **navigate to** (i.e. to open) a special editor showing the relevant position (this other editor is not specified here). The **navigation** and the **prompting** (the related action by means of which such a reference can be easily defined) are integral parts of our editor definition facilities.

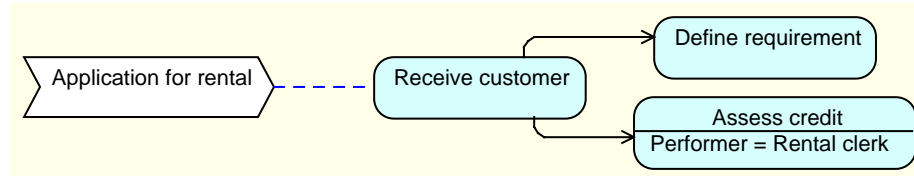


Fig. 3. Business activity diagram example

Roughly speaking, the goal of our definition language is to describe the translation of pictures like Fig. 2 into equivalent pictures like Fig. 3. So it is a sort of graphics translation language.

Now let us start a detailed outline of the EdDL. Like any real language, it contains a lot of details, therefore we will concentrate only on the basic constructs. The language will be presented as an extension of the logical metamodel adhering to the UML class diagram notation. Fig.4 demonstrates the use of EdDL for the definition of the example editor (with some minor details omitted). In this figure rectangles represent the same classes from the logical metamodel in Fig. 1, but rounded rectangles represent classes being the proper elements of EdDL. Classes with class names in bold represent abstract classes, which cannot be modified (they are used mainly for inheritance). Similarly, bold role names of associations represent the fixed ones. We remind that the underlined attributes (to be seen in EdDL classes) are class attributes (the same for all instances) according to UML notation.

The first element added to the logical metamodel is the **diagram** class (*Business activity diagram*), together with standard associations (with the role name **contains**) to the contained diagram objects, and as a subclass of the fixed *Diagram* class. One more standard association for diagram is the refinement association (**refines**), which defines that a *Business Activity* can be further refined by its own *Business activity diagram* (this definition is sufficient for the Editor Engine to enable this service).

Each of the metamodel classes, which must appear as graphical objects in the diagram, are linked by an unnamed association to its **presentation class** – a subclass of standard classes **box** or **line**. The presentation class may contain several class attributes (with predefined names). Thus the presentation class for *Business Activity* – the *Activity box* class says that every business activity must be represented by a rounded rectangle in a light blue default colour. The *Icon* representing this graphical symbol on the editor's **symbol palette** (to create a new business activity in a diagram) is also shown. For presentation classes being lines the situation is similar, but there may be lines corresponding to associations in the metamodel (*Triggering line*) or to classes (*Dependency line*). The latter case is mostly used for lines having associated

texts in the diagram (corresponding to attributes of the class; here the *Message name*). For showing the direction of line (and other similar features) the relevant role names from the metamodel are referenced in the presentation class (e.g. *start=predecessor*).

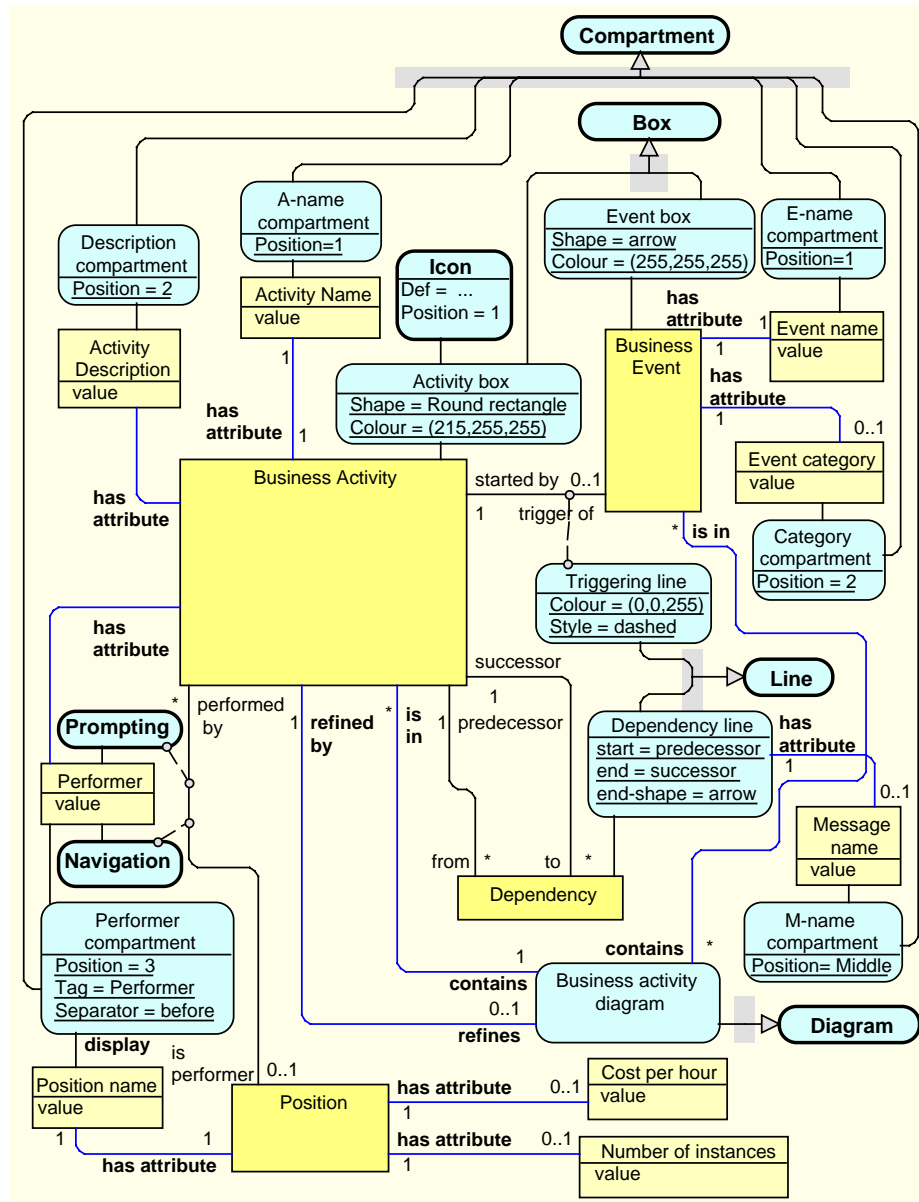


Fig. 4. Business activity diagram editor definition in EdDL

Class attributes are being made visible in diagrams by means of a **compartment** presentation class. The most important attribute of a compartment class is the **position** – for boxes in the simplest case it means the number of the horizontal slice of the box, for lines it means the relative positioning (start, middle, end). Compartment class may contain also style attributes (visible **tag**, separator, font etc).

The most interesting element in this EdDL example is the definition of **prompting** and **navigation**. They are both related to the situation when an attribute (i.e. its value) of a metamodel class (the *Performer* for *Business activity* in the example) actually is determined by an association of this class (a derived attribute in UML terms). Here the *Performer* value actually must be equal to the *Position name* of that *Position* instance (if any) which is linked by the association having the role name *Performed by* (the fact that a *Position* must be represented by its *Position name* is defined by the *display* association). Prompting here means the traditional service found in an editor that a value can be selected from the offered list of values (value of *Performer* selected from the list of available *Position names*), with the automatic generation of the relevant association instance as a side effect. The navigation means the editor feature that double-clicking on the *Performer* field (which presents the *Performer* attribute) in a *Business activity* box automatically invokes some default editor for the *Position* (the target of the association). Both *Prompting* and *Navigation* are shown in the EdDL as fixed classes linking the attribute to the relevant association (they may have also their own attributes specifying, e.g. the prompting constraints). Note that *Position* has no presentation class in the example, consequently its instances are not explicitly visible in the *Business activity diagram*.

Certainly EdDL contains more features than demonstrated in Fig.4, e.g. various uniqueness constraints, definitions for attribute "denormalisation", modes of model/diagram consolidation etc. The EdDL coding shown in Fig. 4 was simplified to make it more readable. The actual coding used for the commercial version of EdDL is much more compact, here the metamodel class attributes are defined in the traditional way, and most of Presentation classes are coded just as UML constraints (properties) inside a metamodel class. Nevertheless the semantics of this language is just the one briefly described in the paper. We assert that EdDL is expressive enough to define practically any types of editor that could be used to build related sets of diagrams in system modelling area. Namely the inter-diagram relations such as prompting and navigation are the most complicated ones, and they successfully managed in EdDL. Finally, Fig. 5 shows the defined editor in action.

3 EdDL implementation principles

EdDL has been implemented by means of an interpreter which in this case is named **Editor Engine**. When an editor has been defined in EdDL the Editor Engine acts as the desired graphical editor for the end user. Here only the main principles of implementation will be discussed. The first issue is the information storage. It is universally accepted nowadays that the logical metamodel describes directly (or very close to it) the physical structure of the tool repository. This repository can be an

OODB, a special tool repository (e.g. Enabler [6]) or a relational DB. This is one more argument why the editor definition should be based on a separately defined metamodel.

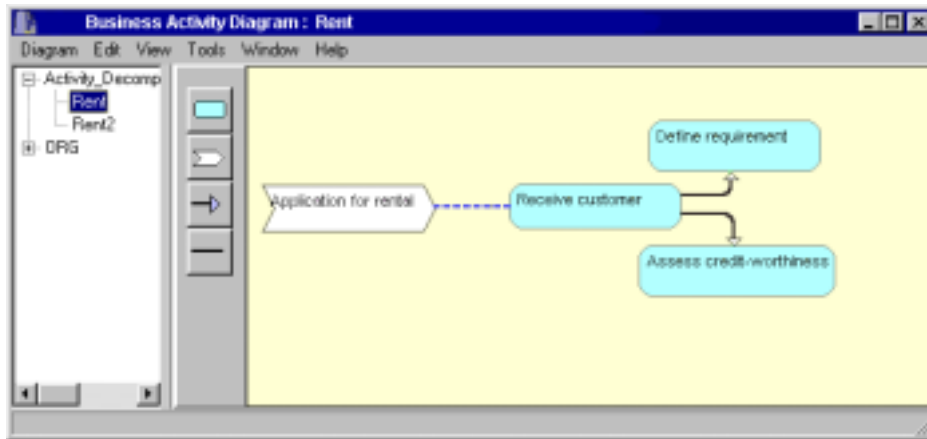


Fig. 5. Editor in action

Fig. 6 shows the general architecture of the EdDL approach. A key aspect is that Editor Engine (EE) relies on **Graphical Diagramming Engine (GDE)** for all diagram drawing related activities. The primitives implemented by GDE - diagram, box, line, compartment etc. and the supported operations on them are very fit for this framework.

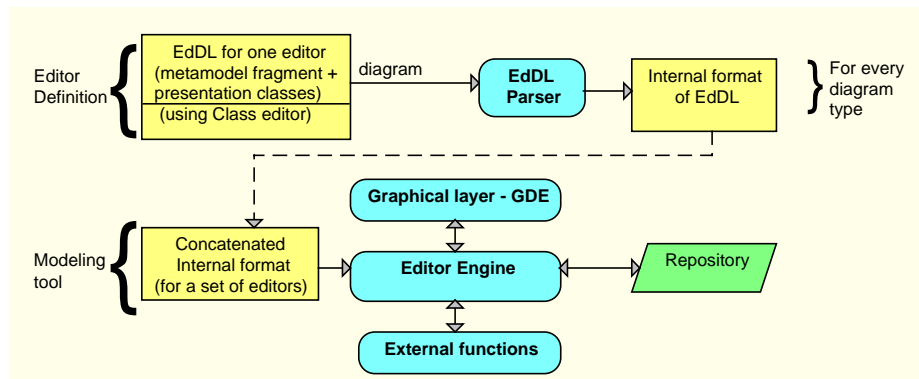


Fig. 6. Architecture of EdDL implementation

Thus the interface between EE and GDE is based on very appropriate high level building blocks, there is no need for low level graphical operations in EE at all. The GDE itself was developed by IMCS UL initially within the framework of ADDE

project, with a commercial version later on. It is based on very sophisticated graph drawing algorithms [7].

The general technology of using EDDL for defining a set of editors is quite simple. For each of the editors its definition on the basis of the relevant metamodel fragment is built. Then these definitions are parsed and assembled into one set. EE "performs" this set, behaving as a modelling tool containing the defined set of diagrams. A lot of tool functionality – "small operations" such as copy-paste and "large operations" such as load and save are implemented in EE in a standard way and need no special definitions. Certainly, external modules can be included for some specific operations.

The practical experiments on using EDDL and EE have confirmed the efficiency and flexibility of approach. The defined editors (like the one in Fig.5) behave as industrial quality graphical editors. The flexibility has been tested by implementing full UML 1.3 and various specific business process related extensions to it.

4 Conclusions

The paper presents a brief outline of the graphical editor definition language EDDL and its implementation. But we can view all this also from a different angle. Actually a new kind of metamodel concept application for a specific area - editor definition - has been proposed. However this approach can be significantly more universal, since it is generally accepted that object model (Class diagram) is a universal means for describing the logical structure of nearly any system. Thus the same approach of extending this model by special "presentation" classes could be used, e.g. to define model dynamics, simulation etc., but this is out of scope for this paper.

References

1. Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit – a flexible graphical environment for methodology modelling. Springer-Verlag (1991)
2. Ebert, J., Suttentbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain (1997)
3. ESPRIT project ADDE. <http://www.fast.de/ADDE>
4. Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: Towards a Metamodel-Based Universal Graphical Editor. Proceedings of the Third International Baltic Workshop DB&IS, Vol. 1. University of Latvia, Riga, (1998) 187-197
5. Rumbaugh, J., Booch, G., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)
6. Enabler Concepts *Release 3.0*, Softlab GmbH, Munich (1998)
7. Kikusts, P., Rucevskis, P.: Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors. Lecture Notes in Computer science, Vol. 1027. Springer-Verlag (1996) 361-364
8. DOME Users Guide. <http://www.htc.honeywell.com/dome/support.htm>
9. Esser, R.: The Moses Project.
<http://www.tik.ee.ethz.ch/~moses/MiniConference2000/pdf/Overview.PDF>

TitleThe First Step Towards Generic Modelling Tool

Audris Kalnins, Janis Barzdins, Edgars Celms, Lelde Lace, Martins Opmanis,
Karlis Podnieks, Andris Zarins

Institute of Mathematics and Computer Science, University of Latvia

Raina bulv. 29, LV-1459, Riga, Latvia

{audris, jbarzdin, Edgars.Celms, lelde, Martins.Opmanis, podnieks, azarins}@mii.lu.lv

Abstract: The foundation of a generic modelling tool is its flexible diagramming facility. The paper proposes a universal graphical editor definition language based on logical metamodel extended by presentation classes. Some more advanced diagram definition facilities such as patterns and diagram cores are also introduced. Implementation principles of this language, based on Graphical Diagramming Engine are briefly described.

Key words: modelling tool, graphical editor, metamodel, editor definition language

1. INTRODUCTION

Why it is not sufficient to use “hard-coded” modelling tools? Let us consider for example the situation in **business modelling**. On the one hand there exist several well-known business modelling languages (IDEF3, ARIS etc), each with a set of tools supporting it. But there are also Activity diagrams in UML, whose main role now is to serve business modelling. There is GRADE BM [1,2] – a specialized language for business modelling and simulation. Thus for the area of business modelling there is no one best or most used language or tool, each of them emphasizes its own aspects. For example GRADE BM presents very convenient facilities for specifying performers of a task and its triggering conditions. However any new language feature does not come for free, the language becomes more complicated for use. Therefore one universal business modelling language which would support all wishes would become extremely difficult for use in simple cases. This issue is even more urgent for domain-specific modelling, where countless special notations are used for separate domains.

UML for this situation offers one ingenious solution – stereotypes for adjusting the modelling language to a specific area. In many cases the idea works perfectly, it is well supported in several tools including GRADE. The latest version of UML - 1.4 extends the notion of stereotype, by assigning tagged values to it and grouping stereotypes into profiles (thus actually extending the metamodel). But currently no tool fully supports it.

In this paper an alternative approach to building flexible modelling environments is used – the **metamodel**-based approach where first the domain metamodel is built. Then the modelling method, notation and tool support is defined declaratively, by means of a special metamodeling environment. Since most of modelling notation in any domain now is diagram based, important part of the approach is the **Editor Definition Language (EdDL)** for a simple and convenient definition of wide spectrum of diagrammatic graphical editors. The paper presents the main ideas and elements of EdDL, as well as basic implementation principles of it. Another component of the approach is a facility for the definition of flexible model content browsing (model tree).

An earlier alternative name for the approach is **meta-CASE**. Let us mention the key research in this area. Perhaps, the first similar approach has been by Metaedit [3], but for a long time its editor definition facilities have been fairly limited. The latest version named Metaedit+ [4] now can support definition of most used diagram types, but via very restricted metamodeling features (non-graphical), the resulting diagrams corresponding to the simplest concepts of labelled directed graph. The most flexible definition facilities (and some time ago, also the most popular in practice, but now the tool is out of market) seem to be offered by the Toolbuilder by Lincoln Software. Being a typical meta-CASE of early nineties, the approach is based on ER model for describing the repository contents and on special extensions to ER notation for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. The diagram itself is being defined by means of a frame describing graphical objects in it and the supported operations. A more academic approach is that proposed by Kogge [5], with a very flexible, but very complicated and mainly procedural editor definition language. Other newer similar approaches are proposed by ISIS GME [6], DOME [7] and Moses [8] projects, with main emphasis for creating environments for domain-specific modelling in the engineering world. The richest diagram definition possibilities of them are in GME. Several commercial modelling tools (STP by Aonix, ARIS by IDS prof. Scheer etc) use a similar approach internally, for easy customisation of their products, but their tool definition languages have never been made explicit.

Our approach in a sense is a further development of the above-mentioned approaches. First, the domain metamodel can be built in a most natural way independently of the diagram definition elements, which are built later and

mapped to it (earlier approaches typically mix up these concepts). The editor definition language (EdDL) is, on the one hand, sufficiently rich for fairly complicated diagrammatic notations (not just simple directed graphs as in GME), and, on the other hand, is sufficiently easy to understand. It offers elements of “definition engineering”, thus making its use more convenient. At the same time it can be implemented efficiently, by means of the universal Editor Engine, resulting in target modelling environments, which support diagramming quality better than many “hard-coded” environments.

Partly the described approach has been developed within the EU ESPRIT project ADDE [9], see [10] for a preliminary report. An earlier version of this research was presented in [11]. The current paper refines these initial ideas and explains the two basic principles of the EdDL language – patterns and cores – which make it easy usable for definition of complicated diagram support required by real-life examples. Another innovative element is a general support for stereotypes.

2. BASIC PRINCIPLES OF EDITOR DEFINITION LANGUAGE

An editor definition for a new diagram type starts with the description of the logical structure of the domain objects to be represented graphically by this diagram. This logical structure is described by a UML class diagram [12], which typically is called the **logical metamodel** (or **domain metamodel**). Fig.1 shows a simple example of a logical metamodel for business activities. This example will be used in the paper to demonstrate the editor definition features.

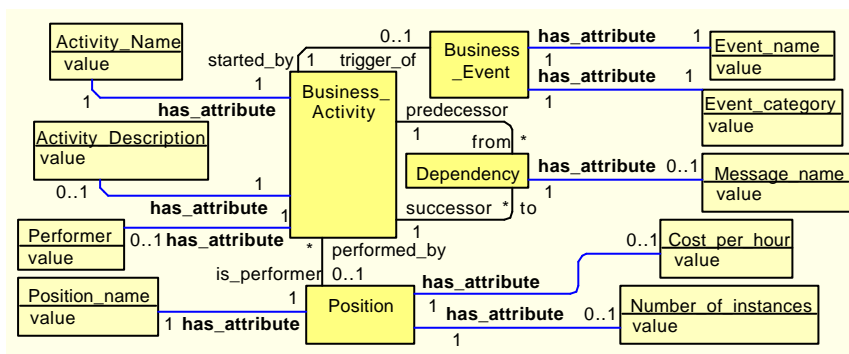


Figure 1. Logical metamodel example

Fig.2 shows an example of an instance diagram (object diagram in UML terms) corresponding to the logical metamodel in Fig.1. Our goal is to define

the corresponding editor, which in this case should be able to present the instance diagram as a highly readable graphical diagram in Fig.3.

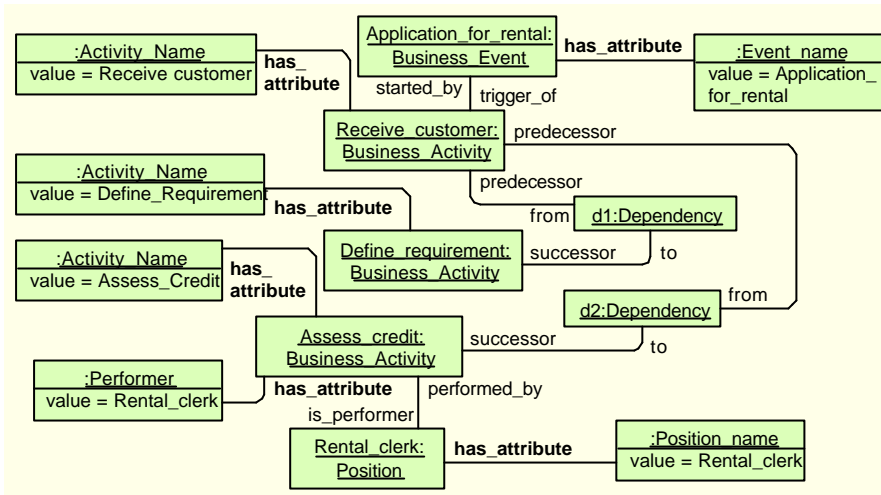


Figure 2. Example of instance diagram

A special remark should be given with respect to *Position*. It is not explicitly represented in diagram in Fig. 3, but double-clicking on the performer name is assumed to **navigate to** (i.e. to open) another editor showing the relevant position (this editor is not specified here). The **navigation** and the **prompting** (the related action by means of which such a reference can be easily defined) are integral parts of our editor definition facilities.

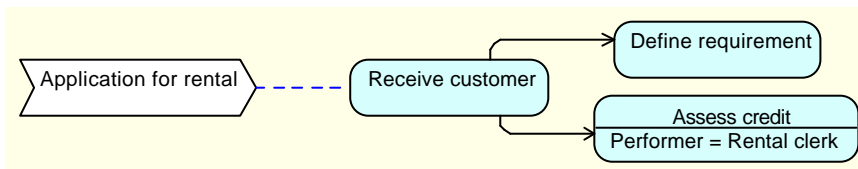


Figure 3. Business activity diagram example

A **diagram definition** in EdDL is an **extension** of the logical metamodel, strictly adhering to the UML class diagram notation. It should be emphasized that the logical metamodel itself is never modified during this process and remains a **separate** fragment, with only new associations attached. Fig.4 demonstrates the use of EdDL for the definition of the example editor. In this figure rectangles represent the classes from the

logical metamodel in Fig. 1, but rounded rectangles represent classes being the proper elements of EdDL. Classes with class names in bold italic represent abstract classes, which cannot be modified (they are used mainly for inheritance), objects with names in bold are the “technical constants” (with fixed class names). Similarly, bold role names of associations represent the fixed ones. We remind that the underlined attributes (to be seen in EdDL classes, with initial values set) are class attributes (the same for all instances) according to UML notation.

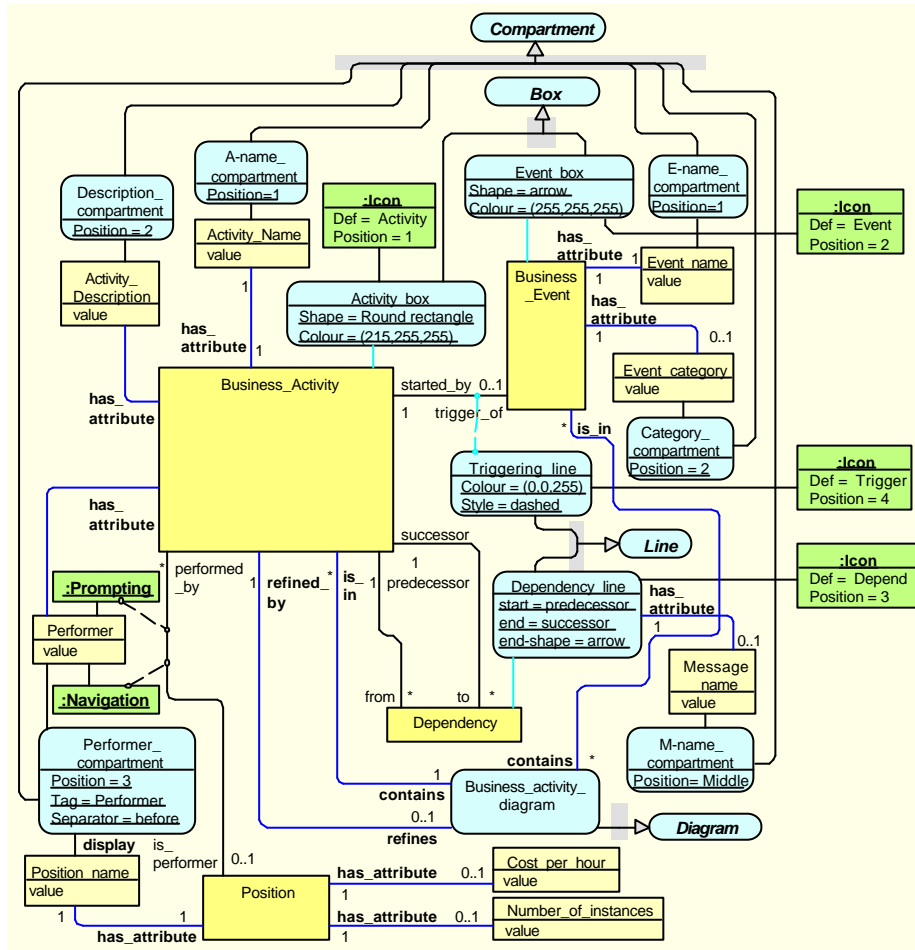


Figure 4. Business activity diagram editor definition in EdDL

Certainly EdDL contains more features than demonstrated in Fig.4, e.g. various uniqueness constraints, definitions for attribute "denormalisation", modes of model/ diagram consolidation etc.

3. A MORE PRACTICAL APPROACH TO DIAGRAM DEFINITION

The previous section demonstrated the simplest features of EdDL – diagram definition was performed by adding presentation classes, which inherited from some fixed abstract classes – box, line etc. This section shows that more knowledge about the diagram world can be incorporated in the editor definition by means of introducing **diagram patterns** – typical metamodel fragments. A certain combinations of diagram patterns frequently are used together, so the concept of a **diagram core** is introduced, from which specific diagram definitions simply can inherit.

3.1 Patterns in diagram definitions

A pattern in Object-oriented development is a typical construct or mechanism (see e.g.[14]), which aside from its purely programming content appears as a standardised fragment of a class diagram, frequently with predefined associations in it. Use of patterns for diagram definition has a similar goal. They are typical fragments of a diagram-oriented metamodel corresponding to typical modelling or diagramming constructs, frequently supported by certain functionality in modelling tools. Similar to patterns in OO programming, classes in a diagramming pattern have certain fixed roles and typically there are some fixed associations.

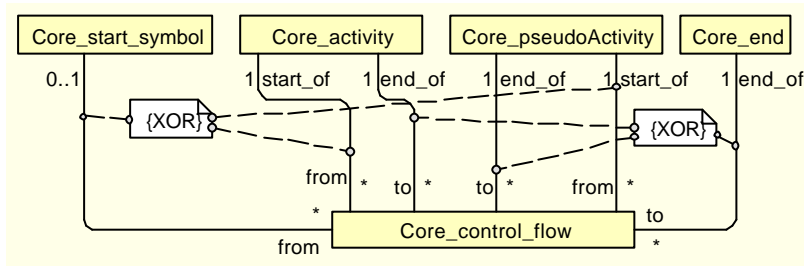


Figure 5. Process flow pattern

Let us consider an example – the **process flow pattern** (see Fig. 5). In any diagram type describing some process – be it UML Activity diagram, IDEF3 process chart, etc., there are process start symbols (one or more), there are regular process elements (e.g., activities in UML activity diagram) and process end symbols. All these symbols are some sorts of boxes. The process flow, represented by directed lines (of one or several types) correspond to *Core control flow* class in fig. 5.

The associations represent the normal relations between lines and boxes, with cardinalities and the explicit XORs giving the most general validity

constraints for a process diagram. For example, these constraints say that flow lines cannot enter a start symbol or exit an end symbol. Typically a process diagram editor functionality is also associated with such a pattern, e.g. process symbols may be automatically positioned so that flows go from top to down whenever possible, automatic flow construction from the current symbol to the symbol built next may be offered etc. Just for defining a specialised editor functionality the pattern contains the *Core pseudoactivity* class (a typical representative of which is fork, join or decision in activity diagram), which in addition to having a meaning different from the basic activity element graphically is represented by a “small symbol”.

Another such pattern is **refinement pattern**, which specifies that typically in process diagrams some of its elements may be refined by a diagram of the same type (Fig. 6). Typical editor functionality here is a support for making a new refinement or attaching existing process as a refinement. Simple navigation to refinement is also usually supported. These actions are not just diagram drawing, some semantic consequences for the model are also typically checked.

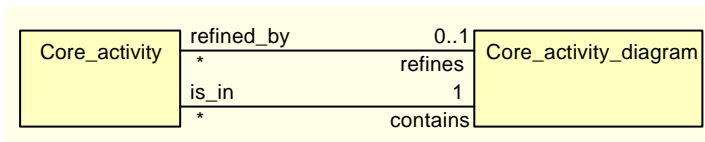


Figure 6. Refinement pattern

One more useful pattern for process diagrams is the **object flow pattern** (Fig. 7), saying that there may be another type of diagram elements (objects, e.g. objects in activity diagrams), which are linked by object flow lines to basic diagram elements – activities.

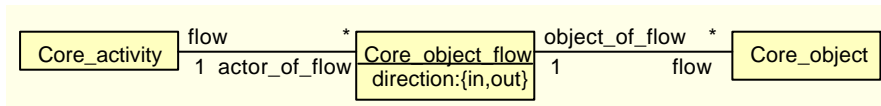


Figure 7. Object flow pattern

Patterns for diagram definition can be used in two ways. Firstly, just as patterns in OO programming, they may serve as manual templates for easy building of correct metamodels. But other more fundamental use is to combine them in diagram cores to be considered in the next subsection. In this case real diagram definitions can simply inherit from patterns, so correct “standard” associations between classes are obtained automatically.

For other types of diagrams patterns are available also. So for hierarchy diagrams (ORG-charts, data structures, function decompositions etc) there are **hierarchy** and **refinement** patterns. Fewer patterns can be defined actually for the most general diagram type – class (class diagram in UML, ER-model etc), here the **multi-occurrence** pattern was found (there may be several occurrences of a class even in the same diagram, and the editor must synchronize them).

3.2 Diagram cores

A lot of diagram notations represent something like a process or flowchart. Actually the example considered in this paper is also of that sort (though rather simple), certainly the most “important” diagram type of this sort is UML activity diagram. We will show how the discussed earlier process patterns typically combine for process-like diagrams. This way the **Activity core** will be obtained.

The core is a standard metamodel fragment for representing elements visible in the diagram. The actual diagram definition for a specific diagram type is obtained by inheriting from elements of the core. Namely, according to the graphical syntax of the diagram to be defined, presentation classes are introduced into the metamodel, and for each a matching core element is found. This way the important associations, which actually determine the diagram structure, need not to be rebuilt – they are inherited from core.

Fig. 9 shows the proposed Activity core and its use for defining the same Business activity diagram considered in section 2.

But at first some comments on the core itself. Since there are several cores possible – **Activity core**, **Hierarchical core**, **Class core** etc., which correspond to different diagram types being a specialised varieties of directed graph, it is reasonable to define a **Supercore**, corresponding to concepts of this graph itself (see Fig. 8). This core contains the basic diagramming concepts mentioned already in section 2. Now they can be reused in any of specific diagram cores by means of inheritance. We could include in this core also the fact that lines start from/end at boxes, but if simply included here it would allow any line start from any box type (to restrict this we should have to introduce a special type of constraints – this is the way things are done e.g. in Metaedit+ [4] – thus making the metamodel significantly less readable, but our approach retains the explicit constraints for smarter cases). So the Supercore expresses only containment.

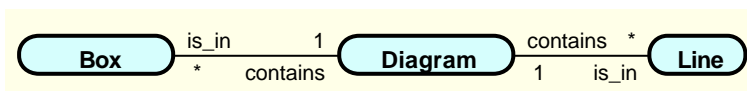


Figure 8. Supercore

The Supercore is visible in the metamodel in Fig. 9 as the top layer of classes. To make the metamodel in Fig. 9 more readable it is separated into horizontal layers of classes (each layer having a slightly different style). The next layer is the Activity core. It combines the elements of the three process patterns from the previous section, with *contains* association inherited from Supercore. By combining patterns we can specify e.g. that only core activities can be refined but not pseudoactivities.

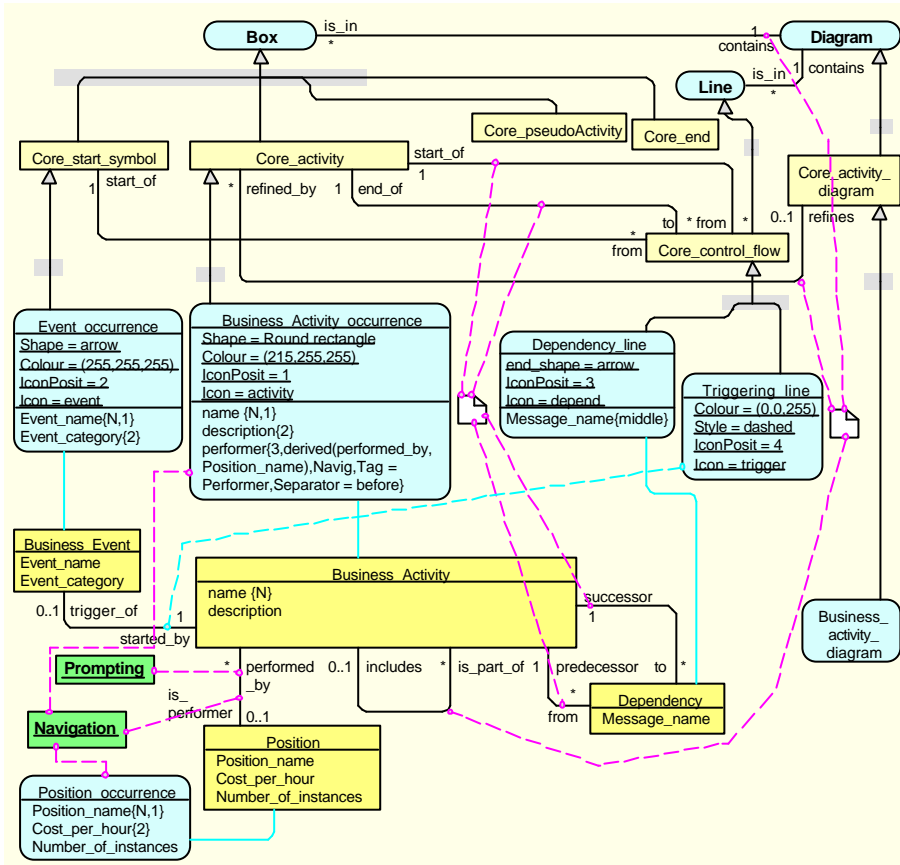


Figure 9. Process core and example of its usage

The next layer is the Business activity diagram definition itself (the presentation classes). It is pretty similar to that in Fig. 4 but not exactly equal. The first difference is purely technical. Since in most practical metamodels attributes are shown inside classes – as it is required by the MOF standard [15], we have switched back to this classical form. But this in turn requires dropping the notation of a compartment as a separate presentation class. Instead, the elements of the diagram definition – the

presentation classes – now contain compartments as attributes (as the second group of attributes, the first group defines the presentation style as before). The attribute name is the compartment name, the properties (in braces) specify the compartment presentation options. Among these options the number specifies the compartment position, **N** specifies that the compartment plays the role of a visible object name, other options are defined as tagged values, just as in Fig.4. The second difference is that *contains* associations now are at presentation class level (they are inherited from the Supercore) – namely the graphical objects are parts of diagrams but not domain objects.

The next layer is the logical metamodel, the same as in Fig. 4, but again with attributes inside classes. Unnamed associations (shown by bold lines) define the **mapping** from these real objects to the corresponding presentation objects in the diagram definition. But a slightly more general meaning can be given to this mapping. The instances of presentation classes are also real objects – boxes visible in diagram instances – the objects manipulated directly by the graphical editor (here – the business activity editor). But the classes of the logical metamodel are the domain model elements in the background (maintained by the modelling tool in the repository). To emphasize this fact, some presentation classes are renamed **occurrences**. In the default mapping (assumed here) one real object has one occurrence in a diagram. But mapping associations can be given explicit cardinalities saying e.g. that one *Business activity* can appear as a box several times in diagrams. The mapping of logical objects to presentation objects requires that their associations also have to be mapped. This derived mapping is shown in Fig. 9 via dashed lines (with a note symbol in between if this is a groupwise mapping).

This generalized concept of mapping permits to define several **alternative** graphical representations for the same fragment of the logical metamodel, which can be automatically kept synchronous by the modelling tool. You simply have to define the corresponding diagrams and define mappings from the domain metamodel fragment to all of them. This feature is a significant improvement when compared to the existing Meta-CASE environments such as Metaedit+[4] or GME[6], where the presentation classes coincide with domain metamodel fragment and this metamodel can not be freely defined.

We conclude the description of EdDL with a note on stereotypes. The example shows the styles of diagram elements (style attributes of presentation classes) strictly defined in diagram definition. But a **user stereotype** concept can be added to EdDL to make this definition more flexible. User stereotype is a named alternative style definition for a presentation class (possibly with a separate icon in the palette). Thus actually

a simplified (1.3 level) UML stereotype concept is implemented, but for any diagram type, not just UML. Via this feature the modelling notation can have “lightweight” extensions during the modelling process (freely combined with the “heavyweight” extensions done during the diagram definition).

4. IMPLEMENTATION PRINCIPLES

The core of the implementation is the **Editor Engine** – an interpreter directly interpreting EdDL. When an editor has been defined in EdDL the Editor Engine acts as the desired graphical editor for the end user. Here only the main principles of implementation will be discussed.

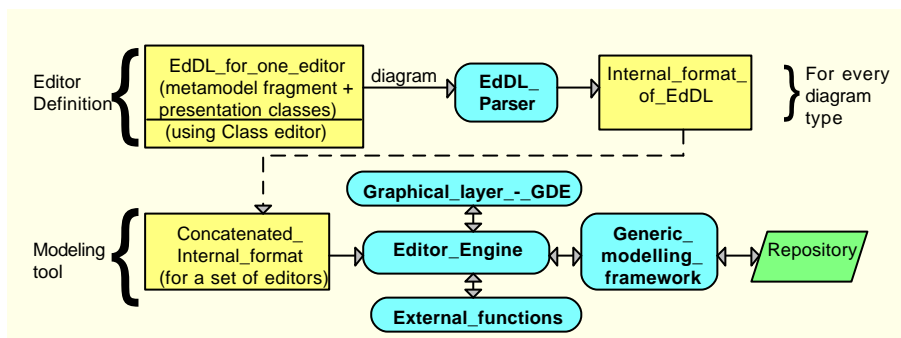


Figure 10. Architecture of EdDL implementation

Fig. 10 shows the general architecture of the EdDL approach. A key aspect is that Editor Engine (EE) relies on **Graphical Diagramming Engine (GDE)** for all diagram drawing related activities. The primitives implemented by GDE - diagram, box, line, compartment etc. and the supported operations on them are very fit for this framework.

Thus the interface between EE and GDE is based on very appropriate high level building blocks, there is no need for low level graphical operations in EE at all. The GDE itself was developed by IMCS UL initially within the framework of ADDE project, with a commercial version later on. It is based on very sophisticated graph drawing algorithms [13].

The general technology of using EdDL for defining a set of editors is quite simple. For each diagram type in the modelling methodology the corresponding editor definition is built on the basis of the relevant domain metamodel fragment and using an appropriate diagram core from the core library. Then these definitions are parsed and assembled into one set. EE

"performs" this set, behaving as a modelling tool containing the defined set of diagrams. A lot of tool functionality, e.g., "small operations" such as copy-paste are implemented in EE in a standard way and need no special definitions. Certainly, external modules can also be included for some domain-specific operations.

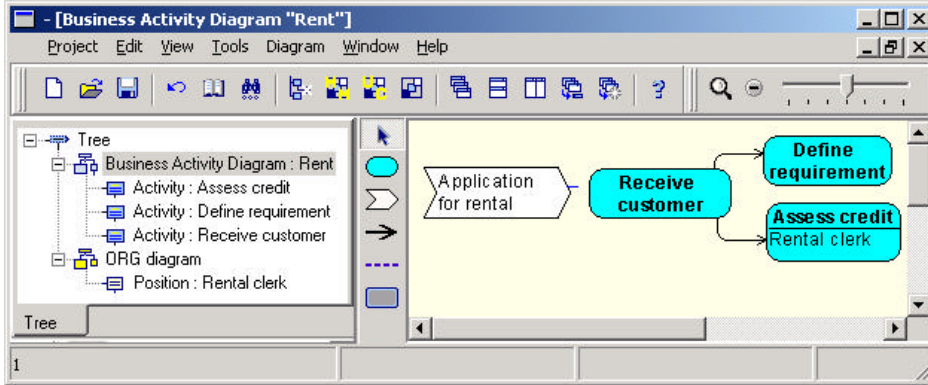


Figure 11. Editor in action

Actually EE runs under the guidance of the **Generic modelling framework (GMF)**, which directly manages the repository and incorporates also other non-diagrammatic generic editors. It is universally accepted nowadays that the logical metamodel describes directly (or very close to it) the physical structure of the tool repository. According to this principle all data-related operations within GMF are metamodel based, various repository formats such as OODB, a special tool repository, a relational DB or even a file in XML format are supported in a uniform way. All model-level operations such as load and save are also automatically supported by the framework. But the most important feature of this framework is the possibility to define a flexible **model content browsing** – via rich **model tree definition** facilities, based on the same metamodel.

A very brief schema for defining a model tree is the following. An arbitrary tree starting with a root node and consisting of two kinds of nodes : **constant nodes** (representing some fixed text) and **object nodes** (representing instances of one class contained in the model) is defined. Then during the usage of this tree an object node is replaced by the set of nodes corresponding to all these instances. But the most non-trivial feature of this approach is the definition of **selection rules** for object nodes. For any object node a selection rule can be specified, saying which of the possible instances actually must appear in the tree. To be short, a selection rule is a Boolean expression on **link conditions** (specifying that a certain sequence of metamodel associations must link the given node to a node higher in the

tree) and **attribute** value **conditions**. The actual expression language can contain some more details (including simple existential quantifiers), but even the mentioned features permit to define nearly any reasonable “treeview” on the model contents. **Recursive tree definitions** are also supported. The definition facilities permit also to define action lists (edit, new, delete etc.) for any node, thus e.g. all the links between the model tree and diagram editors are supported automatically by the framework. The left pane in Fig. 11 shows one of the possible model trees for the Business activity modelling, defined via the described facilities.

The first commercial version of Generic Modelling Framework has been implemented. The flexibility of the approach has been tested by implementing full UML 1.3 (except sequence diagrams) and various specific business process notations. The implementation has a special feature that e.g. the same business process may be alternatively presented as a UML Activity diagram or IDEF3 process chart. The defined editor sets (like the one in Fig.11) have reached the industrial quality of typical modelling tools, with all the required user support.

5. CONCLUSIONS

The practical experiments have demonstrated that the above described metamodel-based graphical editor implementation method is realistic and can reach an industrial quality. However this approach can be made significantly more universal, since it is generally accepted that a metamodel (class diagram) can be used for describing the logical structure of nearly any system. Thus the same approach of extending this model by special “presentation” classes could be used, e.g. to define dynamic semantics of the model, its simulation etc., but this is out of scope for this paper.

References

- [1] Kalnins, A., Barzdins, J., Podnieks, K., Zarins, A. et al. Business Modeling Language GRAPES-BM and Related CASE Tools. *Proceedings of the Second Baltic DB&IS'96*, Institute of Cybernetics, Tallinn, 1996, pp.3-16
- [2] Advanced Business and System Modeling Tools, <http://www.gradetools.com/>
- [3] Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: *Metaedit – a flexible graphical environment for methodology modelling*. Springer-Verlag, 1991.
- [4] MetaEdit+: Technical Summary. <http://www.metacase.com/papers/index.html>
- [5] Ebert, J., Sutzenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. *Proceedings of the 9th International Conference, CAISE'97*, Barcelona, Catalonia, Spain , 1997, pp.203-216

- [6] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment, *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 17, 2001
- [7] DOME Users Guide. <http://www.htc.honeywell.com/dome/support.htm>
- [8] Esser, R.: The Moses Project. <http://www.tik.ee.ethz.ch/~moses/MiniConference2000/pdf/Overview.PDF>
- [9] ESPRIT project ADDE. <http://www.fast.de/ADDE>
- [10] Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: Towards a Metamodel-Based Universal Graphical Editor. *Proceedings of the Third International Baltic Workshop DB&IS*, Vol. 1. University of Latvia, Riga, (1998) 187-197
- [11] Kalnins, A., Podnieks, K., Zarins, A., Celms, E., Barzdins, J. Editor Definition Language and its Implementation. *Proceedings of the 4th International Conference "Perspectives of System Informatics"*, Novosibirsk, 2001, pp.278-281
- [12] Rumbaugh, J., Booch, G., Jacobson, I.: *The Unified Modeling Language Reference Manual*. Addison-Wesley (1999)
- [13] Kikusts, P., Rucevskis, P.: Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors. *Lecture Notes in Computer science*, Vol. 1027. Springer-Verlag, 1996 pp.361-364
- [14] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.
- [15] Meta Object Facility (MOF) Specification, OMG, 2001 (<http://www.omg.org>)

DIAGRAM DEFINITION FACILITIES IN A GENERIC MODELING TOOL

Lelde Lace, Edgars Celms, Audris Kalnins

{Lelde.Lace, Edgars.Celms, Audris.Kalnins}@mii.lu.lv

The foundation of a generic modeling tool is its flexible diagramming facility. The paper proposes a new method for declarative specification of such facility. The proposed method permits to build up several diagrammatic presentations from one domain. The main principles of the proposed method such as mapping, presentation metamodel and some other details of the method are explained.

1 Introduction

Today there are a lot of modeling tools on the market. Modeling tools are designed to provide all what is necessary to support major areas of modeling, including business process modeling, object-oriented and component modeling with UML [1], relational data modeling, and structured analysis and design, etc. Why it is not sufficient to use “hard-coded” modeling tools? Let us consider for example the situation in business modeling. On the one hand there exist several well-known business-modeling languages (IDEF3 [2], ARIS [3] etc), each with a set of tools supporting it. But there are also Activity diagrams in UML, whose main role now is to serve business modeling. There is also GRADE BM [4] – a specialized language for business modeling and simulation. Thus for the area of business modeling there is no one best or most used language or tool, each of them emphasizes its own aspects. The problem with flexible modeling environment is even more urgent for domain-specific modeling, where countless special notations are used for separate domains.

There are probably several ways to solve such problem. You can develop a modeling tool specially for any specific modeling method but this way can be very time and cost consuming. You can make a tool, as universal as possible to support all needs but in practice such a universal tool is difficult for use in simple cases. An alternative way is a completely metamodel-based generic modeling tool (previously called metaCASE). Such a tool has no built-in modeling methodology. It has to be filled up with a specific metamodel and additional information to start modeling something.

Such approaches (metamodel-based) are proposed by ISIS GME [5], DOME [6] and Moses [7] projects. Several commercial modeling tools (STP by Aonix, ARIS by IDS prof. Scheer etc) use a similar approach internally, for easy customization of their products, but their tool definition languages have never been made explicit. Perhaps the richest diagram definition possibilities of them are in GME. According to the GME method the configuration is accomplished through metamodels specifying the modeling paradigm (modeling language) of the target domain. The modeling paradigm contains all the syntactic (defined by UML class diagram), semantic (defined by MCL – a subset of OCL, with some specific extensions), and presentation information regarding the domain. The presentation information is specified by assigning to the domain classes and relationships some special predefined kinds of stereotypes (e.g. <<model>>, <<atom>>, <<connection>>, etc.). In that way the domain classes are mapped to GME modeling objects. This means that the domain metamodel is modified during the specification process and there is no possibility to define various presentation notations for the same domain (for example, UML Sequence and Collaboration diagrams). Perhaps this method is sufficient for creating environments for domain-specific modeling in the engineering world but it is not good enough for the generic diagrammatic modeling environment.

The approach explained in this paper is in a sense a further development of the above-mentioned approaches. In our approach at first the domain metamodel of the modeling area is built independently of the diagram definition elements, which are built later and mapped to it. It should be emphasized that the domain metamodel itself is never modified during this process and remains a separate fragment, with only new associations attached.

2 Use of metamodels for editor definition

What does it mean to define an editor for a new diagram type? The result should have the same functionality as a “hard-coded” editor that would be made for this type of diagram. The result also has to conform to the modeling paradigm of this diagram. The paradigm consists of syntactic, semantic, presentation and interpretation specifications. The main goal of our approach is to solve the following problem – to enable building of several diagrammatic presentations from one domain (for instance, UML Sequence and Collaboration diagrams). Our approach has the following basic principles. First, Graphical Diagramming Engine is built as a separate component. Second, the domain metamodel is extended with other parts. Third, a new concept of Mapping (from one part of metamodel to another) is introduced, with its own syntax and semantics. Our approach is described in detail in following chapters.

2.1 Graphical Diagramming Engine

We use a separate component – Graphical Diagramming Engine (GDE) – to support graphical functionality of diagrams. This component is based upon complicated graph drawing algorithms [8]. This way we separate the graphical functionality of diagram building, which is not domain content related. With such an approach it is easy to add a new graphical functionality and to enforce more sophisticated graph building algorithms.

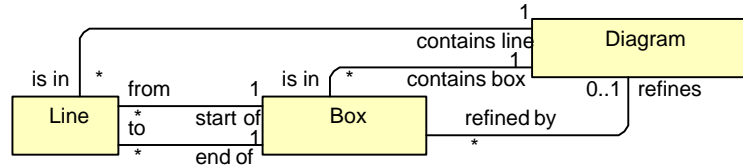


Figure 1. Interface metamodel of Graphical Diagramming Engine

The interface of GDE (see Fig. 1) supports creating a new and opening an existing diagram and allows performing some simple operations with diagram elements (add, delete, move, etc.). GDE supports automatic graph-drawing layouts of a diagram and enables graphical style modification of diagram elements. GDE maintains also the storage of graphical information on diagram elements – position, shape and color, etc.

2.2 Domain metamodel

An editor definition for a new diagram type starts with the description of the logical structure of the domain objects to be represented graphically by this diagram. This logical structure is described by a UML class diagram, which typically is called the domain metamodel.

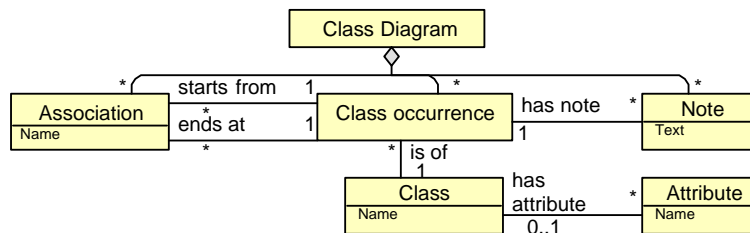


Figure 2. Class diagram metamodel

Figure 2 shows a simple example of such a domain metamodel for UML class diagram. This example will be used in the paper to demonstrate the editor definition features. We have intentionally simplified the domain metamodel (with respect to OMG standard) in order to make the explanation of our method easier to understand.

2.3 Parts of metamodel

It was already mentioned that our approach requires extension of the domain metamodel. The structure of the extended metamodel is shown in Figure 3. The Domain metamodel is a separate part. Two new parts are introduced. The *Core* metamodel corresponds to the Graphical Engine metamodel. The *Presentation* metamodel allows keeping the logical information on diagram graphical structure in the metamodel and corresponds to the specification of diagram presentation. There is only one *Core* in a metamodel but each diagram type has its own *Presentation* metamodel. Relations of *Core* and *Presentation* metamodels are described in chapter 2.4. Each *Presentation* metamodel is related with the corresponding *Domain* metamodel – this link is built based on diagram notation and dealt with in detail in section 2.5.

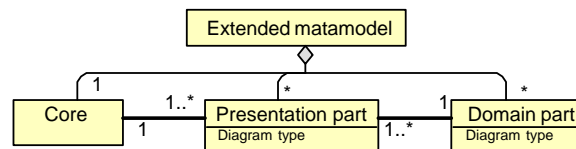


Figure 3. The structure of extended metamodel

2.4 Presentation part of Metamodel

Let's look at the parts attached to the domain metamodel. They provide realization of the specification of diagram presentation. The diagram notation determines which domain classes have to be represented in a diagram of this type. Every such class has to be represented in the *Presentation* part by its own **Presentation class**. The full logical information has to be kept in the metamodel (which boxes and lines are contained in the specific diagram and how they are interconnected). Since the basic connection information is common to

different diagram types, it is reasonable to define *Core*, which corresponds to the concept of an oriented graph. *Core Note* and *Anchor* are added to the *Core* because note symbols are contained in all diagram types.

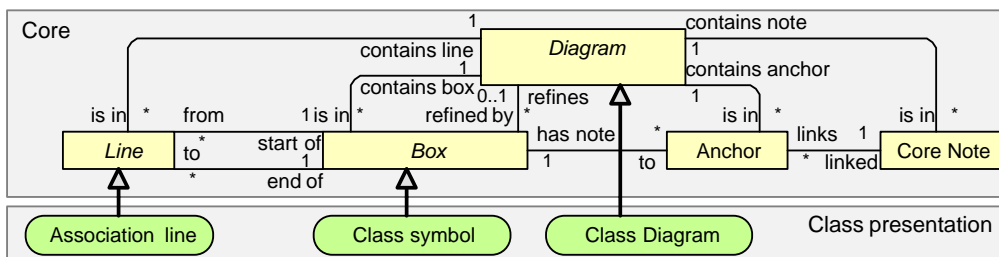


Figure 4. Presentation part of extended metamodel

The metamodel extensions - *Core* and *Presentation* parts of UML class diagram are shown in Figure 4. The abstract classes of *Core* (*Diagram*, *Box*, *Line*) are used as superclasses of Presentation Metamodel classes. By inheritance presentation metamodel classes gain their role in the world of diagrams. It can be seen that three Presentation classes are introduced – *Class Diagram*, *Class symbol* and *Association line*. Notice that *Core Note* and *Anchor* classes are taken from the *Core*. From now on presentation classes are shown as rounded rectangles.

Presentation metamodel is the only place to define graphical representation constraints of a specific diagram, if required by the diagram notation. But if it is necessary to build several diagrammatic representations of the same domain metamodel, each diagram requires its own Presentation metamodel.

2.5 Extended metamodel

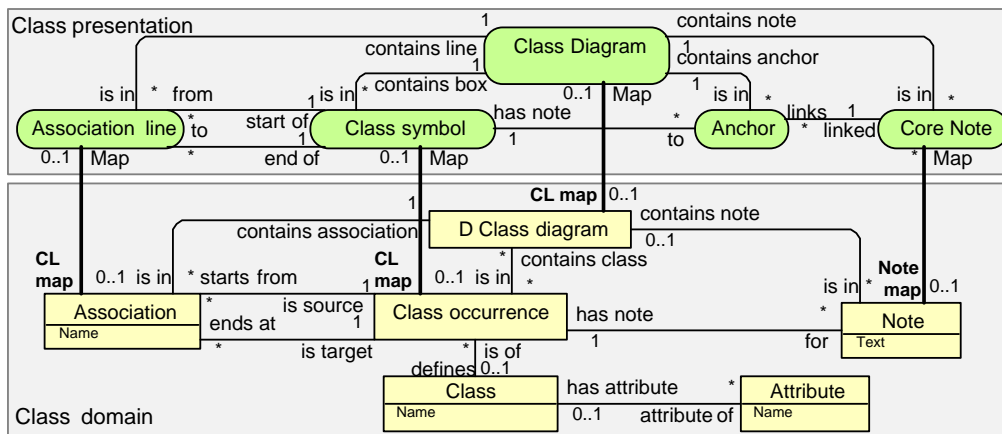


Figure 5. Extended metamodel

Figure 5 contains the complete extended metamodel. *Core* is omitted only for the ease of perception, and the inherited associations are drawn directly into the *Presentation* part. *Core Note* and *Anchor* classes with the respective associations are put into the *Presentation* part. The *Domain* metamodel is in the lowest part of the picture. In this and the following figures rectangles represent classes from the domain metamodel. The modeling paradigm defines which domain classes have to be shown in a diagram. In chapter 2.4 we have built corresponding presentation class for each such domain class. There are special associations called **Mapping links** (bold in the figure) drawn now, which connect presentation classes to the respective domain classes.

3 Diagram definition facilities

The basic ideas of our metamodel-based editor definition facilities have already been outlined in [9]. The most important part of Editor Definition is the Diagram Definition Language. This language specifies the graphical notation and editing functionality of the diagram type to be defined. In doing this it references appropriate elements of the metamodel.

The extended metamodel already contains some information on graphical presentation of the diagram. The Metamodel Presentation part contains Presentation classes. And there are the Mapping links to the Domain classes. But a Mapping link only describes the fact that instances of the given class must be shown in a diagram. The diagram changes during editing cause the following actions: class instances of the Presentation part have to change in accordance with the logical structure of the diagram, class instances of the Domain part have to change in accordance with the diagram notation, and both of these changes have to be mutually correct. Relation patterns between the Domain and Presentation parts of the metamodel are defined by the diagram notation. To support these patterns we have introduced **Mappings** of different types. Mappings are an important part of diagram definition.

3.1 Mapping

Let us explain what is meant by the concept of Mapping in our approach. A Mapping has a syntax and semantics. The syntax is defined by mapping type. A Mapping contains references to metamodel classes and associations (they have to correspond to the required type). The Mapping semantics (also dependent on the type) describes which actions take place with instances of metamodel elements (described by mapping syntax) when diagram is modified. Specific consolidation conditions also correspond to a mapping type. We have made a library of standard mapping types that allows to realize notations of most of popular diagrams. Let us explain some simplest mapping types used in our example.

3.2 Examples of simplest Mapping types

First some notations used in Figures 6 and 7: syntax components of a mapping are denoted by bold (boxes and lines), presentation classes are denoted by rounded rectangles, domain classes denoted by rectangles. Since a mapping is a component of the diagram definition, it can contain only references to the metamodel classes.

The diagram notation in our example demands that a *Class symbol* may be drawn in every diagram for either a new or existing *Class*. *Class symbols* can be connected with the *Association lines* and a *Note* may be added to a *Class symbol*. There are also domain changes corresponding to each of those actions.

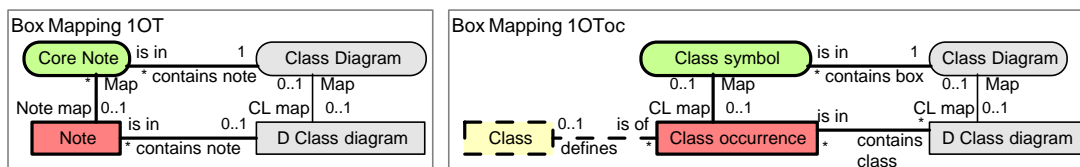


Figure 6. Box Mapping examples

The simplest type of box Mapping is 1OT that corresponds to the situation where one presentation class is mapped to one domain class. In Figure 7 to the left there is the Mapping for notes. The syntax part of the Mapping contains *Core Note* class (from Presentation part), *Note* class (from Domain part) and the Mapping link connecting both classes. To support correct incorporation into the diagram, the Mapping includes links to presentation and domain diagrams. Let's explain the semantics of this mapping. When a new note is created, the following actions take place: an instance of the class *Core Note* located in the Presentation metamodel is created. This instance is connected by *is_in* link to the relevant diagram instance. An instance of the *Note* class is created; Mapping link *Map* is created. Fact that a diagram contains a note must be kept in the Domain part also. So, the relevant instance of the domain diagram must be found. It can be found by using the Mapping link of the presentation diagram. Now we can draw the link between *Note* instance and *D_Class_Diagram* instance.

Box mapping type 1OToc enables the occurrence situation, which differs from 1OT by the possibility to create an occurrence of an existing *Class* instance or a new *Class* instance together with the occurrence. In the case of an existing class the domain metamodel link (*is_of*) to the class instance is added. In the case of a new class both the instance and the link are created.

Presentation parts of both mapping types are syntactically equivalent. The structure of the metamodel Presentation part implies that Presentation parts of all box-mapping types are syntactically equivalent.

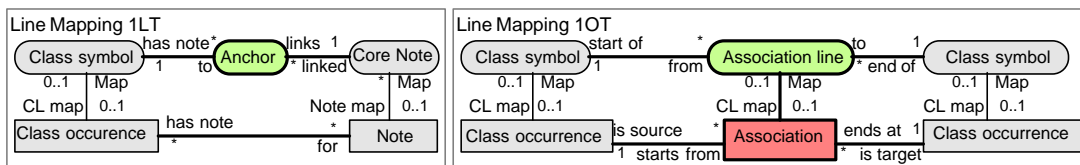


Figure 7. Line Mapping Examples

In line examples (Figure 7) *is_in* links are omitted, they are created the same way as in the case of boxes. Line formation is interesting in that it implicitly uses box Mappings. This is necessary to find the “domain endpoints” for a line at the instance level. Therefore each mapping type has one “main” Mapping link (in general Mappings may contain several Mapping links).

There is a box1 where the line starts, a box2 where the line ends and the line itself if any line is to be drawn. Relevant object instances in the domain part can be found using the existing mappings for box1 and box2. Next actions depend on the Mapping type. If the line Mapping type is 1LT (the case of *Anchor*), the link *has_note* is created which interconnects these object instances at the domain level. If the line is represented in the domain by a class (*Association line*), then the class (*Association*) instance is created and it is connected to the relevant domain object instances (*Class occurrences*) via the links defined by the Mapping.

Presentation parts of all line Mapping types are syntactically equivalent. Therefore, the Presentation part of the Mapping will be called the Mapping source. Each type of Mapping has a syntactically different target part. Mapping target part consists of the Mapping link and the relevant Domain classes and associations.

3.3 Diagram definition

The mapping concept introduced in the previous section permits to define a separate diagram element. The complete information about a specific diagram is contained in the diagram definition. A diagram definition is specified by means of the Diagram Definition Language. The diagram definition for the specific diagram type is built using the diagram configuration tool.

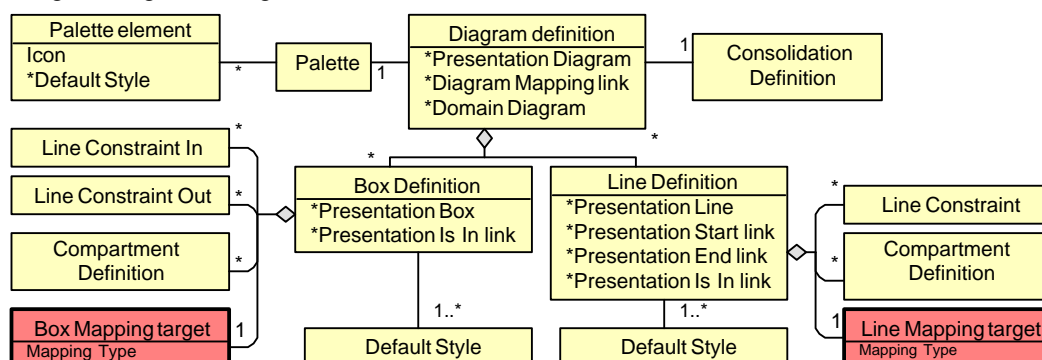


Figure 8. Structure of diagram definition

The structure of a diagram definition is shown in Figure 8. References to the metamodel classes in this figure are denoted by **Name*. A specific *Diagram definition* contains references to the extended metamodel classes (in our example **Presentation diagram=*Class diagram* and **Domain diagram=*D class diagram*). Metamodel classes have to conform to the required type, for instance the *Presentation diagram* has to be a subclass of the *Diagram*. The *Diagram definition* may contain several *Box* and *Line definitions*, which are related to the corresponding *Presentation* classes. Each *Box (Line) definition* has the appropriate mapping type. *Box (Line) definition* contains the Mapping source as an attribute. But the Mapping target is represented by a separate class, because it can have a different structure for each mapping type.

Each *Box (Line) definition* normally has one *Default Style*. The *Default Style* contains all the necessary graphical style information (to be used by the Graphical Engine) to display a box. The element of the palette contains pointer to the *Default Style*. If required by the *Presentation* specification, it is possible to build several *Default Styles* for the same *Presentation Class*, with a separate palette element for each style.

The *Compartment definition* is based on the metamodel, it is defined as a navigation expression. For instance, in our example the name of the class can be found this way: "*Class_symbol.map.class_occurrence.is_of.Class.Name*". More complex compartment definition patterns are also possible.

4 Conclusions

The described modeling tool definition method is powerful enough to define complete UML notation (including alternative presentations of interactions as UML Sequence and Collaboration diagrams). In addition, processes can be presented both as UML Activity diagrams and GRADE BM Process diagrams. The described Mapping library is sufficient to define diagrams of any reasonable type, but in case of necessity this library can be easily extended to more complicated Mapping types. The described method allows also an efficient implementation (in the sense of run-time necessary to build the diagrams from real amounts of data) and the practical experiments have demonstrated that the tools which use the above described definition method can reach an industrial quality.

References

- [1] Rumbaugh, J., Booch, G., Jacobson, I. The Unified Modeling Language Reference Manual, Addison-Wesley (1999).
- [2] Mayer, R., Menzel, C., Painter, M. IDEF3 Process Description Capture Method Report, http://www.idef.com/Downloads/pdf/Idef3_fn.pdf, Knowledge Based Systems Inc (1995).
- [3] Scheer, A.-W. ARIS Business Process Modeling, 3rd edn. Springer-Verlag, Berlin Heidelberg New York (2000).
- [4] Barzdins, J., Tenteris, J., Vilums, E. Business Modeling Language GRAPES-BM (Version 4.0) and its Application, *Dati, Riga* (1998).
- [5] Ledeczki A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P. The Generic Modeling Environment, *Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001*.
- [6] DOME Users Guide, <http://www.htc.honeywell.com/dome/support.htm>.
- [7] Esser, R. The Moses Project, <http://www.tik.ee.ethz.ch/~moses/MiniConference2000/pdf/Overview.PDF>.
- [8] Kikusts, P., Rucevskis, P. Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors, *Lecture Notes in Computer science, Vol. 1027. Springer-Verlag, 1996*, pp.361-364.
- [9] Kalnins, A., Barzdins, J., Celms, E., Lace, L., Opmanis, M., Podnieks, K., Zarins, A. The First Step Towards Generic Modelling Tool, *Proceedings of Baltic DB&IS 2002, Tallinn, 2002*, v. 2, pp. 167-180.

DIAGRAM DEFINITION FACILITIES BASED ON METAMODEL MAPPINGS

Edgars Celms, Audris Kalnins, Lelde Lace

University of Latvia, IMCS, Riga, Latvia

{ Edgars.Celms, Audris.Kalnins, Lelde.Lace }@mii.lu.lv

The paper proposes a new technique for diagram definition in a generic modeling tool, which permits to build several diagrammatic presentations for one domain. The main idea of the proposed method is a mapping from presentation to domain part of a metamodel. The paper describes the semantics of mappings, using a fragment of UML activity diagram as a definition example. In conclusion suggestions are given how the approach could be applied in MDA context.

Introduction

Besides the traditional modeling tools built for a specific modeling notation such as object modeling in UML, business modeling etc. there is a significant niche for generic modeling tools where any modeling notation can be supported without programming in the traditional sense. Typically the modeling notation (language) in generic modeling tools is specified by means of a metamodel, which is then augmented by a tool specific annotation, markup etc, to define the actual tool functionality. The main application area for generic modeling tools is domain-specific visual languages for various industry domains. The current key players in this area are ISIS GME [1], DOME [2], MetaEdit [3] etc, which have gained certain maturity now.

The classical graphical modeling by means of sets of related diagrams can also be considered to be a domain-specific area, especially the business modeling, where there is no one leading modeling language, but a number of quite similar competing notations. UML with its Activity diagrams is just one of the possible notations there. Therefore the generic modeling approach is valid also for the domain of business modeling. This domain poses some specific requirements for the tool, the most important one being the necessity to represent the same domain concepts via several graphic notations simultaneously. The paper discusses the Generic Modeling Tool (GMT), developed by University of Latvia together with Exigen company, built especially for the abovementioned purpose. Namely the requirement for access to the same model data via several graphical notations demands a number of specific solutions for defining the relations between diagrams and domain metamodel (the diagram definition language), which can not be so easily accomplished in the well-known metamodel-based tools [1,2,3]. For example, there it is practically impossible to have some domain object represented as a symbol in one notation and as a line in another.

The main such idea is the mapping between the parts of the metamodel – the domain and the presentation part (the latter ones may be several). Some preliminary presentation of the approach has been given in [4,5], but this paper concentrates on precise definition of the mapping semantics, using UML and OCL. Though developed completely independently, the style of the semantics definition bears some similarity to the more theoretical paper [6], where the concept of set-theoretical relation between metamodel parts is used.

It should be noted that the requirement for alternative graphical notations is present in the UML itself (including the version 2.0) – interactions can be shown both as sequence and collaboration diagrams, there are alternative forms of showing action performers in activity diagrams. The paper will demonstrate the mapping idea on a small fragment of UML 2.0 activity diagram metamodel [7], finally showing how the same model data could be presented as ARIS eEPC [8] diagrams – the most popular business process notation.

Today the hottest area in UML related modeling is MDA [9]. The paper concludes with some suggestions, how the proposed approach can be used for this goal too.

Structuring of metamodels

Strictly speaking, a metamodel for a modeling notation such as UML describes only the domain concepts – the abstract syntax in other terms. But any modeling tool must manipulate also the elements of the diagrammatic presentation – the concrete graphical syntax. UML documents [7] does not specify that part of the metamodel, a very generic description of the presentation part is given in [10], however with another goal in mind – defining an easy interchange format for graphics. Therefore as a rule modeling tools internally use another presentation metamodel, and so it is in our approach. Similarly to [10], we assume a diagram to be a directed graph consisting of nodes (boxes) and edges (lines). The presentation metamodel is also the only natural place, where various constraints on diagram building can be easily specified.

In our approach any metamodel is built according to MOF standards – it is a class diagram using the syntax features permitted by MOF. The elements of a metamodel may be grouped into packages, so we can speak of **domain** and **presentation packages**. Fig. 1 shows the domain package for the UML 2.0 activity diagram (actually a small fragment of the original one in [7], but sufficient for demonstrating the ideas). It should be reminded, that the *Activity* class plays the role of the “domain diagram” – its instances correspond to instances of visible activity diagrams.

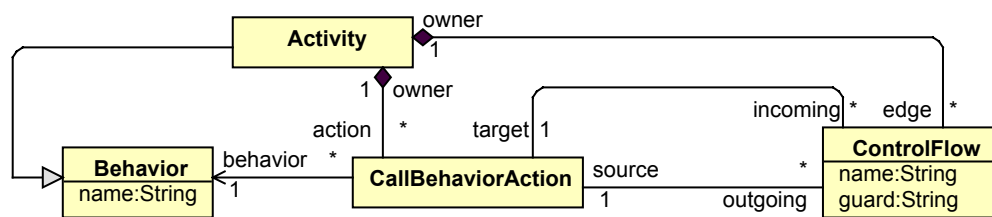


Figure 1. Domain package for UML activity diagram

To reflect the fact that any diagram is a graph, a special **DiagramCore** package is introduced which defines the general properties of a graph. Any specific presentation package inherits these properties from the core and may add some specific features required by the graphic syntax (actually the DiagramCore is more than a graph – it supports element nesting etc., but we omit this for simplicity). The DiagramCore elements also contain attributes characterizing their geometrical properties, but we ignore them here. Fig. 2 shows the DiagramCore package together with the presentation package for activity diagram.

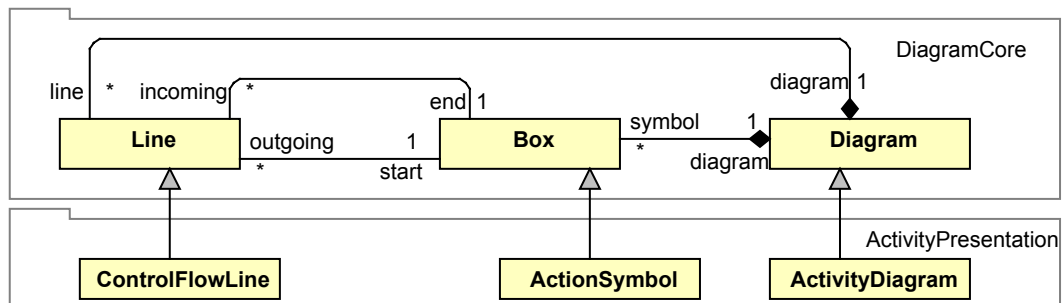


Figure 2. Diagram core and Activity presentation packages

Similarly, for any diagrammatic notation there is one domain package and one or more presentation packages, all of which inherit their essential properties from the DiagramCore. In a sense, it is an extension of the original domain metamodel, but with the domain packages left unmodified. The only elements, which will be added to the domain part of the

metamodel, will be some new associations – mapping associations, which are the basis of our approach and which will be discussed in the next sections.

One more consequence of the DiagramCore is that we can use a separate component – **Graphical Diagramming Engine (GDE)** – to support the graphical functionality of diagrams. This component is based upon complicated graph drawing algorithms [11] and implements all tool features, which can be expressed in terms of extended directed graphs. This way we isolate the pure graphical functionality of diagram building. GDE supports creating a new and opening an existing diagram and allows to perform all graphical operations with diagram elements (add, delete, move, etc.). GDE supports also automatic high quality layouts of a diagram and enables style modification of diagram elements.

General principles of mapping

The domain and presentation parts of the metamodel for a modeling notation must be linked together to define the real modeling functionality. In most cases, a class in the domain part corresponds to a class in the presentation part, but these correspondences may be also more complicated. The main facility for defining a relation between the domain and presentation parts of the metamodel is mapping.

In the simplest case, the **mapping** consists of a **class** in the **presentation** package, the corresponding **class** in the **domain** package and an **association** connecting them. It expresses the fact that as soon as there is a presentation class instance (e.g., action symbol) there must also be the corresponding domain class instance (CallBehaviorAction) and vice versa, and they must be linked by the association instance (link). Typical association multiplicities are 1 – 1 (for one graphical notation). The associations (called **mapping associations**), navigable to both ends, form the base for efficient data management in the Generic Modeling Tool.

However, there is more semantics related to a mapping. Thus, the action symbol must be in the diagram, which is mapped to the activity containing the action. Even more complicated rules constrain mappings for lines, where natural conditions tie a line mapping to its end box mappings. Thus, mappings form hierarchical structures, corresponding to basic diagrammatic constructs or patterns. Each such pattern corresponds to a **mapping type** in our approach. Some basic mapping types will be discussed in the next section. Each of these mapping types will have its **syntax** – the involved metamodel elements, and **semantics** – what constraints must be true for the mapping to be in place (or in other words, what must be done, if one of the mapping ends has changed).

The mapping semantics is based on the hierarchy of mapped elements. There is one common principle in this semantics, so called **scaffolding principle**, explained in Fig. 3. The scaffolding principle specifies how mappings must be consistent with the element hierarchy on both ends. The explanation of the principle to a certain degree reminds the relation principle in [6], but is simpler.

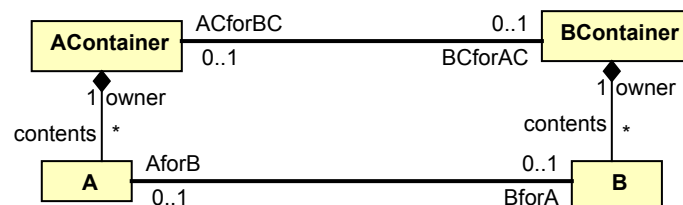


Figure 3. Scaffolding principle for mappings

Let A and B be classes in the presentation and domain packages respectively, involved in a mapping, and let $AContainer$ and $BContainer$ be their corresponding owners in the hierarchy (typically, a diagram and its domain equivalent), also having their own mapping.

Both mapping associations are displayed bold. These elements are assumed to be really present in the metamodel. Fig. 3 actually represents a general **scaffolding schema**.

In a totally correct model the mapping association multiplicities must be 1 – 1 (we consider here the one-one case, the one-to-many case is a completely different pattern). However, for an in-process model (being modified by the tool) some mapping instances may be temporary missing, therefore multiplicities in the schema are set to 0..1.

According to the principle, the following two constraints given by OCL expressions must always hold for the scaffolding schema involving the abovementioned mappings:

Context A inv:

BforA->notEmpty() implies owner.BCforAC = BforA.owner

and

Context B inv:

AforB->notEmpty() implies owner.ACforBC = AforB.owner

These constraints express the fact that *A* to *B* mapping is consistent with the corresponding container mapping – e.g., a symbol maps to an action in the right “domain diagram”.

The most important condition for this schema is the **local completeness** for one container (diagram), which can be expressed by the following OCL constraint:

Context AContainer inv:

contents -> forAll (a | a.BforA->notEmpty()) and

BCforAC->notEmpty() and

BCforAC.contents -> forAll (b | b.AforB->notEmpty())

The constraint says that for this container all its elements are mapped (consistently with the container mapping) and, in addition, are mapped to all elements of the partner (domain) container – the mapping is complete both ways for the given container. Mapping for any diagram type will try to maintain its local completeness for the current diagram instance.

The scaffolding principle is the base for all mapping types to be discussed in the next section.

Mapping types for the activity diagram example

Now the mapping types to be used for the activity diagram example can be defined. The base for all other mappings is the **diagram mapping** – a special singleton mapping. Fig.4 shows the diagram mapping for activity diagrams. It says that each activity diagram instance must correspond to an *Activity* class instance, having a consequence that creating a diagram in the tool must invoke a new *Activity* instance creation.

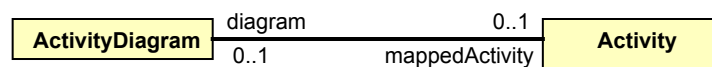


Figure 4. Diagram mapping

The simplest non-trivial mapping type is for a kind of diagram symbols to be mapped to a domain metamodel class. In our simplified activity diagram example there is only one symbol kind – *ActivitySymbol*, which must be mapped to the *CallBehaviorAction* in the domain. This mapping type will be named **IOT** (symbol to 1 Object Type).

Each mapping type defines its mapping schema, based on the scaffolding principle. The mapping schema contains a number of metamodel elements both from presentation and domain packages – the **mapping syntax**, which must be substituted by concrete metamodel elements when concrete mapping is defined. The **mapping semantics** is, firstly, inherited from the scaffolding principle (its constraints), and more constraints can be added for a mapping type definition. But a concrete mapping, as a rule, adds no OCL constraints (however, there is such a possibility in the modeling tool), so concrete diagram definitions

typically requires no explicit use of OCL and constraints inherited from mapping types can be implemented in a more efficient way by the modeling tool.

Fig. 5 shows the mapping schema for the mapping type 1OT. The mapping syntax (list of parameters) contains the classes *Diagram* and *Symbol*, which must be located in the metamodel presentation package, and the classes *DomainDiagram* and *DomainElement*, which must be in the domain package. The two associations (*Diagram* to *Symbol* and *DomainDiagram* to *DomainElement*) are also part of the syntax and must be found in the metamodel. The mapping association for diagrams must already be defined. But the mapping association for the *Symbol* (actually, for its real counterpart in the metamodel) must be specially created before a concrete mapping is defined.

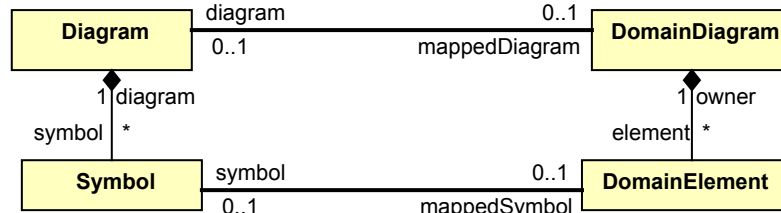


Figure 5. 1OT Mapping schema

The mapping type 1OT adds no new constraints to those inherited from scaffolding schema. Let us remind, that it requires also the local completeness condition to be true for a mapping to be complete. In practice, these constraints imply that creating a new symbol in a diagram means also the creation of the domain element – thus the “operational semantics” required by the tool is also defined by the mapping type.

There is also a variation of the mapping type 1OT named **1OTD** (Object Type with Definition), which adds one more class in the domain, linked by an association to the *DomainElement*, this additional class serves as a common “definition” for the domain elements. Namely the variation 1OTD is used for the activity example – see the concrete action mapping in Fig. 6, with *Behavior* in the role of the definition (the association *refinement* is explained below). Associations inherited from the *DiagramCore* here (and in the next figure) are shown directly between the presentation classes.

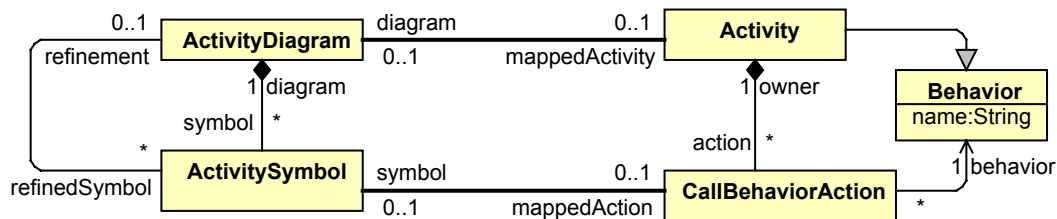


Figure 6. Action mapping according to 1OTD

We will demonstrate one more extension of the type 1OT – **1OTR** (1OT with **refinement**). An action referencing another *Activity* (not a simple *Behavior*) implies that the corresponding *ActionSymbol* must be displayed as refined (with the rake icon in it) and must support hyperlinking to the appropriate diagram. Due to restricted space the definition of 1OTR is not given, just its application is shown in the same Fig. 6. This extension requires new OCL constraints, which here will be demonstrated directly in the application (not in the schema definition, as it in fact is). The idea is that we select an association in the presentation package (with the role *refinement*), which makes the *ActionSymbol* to be refined. This association should be paired to an association in the domain between the relevant classes, in this situation the same *behavior* association may be used, in case if it leads to another *Activity* (as a subclass of *Behavior*). The same *refinement* association also enables hyperlinking to the appropriate diagram. 1OTR schema requires the following additional OCL constraints (here shown in the concrete context of Fig. 6)

Context ActionSymbol inv:

refinement->notEmpty() implies (mappedAction.behavior.ocllsTypeOf(Activity) and refinement= mappedAction.behavior.diagram)

The next essential mapping type is for lines in diagram, which correspond to classes in the domain. This mapping type is named **L1OT**. To save the space, we again do not present separately its schema, but show its application for defining control flows in activity diagram. The basis again is the scaffolding schema, but a similar principle here has to be applied directly in a specific context, in order to specify that line ends (the boxes to which the line is attached) are mapped accordingly. Fig. 7 shows this definition.

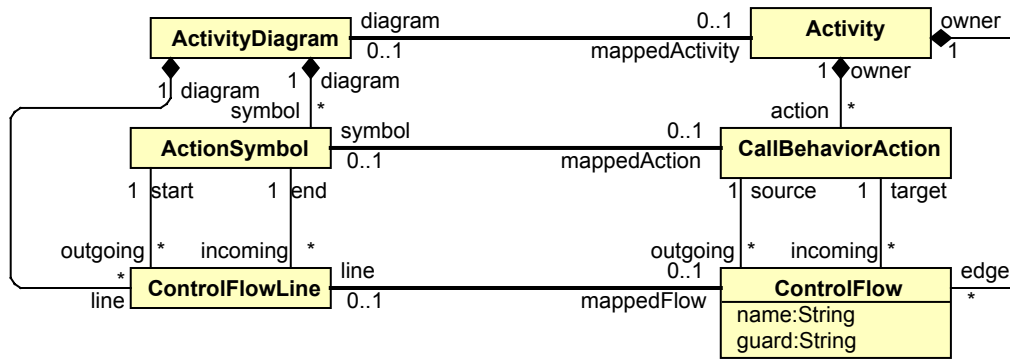


Figure 7. Control flow Line Mapping

A direct application of the scaffolding schema here links the line mapping to the diagram mapping, with the corresponding constraints inherited. But the action mapping is repeated here to specify the line ends. In this simple example there is only one symbol type, but if a control flow could be drawn between different symbol types (as it is in real activity diagrams), all these mappings had to be present in the definition. However, this is required only for the semantics definition, in real diagram definition facility only the relevant pairs of symbol types are to be listed (and also the relevant association roles in the domain), the corresponding OCL constraints are inherited from the L1OT schema (here shown only as its application in Fig. 7). The additional OCL constraints express the fact that both the line start and end boxes are mapped to domain elements, which serve as a source and target respectively for the domain image of the line. The constraints are the following:

Context ControlFlowLine inv:

start=mappedFlow.source.symbol and
end= mappedFlow.target.symbol

Here the assumption is made that *start* leads to the “geometrical” start box of the line.

The described mapping types are sufficient for defining this simplified activity diagram. But certainly our environment has more mapping types, which cover all the typical diagramming patterns (the mapping type library contains about 20 such mappings). All of them can be defined in a way similar to those presented in this paper – their syntax as mapping schemas and semantics as OCL constraints.

Diagram definition facilities in GMT

Mappings between the domain and presentation packages of the metamodel form the basis for diagram definition in our GMT approach. To define a diagram type, mappings for the diagram itself and all its elements – boxes and lines must be defined. To define a mapping in GMT environment, one has to select the appropriate mapping type from the library and to specify all the required syntax elements – appropriate metamodel classes and associations. The semantics of each mapping type is predefined in GMT (in the way presented in this paper), therefore there is no need for explicit OCL in normal cases. Certainly, for each mapping a lot of technical details may be specified – one or more default styles for diagram

elements, corresponding icons in the palette etc. For lines an important aspect is to specify, between which pairs of box types they may be drawn. Normally it is just a list of type pairs. But it is also one of the places where explicit OCL may be of use, in order to specify context dependent constraints, especially on multiplicities, present in some modeling notations. These constraints typically are defined in the presentation package. Some other diagram integrity constraints expressible in OCL are also available. A special case of line mapping is that mapped to a “pseudoline” – box nesting, available in our core and used e.g., for nested states in UML statecharts.

Yet another aspect is how the model data – attributes of domain classes and contents of subordinate classes (e.g., class operations) are mapped to graphical text slots of diagram symbols (lines) – compartments. For each compartment an OCL-style “navigation expression” points to the domain attribute, which supplies that value. For example, in the case of mapping IOT this expression contains just the role name of the mapping link and the attribute name. More than one data supplier expression can be used for complex compartments, and navigation expressions yielding a set of values are used for “list compartments”, such as class attributes or operations in UML. The way how the actual string in a complex compartment is composed from the selected data values is specified by means of a “pattern” - a simple regular grammar. Certainly, the definition component in GMT always prompts the most typical values for compartment definitions, so in simple cases nearly everything is provided automatically.

When the appropriate diagram definitions are supplied, GMT acts as a commercial modeling tool for the given notation, with all typical services enabled.

Alternative diagrammatic representations

One interesting aspect of GMT is the possibility to have different graphical representations for the same domain data. This is accomplished by defining more than one mapping (including a diagram and all of its elements) for a domain diagram, such as *Activity*. Then all of these mappings are active simultaneously, and all the representations can be used to view or edit the model data. Which views are really visible, is defined via the model browser (tree) specifications – a topic out of scope for this paper. When the model (domain) data are modified through one of the diagram views, the alternative ones have to be automatically updated also – this process is called **consolidation** in GMT. Since the mappings actually are bidirectional (due to symmetrical constraints in the scaffolding schema), the updates of diagram elements when domain elements change in general are straightforward. In some special cases OCL preconditions for consolidation may be used. But certainly all this assumes the existence of a “domain diagram” (*Activity*, *StateMachine*, *Interaction* etc), which determines what really must be inside one graphical diagram. Though some other patterns (mapping types) in GMT permit a situation such as for UML class diagrams where no “domain diagram” exists, alternative representations require such one.

To give some insight into alternative graphical representations, we will sketch briefly, how our simple activity domain could be represented via simplified ARIS eEPC [8] diagrams (see example in Fig.10). Here we assume that a (mandatory) named event symbol (hexagon) between two function (=action) symbols actually represents a named control flow (possibly with a guard). The event could be treated also as an object flow in activity notation, but then eEPC had no control flows at all and the alternative mapping would be nearly the same as that for activity diagram with object flows, having only different presentation classes. Fig. 8 shows the set of mappings between a simplified eEPC diagram presentation package and the same activity domain (with DiagramCore in the leftmost column). We remind that mapping associations are displayed bold in the picture.

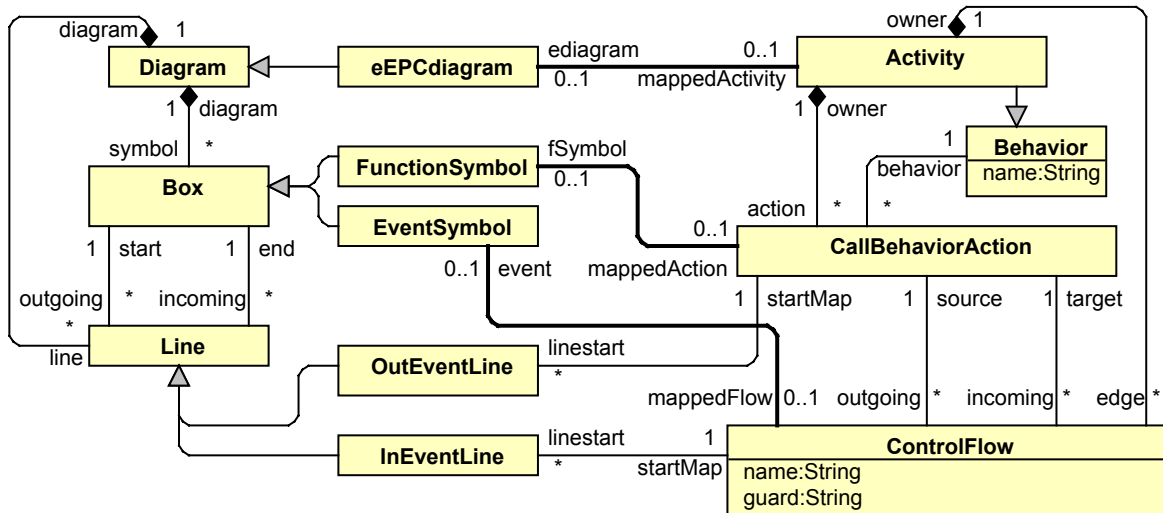


Figure 8. ARIS eEPC Mapping to activity domain

It can be seen, that the eEPC function symbol is mapped in a normal way (as for activity diagram) to *CallBehaviorAction*, using the mapping type 1OTD. The event symbol is mapped to *ControlFlow* (which was mapped to a line in activity diagram!) – the mapping type is 1OT. The attributes of *ControlFlow* – *name* and *guard* are combined into the event name compartment. Both lines types in eEPC (from function to event and vice versa) have a new mapping type L1LT – each of them actually corresponds to a domain association, but not a class. This type of mapping is based on so-called startMap – a specially built association to the domain class, where the desired association starts, with the required association role specified as a constraint (e.g., *target* from *ControlFlow* for the *inEventLine*). It should be noted that L1LT-mapped lines can have no texts – there is no place for data in the domain.

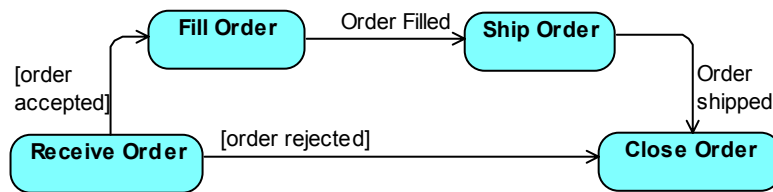


Figure 9. Activity diagram example

Thus the alternative mapping of the activity domain has been defined (certainly, the mapping details are visible only in the real GMT definition component). The consequences for GMT are the following – if we draw an activity diagram (in Fig.9), then the equivalent eEPC diagram (in Fig. 10) is drawn automatically via the consolidation process mentioned above. The activity diagram is clearly a fragment of the real notation – only the elements defined in Fig. 1 and 2 are used.

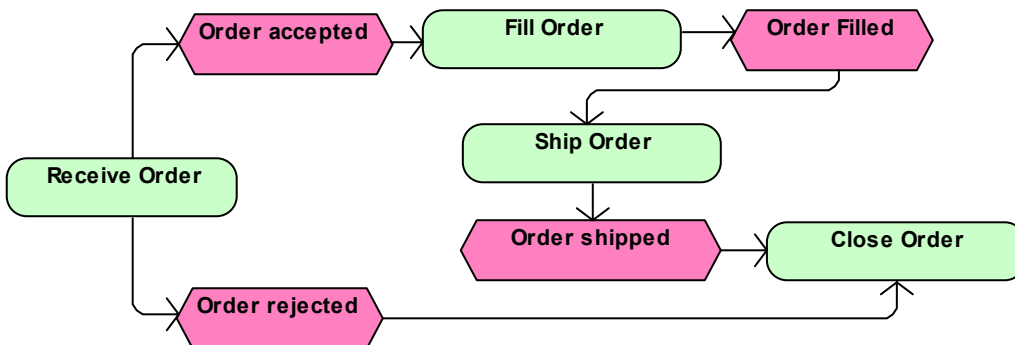


Figure 10. Equivalent ARIS eEPC diagram example

Alternative mappings for the same domain, provided in GMT, can be extended to complete UML 2.0 activity diagram (actually, its business modeling subset, full domain is much richer than that meaningful for ARIS) and complete ARIS eEPC diagram. In most cases the mappings are quite similar (the same domain concept is represented by boxes in both notations, e.g., flow join symbol and AND-rule). The non-trivial cases are when a box is used in one notation and line in another – besides the one explained in Fig. 8 a similar problem is for performer notation.

In general, when a domain for one modeling notation is found to be usable (from the semantics point of view) for another one, the building of alternative mapping can be started. It is done by drawing a candidate mapping association from a presentation class to the domain class, which most naturally corresponds to this presentation class and contains the main data to be shown in the symbol. Then, looking at adjacent domain classes, which may also contain relevant data or determine the connections, the appropriate mapping type from the library is found (there may be cases, when one presentation class maps to a structure of 2 or 3 domain classes). And conversely, two presentation classes may map to the same domain class (the “derived” mapping types are also provided). For lines, the possibility for LILT mapping (as in Fig. 8) must also be checked, as well as “pseudoline” (box nesting) case. All this determines the supported variations between both notations – what is one symbol in one notation, can be several ones in another, a box may become a line and vice versa, but there must be some “local correspondence” anyway. Currently there is no formal procedure for deciding whether an alternative mapping can be built, it is a subject for future research. Simply, in all practical situations we have succeeded – it is more a question of the mapping library completeness. And, certainly, the main question – whether two modeling notations are semantically equivalent and in principle can have a common domain – is completely out of scope for this formal approach.

It should be noted that a price has to be paid for the universality of our approach – even in simple cases where no alternative representations of a domain are planned, two sets of classes – for domain and presentation are required by the basic technology. To avoid this excessive metamodel complexity for simple notations, a special “identity mapping” – domain and presentation classes coincide – is also provided in GMT.

Conclusions

The described method of diagram definition by mappings from presentation to domain packages is powerful enough to define the complete UML notation, including alternative presentations of interactions as UML sequence and collaboration diagrams. In addition, processes can be presented both as UML activity diagrams and traditional business process notations, such as ARIS eEPC diagrams. The approach has been tested in the GMT environment, yielding a modeling tool of industrial quality, including the efficiency for large-scale models. The alternative notations were really used for business modeling purposes, where different members of a team were provided their favorite notation. The mapping library occurred to be sufficient for diagrams of all reasonable types.

However, the approach is applicable also to a completely different area of modeling – that of MDA [9]. There series of models, typically called PIM (platform independent model) and PSM (platform specific model) are built, in order to provide a model transformation based path from requirements to system code. A typical example of a fragment of such path could be the transition from a UML class diagram for persistent data of a system (in the role of PIM) to SQL-based Data model (or ER model) in the role of PSM. Since not only the forward path is important in practice, but also the reverse one, it is reasonable to consider them as two representations of common domain data. Then classes correspond to tables, associations to relations based on foreign/primary keys and so on, each presentation showing

only the relevant aspects of the common domain. This example can completely be covered by the proposed approach and has been tested within GMT. However, some other MDA applications require true transformations of models at the domain level. We expect that the proposed metamodel mapping principles can be applied in this completely new context too.

References

- [1] Ledeczki A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P. The Generic Modeling Environment, *Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001*.
- [2] DOME Users Guide, <http://www.htc.honeywell.com/dome/support.htm>
- [3] MetaEdit resources, <http://www.metacase.com/papers/index.html>
- [4] Kalnins A., Barzdins J., Celms E., Lace L., Opmanis M., Podnieks K., Zarins A. The First Step Towards Generic Modelling Tool, *Proceedings of Baltic DB&IS 2002, Tallinn, 2002, v. 2*, pp. 167-180.
- [5] Lace L., Celms E., Kalnins A. Diagram definition facilities in a generic modeling tool, *Proceedings of International Conference Modelling and Simulation of Business systems, Vilnius, 2003*, pp. 220-224.
- [6] Akehurst D. H, Kent S. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jezequel, H. Hussmann, S. Cook (Eds.), *Lecture Notes in Computer science, Vol. 2460. Springer, 2002*, pp.243-258.
- [7] Unified Modeling Language: Superstructure (version 2.0), <http://www.omg.org/docs/ptc/03-08-02.pdf>
- [8] Scheer, A.-W. ARIS Business Process Modeling, 3rd edn. Springer-Verlag, Berlin Heidelberg New York (2000).
- [9] MDA Guide Version 1.0.1, <http://www.omg.org/docs/omg/03-06-01.pdf>
- [10] UML 2.0 Diagram Interchange, <http://www.omg.org/docs/ad/03-02-07.pdf>
- [11] Kikusts, P., Rucevskis, P. Layout Algorithms of Graph-like Diagrams for GRADE-Windows Graphical Editors, *Lecture Notes in Computer science, Vol. 1027. Springer-Verlag, 1996*, pp.361-364.

Generic Data Representation by Table in Metamodel Based Modelling Tool

Edgars Celms

Institute of Mathematics and Computer Science,

University of Latvia

Raina bulv. 29, LV-1459, Riga, Latvia

Edgars.Celms@mii.lu.lv

Abstract: The foundation of a metamodel based modelling tool is its flexible facility to declaratively define the modelling method, notation and tool support. One of the problems for such tools is a generic data representation by table. The paper proposes a method for declarative definition of a table based on the logical metamodel. The most interesting aspect of this definition is the specification of element selection criterion. Some practical examples of table specification by the described method are also explained.

Keywords: modelling tool, generic table editor, metamodel, editor definition

1. Introduction

Today there are a lot of modelling tools on the market. Modelling tools are designed to provide all what is necessary to support major areas of modelling, including business process modelling, object-oriented and component modelling with UML[1], relational data modelling, and structured analysis and design, etc. Why it is not sufficient to use “hard-coded” modelling tools? Let us consider, for example, the situation in business modelling. On the one hand there exist several well-known business modelling languages (IDEF3[2], ARIS[3] etc.), each with a set of tools supporting it. But there are also Activity diagrams in UML, whose main role now is to serve business modelling. There is GRADE BM [4] – a specialized language for business modelling and simulation. Thus for the area of business modelling there is no one best or most used language or tool, each of them emphasizes its own aspects. For example, GRADE BM presents very convenient facilities for specifying performers of a task and its triggering conditions. However any new language feature does not come for free, the language becomes more complicated for use. Therefore one universal business modelling language, which would support all wishes, would become

extremely difficult for use in simple cases. UML for this situation offers one ingenious solution – stereotypes for adjusting the modelling language to a specific area. In many cases the idea works perfectly, it is well supported in several tools including GRADE. The latest version of UML - 1.4 extends the notion of stereotype, by assigning tagged values to it and grouping stereotypes into profiles (thus actually extending the metamodel). But currently no tool fully supports it and already new version of UML – 2.0 is coming with significant changes, particularly in the area of Activity diagrams.

The problem with flexible modelling environment is even more urgent for domain-specific modelling, where countless special notations are used for separate domains.

There are probably several ways to solve such problem.

You can develop a modelling tool specially for any specific modelling method but this way can be very time and cost consuming.

You can make a tool as universal as possible to support all needs. For example, there is ARIS tool by IDS prof. Scheer, whose "home notation" is the ARIS BM language. But it supports also the UML notation with Activity diagrams, as well as numerous modifications of the main process notation via eEPC (office processes, industrial processes etc). In general, ARIS tool can be characterised to be extremely "wide", with about 110 different types of diagrams, and frequently having about 100 different symbol types per diagram (most, in fact, are predefined stereotypes of the basic ones). At the same time, there are practically no facilities for defining new stereotypes. In practice such a universal tool is difficult for use in simple cases.

An alternative way is a completely metamodel based generic modelling tool (previously called metaCASE). Such a tool has no built-in modelling methodology. It has to be filled up with specific metamodel and additional information to start modelling something.

2. Metamodel Based Modelling Tool

In this paper some aspects of the metamodel based approach to building flexible modelling environments are explained. The metamodel concept has become

popular in recent years especially due to the principle used in UML definition [1]. What is a metamodel in a metamodel based modelling tool? To put it in short, a metamodel is a class diagram containing all modelling concepts, their attributes and their relationships. A metamodel based tool has no built-in modelling methodology. It has to be filled up with specific metamodel and additional information to start modelling something.

An earlier alternative name for the approach is metaCASE. Let me mention the key research in this area. Perhaps, the first similar approach has been made by Metaedit [5], but for a long time its editor definition facilities have been fairly limited. The latest version named Metaedit+ now can support definition of most used diagram types, but via very restricted metamodelling features (non-graphical), the resulting diagrams corresponding to the simplest concept of labelled directed graph. The most flexible definition facilities (and some time ago, also the most popular in practice, but now the tool is out of market) seem to be offered by the Toolbuilder by Lincoln Software. Being a typical metaCASE of early nineties, the approach is based on ER model for describing the repository contents and on special extensions to ER notation for defining derived data objects which are in one-to-one relation to objects in a graphical diagram. A more academic approach is that proposed by Kogge [6], with a very flexible, but very complicated and mainly procedural editor definition language. Other newer similar approaches are proposed by ISIS GME [7], DOME [8] and Moses [9] projects, with main emphasis for creating environments for domain-specific modelling in the engineering world. The richest editor definition possibilities of them are in GME. Several commercial modelling tools (ARIS by IDS prof. Scheer, System Architect by Popkin Software[10], etc.) use a similar approach internally, for easy customisation of their products, but their tool definition languages have never been made explicit.

The approach explained in this paper is in a sense a further development of the above mentioned approaches. In this approach at first the domain metamodel of the modelling area (logical metamodel) is built. Then the modelling method, notation and tool support are defined declaratively, by means of a special metamodelling environment. These activities require also extension of the metamodel (adding some presentation classes), but it is not relevant to the purposes of this paper.

A very significant part of the tool support in metamodel based modelling tools is flexible data representation facilities. There are some main principles how data can be represented in modelling tools:

- Diagrammatic,
- Hierarchical (e.g. “tree views”),
- Data dictionaries,
- Tables,
- Object editors.

Obviously the most popular data representation form (modelling notation) in any domain now is diagram based. Therefore very important part of the metamodel based approach is the Editor Definition Language (EdDL) for a simple and convenient definition of wide spectrum of diagrammatic graphical editors [11, 12]. Nevertheless not all information is convenient to represent by diagrams. There is also necessity to have a possibility in metamodel based modelling tools to define such facilities as flexible model content browsing and flexible definition of an editor for a single metamodel class instance. Perhaps one of the most undervalued data representation manner in modelling tools today is data representation by table. For example, in Rational Rose [13] UML tool, in real size models it is complicate to browse through packages and classes in the model tree. Frequently a package contains more than ten class diagrams with ten to thirty classes in each. That leads to a situation where there are hundreds of classes at one tree level and it isn't easy to find the right one. It would be reasonable to represent such uniform information not in hierarchical (tree) form but in some easily configurable tabular form. Some tools (System Architect) have something like tables but with a very restricted functionality (reports only). More or less usable in practice tables are implemented in GRADE tool but they are “hard-coded”.

Although all of the data representations principles mentioned here are interesting problems in relation to metamodel based modelling tools, but this is out of scope for this paper to explain all of them. The paper introduces a method for declarative definition of a table, based on the logical metamodel.

Partly the described approach has been developed within the EU ESPRIT project ADDE [14], see [15] for a preliminary report.

3. Generic Data Representation by Table

The foundation of a metamodel based modelling tool is its flexible facility to define declaratively the modelling method, notation and tool support. One of the problems for such tools is a generic data representation by table.

The paper proposes a method for table editor definition based on the logical metamodel. Certainly, the logical metamodel for the selected modelling method should be defined before we start defining any of the editors used for the tool. This logical structure is described by a UML class diagram [1]. Fig.1 shows an example of a logical metamodel for UML class diagram (here the UML class metamodel is described by a UML class diagram). This example will be used in the paper to demonstrate the table editor definition features.

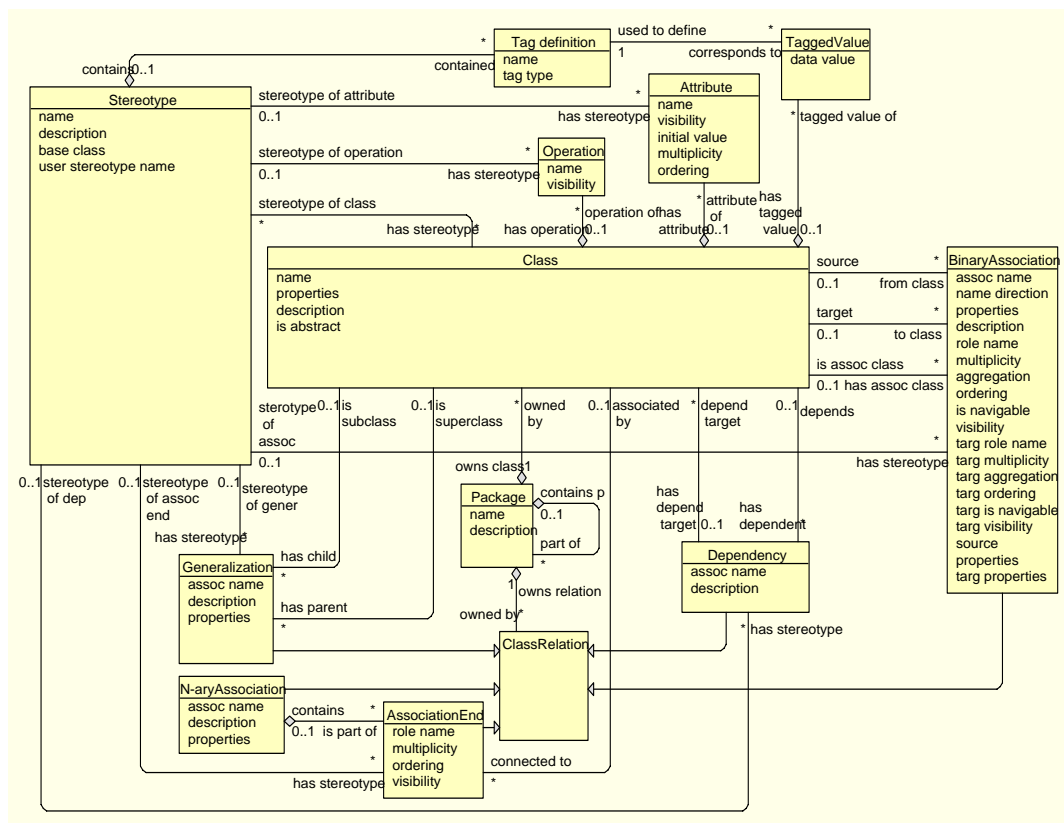


Figure 1. Logical metamodel example (fragment)

A table editor definition for a new table starts with the selection of the domain objects to be represented by the table. It should be emphasized that the logical metamodel itself is never modified during this process. Our goal is to define the corresponding table editor, which in this case should be able to present the selected set of metamodel class instances in a tabular form. For example, Fig.2 shows a table, which represents all instances of *Class* in some model. Classes are grouped by the

association *owned_by* (“Is in Package” column in the table) and ordered by the *Name* attribute.

Table View - Class

Objects - 19 [19]

| Name* | Description | Is in Package* |
|----------------------|--------------|----------------|
| Policy | | Claims |
| Policy_owner_d | | Claims |
| Property claim | this is desc | Claims |
| Witness | | Claims |
| Core entities | | |
| Policy | | Core entities |
| Insurance | | |
| Claim | | Insurance |
| Health claim | | Insurance |
| Insurance company | | Insurance |
| Policy | | Insurance |
| Policy owner | | Insurance |

* Indicates required field.

Sorted by: Is in Package, Name Grouped by: Is in Package

Figure 2. Class table example

Definition of a table consists essentially of two sections: how and what data should be represented. How – it means to define such things as:

- Table columns properties – the user should be allowed to define which columns are visible in the table (based on the metamodel elements). For example, the Class table in Fig.2 is configured to show three columns, which correspond to two attributes (*name* and *description*) and one association (*owned_by*) of the class *Class* in the metamodel. Other attributes and associations of the *Class* are not visible in this table. The user should be also able to define for each column in the table such properties as column title, column edit mode (either it is allowed to do in-place editing or some “native” object editor should be invoked), column ordering and grouping (by which column(s) table is ordered or grouped), etc.
- Prompting/display form for associations – it is a very important feature to make table more usable for end users. For example, the association column *owned_by* for the table in Fig.2 is defined to be shown as the value of the attribute *name* of *Package*. In other words, it is the facility to define the text

assembly rules for columns, which correspond to associations in the metamodel. The user can define the text to display as a concatenation of segments, each consisting of a constant prefix, a variable part and a constant suffix. Each variable part is defined as the object attribute value located at the end of the chain defined by associations (for example, *Class.owned_by.name*). Association chain may be empty, or it must start at the metamodel class corresponding to the objects in the table. Attribute must belong to the metamodel class at the end object of the associations chain.

- Navigation facilities for objects, attributes and associations – what to do in response to user activities. For example, what to do on single or double mouse click on some object attribute or association.
- Popup menu configuration – when defining popup menus for table objects the user should be allowed to define menu items at least for the following actions: *create* a new object (for example, “New Class”), *delete* the object represented in the table, *open* an object viewer/editor (“Properties...”), *copy* and *paste*, *execute* any other program, etc.

The above described facilities for table definition allow the user to define nearly any reasonable table form for practical use.

The second part of the definition of the table is used to specify what data should be represented by the table. The specification of the element selection criterion (selection rule) is the most non-trivial and the most interesting aspect of the table definition. A selection rule can be specified, saying which of the possible instances actually must appear in the table. To be short, a selection rule is a Boolean expression (AND, OR, NOT) on link conditions (specifying that a certain sequence of metamodel associations must link the given object to another object) and attribute value conditions (defined by an association chain, attribute at its end and a fixed attribute value). The actual expression language contains some more details (including simple existential quantifiers).

Examine one more example of the table. Let’s assume that we need to define a table, which contains *Stereotypes* of *Classes*, which are contained in one concrete *Package* and these *Stereotypes* don’t have *Tag_definition* (see Fig.1). There are languages where it is possible to define selections or assertions of such kind. Such languages, for example, are Object Query Language (OQL[16]), which is used for querying an Object Database or Object Constraint Language (OCL[1]), which is used

in the UML to specify the well-formedness rules of the metaclasses comprising the UML metamodel. In OQL and OCL the above mentioned selection criterion can be written as (where the concrete *Package* is given by the parameter *curr_package*):

In OQL: select st from *Stereotype* st

where *curr_package* in *st.stereotype_of_class.owned_by*
and then is_undefined(*st.contains*)

In OCL: *self.stereotype_of_class.owned_by->exists(pck:Package |
pck = curr_package)* and *self.contains->IsEmpty()*

However in practice both these languages are difficult to use for declarative definition of the tables. OQL is a very powerful and extensive language. OCL is a very concise and rather complicated to use language and it is not intended for querying but for specifying static constraints in a metamodel. In order to use OQL or OCL you need to know precise semantics of the languages.

Let's say, that the paper proposes a method to specify selection rule for table, which is a "reasonable subset" of OQL or OCL with a graphical realization (see Fig.3). "Reasonable subset" in this case means – such a subset, which is sufficient for practical use and allows also an efficient implementation.

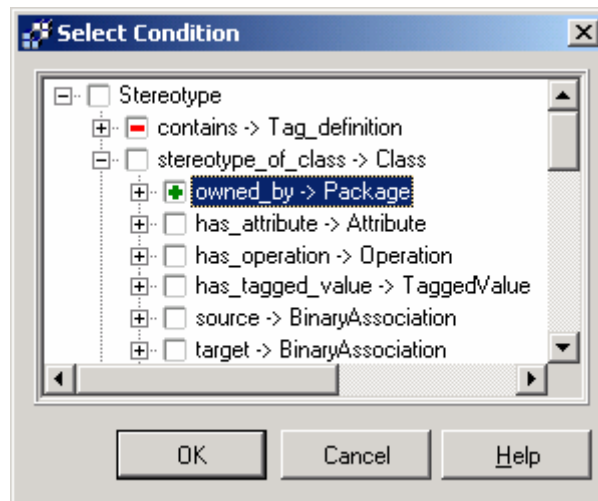


Figure 3. Selection criterion definition example

The definition of the selection rule is done by tree. The tree is some view (fragment) of the metamodel where the root node corresponds to the metamodel class for which we want to define the table (in our example the *Stereotype* class). The every next level of the tree contains nodes (classes), which are linked in the metamodel with associations to the parent node (class) in tree. Fig.3 gives an example of such a tree

where the above mentioned selection criterion is defined. The tree row marked by the red "minus" sign means that there might not be a link to a *Tag_definition* instance.

In other words, according to this approach it is possible in an easy, usable manner to compose textual selection expression, which is based on metamodel elements. We can also specify some additional constraints for attribute values for classes.

It should be emphasized, that unlike to OQL or OCL in the proposed method it is not important to “know” such technical issues as cardinalities of associations or kind of collections (bag, set, list in OQL or bag, set, sequence in OCL). Therefore the proposed method is easy to use in practice. The practical experiments have demonstrated that the described metamodel based table definition method has a realistic implementation and can reach an industrial quality of the defined editors.

4. Conclusions

The approach to table definition described in this paper permits to define nearly any reasonable table for practical use and the described method allows also an efficient implementation (in the sense of run-time necessary to build the tables from real amounts of data). The practical experiments have demonstrated that the table editors obtained by the described definition method can reach an industrial quality. However this approach can be made significantly more universal and applicable not only to tables but also for other similar data representations, since it is generally accepted that a metamodel (class diagram) can be used for describing the logical structure of nearly any system.

References

- [1] Rumbaugh, J., Booch, G., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)
- [2] Mayer, R., Menzel, C., Painter, M.: IDEF3 Process Description Capture Method Report http://www.idef.com/Downloads/pdf/Idef3_fn.pdf, Knowledge Based Systems Inc (1995)
- [3] Scheer, A.-W.: ARIS Business Process Modeling. 3rd edn. Springer-Verlag, Berlin Heidelberg New York (2000)
- [4] Barzdins, J., Tenteris, J., Vilums, E.: Business Modeling Language GRAPES-BM (Version 4.0) and its Application. Dati, Riga (1998)

- [5] Smolander, K., Martiin, P., Lyytinen, K., Tahvanainen, V-P.: Metaedit – a flexible graphical environment for methodology modelling. Springer-Verlag (1991)
- [6] Ebert, J., Sutenbach, R., Uhe, I.: Meta-CASE in Practice: a Case for KOGGE. Proceedings of the 9th International Conference, CAiSE'97, Barcelona, Catalonia, Spain (1997)
- [7] Ledeczki A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary, May 17, 2001 (available in PDF at <http://www.isis.vanderbilt.edu>)
- [8] DOME Users Guide. <http://www.htc.honeywell.com/dome/support.htm>
- [9] Esser, R.: The Moses Project.
<http://www.tik.ee.ethz.ch/~moses/MiniConference2000/pdf/Overview.PDF>
- [10] Popkin Software: System Architect. http://www.popkin.com/products/system_architect.htm
- [11] Audris Kalnins, Janis Barzdins, Edgars Celms, Lelde Lace, Martins Opmanis, Karlis Podnieks, Andris Zarins.: The First Step Towards Generic Modelling Tool. Proceedings of Baltic DB&IS 2002, Tallinn, 2002, v. 2, pp. 167-180.
- [12] Audris Kalnins, Karlis Podnieks, Andris Zarins, Edgars Celms, Janis Barzdins.: Editor Definition Language and its Implementation. Proceedings of the 4th International Conference “Perspectives of System Informatics”, Novosibirsk, 2001, LNCS, v.2244, pp. 530 – 537
- [13] Quatrani, T.: Visual Modeling with Rational Rose 2002 and UML. 3rd edn. Addison Wesley Professional, Boston (2002)
- [14] ESPRIT project ADDE. <http://www.fast.de/ADDE>
- [15] Sarkans, U., Barzdins, J., Kalnins, A., Podnieks, K.: Towards a Metamodel-Based Universal Graphical Editor. Proceedings of the Third International Baltic Workshop DB&IS, Vol. 1. University of Latvia, Riga, (1998) 187-197
- [16] Cattel, R.G.G., Barry, D.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann (2000)

Model Transformation Language MOLA

Audris Kalnins, Janis Barzdins, Edgars Celms

University of Latvia, IMCS, 29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Janis.Barzdins, Edgars.Celms}@mii.lu.lv

Abstract. The paper describes a new graphical model transformation language MOLA. The basic idea of MOLA is to merge traditional structured programming as a control structure with pattern-based transformation rules. The key language element is a graphical loop concept. The main goal of MOLA is to describe model transformations in a natural and easy readable way.

1 Introduction

The Model Driven Architecture (MDA) initiative treats models as proper artifacts during software development process and model-to-model transformations as a proper part of this process. Therefore there is a growing need for model transformation languages and tools that would be highly acceptable by users. Though model transformations would be built by a relatively small community of advanced users, the prerequisite for broad acceptance of transformations by system developers is their easy readability and customizability.

Model transformation languages to a great degree are a new type of languages when compared to design and programming languages. The only sound assumption here is that all models in the MDA process (either UML-based models or other) should be based on metamodels conforming to MOF 2.0 standards.

The need for standardization in the area of model transformation languages led to the MOF 2.0 Query/Views/Transformations (QVT) request for Proposals (RFP)[1] from OMG.

To a great degree the success of the MDA initiative and of QVT in particular will depend on the availability of a concrete syntax for model-to-model transformations that is able to express non-trivial transformations in a clear and compact format that would be useful for industrial production of business software [2].

QVT submissions by several consortiums have been made [3, 4, 5], but all of them are far from a final version of a model transformation language. Currently the proposal most likely to be accepted seems [3] – actually a merge of several initial proposals. Several serious proposals for transformation languages have been provided outside the OMG activities. The most interesting and complete of them seem to be UMLX [6] and GReAT [7]. Some interesting transformation language proposals use only textual syntax, e.g., [15].

According to our view, and many others [2], model transformations should be defined graphically, but should combine the graphical form with text where appropriate. Graphical forms of transformations have the advantage of being able to represent

mappings between patterns in source and target models in a direct way. This is the motivation behind visual languages such as UMLX, GReAT and the others proposed in the QVT submissions. Unfortunately, the currently proposed visual notations make it quite difficult to understand a transformation.

The common setting for all transformation languages is such that the model to be transformed (**source model**) is supplied as a set of class and association instances conforming to the **source metamodel**. The result of transformation is the **target model** - the set of instances conforming to the **target metamodel**. Therefore the transformation has to operate on instance sets specified by a class diagram (actually, the subset of class notation, which is supported by MOF).

Approaches that use graphical notation of model transformations draw on the theoretical work on graph transformations. Hence it follows that most of these transformation languages define transformations as sets of related rules. Each rule has a pattern and action part, where the pattern has to be found (matched) in the existing instance set and the actions specify the modifications related to the matched subset of instances. This schema is used in all of the abovementioned graphical transformation languages. Languages really differ in the strength of pattern definition mechanisms and control structures governing the execution order of rules [8].

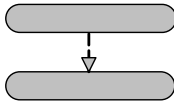
It should be mentioned that an early pioneer in the area (well before the MDA era) is the PROGRES language [9]. This semi-graphical language offered pattern-based graph rewrite rules applicable to “models” described by schemas (actually, metamodels). The execution of rules is governed by the traditional structured control constructs – sequence, branch and loop, though in the form of Dijkstra’s guarded commands.

The current MDA-related graphical transformation languages – UMLX and GReAT use relatively sophisticated pattern definition mechanisms with cardinality specifications (slightly more elaborated in GReAT). The control structure in UMLX is completely based on recursive invocations of rules. The control structure of GReAT is based on hierarchical dataflow-like diagrams, where the only missing control structure is an explicit notation for loops (loops are hidden in patterns). The proposal [3] also offers elaborated patterns, which are combined with a good support for recursive control structures. Since the PROGRES project is now inactive, there currently is no transformation language based on traditional control structures.

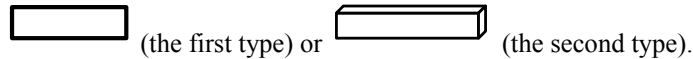
This paper proposes a new transformation language **MOLA (MOdel transformation Language)**. The prime goal of MOLA is to provide an easy readable graphical transformation language by combining traditional structured programming in a graphical form (a sort of “structured flowcharts”) with rules based on relatively simple patterns. This goal is achieved by introducing a natural graphical loop concept, augmented by an explicit loop variable. The loop elements can easily be combined with rule patterns. Other structured control elements are introduced in a similar way. In the result, most of typical model transformation algorithms, which are inherently more iterative than recursive, can be specified in a natural way. The paper demonstrates this on the traditional MDA class-to-database transformation example and on the statechart flattening example – an especially convincing one. Some extensions of MOLA are also sketched.

2 Basic constructs of MOLA

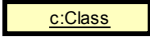
This section presents a brief overview of basic constructs of MOLA. The MOLA language is a procedural graphical language, with control structures taken from traditional structured programming. The elements specific to model transformations can easily be combined with traditional language elements such as assignment statements. A **program** in MOLA is sequence of graphical statements, linked by dashed arrows:



A **statement** can be an assignment or a rule – an elementary instance transformation statement, however the most used statement type in MOLA is a loop. There are two types of **loops**, which will be depicted in the following way:




A loop body always contains one or more sequences of graphical statements. Each body sequence starts with a **loop head** statement declaring the **loop variable** for this sequence. In MOLA the loop variable represents an instance of the given class. In order to distinguish it from other class instances defining its context, the loop variable

is shown with a **bold** frame: . The loop head statement, besides the loop variable, typically contains also instance selection conditions, which constrain the environment of a valid loop variable instance. The UML object (instance specification) notation is used both for the loop variable and its environment description – it expresses the fact that any valid instance from the instance set of the given class in the source model must be used as a loop variable value during the loop execution.

The semantics of both types of loops differ in the following way. A type one loop is executed once for each valid instance from the instance set – but the instance set itself may be modified (extended) during the loop execution. The type two loop continues execution while there is at least one valid variable instance in the instance set (consequently, the same loop variable instance may be processed several times). In an analogy to some existing set and list processing languages, it is natural to call type one loops **FOREACH** loops and type two loops - **WHILE** loops in MOLA.

Another important statement type is the **rule** – the specification of an elementary instance transformation. A rule contains the pattern specification – a set of elements representing class instances and association instances (links), built in accordance with the metamodel. In addition, the rule contains the action specification – what new class instances are to be built, what associations (links) drawn, what instances are to be deleted, what attributes are to be assigned value etc. Its semantics is obvious – locate a pattern instance in the source model and perform the specified actions. When a rule has to be applied – it is determined by the loop whose body contains the rule. A rule can be combined with the loop head – a loop head can also contain actions, which are performed for each valid loop variable instance.

All MOLA statements, except loops, are graphically enclosed in grey rounded rectangles: .

Further, more precise definitions of MOLA syntax and semantics will be given on toy examples.

Let us assume that a toy metamodel visible in Fig.1 is used.

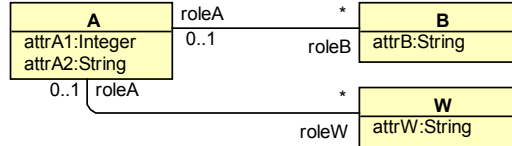


Fig.1. Metamodel for the toy example

Then a MOLA program, which sets the attribute *attrA1* to 1 for those instances of the class *A* that are linked to at least one instance of class *B*, is shown in Fig. 2. The loop (FOREACH type) contains two statements – the loop head and a trivial rule which sets the value of attribute *attrA1* in the loop variable. First, some comments on the loop head statement. The selection condition consisting of an instance of *B* linked by the only available association (*roleB*) to the loop variable (*a:A*) requires that at least one such instance of *B* must exist for a given instance of *A* to be a valid loop variable instance. We want to emphasize that an **association** with no constraints attached in the loop head (or in a rule pattern) always means – there **exists at least one instance** (link) of such an association. The loop head in MOLA is also a kind of pattern.

The second statement in the loop **references** the same instance of *A* – the loop variable, this is shown by prefixing the instance name by the @ character.

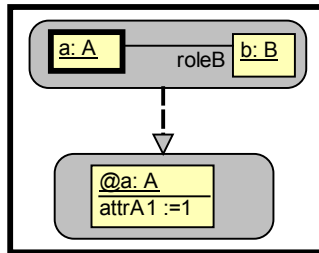


Fig.2. Program finding *A*'s linked to a *B*

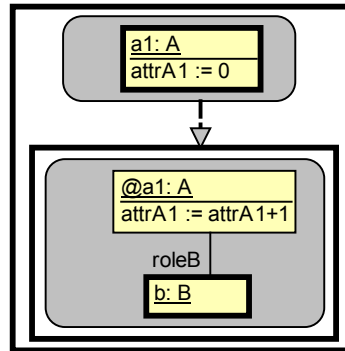


Fig.3. Program counting *B*'s linked to an *A*

The second program example (in Fig.3) finds how many instances of *B* are linked to each instance of *A*.

This example demonstrates a natural use of nested loops. The outer loop (with the loop variable *a:A*) is executed for every instance of *A*. The loop head sets also the initial value of the attribute *attrA1*. The nested loop, which is executed for those instances of *B* which are linked to the current *A*, performs the counting.

The next more complicated task is to build an instance of *W* for each *B* which is linked to an *A*, link it by an association (*roleW*) to the *A* and assign to its string parameter (*attrW*) the concatenation of string parameters in the corresponding instances of *A* and *B*. Fig. 4 shows the corresponding MOLA program.

The same nested loops as in the previous example are used. But here the inner loop head is also a rule with more complicated action – building an instance of W , linking it to the current loop variable instance of the outer loop and setting the required value of $attrW$.

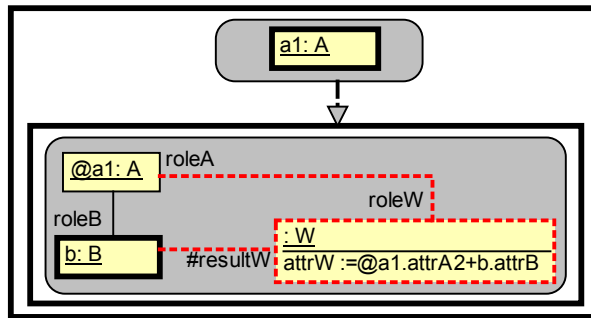


Fig.4. Program building W for each B

The new elements – instances and links are shown with **dotted** lines (and in red color) in MOLA. The expression for $attrW$ references the attribute values from other instances – they are qualified by the corresponding instance names. The association linking the instance of W to the instance of B is a special one – it is the so called **mapping association** (which actually should also be specified in the metamodel). Mapping associations are typically used in MDA-related transformations for setting the context of next subordinate transformations and for tracing instances between models (therefore they normally link elements from different metamodels). Role names of mapping associations are prefixed by the # character in MOLA.

Two more MOLA constructs should be explained here. The first one is the **NOT** constraint on associations in patterns – both in loop heads and ordinary rules. It expresses the negation of the condition specified by the association – there must be no instance with specified properties linked by the given link. Fig. 5 shows an example where an instance of W is built for those A which have no B attached.



Fig.5. NOT constraint

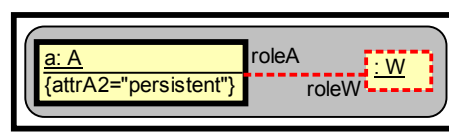


Fig.6. Attribute constraint

Another one is attribute constraints. Fig. 6 shows an example where an instance of W is built for those A where the attribute $attrA2$ has the specified value. The Boolean expression in braces in general uses OCL syntax (in addition, it may contain also explicit qualified references to other instances in the pattern).

There are some more elements of MOLA which are not used in the examples of this paper and therefore will not be explained in detail. Besides the attributes in the source metamodel, instances may have “temporary” computed attributes which can be used as variables for storing values during the computation. These temporary attributes are also defined in the metamodel. Similarly, there may be temporary associa-

tions. There is also one more control structure – an equivalent of the **if-then-else** (or case) statement. There is also a **subprogram** concept in MOLA and the subprogram **call** statement, where the parameters can be references to instances used in the calling program (typically, to loop variables) or simple values. The called subprogram has access to the source model and can add or modify elements in the target model.

3 UML Class Model to Relational Model Example in MOLA

Further description of MOLA will be given on the basis of the “standard benchmark example” for model transformation languages – the UML class model to relational database model transformation example. This example has been used for most of model transformation language proposals (see e.g., [3, 4, 6, 10]). However, no two papers use exactly the same specification of the example. Here we have chosen the version used by A. Kleppe and J. Warmer in their MDA book [10].

The source is a simplified class diagram built according to the metamodel in Fig. 7 (it is a small subset of the actual UML metamodel). Any class which is present in the source model has to be transformed into a database table. Any class attribute has to be converted into a table column. Attribute types are assumed to be simple data types – the problem of “flattening” the class-typed attributes is not considered in this version. We assume here that type names in class diagram and SQL coincide (in reality it is not exactly so!).

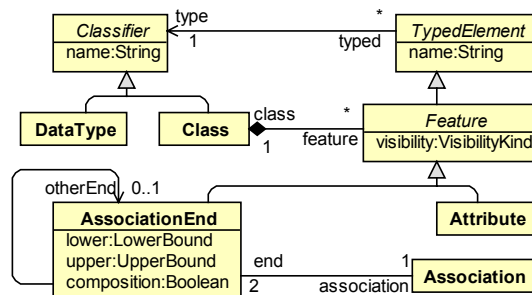


Fig.7. Simplified class metamodel

Each converted table has an “artificial” primary key column with the type *integer*. The treatment of associations is quite realistic. One-one or one-to-many associations result into a foreign key and a column for it in the appropriate table (for one-one – at both ends). A many-to-many association is converted into a special table consisting only of foreign key columns (and having no primary key). Each foreign key references the corresponding primary key.

We should remind that according to UML semantics, in the metamodel the *type* association from an *Association End* leads the *Class* at that end, but *class* association – to *Class* at the opposite end.

The resulting database description must correspond to a simplified SQL metamodel given in Fig. 8.

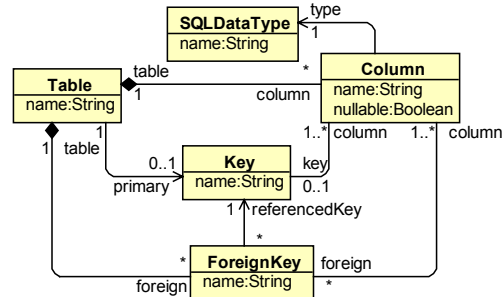


Fig.8. Simplified relational database metamodel

The metamodels and transformation specification are exactly as in [10] except that some inconsistencies and elements unused in the given task are removed.

More formally, in MOLA the source and target metamodels (Fig. 7 and 8) are combined into one common metamodel, where mapping associations can also be specified. We do not present this combined metamodel here, role names of mapping associations can be deduced from MOLA diagrams (Fig. 9 and 10).

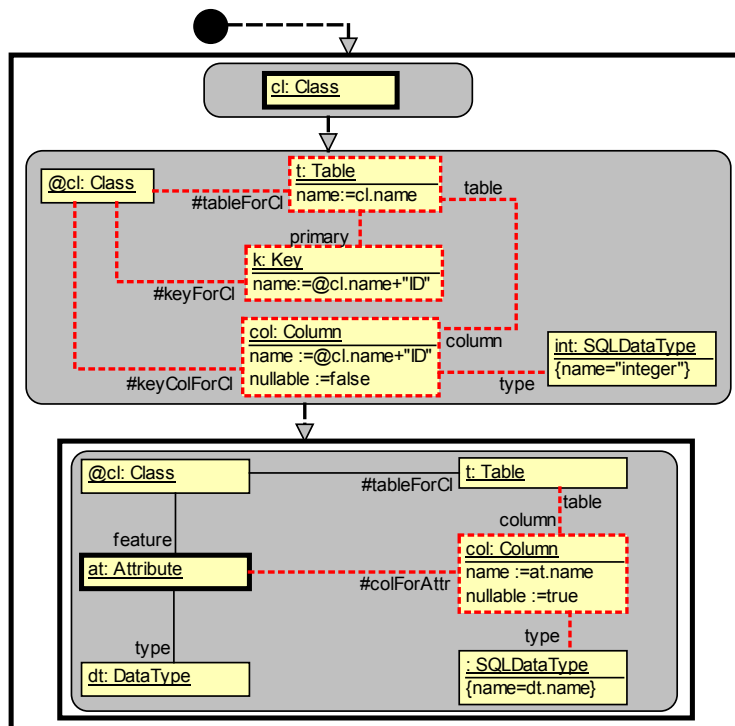


Fig.9. Class to database transformation (part 1)

Fig. 9 and 10 show the complete transformation program in MOLA. The part 1 (Fig. 9) implements the required class-to-table transformations, but the part 2 – the transformation of associations into foreign keys and appropriate columns.

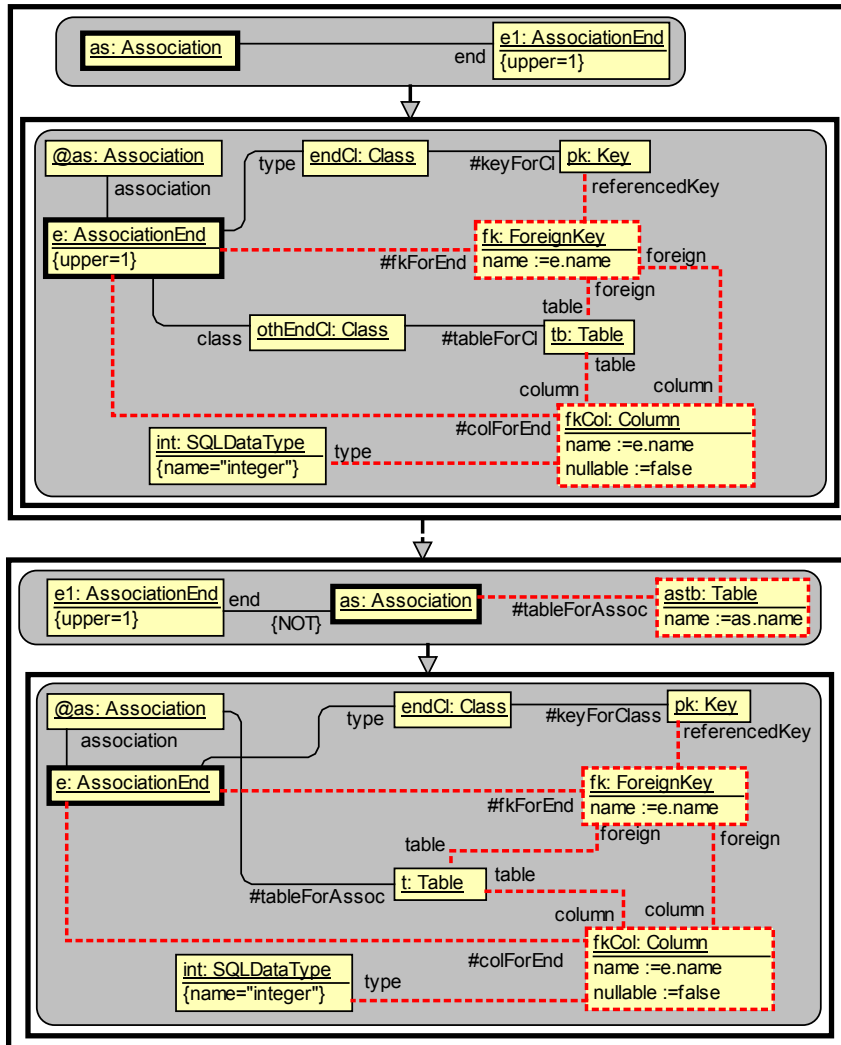


Fig.10. Class to database transformation (part 2)

A complete program in MOLA starts with the UML start symbol and ends with end symbol. In between there are statements connected by arrows; in the given program – three top-level loops (one for class instances and two for associations). All loops are of FOREACH type.

Now some more detailed comments for this program. The first loop is executed once for each class in the source set and during each loop execution the corresponding database elements – the table, the primary key and the column for it are built. The

mapping association *#tableForCl* is used in the condition for the inner loop – to ensure that the correct *Table* instance is taken. This loop is executed once for each attribute and builds a column for each one. Here it is assumed that SQL data types (as instances of the corresponding class) are pre-built and the appropriate one can always be selected.

The second and third loops in totality are executed for each association instance – the second loop for those instances that have multiplicity 0..1 or 1..1 at least at one end and the third one for those which are many-to-many. This is achieved by adding mutually exclusive selection conditions to both loop variable definitions. These conditions are given in a graphical form. The first one uses the already mentioned in section 2 fact that an association in a condition (pattern) requires the existence of the given instance. The other condition uses the {NOT} constraint attached to the association – no such instance can exist. Then both loops have an inner loop - for both ends (even in the first case there may be two “one-ends”). Both inner loops use mapping associations built by previous rules (*#keyForCl*, *#tableForCl*) in their conditions. The type for “foreign columns” is *integer* – as well as that for “primary columns”.

An alternative form of control structure for processing associations could be one loop with an if-then-else statement in the body (Fig. 11).

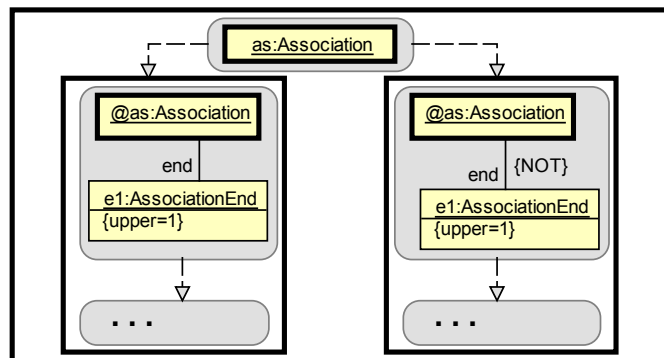


Fig.11. Loop with an if-then-else statement

One more alternative representation could be to make the Fig. 10 a transformation of its own (e.g., *TransformAssociations*) and add the call statement *TransformAssociations* (this time without parameters) to the bottom of Fig. 9. In our case there is no great need in this since the whole transformation example actually fits in one A4 page. However, the subprogram mechanism in MOLA permits to define arbitrarily complicated transformations by well-proven methods of structural programming.

4 Statechart Flattening Example

This section presents another example – the flattening of a UML statechart. This example was first used in [7] to demonstrate the GReAT transformation language. Due to space limits, we use a version where the statechart can contain only composite states with one region (OR-states in terms of [7]). Composite states may contain any

type of states, with an arbitrary nesting level. Such a statechart must be transformed into an equivalent “flat” statechart (which contains only simple states). The informal flattening algorithm is well known (most probably, formulated by D. Harel [11]). A version of this example with much simplified problem statement is present also in [3].

The simplified metamodel of the “full” (hierarchical) statechart is depicted in Fig. 12. There are some constraints to the metamodel specifying what is a valid statechart. There are “normal” transitions for which the event name is nonempty and “special” ones with empty event. These empty transitions have a special role for state structuring. Each composite state must contain exactly one initial state (an instance of *Init*) and may have several final states. There must be exactly one empty transition from the initial state of a composite state (leading to the “default” internal state). The same way, there must be exactly one empty transition from the composite state itself - the default exit. This exit is used when a contained final state is reached. Otherwise, transitions may freely cross composite state boundaries and all other transitions must be named. Named transitions from a composite state have a special meaning (the “interrupting” events), they actually mean an equally named transition from any contained “normal” state – not initial or final. This is the most used semantics of composite states (there are also some variations).

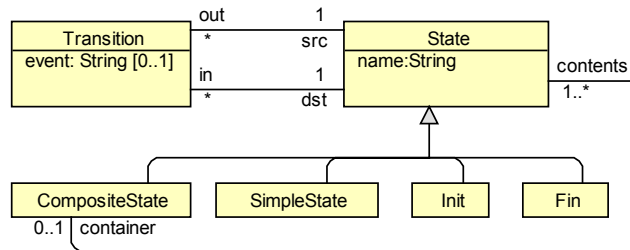


Fig.12. Metamodel of hierarchical statechart

All states have names – but those for initial and final states actually are not used. Names are unique only within a composite state (it acts as a namespace) and at the top level.

The traditional flattening algorithm is formulated in a recursive way. Take a top-most composite state (i.e., one not contained in another composite state). There are three ways how transitions related to this state must be modified:

1. Transitions entering the composite state itself must be redirected to the state to which the empty transition from its initial state leads.
2. Transitions leading to a final state of this composite state must be redirected to the state to which the empty transition from the composite state leads.
3. Named transitions from the composite state must be converted into a set of equally named transitions from all its “normal” states (with the same destination)

Then the name of the composite state must be prefixed to all its contained normal states and the composite state must be removed (together with its initial and final states and involved empty transitions). All this must be repeated until only simple states (and top level initial/final ones) remain.

A simple analysis of this algorithm shows that the redirection of transitions may be done independently of the composite state removal – you can apply the three redirection rules until all transitions start/end at simple states (or top initial/final). The set of simple states is not modified during the process – only their names are modified.

Namely this modified algorithm is implemented in the MOLA program in Fig. 13. It contains two top-level loops – the first one performs the transition redirection and the second – the removal of composite states.

Both top-level loops are WHILE-type – especially, in the first loop a transition may be processed several times until its source and destination states reach their final position. A closer analysis shows that the second loop actually could be of FOREACH type, but the original algorithm suggests WHILE.

The program performs a model update – source and target metamodels coincide, simply, some metaclasses cannot have instances in the target model. Mapping associations are not used in this example.

The first loop contains three loop head statements – all specify the instance *t:Transition* as a loop variable, but with different selection conditions. According to the semantics of MOLA, any *Transition* instance satisfying one of the conditions (one at a time!) is taken and the corresponding rule is applied (note that the conditions are not mutually exclusive). All this is performed until none of the conditions applies – then all transitions have their final positions. The first two rules contain a dashed line – the association (link) removal symbol. The link is used in the selection condition, but then removed by the rule. The third path through the loop contains the instance removal symbol.

Namely the use of **several loop heads per loop** is a strength of MOLA – this way inherently recursive algorithms can be implemented by loops.

The second loop – the removal of composite states also has a recursive nature to a certain degree – it implements the so-called **transitive closure** with respect to finding the deepest constituents (simple states) and computing their names accordingly to the path of descent.

It shows that transitive closure can be implemented in MOLA in a natural way (even the FOREACH loop could be used for this). The other constructs in this loop are “traditional” – except, may be, the fact that several instances may be deleted by a rule in MOLA.

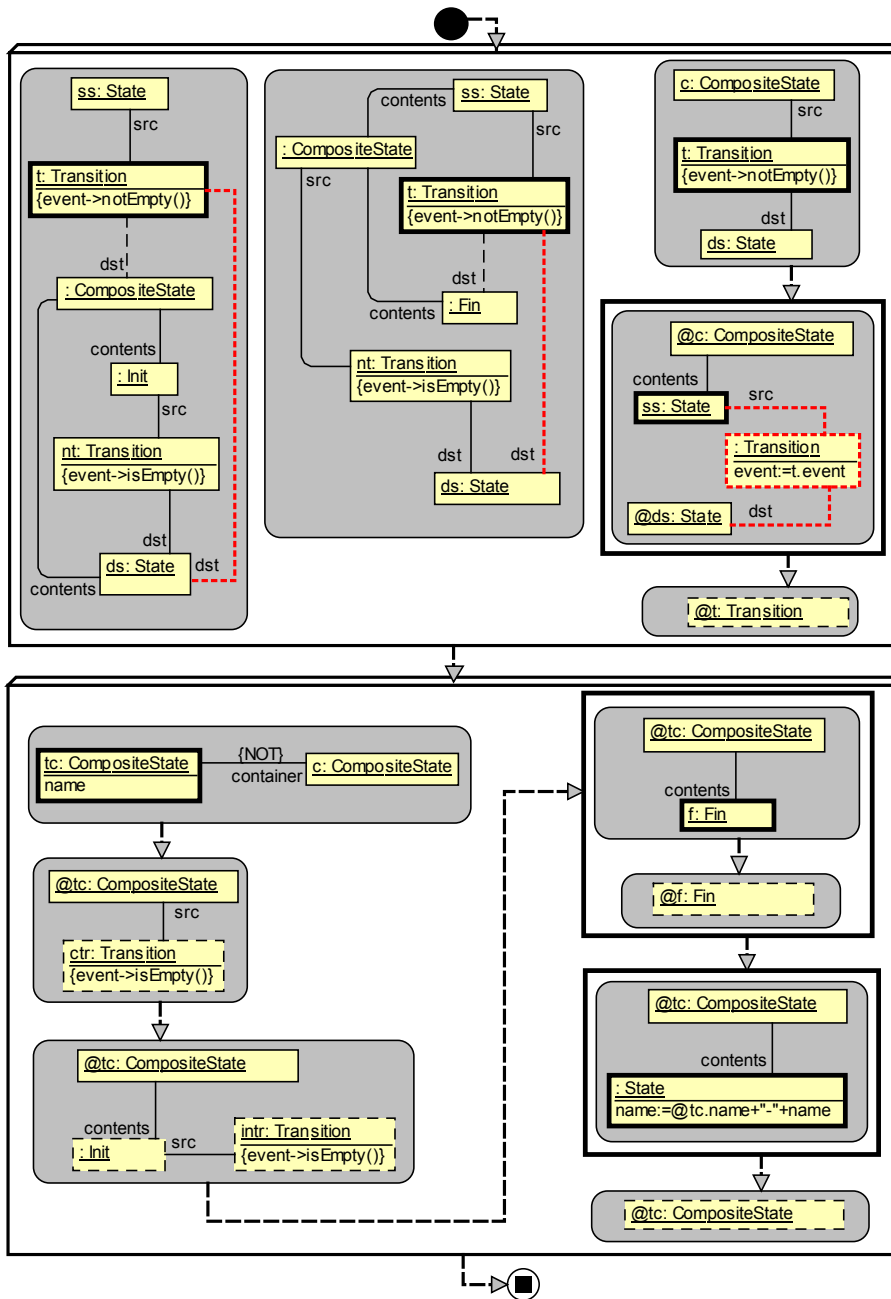


Fig.13. Statechart flattening

5 Extended Patterns in MOLA

The rule in the previous example for computing the name of a state contained in a composite state to be removed actually is the simplest case of a typical transformation paradigm – the transitive closure. Experiments show that transitive closure in all cases can be implemented in MOLA. However, not always it is so straightforward as in Fig. 13, sometimes temporary associations and attributes and nested loops are required for this task. A typical example is the class to database transformation as specified in [3, 6], where the “flattening” of class-typed attributes must be performed – if the type of an attribute is a class, the attributes of this class must be processed and so on. If an attribute with a primitive data type is found in this process, a column with this type is added to the table corresponding to the original (“root”) class. The name of the column is the concatenation of all attribute names along the path from the root class to the attribute. It is easy to see that all such paths must be traversed.

Since the transitive closure is a typical paradigm in MDA-related tasks, an extension of MOLA has been developed for a natural description of this and similar tasks. This extension uses a more powerful – the looping pattern, by which computation of any transitive closure can be implemented in one rule. This feature has been described in details in [12], here we present only the above-mentioned example with some comments.

Fig.14 shows one statement in extended MOLA which is both a FOREACH loop over *Class* instances and a rule with an extended pattern. In contrast to patterns in basic MOLA, this pattern matches to unlimited number of instances in the source model. Most of the associations in this pattern are directed (using the UML navigability mark). The semantics of this pattern is best to be understood in a procedural way. Starting from a valid instance of loop variable (selected by the undirected part of the pattern – one association), a temporary instance tree is being built, following the directed associations.

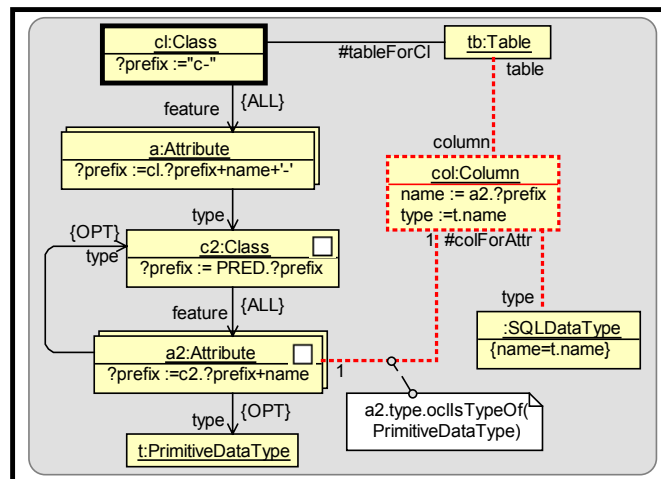


Fig.14. Transitive closure by extended pattern

Associations in this pattern use two new qualifiers – **ALL** and **OPT**. The first one says the instance tree has to contain all possible valid links of this kind (a fan-out occurs), but the second one – that the link is not mandatory for the source instance to be included in the tree (an association without qualifier is mandatory in MOLA). The white square icons in *c2* and *a2* specify that for these pattern elements instance copies are built in the tree (but not the original source model instances used) – it is easy to see that in order to obtain all paths from the root class to primitively-typed attributes namely such copying is required. Another new pattern syntax element is the UML multiobject notation for some elements – to emphasize that a fan-out occurs at these places during the pattern match. The looping part of the pattern – the elements *c2* and *a2* actually are traversed as many times during the matching (tree building) process as there are valid candidates in the source model. The rule uses the temporary attribute *?prefix* (with the type *String*), whose scope is only this rule. The values of this attribute are computed during the building of the match tree (for each of its node) – it is easy to see that the expressions follow the building process (the special PRED qualifier means any predecessor). For this extended pattern the building action also generates many instances of *Column* – one for each instance of *a2* in the tree (it is a copy!) which satisfies the building condition in OCL.

Extended patterns have more applications, however their strength most clearly appears on complicated transitive closures like the one in Fig. 14.

6 Conclusions

MOLA has been tested on most of MDA-related examples – besides the ones in the paper, the class to Enterprise Java transformation from [10], the complete UML state-chart flattening, business process to BPEL transformation and others. In all cases, a natural representation of the informal algorithms has been achieved, using mainly the MOLA loop feature. This provides convincing arguments for a practical functional completeness of the language for various model to model transformations in MDA area. Though it depends on readers' mindset, the “structured flowchart” style in MOLA seems to be more readable and also frequently more compact than the pure recursive style used e.g., in [6]. Though recursive calls are supported in MOLA, this is not the intended style in this language. For some more complicated transformation steps the extended MOLA patterns briefly sketched in section 5 fit in well.

The implementation of MOLA in a model transformation tool also seems not to be difficult. The patterns in basic MOLA are quite simple and don't require sophisticated matching algorithms. Due to the structured procedural style the implementation is expected to be quite efficient. All this makes MOLA a good candidate for practically usable model transformation language.

Initial experiments with MOLA have been performed by means of the modeling tool GRADE [13, 14], in the development of which authors have participated. A separate MOLA tool is currently in development. A graphical editor for MOLA has already been developed, the pictures for this paper have been obtained by this editor. A MOLA execution system is also close to completion.

References

1. OMG: Request For Proposal: MOF 2.0/QVT. OMG Document ad/2002-04-10, <http://www.omg.org/cgi-bin/doc?ad/2002-04-10>
2. Bettin J. Ideas for a Concrete Visual Syntax for Model-to-Model Transformations. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, USA, October 2003.
3. QVT-Merge Group. MOF 2.0 Query/Views/Transformations RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, <http://www.omg.org/cgi-bin/doc?ad/2004-04-01>
4. Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, <http://www.omg.org/cgi-bin/doc?ad/2003-08-07>
5. Interactive Objects Software GmbH, Project Technology, Inc. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-11, <http://www.omg.org/cgi-bin/doc?ad/2003-08-11>
6. 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.
7. Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, November 2003.
8. Czarnecki K., Helsen S. Classification of Model Transformation Approaches. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, USA, October 2003.
9. Bardohl R., Minas M., Schürr A., Taentzer G.: Application of Graph Transformation to Visual Languages. G. Rozenberg (ed.): Handbook on Graph Grammars: Applications, Vol. 2, Singapore, World Scientific, 1998.
10. Kleppe A., Warmer J., Bast W. MDA Explained. The model driven architecture: practice and promise. Addison-Wesley, 2003.
11. Harel D. Statecharts: a Visual Formalism for Complex Systems. Sci. Comput. Program. Vol 8, pp. 231-274, 1987.
12. Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA: Extended Patterns. To be published in proceedings of Baltic DB&IS 2004, Riga, Latvia, June 2004.
13. Kalnins A., Barzdins J., et al. Business Modeling Language GRAPES-BM and Related CASE Tools. Proceedings of Baltic DB&IS'96, Institute of Cybernetics, Tallinn, 1996.
14. GRADE tools. <http://www.gradetools.com>
15. Bézivin J., Dupé G., Jouault F., et al. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. 2nd OOPSLA Workshop on Generative Techniques in Context of MDA, Anaheim, California, 2003.

Basics of Model Transformation Language MOLA

Audris Kalnins, Janis Barzdins, Edgars Celms

University of Latvia, IMCS, 29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Janis.Barzdins, Edgars.Celms}@mii.lu.lv

Abstract. The paper offers basic elements of a new graphical model transformation language MOLA. The language combines the traditional structured programming with pattern-based transformation rules, the key element being a natural loop construct. The prime goal of MOLA is to provide a natural and highly readable representation of model transformation algorithms.

1 Introduction

The success of model driven development to a great degree depends on the availability of appropriate languages and tools for model transformations. It is quite improbable that model driven development could be reduced to one step model-to-code techniques. More likely, in most domains a sequence of models will be required between which there will be many transformations, which at least should be partially automated.

Therefore practical model-to-model transformation languages are of prime importance. Currently in this area there are responses to OMG QVT RFP[1] by several consortiums [2,3,4] and some “standalone” proposals [5,6], including the recent book on MDA[7]. Still the problem seems to be far from adequate solution.

It is a popular view, also strongly supported by us, that the main problem is “the availability of a concrete syntax for model-to-model transformations that is able to express non-trivial transformations in a clear and compact format” [8]. Since now it is universally accepted that all the involved models will be based on metamodels in MOF format, the problem is to define practically usable transformation language for transforming instance sets conforming to source metamodel to respective sets conforming to target metamodel. According to our view, and many others [8], this kind of model transformations should be defined graphically, but combining the graphical form with text where appropriate. From OMG QVT submissions only [2] and [3] use graphical form to a certain degree, but the most well-known graphical transformation languages are UMLX [5] and GreAT[6].

From the logical point of view, transformation languages in most cases consist of transformation rules and control structures governing their application [9]. Transformation rules, in turn, consist of pattern and action part (or LHS and RHS). Desirably, all this should be expressed in a unified diagrammatic form. The remaining space of variation is – how complicated the patterns are, what control structures are used. UMLX is based mainly on recursive calls as a control structure. GreAT uses patterns of approximately the same complexity, but the control structure is based on hierarchical dataflow-like diagrams, where the only missing control structure is an explicit notation for loops (loops are hidden in patterns). Thus there currently is no transformation language based on traditional control structures.

The paper proposes a new transformation language **MOLA (MOdel transformation Language)**. The prime goal of MOLA is to provide an easy readable graphical transformation language by combining the traditional structured programming in a graphical form (a sort of “structured flowcharts”) with pattern-based rules. This goal is achieved by introducing a natural graphical loop concept, augmented by an explicit loop variable. The loop elements can easily be combined with rule patterns. Other structured control elements

are introduced in a similar way. In the result, most of typical model transformation algorithms, which are inherently more iterative than recursive, can be specified in a natural way. The paper demonstrates this on the traditional class-to-database transformation example.

2 Basic Principles of MOLA

The MOLA is a natural combination of traditional structured programming languages and pattern-based model transformation rules – both in a graphical form.

A MOLA program is used to transform a **source model** satisfying the **source metamodel** to the required **target model**, corresponding to the **target metamodel**. Both models actually are treated as class and association instance sets satisfying the relevant metamodel.

MOLA control structure is fairly traditional – a program in MOLA is a sequence of **statements**. A statement is a graphical area, delimited by a rectangle – in most cases, a gray rounded rectangle. The statement sequence is shown by dashed arrows. A MOLA program actually is a sort of a “structured flowchart”.

The simplest kind of statement is a **rule**, which performs an elementary transformation of instances. A rule contains a **pattern** – a set of elements representing class and association instances, built in accordance with the source metamodel. Pattern elements can have attribute constraints (OCL expressions). A rule has also the **action** specification – new class instances to be built, instances to be deleted, association instances (links) to be built or deleted and the modified attribute values (as assignments). Both for the pattern and action part the UML object (instance specification) notation is used. The semantics is standard – locate a pattern instance in the source model and apply the actions.

The most important statement type in MOLA is the **loop**. Graphically a loop is a rectangular frame, containing a sequence of statements. This sequence starts with a special **loop head** statement. The loop head is also a pattern, but with one element – the **loop variable** highlighted (by a **bold** frame). A loop variable represents an arbitrary element of the given class. The semantics of a loop is natural – perform the loop for any loop variable instance which satisfies the conditions specified by the pattern.

Actually there are two types of loops in MOLA, differing in semantics details. The first type (denoted by a **simple** frame) is executed once for each valid loop variable instance, therefore it is called **FOREACH** loop. The second one (denoted by a **3-d** frame) is executed while there is at least one loop variable instance satisfying the pattern conditions – it is called the **WHILE** loop. The second type of loop may be executed several times for the same instance. A loop head may contain also actions (actually it is also a rule), thus the whole loop body may consist of one statement. In other cases, a loop body may contain several statement sequences each having a loop body (typically - with same loop variable). An alternative way is to use one common loop head and the **branch** construct – several frames started by pattern statements as conditions. Loops can be nested to any depth.

The loop is the basic and the most used statement kind in MOLA, which really makes typical model transformation programs look so natural.

Certainly, to scale up for arbitrary complex transformations, MOLA has the subprogram concept. One more statement type is the subprogram call, where the parameters can be references to instances used in the calling program (typically, to loop variables) or simple values. The called subprogram has access to the source model and can add or modify elements in the target model.

We conclude this brief overview by some more comments on patterns (both in loop heads and rules). Class and association instances (possibly containing attribute constraints) in the pattern are meant to be mapped directly to the source model – there must be a match for each

pattern element. However, only for **loop variables** (in loop heads) it is essential to find **all** possible **matches**. The other instances have more the “**exists**” semantics – there must be such an instance. To increase the expressibility, a **NOT** constraint can be added to a pattern association – there can be no link leading to the specified kind of class instance.

3 MOLA on an Example

Due to a very limited space the further details of the MOLA language will be given on one example. The example is the traditional one for model transformation languages – transform a class diagram to database definition. From the many versions of the example [2,3,5,7] the one used in [7] is chosen.

Fig. 1 shows both the source metamodel – a simplified UML class diagram and the target metamodel – a simplified SQL metamodel.

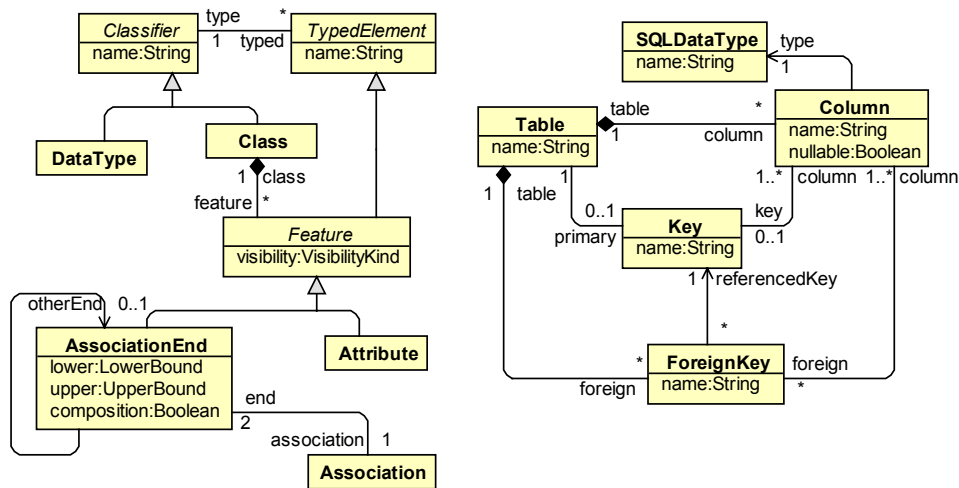


Fig. 1 Source and target metamodels of the example

According to [7], each class in the source model has to be transformed into a database table, with class attributes becoming table columns. All attributes are assumed to have a simple data type and here it is assumed (a sort of simplification!) that UML and SQL data types coincide. Each table must have an “artificial” primary key column with the type *integer*. One-one or one-to-many associations result into a foreign key and a column for it in the appropriate table (for one-one – at both ends). A many-to-many association is converted into a special table consisting only of foreign key columns (and having no primary key). Each foreign key references the corresponding primary key.

The transformation will be performed in two steps – first, the classes will be transformed into tables and attributes into columns, then the associations will be converted into foreign keys and columns supporting them. Fig. 2 depicts the first step and Fig. 3 - the second one. The first step contains one FOREACH loop, but the second – two loops of the same type.

The top-level loop in Fig. 2 is executed for each *Class* instance, since the trivial pattern has no conditions. The next statement is a rule building the *Table*, its primary *Key* and the *Column* for the selected *Class* instance. To show that namely the same instance selected by the loop head is used here, the **reference** notation is used – the instance name (*cl*) is prefixed by the @ character. The action part of the rule builds new class instances for *Table*, *Key* and *Column* and the corresponding association instances linking them. The new elements –

instances and links are shown with **dotted** lines (and in red color) in MOLA. The associations *#tableForCl*, *#keyForCl*, *#keyColForCl* are special ones – they are the so-called **mapping associations**, which are not specified in any of the metamodels (their names start with the # character). These associations link instances corresponding to different metamodels and typically are used in MDA-related transformations for setting the context for next subordinate transformations (e.g., *#tableForCl* will be used in the next statement) and for tracing instances between models (e.g., to record which *Table* from which *Class* actually has been generated). The attribute assignments for new instances use an OCL-like syntax for expressions, with qualifications by instance names. The expression in braces for the *SQLDataType* instance is an example of attribute constraint – here we assume that class instances for SQL data types are pre-built and have to be found.

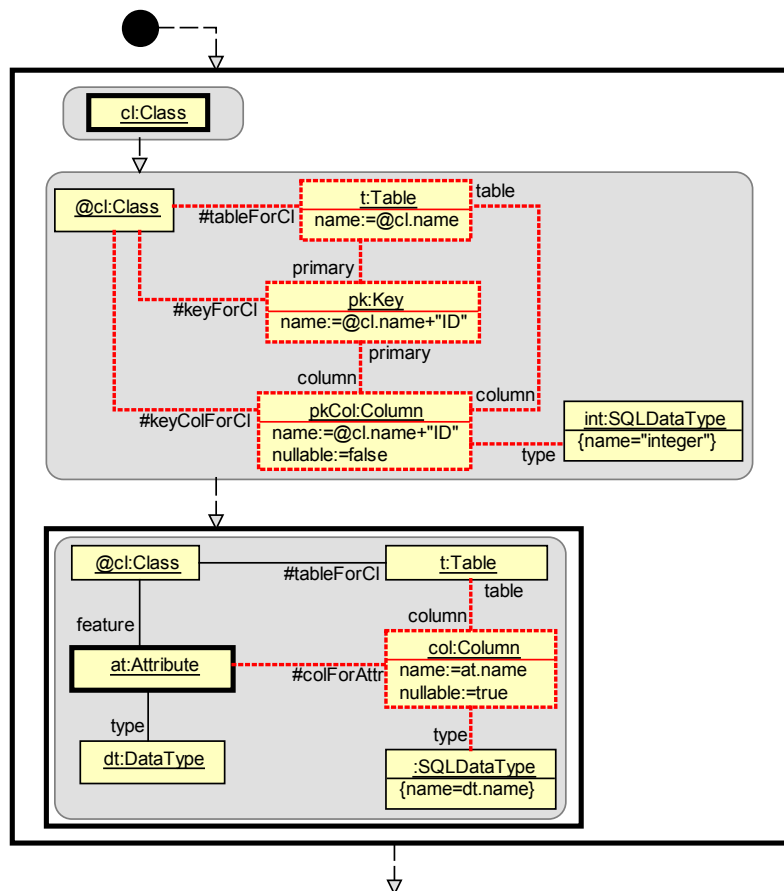


Fig. 2 The first step of the transformation

The next statement in Fig. 2 is a nested loop which is executed for each *Attribute* instance of the current *Class*. Its pattern references the *#tableForCl* mapping association, built by the previous statement and the loop head is combined with building actions.

The second step in Fig. 3 consists of two loops. They in totality are executed for each association instance – the first loop for those instances that have multiplicity 0..1 or 1..1 at least at one end and the second one for those which are many-to-many. This is achieved by adding mutually exclusive selection conditions to both loop variable definitions. These conditions are given in a graphical form. The first one uses the already mentioned in section 2 fact that an association in a condition (pattern) requires the existence of the given instance.

The other condition uses the {NOT} constraint attached to the association – no such instance can exist. Then both loops have an inner loop - for both ends (even in the first case there may be two “one-ends”). Both inner loops use mapping associations built by previous rules (*#keyForCl*, *#tableForCl*) in their conditions. The type of “foreign columns” is integer – as well as that for “primary columns”. Alternatively, one loop on *Association* containing two branches using the above mentioned conditions for selection could be used.

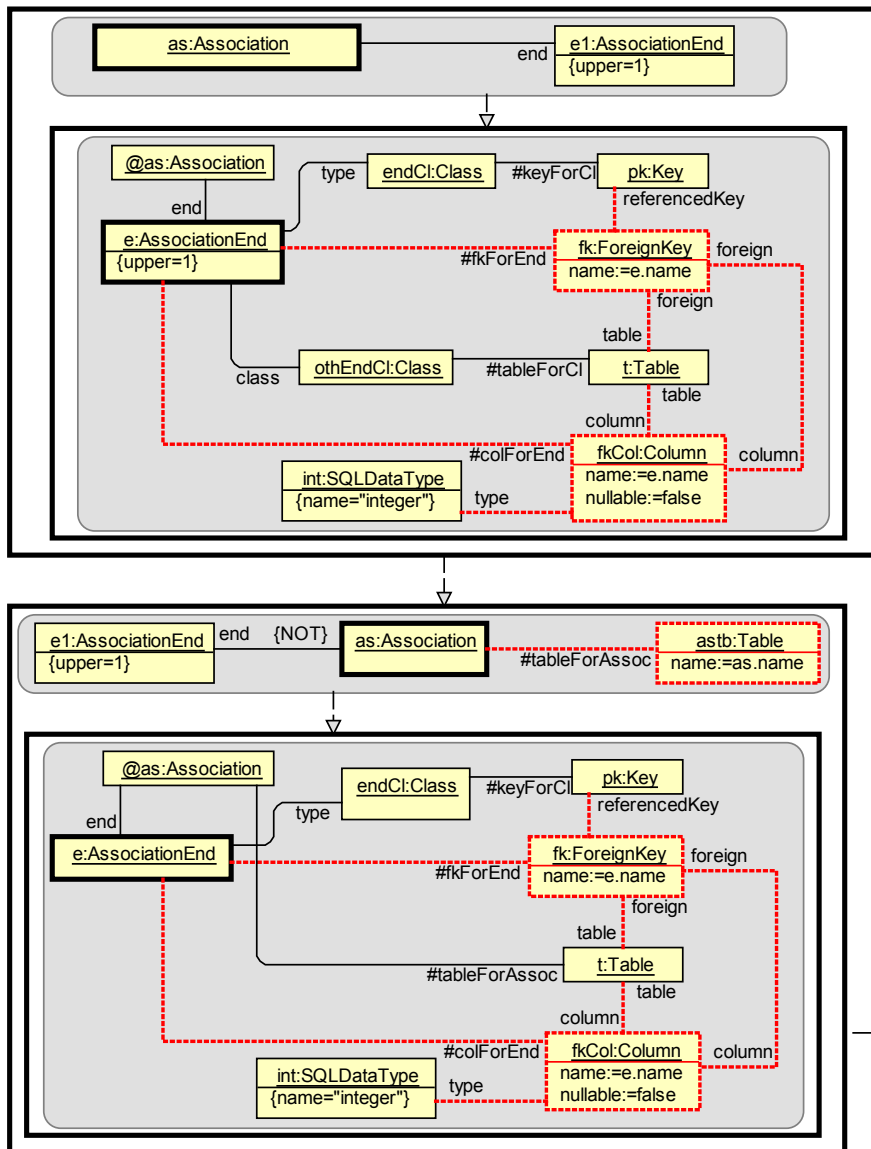


Fig. 3 The second step of the transformation

The example has demonstrated all the main constructs of MOLA and their typical usage. To complete the graphical syntax, deletion of elements is denoted by dashed lines. There are also some less frequently used constructs in MOLA, such as temporary attributes and associations, which permit to implement complicated computations, but there is not enough space to cover them.

4. Conclusions

Authors hope that the given example is a convincing proof of transformation program readability in MOLA. At least for “graphics-minded” readers it can be understood much easier than its OCL-based equivalent in [7].

The language has been tested on most of MDA related standard examples – e.g., class to Enterprise Java in [7], UML statechart flattening from [6]. In all cases a natural representation of the informal algorithms has been achieved, using mainly the MOLA loop feature. An especially adequate representation for the statechart flattening task has been obtained. Though the original algorithm is recursive to certain degree, the use of WHILE loop with several loop heads (not demonstrated in this paper) permits a natural iterative description of it. This provides convincing arguments for a practical functional completeness of the language for various model to model transformations in model driven development area. Though it depends on readers’ mindset, the “structured flowchart” style in MOLA seems to be more readable and also more compact than the pure recursive style used e.g., in [5]. Though recursive calls are supported in MOLA, this is not the intended style in this language. There is one special kind model transformation tasks based on so-called transitive closure pattern (required e.g., for the [2,5] version of class to database transformation). Though this pattern is completely implementable in MOLA, a more direct description of it is available in the extended MOLA (see [10]).

The implementation of MOLA in a model transformation tool also seems not to be difficult. The patterns in MOLA are quite simple and don’t require sophisticated matching algorithms. Due to the structured procedural style the implementation is expected to be quite efficient. All this makes MOLA a good candidate for practically usable model transformation language.

References.

- [1] OMG: Request For Proposal: MOF 2.0/QVT. OMG Document ad/2002-04-10, <http://www.omg.org/cgi-bin/doc?ad/2002-04-10>
- [2] QVT-Merge Group. MOF 2.0 Query/Views/Transformations RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, <http://www.omg.org/cgi-bin/doc?ad/2004-04-01>
- [3] Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, <http://www.omg.org/cgi-bin/doc?ad/2003-08-07>
- [4] Interactive Objects Software GmbH, Project Technology, Inc. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-11, <http://www.omg.org/cgi-bin/doc?ad/2003-08-11>
- [5] E.D.Willink. 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.
- [6] Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, November 2003.
- [7] Kleppe A., Warmer J., Bast W. MDA Explained. The model driven architecture: practice and promise. Addison-Wesley, 2003.
- [8] Bettin J. Ideas for a Concrete Visual Syntax for Model-to-Model Transformations. Proceedings of the 18th International Conference, OOPSLA’2003, Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, USA, October 2003.
- [9] Czarnecki K., Helsen S. Classification of Model Transformation Approaches. Proceedings of the 18th International Conference, OOPSLA’2003, Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, USA, October 2003.
- [10] Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA: Extended Patterns. To be published in proceedings of Baltic DB&IS 2004, Riga, Latvia, June 2004.

Model Transformation Language MOLA: Extended Patterns

Audris Kalnins, Janis Barzdins, Edgars Celms

University of Latvia, IMCS
29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Edgars.Celms}@mii.lu.lv

Abstract. The paper describes a new graphical transformation language MOLA for MDA-related model transformations. Transformations in MOLA are described by combining traditional control structures, especially loops, with pattern-based transformation rules. Besides an overview of the basic MOLA, the paper describes an extension of MOLA – powerful patterns, which may include transitive closure. The paper shows how the usage of these patterns simplifies control structures for typical MDA tasks.

Keywords. Model transformations, MDA, patterns

1. Introduction

The increased use of modeling techniques nowadays requires effective support for model transformations. Perhaps, one of the most actual areas in software engineering today is the Model Driven Architecture (MDA) [1]. MDA is a software development approach in which models are the primary artifacts. According to the MDA, the different types of models are defined (most usually in UML [2,3] notation), mapping between models and towards different targets is formalized, and guidelines are automated, in order to improve efficiency and guarantee that the process of the transformation between models is properly followed. Model-to-model transformation is therefore a key technology for MDA. While the current Object Management Group (OMG) standards such as the Meta Object Facility (MOF) [4] and the UML provide a well-established foundation for defining different types of models, no such well-established foundation exists for transformations between them. The need for standardization in this area led to the MOF 2.0 Query/Views/Transformations (QVT) request for Proposals (RFP)[5] from OMG.

To a great degree the success of the MDA initiative and of QVT in particular will depend on the availability of a concrete syntax for model-to-model transformations that is able to express non-trivial transformations in a clear and compact format that would be useful for industrial production of business software [6].

The submissions by several consortiums have been already made, e.g. [7, 8, 9], and it is somewhat surprisingly, that only a few of them use a natural graphical representation of their language. Currently none of them has reached the status of a complete model transformation language. Several proposals for transformation

languages have been provided outside the OMG activities. The most interesting and complete of them seem to be UMLX [10] and GReAT [11].

According to our view, and many others [6], model transformations should be defined graphically, but combining the graphical form with text where appropriate. Graphical forms of transformations have the advantage of being able to represent mappings between patterns in source and target models in a direct way. This is the motivation behind visual languages such as UMLX, GReAT and the others proposed in the QVT submissions. Unfortunately, the currently proposed visual notations do not provide easy readable descriptions of model transformations.

The common setting for all transformation languages is such that the model to be transformed – the source model is supplied as set of class and association instances conforming to the source metamodel. The result of transformation is the set of instances conforming to the target metamodel – the target model. Therefore the transformation has to operate with instance sets specified by a class diagram (actually, the subset of class notation, which is supported by MOF).

Approaches that use graphical notation of model transformations draw on the theoretical work on graph transformations. Hence it follows that most of these transformation languages define transformations as sets of related rules. Each rule consists of a pattern and action part, where the pattern has to be found (matched) in the existing instance set and the actions specify the modifications related to the matched subset of instances. This schema is used in all of abovementioned graphical transformation languages. Languages really differ in the strength of pattern definition mechanisms and control structures governing the execution order of rules.

The most detailed pattern definition is in the GReAT language. There it is possible to match a set of instances to one element of the pattern (variable cardinality patterns). However, the patterns are still limited in depth but this is compensated by a very elaborated rule control structure specified graphically by dataflow-like diagrams. UMLX has a similar but slightly weaker pattern mechanism. The control structure is completely based on recursive invocations of rules. In the proposal by QvT-Partners [7] graphical patterns are combined with extensive use of textual constraints. The control structure is based on recursive invocation of rules. In the DSTC/IBM/CBOP proposal [12] (now merged with [7]) patterns are specified in a textual (Prolog-like) form, the most interesting feature of this language is the possibility to include a transitive closure in patterns.

This paper proposes a new graphical transformation language MOLA (Model Transformation Language). The main design goal for MOLA has been to make the transformation definitions natural and easy readable, by relying on simple iterative (non-recursive) control structures, based on traditional structured programming. In addition, as far as it improves readability, the intention was to make each rule more powerful. In particular, this requires the strengthening of pattern mechanism. The MOLA project actually consists of two parts – the basic and extended MOLA. The basic MOLA uses simple patterns and more relies on control structures – it has a more procedural style. The main element there is a graphical loop concept, which can easily be combined with a transformation rule. The main new feature of the extended MOLA is the possibility to define looping patterns of “unlimited depth” (in addition to variable cardinality), thus incorporating the mechanism of transitive closure in patterns. In the result, very simple control structure – a sequence of simple loops then is sufficient for many transformation jobs. Certainly, such a pattern

definition requires an adequate definition of the matching procedure, which is also described in the paper. High execution efficiency for the procedure is guaranteed in the typical case when the pattern cardinalities are adapted to the metamodel multiplicities, to which the instance set conforms (the “uniqueness principle” is observed). Patterns in MOLA are defined as directed graphs, to enable the required control over the matching procedure – also a new feature for pattern definition. As a consequence of larger and more powerful patterns, a typical step of a transformation frequently can be described by one rule. This natural non-recursive style of transformation definitions in MOLA has been tested on several real MDA jobs, at the same time using the features of MOLA to keep each rule not too complicated (namely the right balance there ensures the human readability of transformations). As far as we know, the extended MOLA is the only graphical transformation language, which supports transitive closure in patterns. The ideas for pattern definition in MOLA have been partially inspired by the authors’ previous experience in defining mappings for generic modeling tools based on metamodels [13].

This paper describes the basic elements of MOLA. The main emphasis is on the extended pattern concept. Language description is based on a typical MDA example (used also in [7, 8, 10, 14]) – transformation of a simplified class diagram into a database definition. Section 2 describes the general structure of MOLA, section 3 - the example. Section 4 describes the basic MOLA – rules with simple patterns and the new concept of loop. The complete pattern mechanism in extended MOLA, including cardinality constraints, looping patterns and the corresponding matching procedure is described in section 5.

2. Overview of language elements in MOLA.

The MOLA language is a natural combination of pattern-based model transformation rules with control structures from traditional structured programming, both specified in a graphical form.

MOLA is meant for transforming models built according to one metamodel – the **source metamodel** (SMM) to models conforming to another metamodel – the **target metamodel** (TMM). In a special case, SMM and TMM may coincide. Both the source and target models actually are treated as instance sets of the corresponding metamodel classes and associations.

A **transformation definition** in MOLA consists of the both metamodels and the transformation program. A **transformation program** in MOLA is a sequence of **statements**. A statement is a graphical area, delimited by a rectangle – in most cases, a gray rounded rectangle. The statement sequence is shown by dashed arrows. The program starts with the UML start symbol and ends with an end symbol.

The simplest kind of statement is a **rule**, which performs an elementary transformation of instances. A rule contains a **pattern** – a set of elements representing class and association instances, built in accordance with the source metamodel. Pattern elements can have **attribute constraints** (OCL expressions). The pattern specifies what kind of instance group must be found in the source model, to which the rule must be applied. A rule has also an **action** specification – new class instances to be built, instances to be deleted, association instances (links)

to be built or deleted and the modified attribute values (as assignments). Both for the pattern and action part the UML object (instance specification) notation is used.

The most important statement type in MOLA is the **loop**. Graphically a loop is a rectangular frame, containing a sequence of statements. This sequence starts with a special **loop head** statement. The loop head is also a pattern, but with one element – the **loop variable** highlighted (by a **bold** frame). Informally a loop variable represents an arbitrary instance of the given class, which satisfies the conditions specified by the pattern. Actually there are two types of loops in MOLA, differing in semantics details. The first type (denoted by a **simple** frame) is executed once for each valid loop variable instance, therefore it is called **FOREACH** loop. Mainly this type of loop will be used in the paper. The second type (denoted by a **3-d** frame) is executed while there is at least one loop variable instance satisfying the pattern conditions – it is called the **WHILE** loop. Loops can be nested to any depth. A loop head can contain actions – it is also a rule. Such a combined statement will be widely used in the examples of this paper.

Other control structures in MOLA are the **branch** construct – several frames started by pattern statements as conditions and the **subprogram** concept together with the **subprogram call**, where the parameters can contain references to instances used in the calling program. However, these control structures actually will not be used in the paper – the aim of this paper is to demonstrate how extended patterns in MOLA allow to use a very simple control structure – a sequence of simple loops.

Section 4 discusses in detail the syntax and semantics of basic MOLA on an example. Then the extended patterns are introduced – the section 5 describes the extended MOLA.

The general execution schema in MOLA is simple – when a source model is supplied, statements are applied to it in the specified order. A statement is always applied to the whole instance set which is being gradually transformed by the rule actions. The potential execution efficiency is ensured by the corresponding features of the pattern language, where it is easy to specify that only “useful” pattern matches occur.

During the transformation process one more optional metamodel – the **intermediate metamodel** (IMM) may be used. IMM contains both SMM and TMM as a subsets, and additional elements – classes, attributes and associations necessary for performing the transformation. There is a special kind of additional associations – **mapping associations** which in fact are present in every transformation. These associations physically implement the mapping from the elements of source model to target elements, therefore they link classes from SMM to the corresponding classes in TMM. See more on the role of mapping associations in 4.2. Namely due to a large set of mapping associations it is recommended to use IMM for non-trivial transformations (it is also permitted in MOLA to define mapping associations “on the fly” – directly in rules, if IMM is not used). Another important elements of IMM are **computed attributes** – “temporary” attributes added to classes of IMM for storing intermediate values. There may be several kinds of computed attributes, the most used here are the rule-local ones. Names of rule-local attributes start with “?” in the IMM, see more on them in 5.2. Non-local temporary attributes (with the scope of several statements) can be also defined/created and destroyed by special statements.

3 Example - the class model to relational model transformation

The paper will be based on a typical MDA example, considered also in [7, 8, 10, 14] – the transformation of a simplified class diagram into a semantically equivalent relational database definition. For all the different versions of the example the one in [7] is used here. This version permits to demonstrate the easy definition of transitive closure in extended MOLA. The SMM for this task is shown in Fig.1.

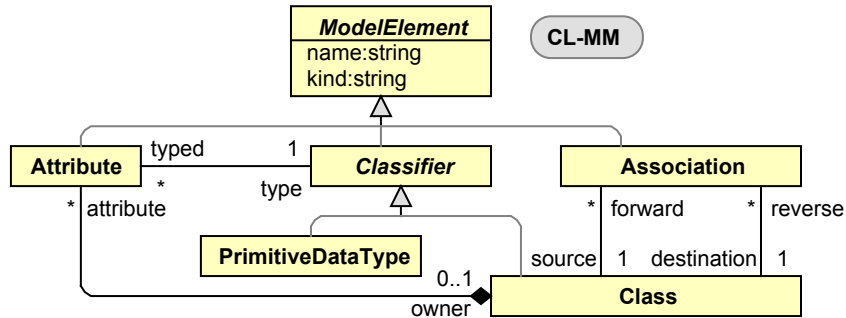


Figure 1. Source metamodel for simplified class diagram.

All elements can have a *name*. The metaattribute *kind* is applicable to metaclasses *Class* and *Attribute*, only a *Class* where its value is “*persistent*” must be transformed into a database *Table*, the value of *kind* equal to “*primary*” determines that an *Attribute* actually is a part of a primary key (all other values of *kind* are irrelevant). The *type* of an *Attribute* can be either a *PrimitiveDataType*, or another *Class*. Fig.2 shows the TMM - a simplified relational database definition metamodel.

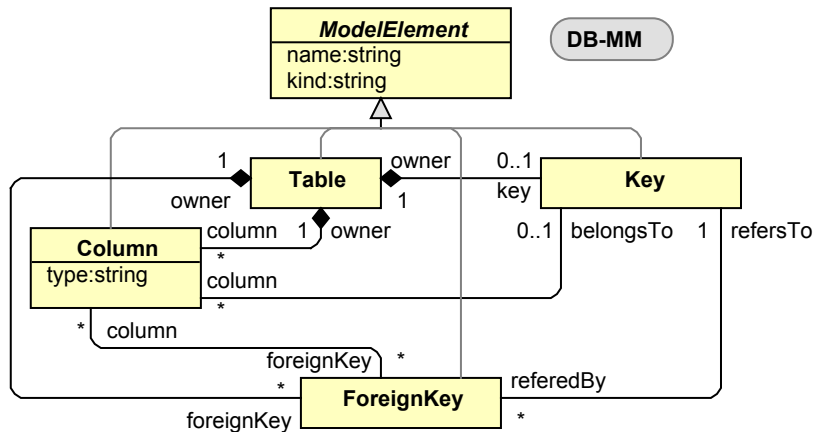


Figure 2. Target metamodel for database definition.

A *Table* consists of *Columns*, and it can have a (primary) *Key*, which contains some of the *Columns*. The *ForeignKey* for a *Table* always refers to a *Key* of another *Table*.

The informal transformation algorithm is quite straightforward. For each *persistent* class a table and its key must be built. The primitive-typed attributes of this class become columns of the table (with the same name and type name). The columns which correspond to primitive-typed attributes of *primary* kind become parts of the key. In addition, for class-typed attributes, the primitive attributes of the target class are also transformed to columns of the table for the original class (as “indirect attributes”), and this process of finding indirect attributes is continued down until no more indirect attributes can be found (so-called class flattening – a transitive closure-like process). A column for an indirect attribute has a compound name – the concatenation of all attribute names down to the primitive one. An association is converted into a foreign key for the source class (table), and this foreign key refers to the key for the destination class. In addition, new columns are added to the source table (and to the foreign key) – one (equally named and typed) for each column of the corresponding key. The problem of ordering the columns in keys is ignored in the example. Association multiplicities also are not used in this simplified example – only the association direction matters. It should be noted that the processing of indirect attributes and associations is independent – it may be done in any order.

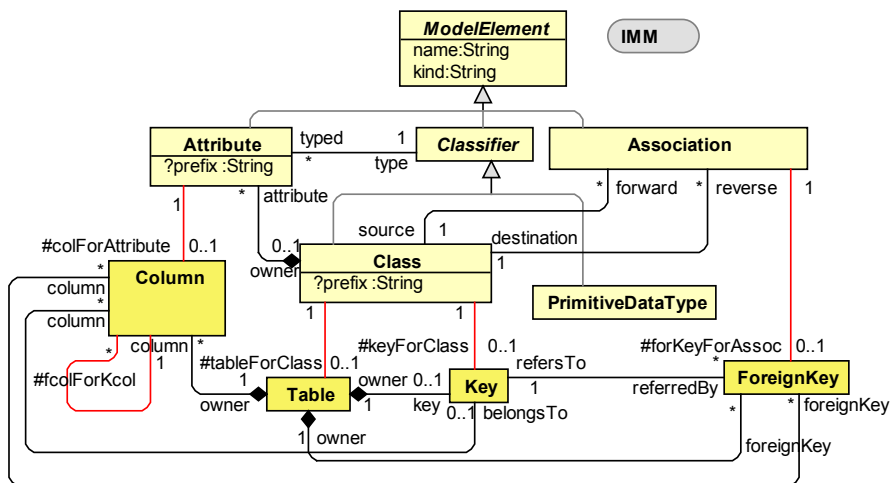


Figure 3. Intermediate metamodel for the transformation.

The one remaining element to be described is the IMM – see Fig.3. In addition to a copy of SMM and a copy of TMM (the classes of TMM are in a darker color), it contains mapping associations from source to target elements, e.g., from *Class* to *Table*. These mapping associations will be used in rules in sections 4 and 5, they have the # character prefixed to role names and are in red color in Fig.3. IMM contains also a computed attribute *prefix* of rule-local kind (names of rule-local attributes start with “?” in MOLA), it will be used in section 5.2.

4. Structure of simple loops and rules

As it was stated, both loops and rules rely on patterns in MOLA. In this section the structure of simplest patterns will be described in detail. These patterns, for which only a fixed-size match is possible, have a very simple matching algorithm. The patterns in this section actually are weaker than those described in [10, 11], the goal of this section is just to demonstrate the general principles of MOLA in a very simple case. Non-trivial patterns of MOLA will be described in section 5.

4.1. Basic patterns

A pattern in MOLA specifies the instance set which can be matched to it. From a syntax point of view, it is similar to UML 2.0 collaborations or structured classifiers. The main **element** of a pattern is a source **metamodel class**, specified in UML **instance notation**. Each element has an optional **instance name** and the **class name**, the same class may be used several times in a pattern. In totality, they must be unique within a pattern. Each element matches to an appropriate instance of that class. Since a typical use of pattern in MOLA is in a loop head, we start with this case. There one pattern element – the loop head (a bold one) has a special meaning. All other elements of the pattern are used to specify, namely which instances of this class in the source model can be used as loop variable instances. The other pattern elements (which may correspond also to target metamodel classes) also must match to an appropriate instance – they specify the context of a loop variable.

In addition to elements, a pattern contains **pattern associations** – selected metamodel associations between the used classes and **attribute constraints** – OCL constraints specified within elements (in braces). The specified association instances must exist between the matched model instances and the attribute constraints must evaluate to true. Pattern associations can have also a {NOT} constraint – this means that no specified instance can be linked to the “main match” by the given link.

Thus a pattern in a loop head specifies which instances of the given class in the source model qualify as valid instances for the loop variable. The other pattern elements have the “exists” semantics – there must be (or must not be) an appropriate instance in the selected match, but there is no need to find all possible matches for them.

The loop variable in a pattern in fact plays the role of its **root** – the match is started from it.

Fig.4 shows the simplest pattern consisting just of the loop variable. This pattern says that an instance of *Class* in the source model (i.e., the instance set corresponding to the class diagram to be transformed into a database definition) matches to this pattern, if its *kind* has the value “persistent” (*kind* – a string-typed attribute of *Class*).

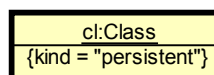


Figure 4. Simplest pattern example.

Fig. 5 shows a more non-trivial pattern, involving several elements and associations. In this pattern only these instances of *Attribute* qualify as a loop variable instances, which have an *owner* link to a *Class* instance, which in turn has a *#tableForClass* link to a *Table* instance, and also have a *type* link to a *PrimitiveDataType* instance. The *Table* class is from the target metamodel, and the *#tableForClass* is a mapping association – this means that these instances have to be already built by previous statements. Pattern associations typically specify only one of the role names – that leading away from the root.

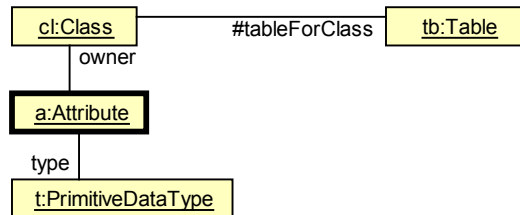


Figure 5. Associations in a pattern.

The simple patterns described so far do not require a more formal match definition. However, for extended patterns in section 5 such a definition will be used.

Here one principle of a good programming style in MOLA should be given. To achieve a high execution efficiency, pattern associations leading away from the loop variable (the pattern root) should have the **0..1 multiplicity** at this end in the corresponding metamodel. This means that we test the **existence of one possible instance**. In addition, in the case of existence, the match is unique then and we can reference this instance for various purposes, e.g., to use its attribute values in a deterministic way. Actually, all examples in the paper use this principle. For other multiplicities the extended patterns in section 5 serve well.

Patterns can use also the **reference** notation – an element whose name is prefixed by the **@** character – this means that an already selected instance (by a previous statement, typically a loop head) must be used. This way patterns can be structured – similarly as, e.g., in [11]. See an example in Fig. 8.

4.2. Actions of the rule, complete rule examples

Pattern matching is only one part of the rule application. Another one is to perform the actions specified in the rule (on the basis of the current match). These actions modify the current instance set – typically, the target model. The following actions can be specified in a rule:

- building new class instances
- building new association instances (connecting new as well as existing instances)
- changing the attribute values – both for new and existing class instances
- deleting instances

The **action specification** (the “RHS part”) of a statement has a structure similar to the pattern. It also consists of elements to build (in the instance notation) and

associations linking the new elements between themselves or to the pattern elements. Syntactically the action part is distinguishable by dotted lines and the line color – it is in **red**. Actions can also specify the deletion of an existing element (matched by the pattern) – this is shown by dashed lines.

The most typical action is the **building of a new class instance**. Building of class instance in MOLA is always accompanied by building of one special association – the **mapping association**, which in the rule must be linked to a pattern element (e.g., an association with the role name *#tableForClass* is linked to *cl:Class* in Fig.6). The role name of this association is specified in the intermediate metamodel (here – Fig.3) if that exists, but anyway its name must be prefixed by the # character. At the instance level it means that one instance of the new class is built and linked by the mapping association to the existing instance of the corresponding pattern element.

One goal of the mapping association is to serve for matching in the patterns of next rules. It is very typical in MDA model transformations that the transformation of a “higher level” element – package, class etc. determines how its subordinates – classes, attributes etc. must be transformed. The mapping association is namely the element linking such subordinate rules and ensuring their consistency. In addition, the mapping association reifies physically the mapping between the source and target model (and serves for tracing), hence such a name is used for this concept in MOLA. In our simple patterns, the cardinality of the mapping association is 1 – 1, but in more advanced patterns of MOLA it may have cardinality 1 – 1..* (and serve for determining the instance set of the new class which must be built).

The remaining action element is the **assignment of attribute values** (done by Pascal-like assignment statements). For an attribute to be set the new value is defined by an OCL style expression, which can contain one extension – attributes from pattern elements may be referenced, just by prefixing them with the instance name. The semantics is straightforward – take the attribute value from the existing instance matched to the element. The attribute assignments can be done for the “new” instances, but attributes of existing instances (in the pattern elements) can also be modified this way.

Fig.6 shows a complete example of a statement in MOLA. This is the first statement in the program for building the database definition from a class diagram.

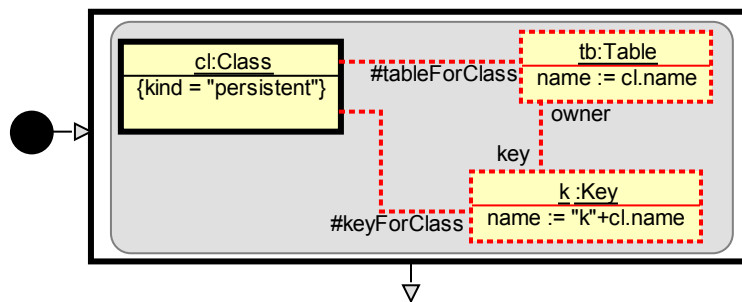


Figure 6. Simple statement in MOLA.

This statement is a FOREACH loop consisting of its loop head only, this loop head is also a rule which builds new instances. It does the first job in the transformation process – builds instances of both *Table* and *Key* for any *Class*

instance whose *kind* has the value *persistent*. The *name* attribute in each of the new instances is set to the specified value – to the *name* value in the matched *Class* instance. In addition, the two mapping associations are built, as well as an association instance with the roles *key* – *owner* between the new instances.

Fig. 7 shows the next two statements of the transformation program – both FOREACH loops too. The first one builds columns corresponding to primitive-typed attributes of persistent classes from the source model. Its pattern (discussed in section 4.1) selects the appropriate table in the target model (built by the previous statement) and the rule associates the new column to it.

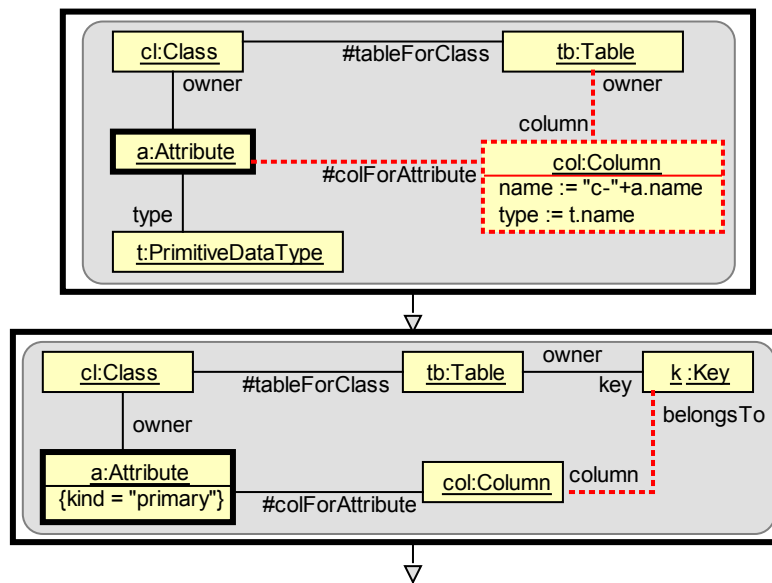


Figure 7. Statements transforming primitive attributes to columns and associating key columns.

The other statement in Fig.7 has the intention to attach the *belongsTo* association to each *Column* which corresponds to an *Attribute* with *kind* = *primary* (the other end of the association must be the *Key* for the relevant *Table*).

One more task to be done is to process associations in the source model. Fig.8 shows the corresponding program statement – a nested FOREACH loop. The top level loop builds the foreign key for each association in source model and associates it to the relevant primary key (built by the first statement). The nested loop builds a new column (in the table corresponding to the source class) corresponding to each column in the referred primary key. The pattern of this loop uses three references to instances selected in the top loop – all prefixed by the @ character. We remind that this means that namely these referenced instances must be used in any match for the subordinate pattern. Thus there is no need to repeat the corresponding selection conditions in the nested loop – its pattern becomes simple.

Certainly, the building of columns for a foreign key could be done by a separate independent loop, but then its pattern would be more complicated. In general, a certain “**breadth first programming style**” – each action a separate top level loop –

is in most cases usable for typical MDA jobs. But sometimes nested loops help to structure complicated patterns.

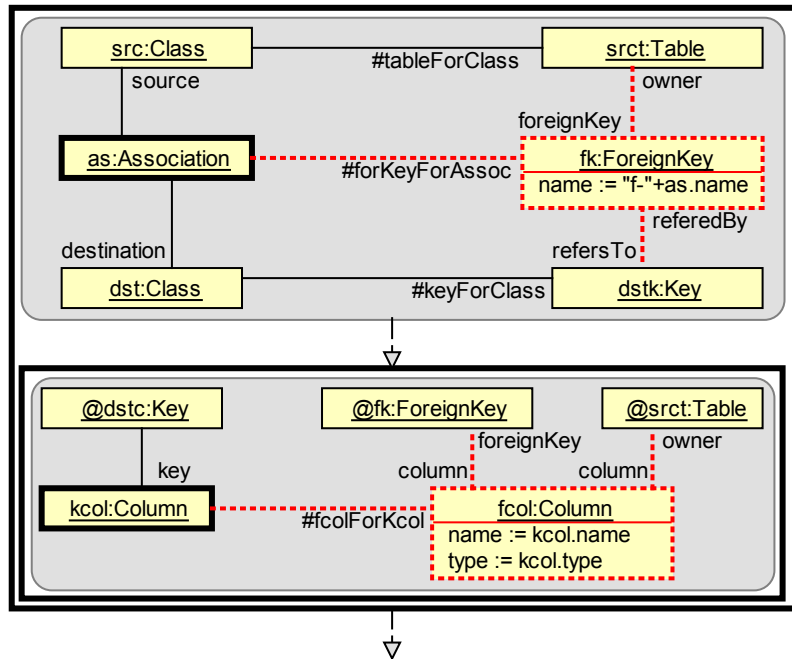


Fig. 8 Statement processing source model associations.

The remaining task – flattening class-typed attributes can also be implemented in the basic MOLA described so far. However, since the flattening is a true transitive closure task, and in its most complicated form – find all possible paths in the source model from a class to its indirect primitively-typed attributes and compute a name along each of the paths, it requires creating copies of attribute instances, building temporary associations and attributes and using depth three loops. In other words, the standard algorithm for building all paths in a graph has to be implemented. An alternative more readable solution is to use extended patterns to be considered in the next section.

5. Extended patterns

The patterns considered so far have one limiting property – only one instance can be matched to an element. Since this is too restrictive for some tasks in real transformations, especially those related to transitive closure, various ways to extend the pattern notation will be considered in this section. First, patterns with cardinality constraints will be considered – they may have unlimited number of instances associated with one pattern element, but the matching depth is still limited. An efficient match building procedure for this case is defined in a completely different way – as a stepwise algorithm on a graph. The efficiency of this procedure is

guaranteed if uniqueness principle is observed – pattern cardinalities match to the metamodel multiplicities. Finally, the looping patterns with unlimited matching depth are introduced – namely these patterns are more powerful than those in [10, 11] and permit to perform nontrivial actions, including transitive closure, in one rule.

5.1. Cardinality constraints for navigation associations

In section 4 the simplest case was considered where each pattern element was matched to a single instance and each association to a single association instance linking the matched class instances. In order to have larger fragments of the instance set mapped to the pattern, with several instances associated to one pattern element (what is really required by transformation rules), the extended MOLA uses **cardinality constraints** attached to pattern associations (actually something similar is used also in [10, 11]). In addition, the associations with cardinality constraints are treated as **directed** graph edges – using the UML navigability notation.

One of the constraints - **ALL**. It is used when the association has * or 1..* multiplicity at the appropriate end in the metamodel. It corresponds to * in UML – take all what you can, but nothing bad, if none. Another constraint is **OPT**. It is used with associations having 0..1 multiplicity at the appropriate end and means – take the instance if it exists, but the pattern does not fail if none exists. Actually OPT is a decorative version of ALL for the 0..1 case – to improve the readability. And we remind that empty cardinality constraint actually means **just one**. **NOT** also can be used as a cardinality constraint – there is none. In fact, there are constraints in MOLA which correspond to all possible UML multiplicities, but currently we don't need the other.

Now, more precisely, what is a valid extended MOLA pattern. Here we consider only the case when the pattern is used as a loop head. Then there is a loop variable, which serves as a pattern **root**. The pattern consists of **two** parts, having the **loop variable** as the **sole common node**. The first part, which uses undirected associations (and no additional cardinality constraints), is the same as before. It expresses (as before) the conditions for selecting valid instances for the loop variable. The other – the **extension** part uses **directed** associations and cardinality constraints and is used for matching to a set of instances. It must be a **directed acyclic graph (DAG)** starting from the loop variable as a root (a more complicated case with loops is considered in the next section). This part can be built by taking classes from the metamodel (in fact, IMM), converting them to pattern elements (adding instance names) and adding metamodel associations as directed edges. The extension part must be **distinguishable by associations** – if several edges with the same role name leave a node, they must lead to different classes (this requirement is essential for having an efficient match procedure, a weaker version of this restriction will be given in 5.2). The next step is to add appropriate cardinality constraints to the **navigable ends** of associations – ALL if the multiplicity in the metamodel is * or 1..* and OPT or nothing (one) if multiplicity is 0..1 or 1. When cardinality constraints are set this way, we say that the “**uniqueness principle**” is observed (the pattern fits to the metamodel). The pattern in Fig. 9 obviously satisfies the uniqueness principle – ALL is at the *attribute* end of the association from *Class*

(where the multiplicity in IMM is *), all other multiplicities are 0..1. To emphasize the fact that we expect many instances of *Attribute* to be matched, a decorative element in the pattern – the **multioject** notation (from UML collaborations) can be used for the *Attribute* element.

We make here also one assumption – the extension part edges leaving the root node have the constraint ALL or OPT (the extension part should express an unlimited, but optional at the same time part of the match).

Let us consider an example for the usage of ALL constraint in an extended pattern – the first statement from Fig.7 but defined in an alternative way – in Fig.9.

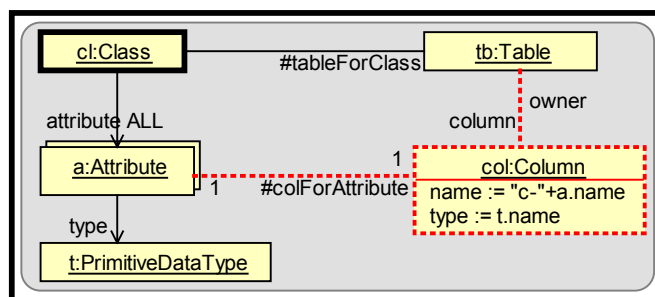


Figure 9. Rule with cardinality constraint.

The loop variable accepts as valid those instances of *Class*, for which a *Table* has been built. Now, when the loop is executed for a valid instance of the loop variable, the following new action is performed. For the extension part of the pattern (containing the elements *a:Attribute* and *t:PrimitiveDataType*) a temporary instance DAG is built containing all matches for the given root instance. For the given example this DAG is very simple – **all** those *Attribute* instances for the given *Class* instance (root), which are linked to a *PrimitiveDataType* instance, together with the corresponding association instances (*attribute* and *type*). The result is indeed a DAG, but not a tree, because a *PrimitiveDataType* instance can be used for several *Attribute* instances. Here the original instances from the source model are used as nodes for the DAG (we can assume that the nodes and edges of the DAG are highlighted, e.g., by “coloring” them green), but in some cases node copies are built – see section 5.2.

When the instance DAG is built, the rule actions are performed. Here a *Column* instance is built for each *Attribute* instance in the DAG (i.e., an instance associated to the element *a:Attribute*). This is specified by the mapping association (*#colForAttribute*) which now has an explicit multiplicity 1-1. Then the DAG is discarded (the highlighting is removed).

Now we will define the match building for an arbitrary pattern. The most natural way is to use a **procedural match definition**. We treat the pattern (its extension part) as a DAG from the root and build the instance DAG starting from its root – the current loop variable instance. Valid instance nodes are added to it layer by layer, in strict accordance with the pattern – so that each instance node can be assigned to a pattern node at that layer and the edges also match. Here the number of layers is fixed (determined by the pattern). Cardinality constraints must be taken into account – if the constraint is ALL, it doesn’t matter how many edges exit a node in the

current layer, but for the default constraint (just one) there must be the corresponding edge to a node in the next layer. If that edge is not found, the node must be removed from the current layer as invalid. The pattern edges with ALL constraint actually generate fan-out cases in the instance DAG.

The formal definition of the **matching procedure** (generating a complete valid match – the instance DAG) is the following:

1. mark the current instance of the loop variable as the only instance in the layer one of the instance DAG and associate it to the pattern root.
2. take a node in the current layer of the instance DAG. For each directed association leaving the pattern node, to which the instance node is associated, find **all** association instances from the current node and select those where the target instance satisfies the corresponding attribute constraint; add this “filtered neighborhood” to the next layer. Repeat this for all nodes in the layer. If a node in the next layer has been reached twice, mark it only once (the path history is not important in this mode).
3. assign instances in the next layer to the corresponding elements in the pattern (it can be done uniquely due to the required distinguishability by associations).
4. check cardinalities – for each node in the current layer and for each navigation association (which has the default cardinality constraint - i.e., “just 1”) from the associated pattern node check whether there is an instance of this association. If there is none, remove the instance node from the current layer, and recheck the previous layers (for layer one it cannot occur due to our assumption). ALL and OPT constraints require no check.
5. repeat steps 2, 3, 4 for each layer of the pattern

The semantics of ALL guarantees that always the maximal match is selected – no subset of a match can be a valid match. Even more, for a given root the procedure result is **deterministic** – it is due to the “uniqueness principle” for the pattern, that from several possible instances all are selected, and one instance must be selected from possible one. Namely this would permit also an efficient match implementation in MOLA.

5.2. Looping patterns

In this section we introduce the final elements of extended patterns in MOLA. First, we permit patterns to have **directed loops** in the extension part when a pattern is built on the basis of a metamodel fragment. This extension is essential for defining a **transitive closure in a pattern**. Features will also be provided for defining a closure involving all possible paths.

The requirement introduced in 5.1 that the extension part must be **distinguishable by associations** is still in place. This requirement is sufficient for the example in Fig. 10 and many similar ones. A weaker restriction – the K-distinguishability – sufficient for any reasonable MDA task will be considered at the end of section (however, it makes the matching procedure more complicated).

Certainly, the **uniqueness principle** from 5.1 must be observed when assigning cardinality constraints to pattern edges – violating this principle would lead to a much more complicated and inefficient matching procedure.

Though a pattern now may have loops, the instance graph for it will be required to be a DAG anyway. From the theoretical point of view, we may be interested in finding instance-level loops via patterns, but no MDA related job was found where it makes sense. Therefore loops at the instance level will be simply forbidden by the matching procedure.

One more remark refers to nodes of the instance DAG. In 5.1 a simple case was considered where the original model nodes were used (just highlighted). But this implies that the path history cannot be stored in the instance DAG – if a node is reachable via two or more paths, data from the path cannot be stored in the node. The only way for storing this data is to make copies of the original instances and store them in the DAG. In an extended MOLA statement it can be specified which **pattern nodes must be copied** (such a node is marked by a **square icon**) during the building of the DAG (it makes sense only for the looping nodes). The temporary copies in the DAG are related to their originals and “inherit” all attributes and association instances from them. When the statement completes, the temporary copies are discarded. Copying selected pattern nodes is the easiest way to implement transitive closures where all paths from a node must be traversed – as the one in Fig. 10.

The same **matching procedure** from the previous section is usable, but with the following extensions:

- step 2. New instance which is already present in the DAG on the path (from the root to the current instance) is not added to the next layer – a safeguard against infinite loops. If the target instance corresponds to a pattern element in the “copy list”, make a copy of the instance and of the relevant associations, relate the copy to the original.
- step 5. Repeat steps 2, 3, 4 until no instances are placed in the next layer (the repetition is no longer limited by the number of layers in the pattern due to possible loops)

A typical use of a looping pattern is for performing a **transitive closure** along a metamodel association. Transitive closure is directly supported in the textual languages [12, 14], but no other graphical pattern languages [7, 10, 11] support it.

Fig.10 shows an example of a looping pattern. It is the last statement in the class to database transformation program and performs a recursive “flattening” of the class diagram – adding indirect attributes (columns) to a class (table), whose direct attributes have another class as a type. In this example actually a transitive closure on the *attribute* association is performed. The only loop in the pattern is formed by the *type* association from *a2*. The *type* edge can lead either to *t* or to *c2*, the situation is distinguishable because they are of different classes. Both of the edges have the OPT cardinality constraint. During the pattern match, just one of these possible continuations will occur (because an *Attribute* always must have a *type*). If no *type* leads back to a *c2*, then looping along this path is finished. If all the looping is finished, a large instance tree (it is indeed a tree due to instance copying, except the “terminal” instances of *t*) with the root *Class* as a root and certain number of *Attribute* instances as leaves is built as the result of the match. The tree represents all possible paths via indirect attributes from the given class instance - due to the copying of *c2* and *a2* two paths never join. The leaves have a primitive type – they will be used for columns. However, it should be noted that both leave and non-leave

Attribute instances are assigned to *a2* – they must be sorted out to find leaves only. This tree represents graphically the transitive closure of the *attribute* association.

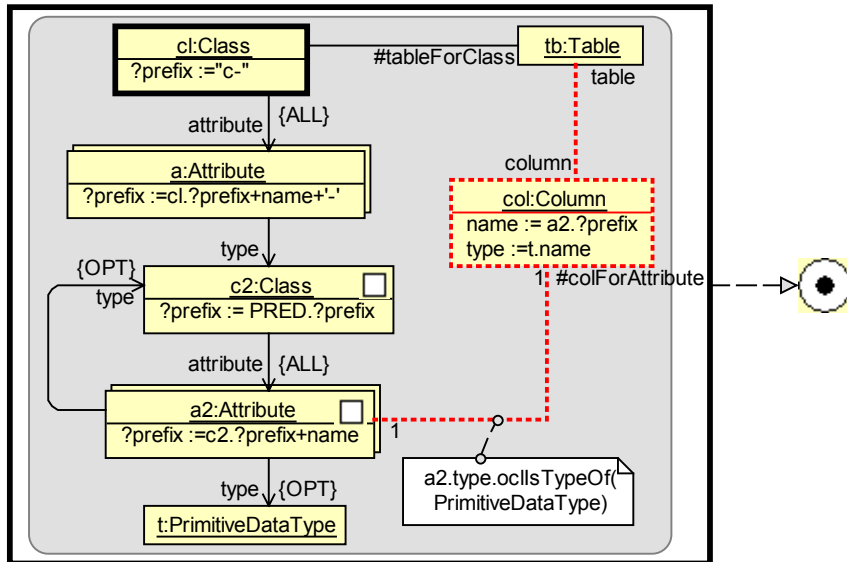


Figure 10. Rule with looping pattern.

Looping patterns typically involve complicated **assignments** to **computed attributes** of metamodel classes.

There are several kinds of computed attributes in MOLA, differing in their scope. Here the most used attributes are **statement-local attributes**, their scope being actions within one statement execution (here – one iteration of the loop, during which the extended match is built). Their values are discarded after the loop iteration is complete. Their main use is for finding various qualified or compound names, typically appearing in MDA tasks.

If the intermediate metamodel is used (here – Fig. 3), rule-local attributes are defined in it for all relevant classes, their **names start with “?”**. Their computation is performed immediately after the pattern match – when the instance DAG is complete. Rule-local attributes are computed in the same order as the match itself was built – starting from the root and moving away from it. If the instance DAG contains copies, these attributes are located in copies – not in the original instances. The assignment statement for a computable attribute in its expression part can contain the value of this attribute in the predecessor node and any values of normal (source) attributes in the current node (these are unqualified). The attribute value from the predecessor node can be qualified either by its instance name or by a **PRED** keyword. The use of PRED is required in cases when a node may have several nodes as predecessors – due to loops in patterns. Each node in the path must have the corresponding assignment statement, otherwise the attribute computation is terminated there. Several attributes may be computed simultaneously in a rule.

The example in Fig.10 contains assignments for the single computed attribute – *?prefix*, which is contained in *Attribute* and *Class*. The computation starts in the

root, where the constant value – the string “c-“ is assigned. When the value is propagated through an *Attribute*, the value of its *name* and the constant “-“ is concatenated to it. The propagation through the *Class* node does not change the value. It is easy to see, that in the result the value of *?prefix* for each leaf of the match tree is the concatenation of all *Attribute* names along that path, separated by “-“ and prefixed by “c-“ – namely the value specified in this task as a *Column* name for indirect attributes. The obtained values are used namely for this purpose – they are used in the assignment for the *name* of the new *Column* instance. A *Column* is built only for those instances assigned to *a2* which are of primitive type (are leaves in the tree) – this is specified by the OCL constraint at the mapping association.

Fig. 10 completes the example transformation program in extended MOLA – the complete transformation consists of Fig. 6, 7, 8 and 10.

We conclude the section with a weaker restriction for patterns, than the distinguishability by associations. Namely, if in a pattern an association with the same role name can lead to several nodes, these nodes must be **K-distinguishable** – their neighborhood of order $\leq K$ (in the sense of directed graphs) must contain **mutually exclusive** elements – different local constraints, different mandatory (just one) associations, mandatory associations leading to different classes etc. The distinguishability by associations can be considered to be 0-distinguishability.

For most MDA examples – e.g., more complicated versions of the example in this paper, and many similar ones, typically K is 1 or 2. In order to use patterns with K -distinguishable elements, a **look-ahead** (in the instance set, but along the pattern edges) not longer than K has to be included in the matching step 3.

6. Conclusions

The paper describes the basic principles of the graphical model transformation language MOLA. There are two innovative elements in MOLA. One is natural combination of simple control structures with pattern based rules. The other one is the powerful pattern mechanism supporting variable cardinality and looping patterns, thus enabling transitive closure in patterns and simplifying even more the control structure of the language. The complete language MOLA is tested on several real world MDA examples, such as converting statecharts to FSM and realistic class-to-database transformation including class inheritance, transformation of business process models to workflows etc. The results show that in most cases more compact and readable rule definitions have been obtained, when compared to e.g., pure recursive style in [10]. The extended patterns permit to strike a right balance between complexity of rules and control structures governing them, thus providing the required transformation readability. Simple recursions, typical e.g., to statechart flattening job, can be specified in a readable way using the WHILE loop in basic MOLA.

The extended patterns, though more complicated than patterns of basic MOLA, are defined in a way to permit also an efficient implementation.

Acknowledgements

Authors of the paper are grateful for valuable discussions and comments provided by their colleagues at the IMCS Modeling and Model Transformations seminar. This research was partially funded by Science Council of Latvia under the project Nr.02 0002.

References

- [1] OMG: MDA Guide Version 1.0.1, <http://www.omg.org/docs/omg/03-06-01.pdf>
- [2] OMG: Unified Modeling Language: Superstructure. Version 2.0 (Final Adopted Specification). <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02> (2003)
- [3] Booch G., Jacobson I., Rumbaugh J. The Unified Modeling Language. Reference Manual, Addison-Wesley, 1999.
- [4] OMG: Meta Object Facility (MOF) Specification. Version 1.4. <http://www.omg.org/cgi-bin/doc?formal/2002-04-03>
- [5] OMG: Request For Proposal: MOF 2.0/QVT. OMG Document ad/2002-04-10, <http://www.omg.org/cgi-bin/doc?ad/2002-04-10>
- [6] Bettin J. Ideas for a Concrete Visual Syntax for Model-to-Model Transformations. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, USA, October 2003.
- [7] QVT-Merge Group. MOF 2.0 Query/Views/Transformations RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, <http://www.omg.org/cgi-bin/doc?ad/2004-04-01>
- [8] Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, <http://www.omg.org/cgi-bin/doc?ad/2003-08-07>
- [9] Interactive Objects Software GmbH, Project Technology, Inc. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-11, <http://www.omg.org/cgi-bin/doc?ad/2003-08-11>
- [10] Willink E., 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.
- [11] Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, November 2003.
- [12] DSTC/IBM/CBOP. MOF Query/Views/Transformations RFP, Second Revised Submission. OMG Document ad/2004-01-06, <http://www.omg.org/cgi-bin/doc?ad/2004-01-06>
- [13] Celms E., Kalnins A., Lace L. Diagram definition facilities based on metamodel mappings. Proceedings of the 18th International Conference, OOPSLA'2003, Workshop on Domain-Specific Modeling, Anaheim, California, USA, October 2003, pp. 23-32.
- [14] Kleppe A., Warmer J., Bast W. MDA Explained. The model driven architecture: practice and promise. Addison-Wesley, 2003.

MOLA Language: Methodology Sketch

Audris Kalnins, Janis Barzdins, Edgars Celms

University of Latvia, IMCS, 29 Raina boulevard,
Riga, Latvia

{Audris.Kalnins, Janis.Barzdins, Edgars.Celms}@mii.lu.lv

Abstract. The paper demonstrates the MOLA transformation program building methodology on an example. The example shows how to obtain self-documenting model transformation programs in MOLA by means of standardized comments. The proper usage of loops in MOLA is also discussed.

1. Introduction

There is no doubt that model transformation languages and tools are the key technology elements for MDA. Due to OMG initiatives, currently there are several proposals for model transformation languages, both as responses to OMG QVT RFP [1,2] or “independent” ones [3,4]. Among the independent languages there is also the MOLA language proposed by the authors of this paper [5,6]. Each of the proposed languages has its strengths and weaknesses, there is no clear adoption of any of the languages in the MDA community yet. The main distinguishing feature of MOLA is a natural combination of traditional structured programming in a graphical form with pattern-based rules. Especially, the rich loop concepts in MOLA enable the iterative style for transformation definitions, while most of other languages rely on recursion. A more detailed comparison of MOLA to other MDA languages is provided in [5,6].

Transformation languages have two essential requirements. On the one hand, transformations should be easy to write – to implement the intended algorithms in an adequate manner. On the other hand, transformations should be easy readable by much broader user community – those wanting to apply a transformation to their models in a safe and controllable manner. Transformation readability has been one of the design goals of MOLA.

The only way to evaluate different languages is to compare them on generally accepted benchmark examples. Since transformation development actually is a completely new domain, there are no proven methodologies and design patterns, as there are in more classical domains.

The goal of this paper is to analyze the MOLA language from the above-mentioned perspectives. Using one of the standard benchmark examples - Class to Relational Database transformation, it will be shown how the readability can be achieved in MOLA, including also standardized comments. Transformation design methodology will also be sketched, especially the proper use of loops. Certainly, the paper does not claim to provide a methodology for MOLA-based system design, just some advices how the model transformations themselves should be programmed in MOLA.

2. Brief Overview of MOLA

This section gives a very brief overview of the MOLA language. A more complete description of MOLA is to be found in [5,6]. Authors also hope that the example in section 4 will help significantly to understand the language.

A MOLA program, as any other transformation program, transforms an instance of **source metamodel** into an instance of **target metamodel**. These metamodels are specified by means of UML class diagrams (MOF compliant).

More formally, source and target metamodels are part of a transformation program in MOLA. But the main part of MOLA program is one or more MOLA diagrams (one of which is the main). A **MOLA diagram** is a sequence of **graphical statements**, linked by arrows. It starts with a UML start symbol and ends with an end symbol.

The most used statement type is the **loop statement** – a bold-lined rectangle. Each loop statement has a **loop head** – a special statement (grey rounded rectangle) containing the **loop variable** and the **pattern** – a graphical condition defining which instances of the loop variable must be used for iterations. The pattern contains **elements** – rectangles containing `instance_name:class_name` – the traditional UML instance notation, where the class is a metamodel class. The loop variable is also a special kind of element, it is distinguished by having a **bold-lined** rectangle. In addition, a pattern contains metamodel associations – a pattern actually corresponds to a metamodel fragment (but the same class may be referenced several times). Pattern elements may have attribute constraints – OCL expressions. Associations can have cardinality constraints (e.g., NOT). The semantics of this loop statement (called the **FOREACH** loop) is natural – the loop is executed **once** for each instance of the loop variable, where the condition is true – the pattern elements can be matched to existing instances and attribute constraints are true on these instances. There is also another kind of loop – **WHILE** loop, which is denoted by a 3-d frame and continues execution while a valid loop variable instance can be found (it may have also several loop heads). Loops may be nested to any depth. The loop variable (and other element instances) from an upper level loop can be referenced by means of a **reference** symbol – the element with @ prefixed to its name.

Another widely used statement in MOLA is **rule** (also a grey rounded rectangle) – a statement consisting of pattern and actions. These actions can be **building actions** – an element or association to be built (denoted by red dotted lines) and **delete actions** (denoted by dashed lines). In addition, an attribute value of an element (new or existing) can be set by means of **attribute assignments**. A rule is executed once – typically in a loop body (then once for each iteration). A rule may be combined with a loop head, in other words, actions may be added to a loop head, thus frequently the whole loop consists of one such combined statement.

To call a subprogram, a **call statement** is used (possibly, with parameters - instances in the same reference notation). A subprogram, in turn, may have one or more **input parameters**. The same loop statement notation can be used to denote control branching – with a guard statement instead of loop head.

In this paper an additional MOLA element – **standardized comments** are introduced. These comments are text boxes associated to a MOLA diagram (its start symbol) and its statements. Comments can contain any text, but references to loop variables are shown in **bold**, and references to other elements – in *italic*. The comment for

the whole diagram is intended to describe its informal pre- and post-conditions. The comments to separate statements are meant to describe their goal in an informal way.

Our goal is to make a MOLA program self-documenting, i.e., so easy readable that any one can ascertain that a MOLA program actually performs the intended transformation. Our experience shows that a well-written MOLA program with such comments is self-documenting really and we hope that the example in section 4 confirms this.

3. The Benchmark Example

The most popular transformation benchmark example – transformation of UML class model to relational database is used here. There are several versions of this example originally proposed by OMG – nearly each paper uses its own version. We use here the version from the QVT-P proposal [1].

The source metamodel is a significantly simplified fragment of the UML class diagram metamodel, it is visible in the upper part of Fig. 1. The target metamodel is a simplified relational database metamodel, it is given in the lower part of Fig.1. Next, the precise informal specification of the transformation task will be given (since there are some minor deviations from [1] due to some inconsistencies in it).

Any persistent *Class* (with *kind*="persistent") must be transformed into a database *Table*. In addition, a (primary) *key* is built for this *table*. *Attributes* of the class, which have a primitive data type, must be transformed into *columns* of the corresponding *table* (we assume here that types in UML and SQL coincide). *Attributes* whose type is a *class*, must be "drilled-down": primitively-typed attributes of this new class are added as columns to the table for the original class. Class-typed attributes are processed as before. The process is repeated until no new columns can be added to the table for the original class. In other words, a transitive closure is performed, which finds all "indirect" attributes of the class. The added columns have compound names consisting of all attribute names along the path. One special issue must be reminded here: several attributes of a class may have the same class as a type, in this case the added columns are duplicated for each of them (they have unique names!). In other words, any path leading to a primitively-typed attribute results into a separate column.

For primitive-typed "direct" attributes of a persistent class with *kind*="primary", the corresponding columns are included in the relevant (primary) key. An association (with multiplicities ignored, but direction taken into account) is transformed into a foreign key for the "source end" table. The same table is extended with columns corresponding to columns of the (primary) key at the target end. For both "primary" and "foreign" columns their *kind* is set accordingly.

4. MOLA Solution

4.1. Building the Workspace Metamodel

The first step in building a MOLA program (transformation) is to define the **workspace** metamodel (see Fig.1). This metamodel includes both the **source metamodel**

(light yellow classes – the upper part) and the **target metamodel** (dark yellow classes – the lower part). Both metamodels are taken from the problem domain without modifications – they describe the corresponding input data (source model) and the result (target model) of the transformation.

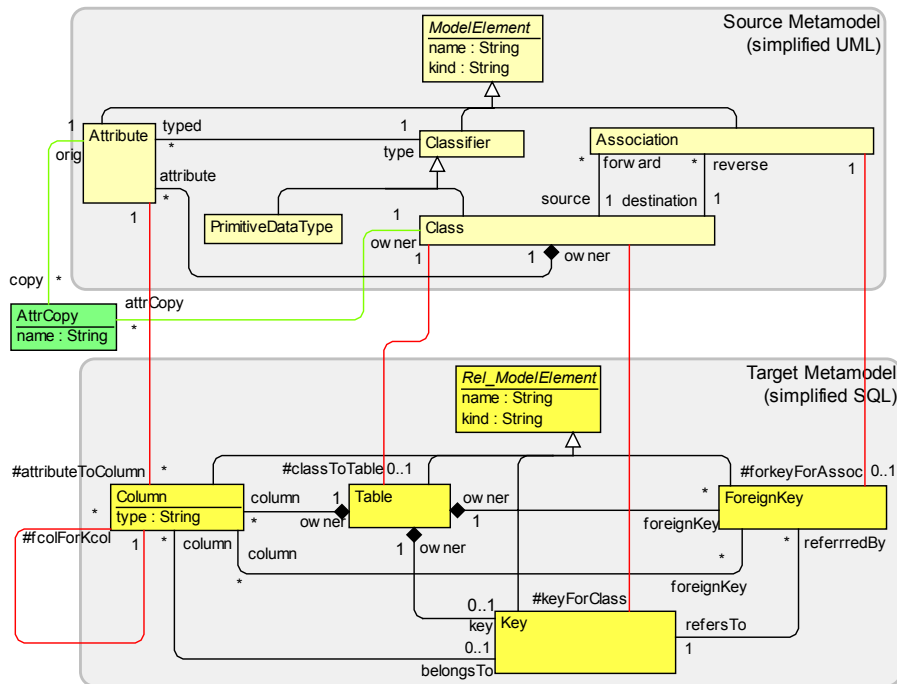


Fig. 1. The workspace metamodel

However, some elements typically are added to the workspace metamodel. First, there are **mapping associations** – associations linking classes in the source and target metamodels (red lines in Fig. 1). They serve two different purposes – on the one hand, they document relations between the corresponding source and target elements of the transformation (e.g., *Class* and *Table*, *Attribute* and *Column*, etc.) and thus enable the **traceability** at the instance level (which *Table* was obtained from which *Class*). On the other hand, they have a technical role in MOLA – after being built by one rule, they frequently are used in patterns of subsequent rules. It is recommended in MOLA to start the role names of mapping associations with “#”.

Another possible metamodel extensions are temporary classes – *AttrCopy* in the example and temporary associations (associations linking *AttrCopy* to base classes of the metamodel, all temporary elements are in green color in the example). This temporary class will be used to store copies of an attribute – indirect attributes. Temporary elements serve as a “workspace” for transformations, they have instances only during the transformation execution, and they are not supplied at input and are discarded at output. Base metamodel classes may have also temporary attributes added (attributes which have value only during the transformation execution) – this example does not use them.

4.2. MOLA Program Implementing the Transformation

The transformation is specified in MOLA by means of one main diagram (Fig. 2) and four subprograms (subdiagrams) – Fig. 3 to 6. The implemented transformation corresponds to its informal specification in a quite straightforward manner. The specification requires to perform a transitive closure – to find all indirect attributes of a class, and with duplicates included (therefore attribute copying is required). We use an idea that each instance of indirect attribute actually is a path in the “instance graph” from the “root class” to an attribute. The iterative algorithm (Fig. 3) for finding all indirect attributes of a class is inspired by the well known algorithm for finding all paths from a node.

All diagrams (Fig. 2 to 6) are annotated by standardized comments and, we hope, will require no other explanations.

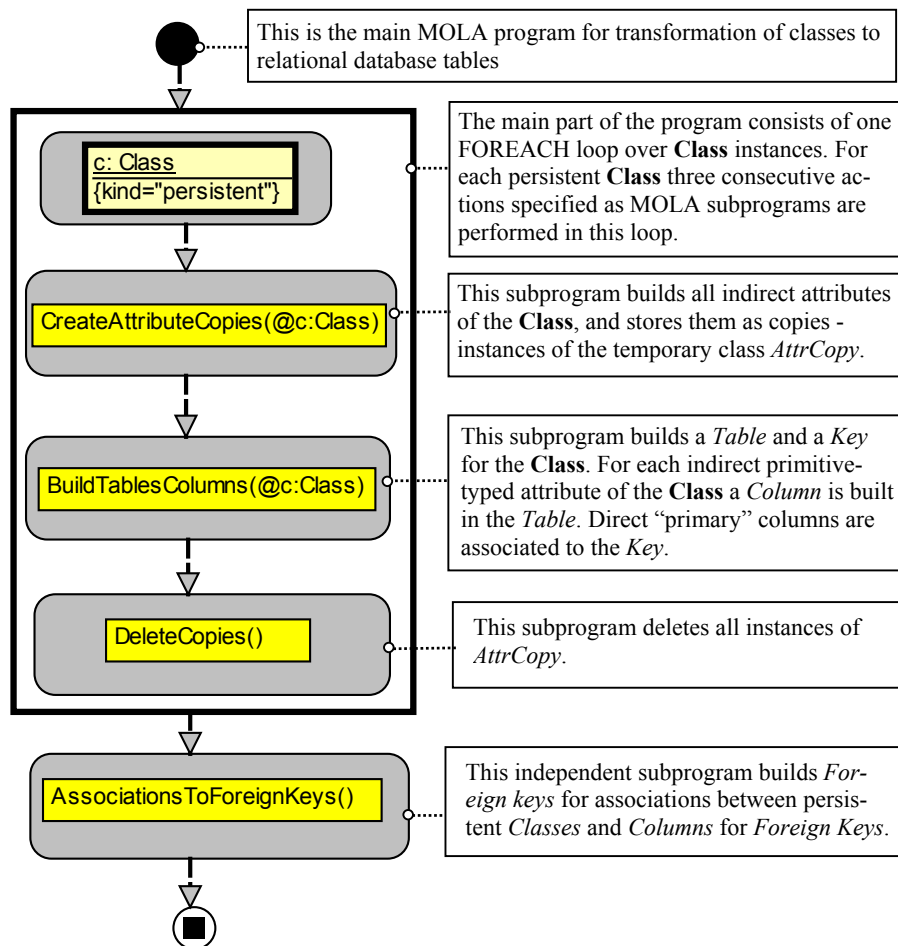


Fig. 2. The main diagram of the transformation

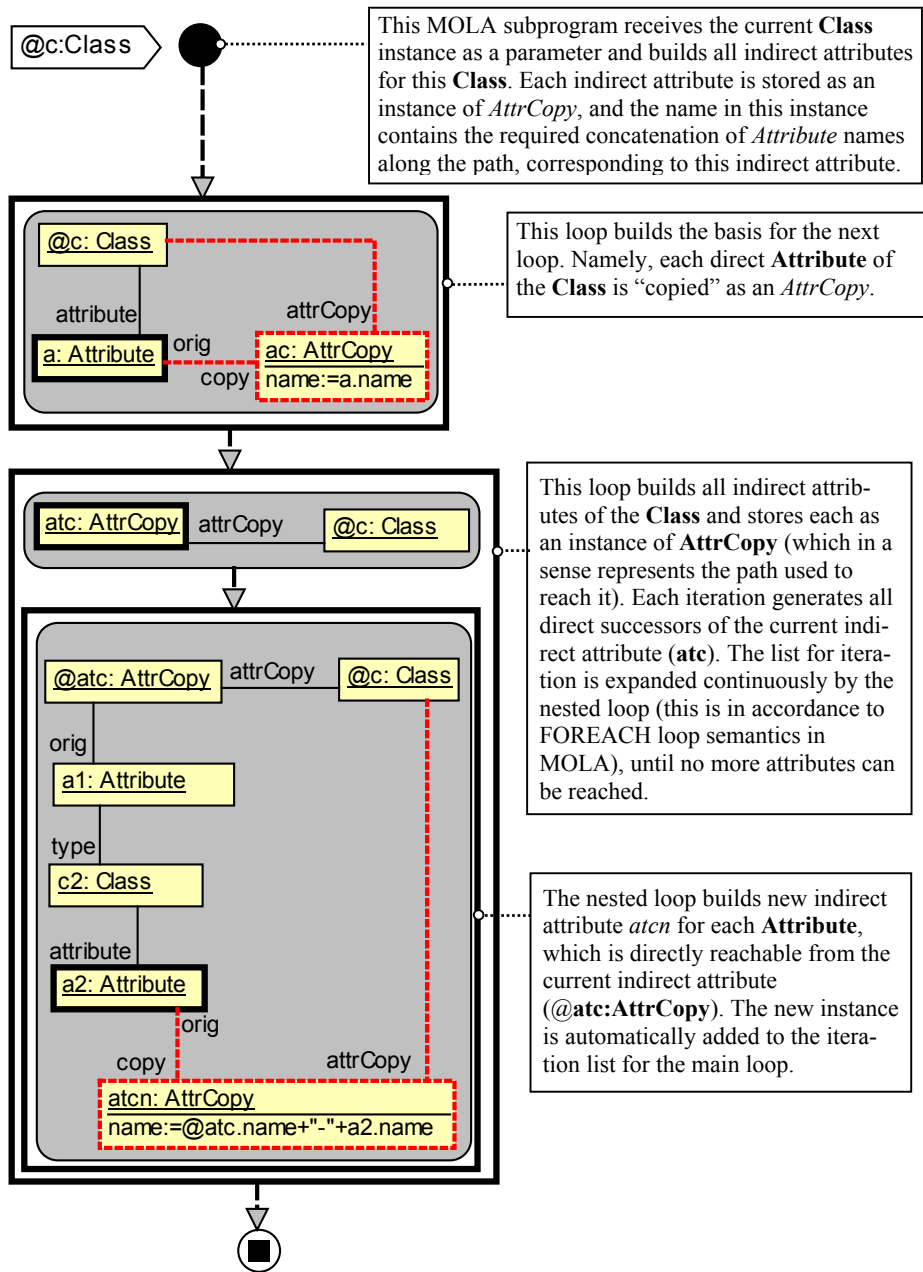


Fig. 3. Subprogram *CreateAttributeCopies*

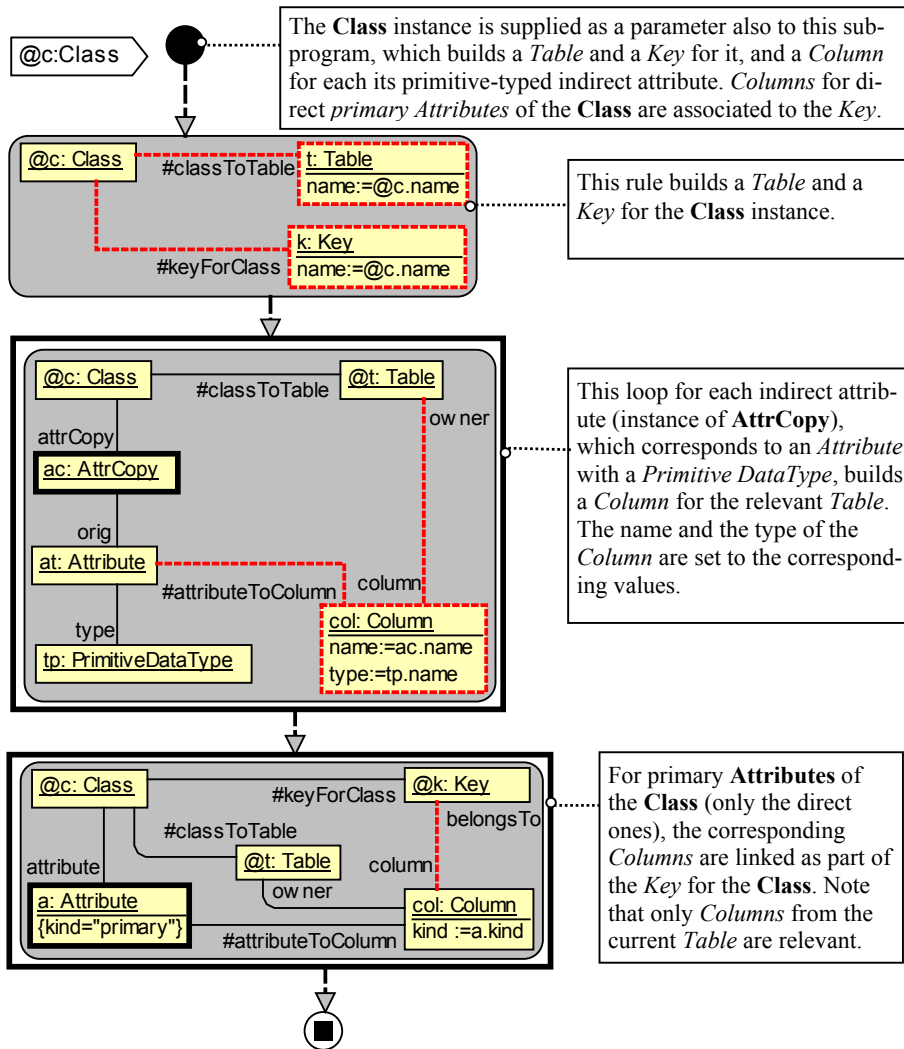


Fig. 4. Subprogram *BuildTablesColumns*

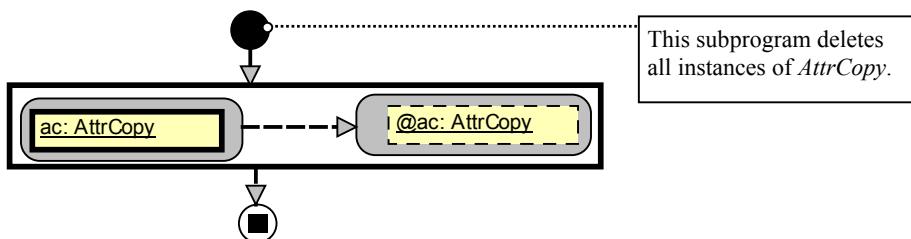


Fig. 5. Subprogram *DeleteCopies*

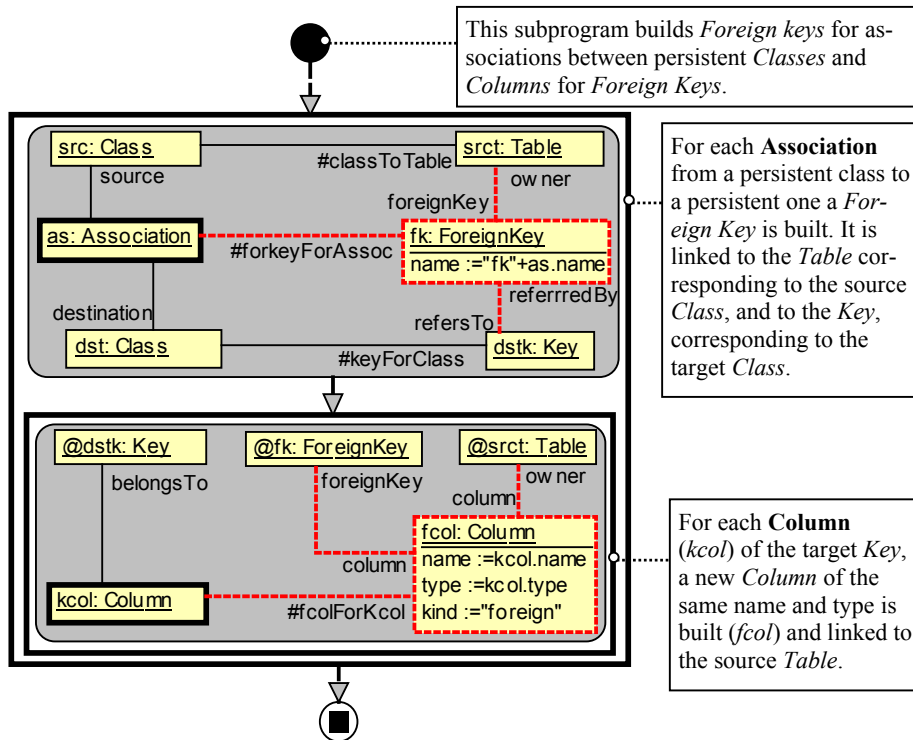


Fig. 6. Subprogram *AssociationsToForeignKeys*

5. Some Remarks on MOLA Methodology

Using the previous example as a basis, some elements of transformation program design methodology in MOLA will be provided.

Firstly, in MOLA, like most model transformation languages, a top-down approach should be used. Namely, the most coarse-grained elements of the source model (in our case, *Classes*) must be processed first. Only in this way, the mapping associations built for transforming them (e.g., *#classToTable*) can be used in patterns for transformations of contained elements (*Attributes*) or related ones (*Associations*). Thus, there is no need to repeat higher-level constraints (e.g., {kind="persistent"}) at lower level.

Since the main "processing element" in MOLA is loop, a correct design of loops is of prime importance. Typical algorithm steps frequently contain statements "for each A do ..." and FOREACH loops (the loops used in section 4) should be used to implement these steps. Besides being a natural formalization of the step, FOREACH loops are easier to use, because their semantics already includes "iterate once for each element ..." and there is no need for marking instances already processed. In most cases, the possible infinite loop problem is also eliminated due to a finite set of instances of the class (all loops in Fig. 2, 4, 5, 6). However, MOLA FOREACH loop can have its instance set replenished dynamically (the second loop in Fig.3), in this

case additional considerations should be used (during the building of indirect attributes we assume that no class is used as the type of its own indirect attribute). On the contrary, WHILE loops should be used for steps which are a mix of iteration and recursion (such as the moving of transition ends during the flattening of a UML state-chart in [5]), there the use of several loop heads per loop enables a natural and compact at the same time formalization for this kind of algorithm step.

Yet another important design element in MOLA is the selection of loop variables so that patterns in loop heads do not become complicated. Especially, for nested loops the deepest repeating element must be used. Use of referenced elements from upper level loops helps to simplify patterns in nested loops (see the nested loop in Fig. 6).

Actually, there are more design hints in MOLA and eventually “GOF-style design patterns” could be defined, but this is a topic of another paper.

And finally, nearly any non-standard transformation element can be described in MOLA using low-level facilities such as temporary classes and associations.

6. Conclusions

We have shown that by selecting an appropriate design style, the transformation programming in MOLA is relatively simple, as it is demonstrated by the complete example in section 4.

By adding standardized comments (even quite short ones, as in section 4), the readability of MOLA programs really reaches the level of self-documenting – one of main goals for the design of MOLA.

The implementation of MOLA is expected not to be very complicated due to relatively simple constructs in it. The implementation efficiency is also expected to be high enough – if the programming guidelines from section 5 and some more natural assumptions are observed, typical pattern matching problems of graph transformations are avoidable in MOLA.

References

1. QVT-Merge Group. MOF 2.0 QVT RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, <http://www.omg.org/cgi-bin/doc?ad/2004-04-01>
2. Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, <http://www.omg.org/cgi-bin/doc?ad/2003-08-07>
3. 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
4. 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
5. Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDAFA 2004, University of Linköping, Sweden, 2004, pp.14-28. (see also http://melnais.mii.lu.lv/audris/MOLA_MDAFA.pdf)
6. Kalnins A., Barzdins J., Celms E. Basics of Model Transformation Language MOLA. Proceedings of WMDD 2004, Oslo, 2004, <http://heim.ifi.uio.no/~janoa/wmdd2004/papers/kalnins.pdf>

Efficiency Problems in MOLA Implementation

Audris Kalnins, Janis Barzdins, Edgars Celms
University of Latvia, IMCS, 29 Raina boulevard,
Riga, Latvia
{Audris.Kalnins, Janis.Barzdins, Edgars.Celms}@mii.lu.lv

Abstract. Efficiency of pattern matching for MOLA model transformation language is analyzed in the paper. A virtual machine and pattern matching procedure based on it is proposed, which takes into account the specific requirements for efficient pattern matching in MOLA. On the basis of a typical MDA example it is shown that the proposed solution is optimal and the conclusions are generalized to typical MOLA programs.

1. Introduction

Model transformation languages are the main logical support for model driven software development (MDSO). Due to OMG initiatives, currently there are several proposals for model transformation languages, both as responses to OMG QVT RFP [1,2] or “independent” ones [3,4,5]. Among the independent languages there is also the MOLA language proposed by the authors of this paper [6,7,8]. The main distinguishing feature of MOLA is a natural combination of traditional structured programming in a graphical form, especially, the rich loop concepts with pattern-based rules.

A model transformation is applied to a source model – an instance set corresponding to the source metamodel and produces target model, corresponding to the target metamodel. The source model can be treated as an instance graph for the source metamodel – it consists of typed nodes – instances of metamodel classes and edges – links corresponding to metamodel associations.

Model transformation languages – be they textual or graphical – contain rules based on pattern matching and control structures which govern the execution order of rules. It should be noted that facilities for defining pattern matching are quite similar from the semantics point of view for most of model transformation languages, including MOLA. Since pattern matching is performed in the source instance graph, which can be of quite substantial size, problems typical to graph transformation languages may appear, especially those of pattern matching efficiency. These problems e.g., for the graph transformation language Progress are discussed in [9]. The proposed solution there is an appropriate programming style.

What refers to model transformation languages, the most thorough efficiency analysis has been done for GReAT language ([4], and especially, [10]). The main result there is that pattern matching can be made sufficiently efficient by passing already matched nodes from one pattern to another (“pivoting” and reusing). Certainly, this result relies significantly on the specific control structures (data flows, input and output ports) and semantics of GReAT.

In this paper we try to solve the pattern matching problem for MOLA, relying on its specific control structures – loops. MOLA loops contain loop variables – pattern elements which must be matched to all possible relevant nodes in the source graph. At the same time the other pattern elements, according to MOLA semantics, must just have any one feasible match. There is also an observation (discussed in the paper to some detail) that in a correctly built MOLA program the match of any pattern element tends to be deterministic – thus typically leading to a simplified backtracking during the match. Another important fact is that nested loops in MOLA contain references to

already matched elements in upper level loops. All this has led to the necessity to build a specific matching procedure for MOLA, which could be optimal namely in these circumstances. The paper proposes such a procedure, which in turn is based on a virtual machine for accessing source model elements and the MOLA program itself. In order to ascertain that the proposed solution is indeed efficient for MOLA, a typical benchmark example – class-to-relational database transformation (a MOLA program for which was proposed already in [8]) is analyzed from the complexity point of view. It is shown that both the program and the proposed MOLA implementation is optimal for this example – the number of virtual machine operations (which themselves are simple) is the best possible – proportional to the source model size.

The main conclusion of the paper is that the example reveals a typical situation for MOLA and there are no efficiency problems expectable if adequate programming style and adequate pattern matching is used. Thus the proposed matching procedure and the sketch of virtual machine indeed can serve as the basis for MOLA implementation.

Sections 3 and 4 of the paper describe the virtual machine and the pattern matching respectively. The example program is provided in section 5 and its performance analysis in section 6.

2. Brief Overview of MOLA

This section provides a very brief overview of MOLA syntax and semantics. A more complete description of MOLA language is given in [6,7].

A MOLA program, as any other transformation program, transforms an instance of source metamodel into an instance of target metamodel. These metamodels are specified by means of UML class diagrams (MOF compliant).

More formally, the combined source and target metamodel is part of a transformation program in MOLA. To avoid any confusion, classes in this combined metamodel will be called **metaclasses** in the paper. But the main part of MOLA program is one or more **MOLA diagrams** (one of which is the main). A MOLA diagram is a sequence of graphical **statements**, linked by arrows. It starts with a UML start symbol and ends with an end symbol.

The most used statement type is the **loop statement** – a bold-lined rectangle. Each loop statement has a **loop head** – a special statement (grey rounded rectangle) containing the **loop variable** and the **pattern** – a graphical condition defining which instances of the loop variable must be used for iterations. The pattern contains **elements** – rectangles containing `instance_name:class_name` – the traditional UML instance notation, where the class is a metaclass. The loop variable is also a special kind of element, it is distinguished by having a bold-lined rectangle. In addition, a pattern contains metamodel **associations** – a pattern actually corresponds to a metamodel fragment (but the same class may be referenced several times). Pattern elements may have attribute constraints – OCL expressions. Associations can have cardinality constraints (e.g., NOT). The semantics of this loop statement (called the **FOREACH** loop) is natural – the loop is executed once for each instance of the loop variable, where the condition is true – the pattern elements can be matched to existing instances and attribute constraints are true on these instances. The valid instance set for the loop variable may be replenished during the loop execution – these additional instances are also used for iterations, but certainly, each instance only once. There is also another kind of loop – WHILE loop, which is denoted by a 3-d frame and continues execution while a valid loop variable instance can be found (it may have also several loop heads). Loops may be nested to any depth. The loop variable (and

other element instances) from an upper level loop can be referenced by means of reference symbol – the element with @ prefixed to its name.

Another widely used statement in MOLA is rule (also a grey rounded rectangle) – a statement consisting of pattern and actions. These actions can be building actions – an element or association to be built (denoted by red dotted lines) and delete actions (denoted by dashed lines). In addition, an attribute value of an element (new or existing) can be set by means of attribute assignments. A rule is executed once – typically in a loop body (then once for each iteration). A rule may be combined with a loop head, in other words, actions may be added to a loop head, thus frequently the whole loop consists of one such combined statement.

To call a subprogram, a call statement is used (possibly, with parameters - instances in the same reference notation). A subprogram, in turn, may have one or more input parameters. The same loop statement notation can be used to denote control branching – with a guard statement instead of loop head.

3. Basic Principles of MOLA Implementation

A detailed description of MOLA implementation is quite lengthy (the same way as implementation of any model transformation language) and is not the goal of this paper. Here we will provide only some elements of this implementation – those which are necessary to convince that an efficient implementation of MOLA is possible. As for any of the transformation languages, the most difficult part is the implementation of pattern matching. In turn, the most critical use of patterns in MOLA is within loop statements, therefore we will concentrate on the implementation of loop statements, mainly the FOREACH loop – the most used one. The implementation of all other MOLA statements is relatively straightforward, and no special efficiency gains can be obtained there, therefore we hope the reader will believe that complete MOLA can be implemented in the way sketched here.

To implement a transformation language, some sort of **model/metamodel repository** is required. In this paper we assume that such a repository is available – it can be a properly defined SQL database or a special repository based on hash tables. All we will need from this repository here is that some natural queries (to be described later) can be executed in a "nearly-constant" time with respect to the size of the model data. Certainly, a proper design of such repository for MOLA is not trivial (compare, e.g., to [11]) and could be a topic of another paper.

We assume that the same repository contains also the MOLA program to be executed. Again, the exact format for the program storage will not be provided, except for some sketch of the pattern storage – the most used part. Some queries for retrieving the program elements will also be described. In totality, all the queries mentioned here form a **virtual machine** for MOLA execution. Certainly, this virtual machine must contain more functionality (e.g., for creating or updating model elements), but we hope that the reader will believe that the sketch of the machine provided in the paper can be properly extended, while preserving the requirements for efficiency to be described later. The rest of the section will be devoted to the sketch of the MOLA virtual machine.

We start with the pattern storage and queries for it. Since a MOLA pattern – a slightly modified fragment of a UML class diagram – actually is an undirected graph, it could be stored in a quite straightforward way. However, in order to simplify the description of pattern matching algorithm used for MOLA, we will use another representation, also based on graph theory. Thus, we assume that an “**optimizing compiler**” is available for MOLA, which builds this representation.

Fig. 1 presents a typical MOLA pattern in a FOREACH loop (actually part of Fig. 7).

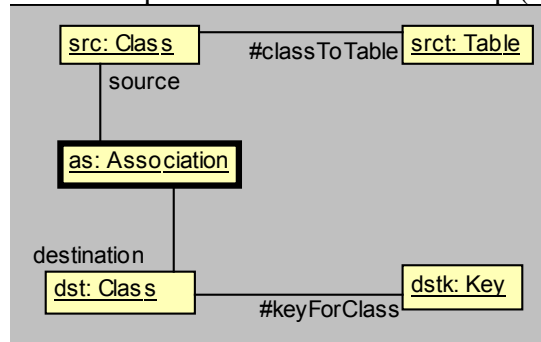


Fig. 1. Pattern example

When viewed from the loop variable (the node *as: Association*), it can be treated as a directed rooted tree with two branches. We want to code this tree as an ordered list of edges/nodes, in a depth-first manner, starting with the root. The list for the example will contain 5 elements:

- the root node *as: Association*
- association *source* leading to *src: Class*
- association *#classToTable* leading to *srct: Table*
- association *destination* leading to *dst: Class*
- association *#keyForClass* leading to *dstk: Key*

The described order will be especially fit for matching the pattern to the instance set: we start with the root (an instance of the metaclass *Association*), then proceed via the link *source* to an instance of *Class* etc.

When the pattern is not a tree (as in the schematic example in Fig. 2), the compiler selects a spanning tree from the root and codes it as already shown. The basic part of the pattern code from Fig.2 could be the following: $A, (A,rb,B), (B,rc,C), (B,rd,D), (A,re,E)$. The edges not in the tree will be coded by a special sublist at relevant nodes, so that each such edge goes “backwards” in the main list. For example, the following sublist of edges is attached to the node E: $(E,sc,C), (E,sd,D)$ in Fig. 2. The selected coding reduces the checking of the existence of relevant “crosslinks” to a simple additional constraint during the pattern matching.

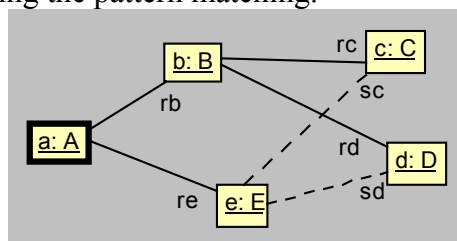


Fig. 2. Another pattern example

To put it more formally, the pattern code is the list *Pattern*, where each element is a structure consisting of:

- assoc* – the association
- sourceIndex* – index of the source node in the list
- metaClass* – the metaclass instance of which is sought
- constraint* – the local OCL constraint on attributes of the metaclass
- crossList* – the sublist of “crosslinks”
- instanceSet* – a pointer to a “restriction set”, necessary at runtime for nested loops.

The virtual machine must contain one main operation for patterns:

getPatternElement(int i) .

The root element (the loop variable) is coded as the 0-th element of list (with less fields filled), and is available via

getPatternRoot() .

Some more data are generated by the compiler for reference elements of the pattern, they will be explained in the next section.

Now the operations of the virtual machine for querying “model elements” – instances of metaclasses and associations in the repository are described. As it was already mentioned, the repository contains the current model to be processed – instances of metaclasses and links – instances of associations. Thus the model actually is also a graph – a graph of instances and links. We assume that the repository also supports a “list-like” behavior – you can query specified kind of instances and get them one by one – an SQL cursor-like behavior. All operations are assumed to be “static” – they remember the previous calls. The simplest required virtual machine operation is

getNext(metaClass mcl).

This operation returns the next instance of metaclass *mcl* upon each call (it does not matter how many instances of this operation are used in the program for the virtual machine). The **null** constant is returned when there are no more instances.

The operation most used for implementing pattern matching is

getNextByLink(association assoc, instance sourceInst, metaclass mcl).

This operation returns one by one the instances of metaclass *mcl*, which can be reached by links corresponding to *assoc* from the fixed instance *sourceInst*. *Null* is returned in case of absence. There is also an initialization for it, with similar parameters

initializeGetNextByLink(association assoc, instance sourceInst, metaclass mcl)

Two more auxiliary operations are:

eval(instance inst, oclExpression expr) – evaluate a local constraint on attributes

checkLink(instance sourceInst, instance targInst, association assoc) – check whether a link of required type is between these instances.

These operations are sufficient for programming the pattern matching for top-level FOREACH loops. There is no doubt that at least for an SQL-based repository they are of “nearly-constant” complexity with respect to the repository size (i.e., growing much slower than the repository size).

Two similar more special operations are required for nested loops (using references):

getNextFromSet(metaClass mcl, set instSet) and

getNextByLinkFromSet(association assoc, instance sourceInst, metaclass mcl, set instSet) .

These operations actually provide relevant instances from the specified set. Corresponding initializations are also available.

This completes the description of virtual machine, used for pattern matching.

4. Pattern Matching in MOLA

A Java-style pseudocode is provided for the main part of the implementation schema - the pattern matching. Pattern matching is required for all kinds of MOLA statements, but here we consider only the most used and also the most sensitive from the performance point of view statement – the FOREACH loop. Besides some well-known Java-like constructs, the pseudocode will contain only calls to MOLA virtual machine operations, defined above. The specific requirements for matching, outlined already in the introduction, are all taken into account in matching procedure design.

The matching procedure uses a runtime list *BoundInstances*, with the same length as the *Pattern*, which contains metaclass instances matched to the corresponding pattern elements. The 0-th element of it contains the current instance of the loop variable (the root).

We start with the simplest case – a top-level loop (having no parameters). Then all instances of the loop variable metaclass must be browsed and for each such instance the pattern must be matched. If a valid match (according to MOLA semantics, any of them, e.g., the first, if, in fact, there exists more than one) is found, the actions of the iteration are performed, using the matched instances.

The main idea is quite simple. We try to advance along the *Pattern* list, by finding on each step an instance of the metaclass required by the current *Pattern* element. An appropriate instance is sought, using the already known source instance and browsing instances reachable from it via links of the specified type (*Pattern[i].assoc*). If an instance is found which satisfies constraints, it is stored in *BoundInstances* and we advance to the $i+1^{\text{st}}$ pattern element. If no valid instance is found this way, we backtrack to the previous pattern element in the list – select a new instance for it. It should be noted that the specified backtracking strategy is not optimal – it is chosen to simplify the pseudocode and its complexity evaluations for "good cases". For example, if a pattern element has no crosslinks, we could backtrack to the pattern element with index equal to the current *sourceIndex*. However, these optimizations are not essential for our performance evaluations, since actually only a trivial backtracking is typically used in MOLA. If backtracking reaches the loop variable (root), we start a new pattern matching for a new instance of it.

```
lv = getPatternRoot( );
while ( lv_inst = getNext(lv.metaClass) ) // browse instances of the loop variable
{
  if ( !eval(lv_inst, lv.constraint) ) // if the eval operation returns false (constraint fails)
    continue; // then start next iteration
  BoundInstances[0] = lv_inst; // store the current loop variable instance
  failed = false; // failed is a boolean tag signaling match exhaustion
  mustInitialize = true; // getNextByLink must be initialized – a new context is started
  i = 1; // start matching
  while ( i < Pattern.Size )
  {
    pattern_element = getPatternElement(i);
    if ( mustInitialize ) // initialize local search if it is not backtracking
      initializeGetNextByLink(pattern_element.assoc,
        BoundInstances[pattern_element.sourceIndex],
        pattern_element.metaClass);
    curr_inst = null;
    while ( curr_inst = getNextByLink(pattern_element.assoc,
      BoundInstances[pattern_element.sourceIndex],
      pattern_element.metaClass) ) // take current candidate instance
    {
      if ( validate(curr_inst, pattern_element) ) // validate is a subprocedure checking
        break; // the local constraint and existence of crosslinks
    }
    if ( !curr_inst ) // curr_inst not found, i.e., equal to null, local search is exhausted
    {
      if ( i == 1 ) { failed = true; break; } // no more backtracking possible, select
        // new root instance
      else { i = i-1; mustInitialize = false; continue; } // backtracking must be
```

```

// performed!
}
BoundInstances[i] = curr_inst; // successful match step
i = i+1; // advance to the next step
mustInitialize = true;
}
if ( !failed ) // match successful, BoundInstances contain the result
executeRule();
}

```

It is not difficult to ascertain that the described procedure indeed implements the matching algorithm outlined in the beginning. What refers to the subprocedure *validate*, it is easy to see that it can be implemented directly using *eval* and *checkLink* operations, the number of required steps depends only on the pattern element size.

Nested loops typically contain references to elements in upper level loops. From the point of view of a nested loop, all references have fixed instances during the match, so we will call them **fixed elements** in this section. In principle the same matching procedure could be used for nested loops, with fixed elements playing the role of additional constraints. But this approach is too suboptimal, requiring excessive searches proportional to the model size. Therefore we propose a more optimal approach, where the search space is limited on the basis of fixed elements. For this purpose **restriction paths**, leading from fixed elements to the root in the pattern are also built by the compiler.

Additional preparatory pass is added to the matching procedure. During this pass for each path the **sets of feasible instances** are built. For fixed elements themselves the set consists of just one instance, but for subsequent path nodes the set is determined by the pattern association (i.e., by links corresponding to this association). Finally, a set for root is also found. If two paths have a common node in the pattern, then set intersection is taken at this node. For example, the root is common to all paths, so at least there the intersection will be taken. To implement this principle, sets must be kept separate from paths, therefore the sets are attached to the corresponding elements of *Pattern* (via *instanceSet*). The described set building algorithm can be implemented easily, using the list of restriction paths. The same *getNextByLink* operation is used to retrieve instances to be placed in sets. Certainly, some obvious operations for adding an element to a set and building a set intersection are also necessary. Since the total size of the sets built in this process will be limited by a constant in our performance evaluations (see section 6), there is no need to elaborate more on this set building.

Now a procedure very similar to the one described above can be used for pattern matching (and yield the required performance). The only difference is that *getNextFromSet* is used instead of *getNext* and *getNextByLinkFromSet* instead of *getNextByLink* (and the corresponding initializer is replaced too). These operations select instances only from their set argument. The sets for instance selection (including the root) are those found in the preparatory pass. If a pattern element has no instance set attached (it is not on a restriction path), *getNextByLinkFromSet* behaves the same way as its simple counterpart *getNextByLink*. Thus for those pattern elements, where fixed elements restrict the search space, the restricted search is used, while for others the full search is applied, as in the previous case. The analysis in the next section shows that the proposed principle for building restriction sets is indeed optimal – in some cases the exact required instance set is obtained.

5. Example

The same Class-to-Relational DB example from [8] is used to evaluate the performance of the proposed MOLA implementation. Here we repeat only a very short description of the transformation, a complete description is to be found in [8], the specification originally comes from [1].

Any persistent Class (with kind="persistent") must be transformed into a database Table. In addition, a (primary) key is built for this table. Attributes of the class, which have a primitive data type, must be transformed into columns of the corresponding table. Attributes whose type is a class, must be transitively "drilled-down": primitively-typed attributes of the new class are added as columns to the table for the original class. For primitive-typed "direct" attributes of a persistent class with kind="primary", the corresponding columns are included in the relevant (primary) key. An association (with multiplicities ignored, but direction taken into account) is transformed into a foreign key for the "source end" table. The same table is extended with columns corresponding to columns of the (primary) key at the target end.

Fig. 3 presents the combined metamodel: the upper part is the source – a simplified UML class diagram metamodel, the lower part is the target – a simplified relational database metamodel. In-between are temporary elements. Fig. 4 to 8 present the transformation as MOLA programs, Fig.4 shows the main program, which invokes the subprograms. All programs actually are simple or nested FOREACH loops. The temporary metaclass *AttrCopy* is used to implement the transitive closure (Fig.5).

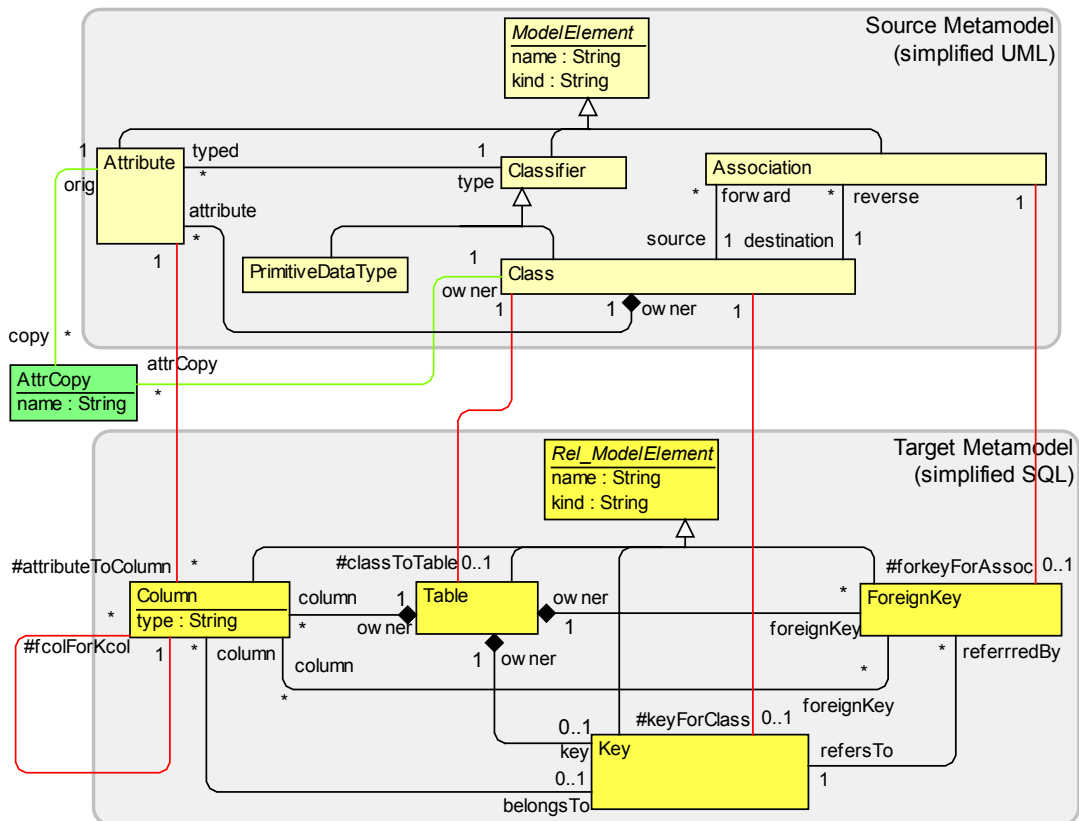


Fig. 3. Combined Metamodel

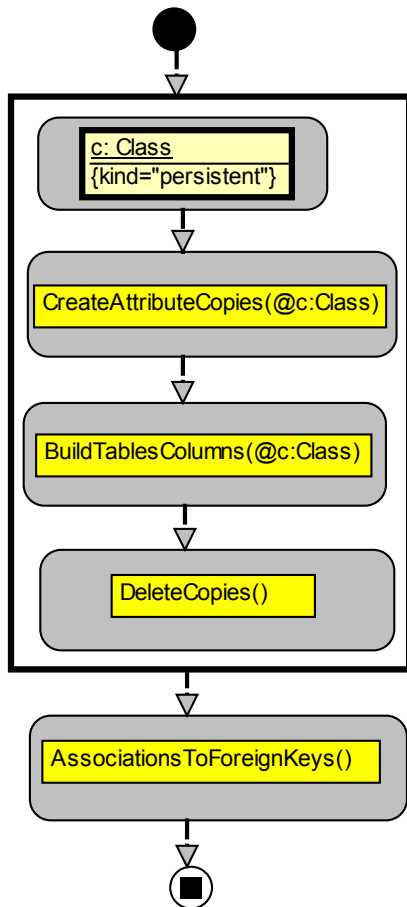


Fig. 4. Main Program

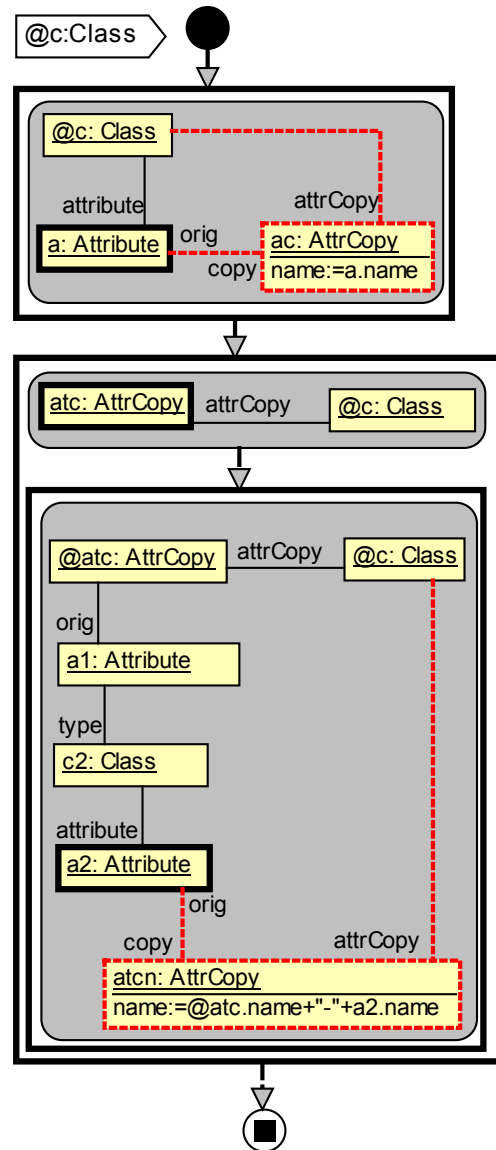


Fig. 5. Subprogram CreateAttributeCopies

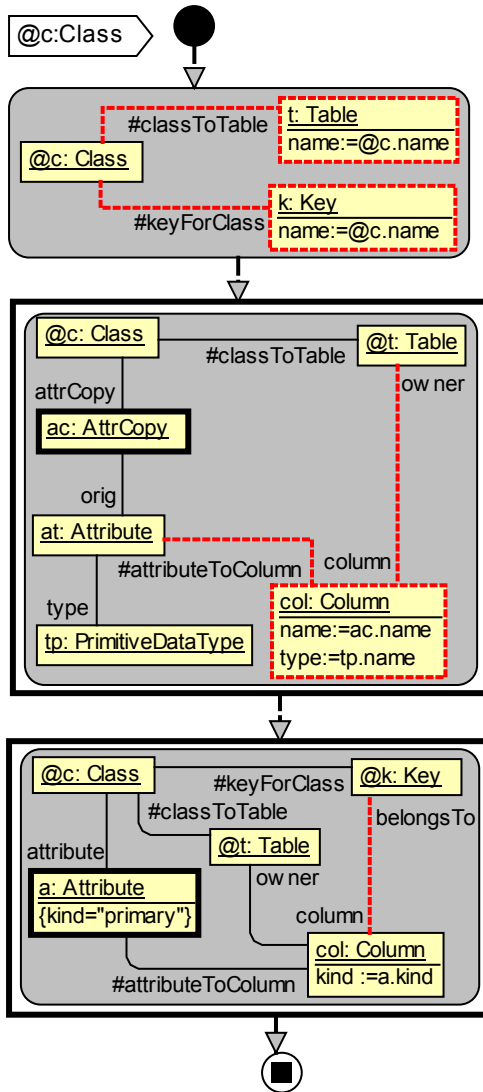


Fig. 6. Subprogram BuildTablesColumns

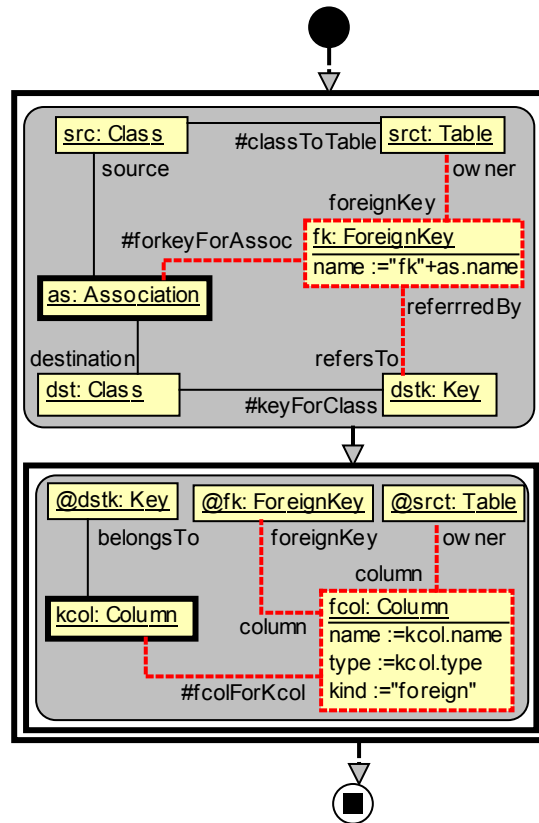


Fig. 7. Subprogram AssociationsToForeignKeys

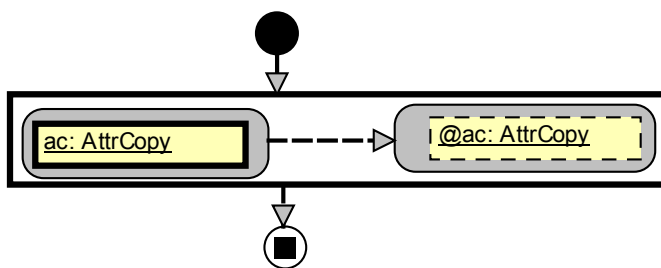


Fig. 8. Subprogram DeleteCopies

6. Performance evaluation for the example

Aside from being a standard benchmark for MDA languages, the example is very appropriate for performance evaluation. For example, the procedure in Fig. 5 actually has the loop depth 3, which could lead to a bad performance.

We start with one general observation on MOLA programs, which is true for the example and also for all MOLA programs built so far. A pattern in a correct MOLA program is typically built so that any **nondeterministic choice** is **excluded** during the pattern match. More precisely, each pattern element (except for the loop variable) can

be matched to 0 or 1 instance (if 0, the pattern fails for the given instance of loop variable). This can be achieved by syntactic means, e.g., by selecting associations with multiplicity 1 in the appropriate direction. Or some semantic considerations specific to the example may be used. For nested loops such unambiguous matching typically is achieved by proper use of references to elements of upper level loops. During the analysis of the example the specific reason for each loop will be shown.

The common principle, which will be applied to analysis of all loops, is that **no proper backtracking** occurs in the situation described above. Namely, either one instance can be matched to a pattern element or there is no instance at all and the pattern fails. The pattern matching procedure proposed in the previous section is specially built so that only a **constant "overhead"** can occur in this situation. More precisely, if a pattern element cannot be matched, the procedure has just to backtrack formally (without finding any new match) over all preceding elements in the pattern list.

To start with performance evaluation for the example, some reasonable assumptions about the source models (class models built according to the metamodel in Fig.3) must be made. We assume that all **instance level multiplicities** (number of the specified links per instance) **are bounded by some fixed constant**, while the size of the source model (instance set) may grow unboundedly. For example, it means that if we have n classes (i.e., instances of the metaclass *Class*), then we can have at most $c_1 * n$ instances of *Attribute* and $c_2 * n$ instances of *Association* in the source model. In other words, the instance graph has bounded degrees for all types of edges. The described transformation includes a transitive closure, and even for the abovementioned assumptions the target model (or instance set) could have unlimited cardinalities. More precisely, there could be an unlimited number of *Columns* per *Table*. Since this is untypical in practice, we assume this number also bounded by a constant.

The "units", in which the number of required steps for a MOLA procedure will be measured, are the calls to virtual machine operations (actually those dealing with instances, not MOLA code).

The simplest case for performance evaluation is top-level FOREACH loops (not using any references). There are two such loops in the example – the main one in Fig.4 and the top loop over *Associations* in Fig.7. The main loop over *Class* in Fig.4 is obviously executed for each *Class* instance, i.e., n times, creating a proper iteration (subprogram invocation) for *persistent* instances – we assume that it is also $O(n)$ times. Our performance measure – virtual machine calls is obviously the same since actually there is no pattern in this loop.

The top loop in Fig.7 is more interesting since it has a pattern – reproduced also in Fig.1. If we had to evaluate the number of steps in matching this pattern without any special considerations, we would obtain $O(n^5)$ – there are five elements in the pattern all having corresponding instance sets of size $O(n)$. Fortunately, the pattern is a very correct one for our evaluations – the no-backtracking principle applies in the simplest way, since all pattern associations have multiplicities 1 or 0..1 in the required direction (away from the root) in the metamodel (see Fig.3). Thus, an *Association* has exactly 1 *source Class*, a *Class* has 0 or 1 *#classToTable* link to *Table* etc. According to the constant overhead in matching procedure described above, this yields an estimate $O(n)$ for pattern matching in this FOREACH loop. The real number of iterations (executions of the nested loop) can also be evaluated as $O(n)$ according to our assumptions – since *persistent Classes* have *Tables* and *Keys*.

Now some general comments on pattern matching evaluation for nested loops, containing references. Patterns in such loops contain elements of two kinds. Elements, which are on restriction paths (paths linking a reference to a loop variable, see section 4), can be matched to instances from feasible instance sets, built according to the principles described in section 4. Since these sets are built, starting from one instance (the reference) and the size of the next set is limited by the number of specified links from instances of the previous set, the size of any instance set is also limited by a constant in our case. This implies that there is no need for precise evaluation for this kind of pattern elements, if we want to obtain just order-of-magnitude type results. By the way, this implies also that the given kind of loop is iterated no more than constant number of times on each invocation. Certainly, we have to check whether the loop variable is indeed reachable by a restriction path. Pattern elements not on restriction paths should be evaluated according to the same no-backtracking principle as above – the general matching procedure is used for them.

We start with the evaluation for MOLA subprogram in Fig.5 – it has loop nesting depth 3 (it is invoked in the main loop). Fortunately, all pattern elements (including loop variables) in all FOREACH loops in this procedure are on restriction paths. For example, in the first loop only *Attributes* of the given *Class* may be iterated. Similar situation is for the next loop, both at upper level and the nested loop (there the restriction path is longer, but actually the number of *Attribute* instances per loop invocation is limited by the original constant c_1 , the other sets have size 1). According to the general evaluation principles for nested loops described above, we can conclude that the total complexity evaluation for this subprogram is just a constant (for one invocation). However, we must be careful in one respect. The second loop (upper level) is a self-replenishing one, the instances of *AttrCopy* are generated within the nested loop (a typical situation for transitive closure). Therefore we must be sure that the total number of *AttrCopy* instances per *Class* is also limited by a constant (only in this case our general principles are applicable). This is not a MOLA evaluation problem, it is more the domain problem. Since any primitive-typed *AttrCopy* generates a *Column* (see Fig.6) and we have assumed a constant limit for *Columns* per *Table*, it is natural to assume also a similar limit for *AttrCopy* (which makes our conclusions completely valid).

A more interesting situation is for *BuildTablesColumns* subprogram in Fig.6, where patterns contain both kinds of elements. The first statement of this subprogram is a simple rule (executed once), the only fact to be noticed is that elements of a rule (*Table* and *Key*) may be used as references in subsequent statements. The second statement is a loop, where the loop variable (*AttrCopy*) is on a restriction path, but two other elements are restricted by metamodel multiplicities. This in totality again yields a constant evaluation. The third statement is a loop, where the loop variable (*Attribute*) is on two restriction paths – from *Class* and *Table*. The general evaluation principle applies in a standard way, but it is interesting to note that the set intersection from two paths supply the exactly desired instance set for *Attribute*, which in turn ensures matching uniqueness for *Column* (by semantic considerations, not by multiplicities). The program would be incorrect if the uniqueness were not achieved.

The remaining loops (the nested one in Fig.7 and the one in Fig.8) obviously satisfy the general evaluation principle and have a constant evaluation. Thus all nested loops in the program do have a constant evaluation for pattern matching, and the **total estimate** for the **whole program** evidently is **O(n)** – both for pattern matching and any other kind of operations. In other words, the implementation is optimal for the example (with respect to the measures used).

7. Conclusions

We have demonstrated on one example, how reasonable programming style and appropriate implementation of pattern matching together yield a very efficient performance. Any of pitfalls of pattern matching, typical to graph rewriting languages [10], are automatically avoided in MOLA for this example.

In the conclusion we want to generalize and comment this situation more broadly. Firstly, it is the completely deterministic control structure of MOLA – sequence, the two types of loops and branching, which forces us to use a "deterministic" approach to programming in MOLA. The only non-determinism is that no one is interested in the order, in which valid instances of a loop variable are processed (even concurrent processing could be used).

Consequently, in a typical MOLA program, where patterns have no redundant elements, we expect a deterministic match for pattern elements. To achieve this, good programming style for MOLA patterns should be used. The **elements of this style** are: use metamodel associations with multiplicity 1 in the appropriate direction (away from loop variable), appropriate semantic considerations (the nested loop in Fig.6, statechart flattening example – Fig.13 in [6]) or sufficient references in nested loops (Fig.5,6,7). The matching determinism is especially important if we have some additional use of the pattern element (and typically it is so) – we reference it in a nested loop, use its attributes, use it as a base for instance creation etc. If several "useful" instances of a metaclass correspond to a pattern element, then most probably actually all of them must be processed in the same way. In MOLA this should be implemented by one more nested loop using the metaclass as the loop variable and references to elements in the previous loop for specifying the context.

If the described means ensuring deterministic match are used in a MOLA transformation program, then its performance can be evaluated in a way similar to the example in section 6. Such an analysis has been performed for all MOLA examples built so far, and in all cases the evaluation showed results similar to that in the paper – there was no loss with respect to the natural complexity of the implemented transformation algorithm. For many MDA-related transformations the estimate $O(n)$ with respect to data size is typical, but there are also more complicated ones.

Thus a **correct** transformation program in MOLA becomes **efficient** at the same time. There is no special need to bother on program efficiency – just concentrate on correctness and natural use of MOLA constructs. Certainly, an appropriate implementation of MOLA must be used – the one that takes the described above feature into account. A possible implementation of pattern matching has been sketched in section 4. The matching procedure may have more optimizations – the one described here is sufficiently "rough", but it would have no great effect in typical case, when no proper backtracking occurs. Some heuristics specially oriented towards "typical trivial backtracking" could also be added. For example, if an instance is found by *getNextByLink*, the association multiplicity is 1 (this fact can be marked by the compiler) and another instance is required, immediate backtracking (to the deepest feasible level) could be done. However, only constant improvements can be achieved this way.

All the abovementioned suggests that an efficient and at the same time relatively simple implementation of MOLA is possible. The implementation schema for pattern matching sketched in this paper can be used as the basis for such implementation. By the way, this suggests that an implementation based on an interpreter for MOLA virtual machine is quite feasible from the performance point of view.

References

- [1] QVT-Merge Group. MOF 2.0 QVT RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, <http://www.omg.org/cgi-bin/doc?ad/2004-04-01>
- [2] Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, <http://www.omg.org/cgi-bin/doc?ad/2003-08-07>
- [3] 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
- [4] 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
- [5] Bézivin J., Dupé G., Jouault F., et al. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. 2nd OOPSLA Workshop on Generative Techniques in Context of MDA, Anaheim, California, 2003.
- [6] Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDFA 2004, University of Linköping, Sweden, 2004, pp.14-28. (see also http://melnais.mii.lv/audris/MOLA_MDFA.pdf)
- [7] Kalnins A., Barzdins J., Celms E. Basics of Model Transformation Language MOLA. Proceedings of WMDD 2004, Oslo, 2004, <http://heim.ifi.uio.no/~janoa/wmdd2004/papers/kalnis.pdf>
- [8] Kalnins A., Barzdins J., Celms E. MOLA Language: Methodology Sketch. To be published in proceedings of EWMDA-2, Canterbury, England, 2004. (see also http://melnais.mii.lv/audris/EWMDA_MOLA2.pdf)
- [9] A. Schürr. PROGRES for Beginners. Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.
- [10] Vizhanyo A., Agrawal A., Shi F. Towards Generation of High-performance Transformations. Generative Programming and Component Engineering, (accepted), Vancouver, Canada, 2004.
- [11] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anaheim, California, 2003.

Tool support for MOLA

Audris Kalnins¹, Edgars Celms^{2,3}, Agris Sostaks⁴

*IMCS
University of Latvia
Riga, Latvia*

Abstract

The paper describes the MOLA Tool, which supports the model transformation language MOLA. MOLA Tool consists of two parts: MOLA definition environment and MOLA execution environment. MOLA definition environment is based on the GMF (Generic Modeling Framework) and contains graphical editors for metamodels and MOLA diagrams, as well as the MOLA compiler. The main component of MOLA execution environment is a MOLA virtual machine, which performs model transformations, using an SQL database as a repository. The execution environment may be used as a plug-in for Eclipse based modeling tools (e.g., IBM Rational RSA). The current status of the tool is truly academic.

Key words: Model transformations, MDD, MOLA, MOLA tool.

1 Introduction

Practical use of Model Driven Development (MDD) for building systems is impossible without appropriate tools. Principles of MDA and MDD are known for quite a time and several model transformation languages, including the emerging OMG standard (QVT-Merge) [15] have got certain publicity. However, there are very few truly MDD tools available. At the time of writing, the available commercial tools supporting MDD (OptimalJ[14], ArcStyler[2], Objecteering[13] and other) do it well for specific kinds of PSM (frequently, J2EE) and specific design methodologies, but modifying the used default model transformations is as hard as extending traditional modeling tools - in most cases conventional OOP languages are used to define transformations. On the other hand, the experimental model transformation tools - ATL[3],

¹ Email: Audris.Kalnins@mii.lu.lv

² Email: Edgars.Celms@mii.lu.lv

³ supported partially by ESF

⁴ Email: agree@os.lv

MTF[12], Tefkat[16], etc. which are mainly Eclipse EMF based and use various (mainly textual) transformation languages, are not well linked with the model providers - the modeling tools.

In this paper the academic MOLA tool, which is being developed at the University of Latvia and supports the graphical model transformation language MOLA [7], is described. The goal of the design has been to have a simple implementation, which nevertheless would be practically usable in the MDD context. The structure and main principles of the tool are described, and also its links with modeling tools. In a more detailed way, it is shown how MOLA execution environment can be linked to Eclipse EMF based modeling tools. This is illustrated by a case study - an application of MDD principles to IS design based on Hibernate framework.

2 Brief Description of MOLA

The MOLA tool is based on the MOLA model transformation language, developed at the University of Latvia, IMCS [7,8,9,10]. MOLA is a graphical procedural transformation language. Its main distinguishing features are advanced graphical pattern definitions and control structures taken from the traditional structural programming. To facilitate the understanding, we briefly remind the main concepts of MOLA.

Like most of the model transformation languages, MOLA is based on source and target metamodels, which describe the source and target models respectively. The used metamodeling language is EMOF [11](with some slight restrictions). In MOLA source and target metamodels are combined in one class diagram, but packages may be used for structuring. The source and target may coincide. Special **mapping associations** linking the corresponding classes in source and target metamodels may be added to the metamodel. Their role is similar to relations in other transformation languages - for structuring the transformation and documenting the transformation traceability.

The transformation itself is defined by one or more **MOLA diagrams** (see examples in Fig. 6 and 7). A MOLA diagram is a sequence of graphical statements, linked by arrows. The most used statement in a MOLA diagram is the **FOREACH loop** - a bold-lined rectangle. A loop has a **loop head** (a grey rounded rectangle), which contains the **loop variable** (bolded element) - a class, instances of which the loop has to iterate through. In addition, the loop head contains a **pattern**, which specifies namely which of the instances qualify for the given loop. A pattern is a metamodel fragment, but in instance notation - `element_name:class_name`, therefore classes may be repeated. Links just correspond to metamodel associations. A pattern element may contain an attribute-based constraint - an expression in OCL subset. The semantics of loop is quite natural - the loop must be executed for all instances of the loop variable for which there exist instances of other pattern elements satisfying their constraints and linked by the specified links (pure existence

semantics). Loops may be nested, the instance of the loop variable (and other elements) matched in the parent loop may be referenced in the nested loop by the reference notation - the element name prefixed by @ character.

Another kind of graphical statements is the **rule** (a grey rounded rectangle too), which also contains a pattern but without loop variable. A rule typically contains actions - element or association building (red dotted lines) and deletion (dashed lines). A rule is executed once in its control path (if the pattern matches) or not at all - thus it plays the role of an if-statement too. A loop head may also contain actions. MOLA subprograms are invoked by the call statement (possibly with parameters).

One year experience of using MOLA (mainly in academic environment - from undergraduate to PhD students) has confirmed its ease of learning and high readability of defined transformations - especially when compared to the current QVT-Merge proposal [15].

Actually, quite a few graphical model transformation languages are now in use - besides the graphical form of QVT-Merge, Fujaba Story diagrams (SDM) [6] and the GME-based GReAT notation [1] is used. The pattern definition facilities are approximately of the same strength in all these approaches, including MOLA. There are differences in defining the rule control structure, the Fujaba approach is the closest one to MOLA, but is less structured, while GReAT is more based on data flows. Actually we don't mention here the graph transformation languages, which have slightly different goals. A more comprehensive comparison of MOLA to other languages has been already given in [7].

3 The Architecture of MOLA Tool

The current version of MOLA tool has been developed with mainly academic goals - to test the MOLA usability, teach the use of MDD for software system development and perform some real life experiments. This has influenced some of the design requirements, though with easy usability as one of the goals and sufficient efficiency the tool has confirmed its potential as an industrial tool too.

Like most of the model transformation tools, the MOLA tool has two parts - the **Transformation Definition Environment (TDE)** and the **Transformation Execution Environment (TEE)**. Both these environments have a common repository for storing the transformation, metamodels and models (in the runtime format). Fig. 1 shows the general architecture of the tool.

The definition environment is related to the metamodel level - M2 in the MOF classification. Its intended users are methodology experts who provide the metamodels and define the transformations for development steps which can be automated. Since MOLA is a graphical language, TDE is a set of graphical editors, built on the basis of GMF [4] - a generic metamodel based modeling framework, 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 MDD methodology perform the automated development steps and obtain the relevant target models. Currently two forms of TEE are available. The form closer to an industrial use is an Eclipse plug-in, which can be used as a transformation plug-in for UML 2.0 modeling tools, including the commercial IBM Rational tool RSA. This use is described in more details in section 5 and demonstrated on a case study. Another form is a more experimental one. It is based on GMF as a generic modeling environment and is intended for various domain specific modeling and design notations. It is described more closely in section 6.

Fig. 1 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.

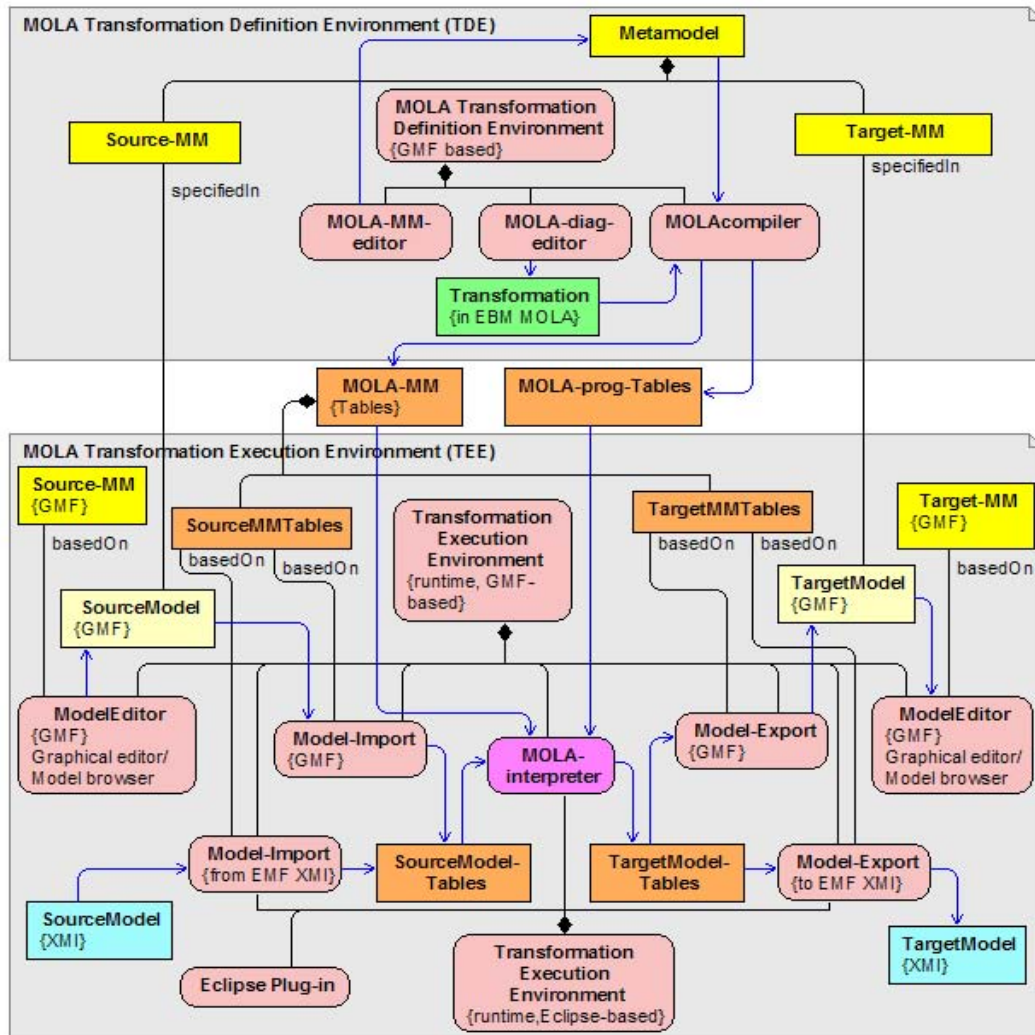


Figure 1. MOLA Tool environment architecture.

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 GMF, 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 GMF repository format to the MOLA runtime repository format. Fig. 2 shows a screenshot of MOLA TDE, with both metamodel and MOLA diagram editors open.

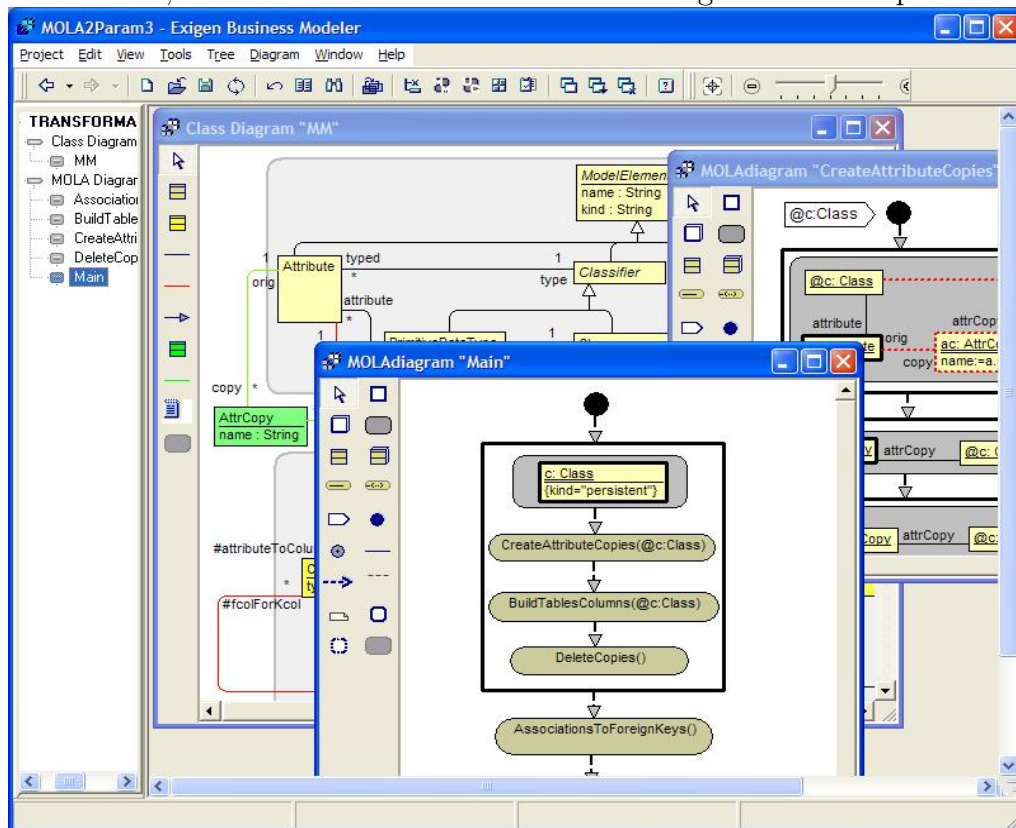


Figure 2. Screenshot of the MOLA TDE.

4 MOLA Virtual Machine and Repository

The core of the MOLA TEE is MOLA Virtual Machine (VM) - an interpreter performing the model transformation. Certainly, it is closely linked to the MOLA repository, whose main function is to ensure the efficiency of MOLA VM. The most crucial factor in implementing MOLA VM is the implementation of pattern matching - the most "expensive" part in any transformation implementation.

Model transformation tools [3,12,16] typically are implemented on meta-

model based repositories such as Eclipse EMF. Such an implementation typically uses low level repository operations for pattern matching and is more compiler than interpreter. Authors of this paper have shown [10] that a very efficient MOLA implementation is possible this way. However, for this academic implementation of MOLA another goal was set - the implementation must be as simple as possible, but still usable on examples of reasonable size, e.g., to transform a model containing several hundred classes.

Therefore another solution was adopted - to use an SQL database as a repository. The key rationale for this solution has been that a complete MOLA pattern match operation can be implemented by one SQL Select statement. In addition, this Select statement can be generated very easily from the MOLA pattern definition also stored in SQL tables in an appropriate way by the MOLA compiler. Thus MOLA VM would be quite close to a pure interpreter and therefore simple. The implementation of other MOLA elements is not so sensitive to repository format because it is a fairly classic implementation of traditional control structures. Namely these principles are used for MOLA VM implementation in this version of MOLA tool.

The only problem which remained to be solved was efficiency. The query generated from a MOLA pattern is quite untypical for standard SQL databases - it is a “self-join” of two tables (representing class and association instances in the model) as many times as there are elements and links in the pattern. Not all database engines occurred to process such joins satisfactorily. Since the desire was to build the MOLA tool as open source based as possible, the first candidate was MySQL. However, it occurred that for large patterns the performance was not satisfactory - the query optimization itself used by the MySQL engine was too lengthy for this type of queries. Another candidate was the free version Microsoft SQL - the MSDE engine. It performed quite efficiently for patterns occurring in reasonable MOLA programs and quite large example models. This way the stated goals for both simple and efficient MOLA VM were reached.

5 MOLA Transformation Execution Environment as an Eclipse plug-in

The MOLA TEE besides the MOLA VM must contain components for fetching the source models to be transformed and passing the transformation results. In a typical MDD scenario there must be also a modeling environment where the source models are prepared and the obtained target models are processed further. In the approach where MOLA transformations are used as plug-ins for Eclipse EMF namely this scenario is assumed. The environment was selected due its popularity as a model transformation testbed and because there is a publicly available UML 2.0 metamodel [17] for this environment. The MOLA TEE in this case, in addition to MOLA VM, contains XMI import component, XMI export component and a simple Eclipse plug-in (see the lower

layer of TEE in Fig. 1). The XMI import component currently supports a reasonable subset of UML 2.0 metamodel (in its EMF version). The XMI export component also supports a subset of UML 2.0, but in addition some other metamodels available in EMF are supported (e.g., the SQL database definition metamodel). The plug-in currently is very simple - it is used just to activate the TEE and select the source and target XMI location and the required MOLA transformation. The source model must be exported by the Eclipse modeling tool export facility and the generated target model imported by the import facility.

Currently this schema has been tested with the only professional Eclipse EMF based modeling tool truly supporting UML 2.0 - IBM Rational RSA. As soon as more Eclipse based modeling tools support the UML 2.0 metamodel, the same plug-in would be applicable to them. At the time of writing, there is no true transformation plug-in for RSA (the embedded RSA transformation extension facilities require coding in Java), so the developed plug-in could present also some practical interest. Certainly, a more user friendly solution would be to acquire the relevant source model directly via Eclipse API and pass the result this way too, but this solution is much more complicated and more tied up to a specific modeling tool.

The proposed solution seems to be practically usable for various MDD style development scenarios. Section 7 contains one such case study - a scenario where the Hibernate persistence framework is used. Since MOLA is well suited for model-to-model transformations, but not so well for model-to-code, the built-in code generation facilities of RSA (or other modeling tool) are used for this purpose.

6 Standalone MOLA Transformation Execution Environment

Another possibility to have a usable transformation execution environment is to tie the MOLA VM up to a generic modeling environment where an arbitrary graphical modeling notation can be supported. Since the GMF environment [4] is such one, another MOLA TEE has been based on it. Truly speaking, Eclipse+EMF+GEF [5] is also such an environment, but the development there requires much more effort. The MOLA TEE version based on GMF is meant for various experiments in applying MDD and model transformations to domain-specific notations, including non-UML ones. The top layer of TEE in Fig. 1 shows the corresponding components. In GMF it is possible to define the graphical presentation of a domain model as a sort of transformation (though not very universal, see more in [4]), therefore for many modeling notations usable graphical editors can be defined without proper programming at all. In any case, Eclipse EMF style model browsers/editors, but more flexible ones, can be built very easily with GMF. Universal metamodel-controlled export and import components from/to GMF repository have been built. This

task has not been so hard since the GMF repository is functionally close to EMOF. The relevant MOLA transformation can be invoked directly from this environment (MOLA VM is used as a GMF plug-in).

Several such experiments have been performed. In one case, a UML activity diagram profile (a complicated one and represented graphically) meant for defining workflows in UML was implemented in GMF, and a transformation to a specific workflow notation was defined in MOLA. GMF has a special feature of generating readable diagrams automatically, therefore in many cases the transformed target model can be automatically presented as a diagram.

Another GMF based experiment has been in converting a special profile of class diagrams to OWL notation for ontology definitions.

7 Case Study: Use of MOLA Tool for Building an IS within Hibernate Framework

In this section we show how the Eclipse plug-in form of MOLA tool could be used for MDD style development of information systems in Java within the Hibernate persistence framework. This framework provides a “classical” object-relational mapping between Java classes and database tables, permitting a developer to access instances of such classes (actually stored in a database) as if they were true Java objects. The modeling tool where the models are built is assumed to be IBM Rational RSA. Just one nontrivial step in the methodology is illustrated. We assume that a PIM - an IS domain model in the form of a standard UML 2.0 class diagram has been built (see a small example in Fig. 3).

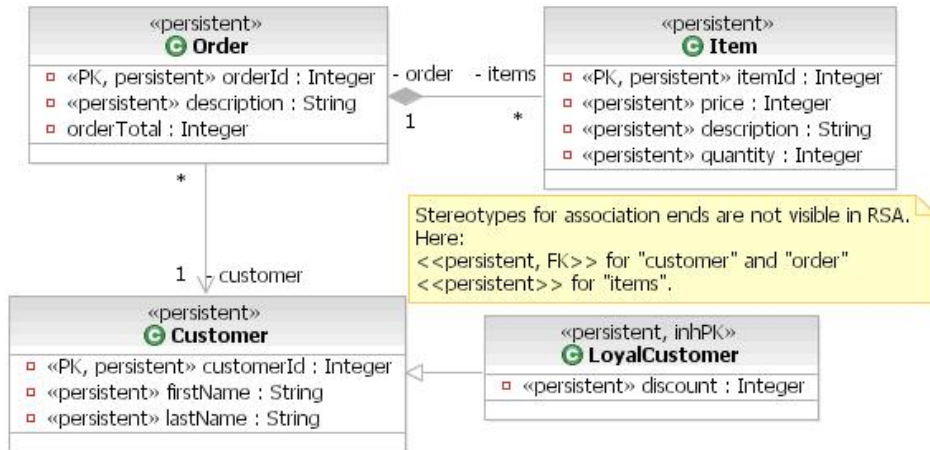


Figure 3. UML class diagram with stereotypes in RSA (PIM model).

Some of the classes must be persistent - stored in a relational database as tables. The standard Hibernate mapping is assumed for these classes, which requires “standard” Java getters and setters to be added for the persistent attributes and association ends of a class, with other attributes and operations unmodified. In addition, for each such mapping the Hibernate mapping de-

scriptor (an XML file) must be built. Thus the task is to build a PSM model consisting of three parts - the augmented UML classes, database schema definition and Hibernate mapping descriptors.

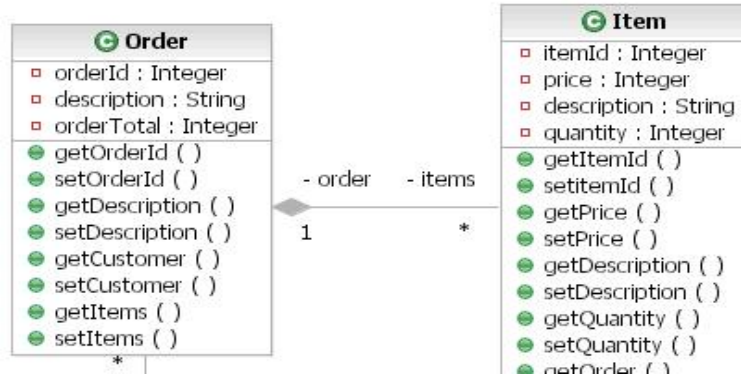


Figure 4. Part of class diagram for Hibernate framework in RSA(PSM model).

In order to specify adequately the logical design decisions at the PIM level, a custom profile (`HibernateProfile`) is required. This profile should contain the stereotypes `persistent` (both for classes and properties, representing either attributes or association ends), `PK` - for attributes (properties), `FK` - for association ends (properties) and `inhPK` - for defining Hibernate-style storing of persistent subclasses. If these stereotypes are appropriately applied to the PIM model, then the three-part PSM can be generated automatically by a MOLA transformation - for each persistent class a table will be defined (containing persistent attributes and associations), getters/setters will be added to the class and the Hibernate descriptor will be defined. Fig. 3 shows these stereotypes applied (RSA does not visualize stereotypes for association ends). All classes there are assumed to be persistent, but not all attributes. Classes in a PIM normally should contain also business operations, we don't show them for brevity. We assume also that primary keys consist of one column (Hibernate uses a complicated mapping for complex keys).

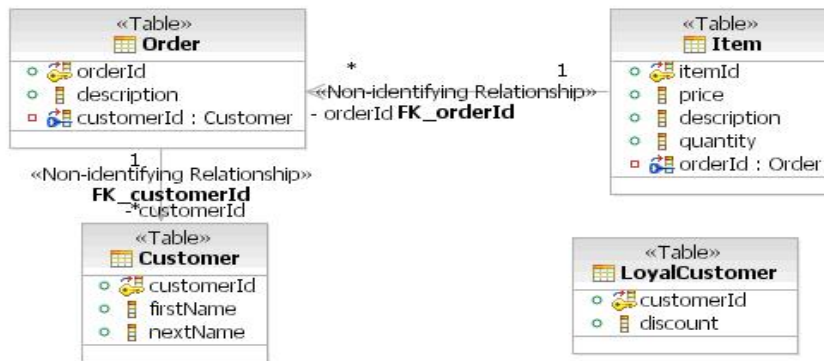


Figure 5. Data model visualization in RSA (PSM model).

Fig. 4 shows the first component of the result - the updated class diagram (fragment). Getters and setters are added where appropriate, but custom stereotypes are removed - RSA does not use them for code generation.

Fig. 5 shows the second component of the result - the database schema.

The model is built according to the EMF SQL metamodel, but RSA data model visualization feature is used to show the schema as a diagram.

Finally, Fig. 6 and 7 show the part of the MOLA transformation - the main program and SQL table building.

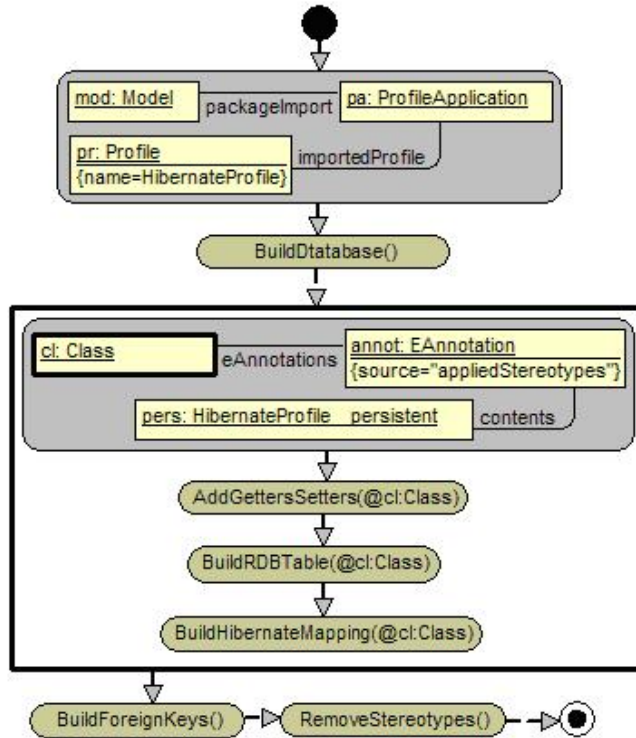


Figure 6. MOLA transformation program (main).

The metamodels are not shown due to lack of space. The source metamodel is the standard UML 2.0 metamodel. However, in EMF a special coding (not the OMG standard, but eCore defined via the `EAnnotation` metaclass) is used for applied stereotypes. Namely this coding is used in transformations - in UML 2.0 the applied stereotypes show up as instances in the model, therefore model transformations must treat them as instances of special temporary metaclasses (note the MOLA pattern for finding a persistent class in the `FOREACH` loop of Fig. 6). The `AddGettersSetters` transformation (not shown here) uses the same UML metamodel as a target - it is an update transformation, which simply attaches new `Operation` instances to existing `Class` instances. The `BuildRDBTable` program (Fig. 7) builds a table for the class and then performs a loop, which builds a column (including its type and key constraints) for each persistent property. The target metamodel for this program is the SQL metamodel in EMF, but for `BuildHibernateMapping` - the metamodel obtained from the Hibernate XML schema definition. Actually in MOLA all these metamodels appear as a common class diagram, but packages are used to separate them. The same packages are used to guide the MOLA tool XMI exporter component - in this case several separate XMI files must be generated, but for Hibernate mapping a non-XMI XML coding is required.

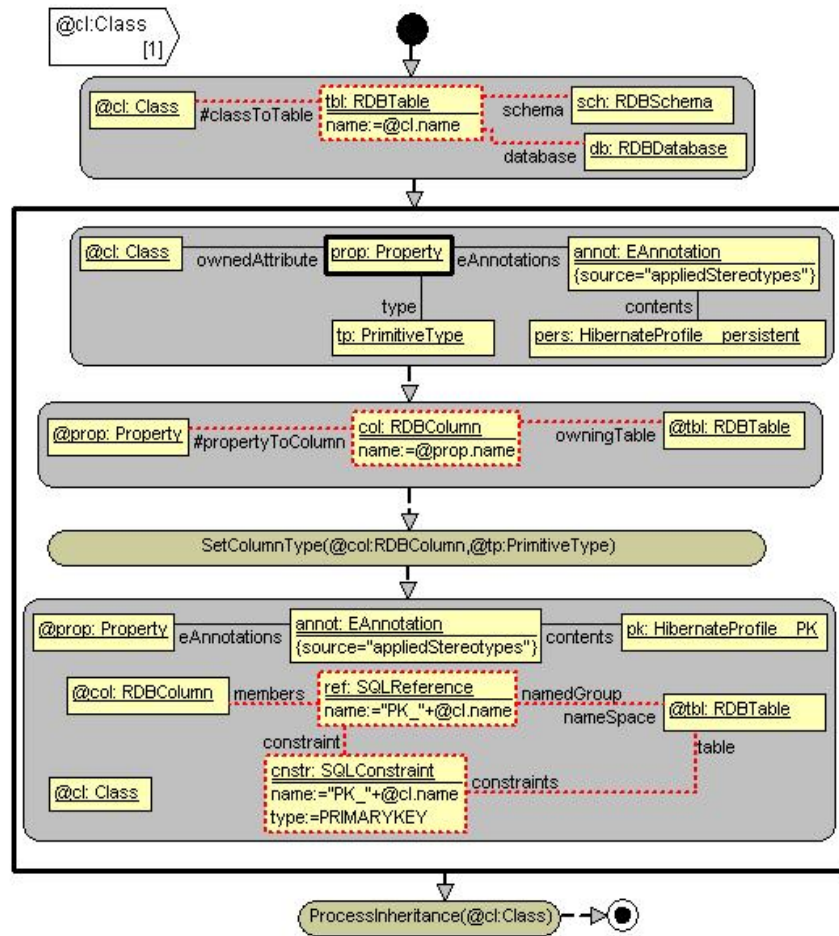


Figure 7. MOLA transformation program (BuildRDBTable).

8 Conclusions

The structure and some use cases of the experimental academic MOLA tool have been described in the paper. The existing experience of using the tool has shown that the adopted solutions are appropriate and MOLA transformations fit in well both in the traditional UML based MDD style development and in domain specific modeling. Certainly, the practical tool usability has to be improved, especially the links with the modeling tools. One more issue to be solved is the “round-tripping”, because in MDD setting the target models are also sometimes updated manually. MOLA transformations have no “native reversibility”, but it is clear that for typical MDD tasks reverse transformations are easy to build, using either mapping associations (in standalone environment) or special annotations (in EMF environment). Yet another task is to build a MOLA transformation library for typical MDD use cases.

References

- [1] Agrawal A., G. Karsai, F. Shi. “Graph Transformations on Domain-Specific Models”. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003
- [2] ArcStyler. URL: <http://www.interactive-objects.com/>
- [3] ATL. URL: <http://www.sciences.univ-nantes.fr/lina/atl/>
- [4] Celms E., A. Kalnins, L. Lace. “Diagram definition facilities based on metamodel mappings”. Proceedings of the 18th International Conference, OOPSLA’2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23-32.
- [5] Eclipse GEF. URL: <http://www.eclipse.org/gef/>
- [6] Fujaba User Documentation. URL: <http://www.cs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>
- [7] Kalnins A., J. Barzdins, E. Celms. “Model Transformation Language MOLA”. Proceedings of MDFA 2004 (Model-Driven Architecture: Foundations and Applications 2004), Linköping, Sweden, June 10-11, 2004. pp.14-28.
- [8] Kalnins A., J. Barzdins, E. Celms. “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.
URL: <http://heim.ifi.uio.no/janoa/wmdd2004/papers/>
- [9] Kalnins A., J. Barzdins, E. Celms. “MOLA Language: Methodology Sketch”. Proceedings of EWMDA-2, Canterbury, England, 2004. pp.194-203.
- [10] Kalnins A., J. Barzdins, E. Celms. “Efficiency Problems in MOLA Implementation”.
19th International Conference, OOPSLA’2004 (Workshop “Best Practices for Model-Driven Software Development”), Vancouver, Canada, October 2004.
URL: <http://www.softmetaware.com/oopsla2004/mdsd-workshop.html>
- [11] MOF 2.0 Core Final Adopted Specification.
URL: <http://www.omg.org/docs/ptc/03-10-04.pdf>
- [12] MTF. URL: <http://www.alphaworks.ibm.com/tech/mtf>
- [13] Objectteering. URL: <http://www.objectteering.com/>
- [14] OptimalJ. URL: <http://www.compuware.com/products/optimalj/>
- [15] QVT-Merge. URL: <http://www.omg.org/docs/ad/05-03-02.pdf>
- [16] Tefkat. URL: <http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/>
- [17] UML 2.0 Eclipse EMF. URL: <http://www.eclipse.org/uml2/>

Model Transformation Approach Based on MOLA

Audris Kalnins, Edgars Celms¹, Agris Sostaks

University of Latvia, IMCS, 29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Edgars.Celms}@mii.lu.lv, agree@os.lv

Abstract. This paper provides a solution to the mandatory transformation example specified in MOLA – a graphical model transformation language developed at the University of Latvia. The solution is validated by executing it via the MOLA execution environment on several examples. In addition, a solution to one of the optional examples – determinization of a non-deterministic automaton is provided.

1 Introduction

The idea of model transformations as the main support for model driven software development is already gaining some maturity now. First and foremost, it appears in the area of model transformation languages. The emerging OMG standard model transformation language, QVT-Merge [1], most probably will reach its final shape at the end of this year. But while waiting for this, various independent model transformation languages gain their maturity too. Most of the languages use some sort of the pattern concept (to be matched in the source model) and rules controlling the application of patterns.

According to a very rough grouping, model transformation languages can be divided into textual and graphical languages. The QVT-Merge language fits into both groups since it has both textual and graphical form. Textual languages such as ATL[2], MTF[3], Tefkat[4], MT[5] and many other, though very different in details, typically use recursion as the main control structure.

Graphical transformation languages are significantly less in number. Besides QVT-Merge, Fujaba Story diagrams (SDM) [6] and GME-based GReAT [7] notation should be mentioned. The MOLA transformation language, which is the topic of this paper is namely in this category. In addition, graph transformation languages (such as AGG [8]), though originally built for different goal, actually have similar characteristics. It should be noted, that many characteristics of the graphical languages are somewhat similar too.

An unbiased comparison of qualities of transformation languages is not so easy to obtain, because there are so many different subjective viewpoints. Therefore this workshop, where very precisely defined requirements for a mandatory transformation example are given in its CFP [9], could provide the first such impartial comparison.

¹ supported partially by ESF (European Social Fund), project 2004/0001/VPD1/ESF/PIAA/04/NP/3.2.3.1/0001/0063

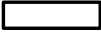


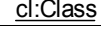
This paper presents a solution to this transformation task specified in MOLA – a graphical language developed at the University of Latvia, IMCS. Though the description of the same transformation in graphical languages is longer than in textual ones, authors consider the provided solution to be optimal from the readability and clarity point of view (though this is subjective too). The solution is validated on test models, by executing it via the MOLA tool. The paper describes how a problem specific modeling environment (for building test models) linked to the MOLA execution environment can be built using GMF – a generic modeling framework also developed at the University of Latvia (unfortunately, a name clash has occurred – an Eclipse project also named GMF [10] has been recently started).


Sections 2 and 3 provide a brief introduction to MOLA and its tools. The section 4 presents the solution of the mandatory transformation example, but section 5 – its validation via MOLA tool. Section 6 provides MOLA solution for one of the optional examples – the determinization of a non-deterministic automaton.

2 Brief Description of MOLA Language

The MOLA model transformation language has been developed at the University of Latvia, IMCS [11,12,13,14], the most complete description is given in [11]. MOLA is a graphical procedural transformation language. Its main distinguishing features are advanced graphical pattern definitions and control structures taken from the traditional structural programming. In this section we briefly remind the main concepts of MOLA. Later on in the examples sections example diagrams will be annotated by comments, which will allow easily to follow the notation.

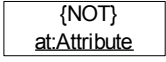
Like most of the model transformation languages, MOLA is based on source and target metamodels, which describe the source and target models respectively. The used metamodeling language is EMOF [15] (with some slight restrictions). In MOLA source and target metamodels are combined in one class diagram, but packages may be used for structuring. The source and target metamodels may coincide. Special mapping associations linking the corresponding classes in source and target metamodels may be added to the metamodel. Their role is similar to relations in other transformation languages – for structuring the transformation and documenting the transformation traceability. If necessary, temporary classes and/or associations for storing intermediate data may be added.

The transformation itself is defined by one or more MOLA diagrams. A MOLA diagram is a sequence of graphical statements, linked by arrows. The most used statement in a MOLA diagram is the **FOREACH loop** – a bold-lined rectangle(). A loop has a **loop head** (a grey rounded rectangle - ) , which contains the **loop variable** (a bolded element – e.g., ) – a class, instances of which the loop has to iterate through. In addition, the loop head contains a **pattern**, which specifies namely which of the instances qualify for the given loop. A pattern is a metamodel fragment, but in instance notation – it contains **elements**, e.g.,  , therefore classes may be repeated. Pattern links just correspond to metamodel associations. A pattern element may contain a **constraint** – an expression

in OCL subset, which must be true for an instance to qualify. The semantics of loop is quite natural – the loop must be executed for all instances of the loop variable for which there exist instances of other pattern elements satisfying their constraints and linked by the specified links (pure existence semantics). Loops may be nested, the instance of the loop variable (and other elements) matched in the parent loop may be referenced in the nested loop by the **reference** notation – the element name prefixed by @ character. Besides the FOREACH loop, there is also the less used WHILE loop () , which is executed while there is at least one instance of loop variable for which the pattern matches, i.e., the same instance may be processed several times.

Another kind of graphical statement is the **rule** (a grey rounded rectangle too), which also contains a pattern but without loop variable. A rule typically contains actions – element or association building (red dotted lines) and deletion (dashed lines). A rule is executed once in its control path (if the pattern matches) or not at all – thus it plays the role of an if-statement too. A loop head may also contain actions. MOLA subprograms are invoked by the **call** statement (possibly with parameters), recursive calls are permitted. The parameters may be references to elements or primitive values.

One year experimental usage of MOLA, mostly in academic environment, has suggested few extensions with respect to the original definition of MOLA in [11]. Firstly, the use of **NOT** constraint in patterns has been clarified and extended. A

MOLA element in a pattern may have the NOT constraint, e.g., . The meaning is that the whole pattern matches, if there is **no** instance of the given class, which satisfies the local OCL constraint (if any) and has the specified links with the other ("positive") elements of the pattern. In addition, there may be a NOT constraint on a pattern link (no such link may exist between the matched instances) and a NOT-region – a rectangle containing several pattern elements (then there may be no properly linked match for the whole subpattern). Since the last two cases are not used in this paper, we present no more details of semantics for them.

Other extensions are related to control flows – now there are graphical equivalents for most of structured control constructs of modern programming languages. A rule may have two exits – one unmarked and the other one marked {**ELSE**} (any of them may be absent). If the rule pattern matches (and the rule actions are performed), the unmarked exit is taken. Otherwise, the ELSE exit is taken. If the required exit is absent, there is a default transition – if inside a loop body, then to the next iteration ("implicit continue"), if at the top level of a MOLA program, then it means the program end ("implicit return"). Thus a true if-then-else construct is provided. Branched control flows may merge again, but it is forbidden to build a "proper goto" – to branch backwards. Elements matched in a rule may be referenced only in its "positive path". In the context of a loop, some more options are available. A flow may reach the loop rectangle from inside – it means an "explicit continue". A flow may also cross the loop border – this is an "explicit break" (or "explicit return", if the target is an end symbol). In any case, no backward loops are permitted this way.

3 MOLA support tools

A MOLA tool supporting the MOLA transformation language has been built at the University of Latvia (see the first report on it in [16]). MOLA tool has two parts – the **Transformation Definition Environment (TDE)** and the **Transformation Execution Environment (TEE)**. Both environments use a common **runtime repository**, which currently is a relational database. There transformations, metamodels and models all are stored.

The definition environment (TDE) is at the metamodel level (M2 in the MOF classification). Since MOLA is a graphical language, TDE is a set of graphical editors, built on the basis of GMF [17] – a generic metamodel based modeling framework, developed by University of Latvia, IMCS together with the Exigen company. It contains graphical editors for class diagrams (EMOF level) and MOLA diagrams. Both the source and target metamodels currently 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. 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 GMF repository format to the MOLA runtime repository format. All MOLA examples in this paper have been taken from the MOLA TDE.

MOLA TEE is based on the MOLA Virtual Machine (VM) – an interpreter performing the model transformation, with instance data kept in the runtime repository (RDB). MOLA VM performs MOLA statements by converting them to SQL queries. It should be noted, that the most complicated element of MOLA – a pattern in a loop head or rule can be converted to a single SQL query. Thus the given implementation of MOLA is sufficiently simple (see more details in [16]). At the same time the experience with MOLA tool shows that it is also efficient enough – models with hundreds of instances may be transformed in seconds, if an appropriate RDB is used for the repository (currently – MSDE [19], the free version of MS SQL).

There are several ways how a complete MOLA TEE can be built because it must have close links with the supplier/consumer of models – a modeling environment. One of the ways is to use MOLA TEE as a plug-in for a modeling tool, with model data being exchanged in XMI format. It is sufficiently easy in the case of Eclipse and EMF [18] based tools. In [16] it is described in sufficient detail, how MOLA TEE can be used as a plug-in for the commercial IBM Rational modeling tool RSA. It should be noted that this approach requires at least one of the models (source or target) to be in standard UML 2.0.

Another approach, which is more relevant to the goals of this paper, is to use a generic modeling environment where an arbitrary graphical modeling notation can be supported. Since the GMF environment [17] fulfills these requirements, a reasonable solution is to link MOLA TEE to this environment. In GMF it is possible to define the graphical presentation of a domain model as a sort of transformation (though not very universal, see more in [17]), therefore for many modeling notations usable graphical editors can be defined without proper programming at all. In addition, Eclipse EMF style model tree browsers/editors, but more flexible ones (e.g., with several instances combined in one tree node), can be built very easily with GMF. Thus a readable visual representation of a model (source or target) can be obtained. This approach is

adequate for domain specific notations, including non-UML ones, where frequently standard editing facilities simply are not available. Since the examples of this paper are in this category, namely such an approach is used. It should be noted that a somewhat similar approach is used for GReAT transformation language, combined with the generic GME modeling environment [7].

To apply the approach, two visual editors (diagrammatic or model tree based) must be defined in GMF for the source and target models respectively (if the source and target is different). They are based on the same metamodels which are used to define the model transformation in MOLA. Currently these metamodels must be ported manually to the GMF environment (GMF metamodels are in a slight variation of EMOF notation), but in the near future an automatic support will be provided. Then the editor definitions must be provided (e.g., which "domain metamodel pattern" maps to a presentation class, which pattern maps to a tree node etc., see more in [17]). The GMF-based MOLA TEE contains universal metamodel-controlled instance export and import components from/to GMF repository. The relevant MOLA transformation can be invoked directly from the GMF environment (MOLA VM is used as a GMF plug-in). The general schema of GMF based MOLA TEE is shown in Fig. 1.

The outlined here approach will be demonstrated in section 5 for the mandatory example – both tree-form and diagrammatic editors for source and target models will be shown. The convenient graphical facilities for building source models are used to test the correctness of defined MOLA transformations (see more in section 5).

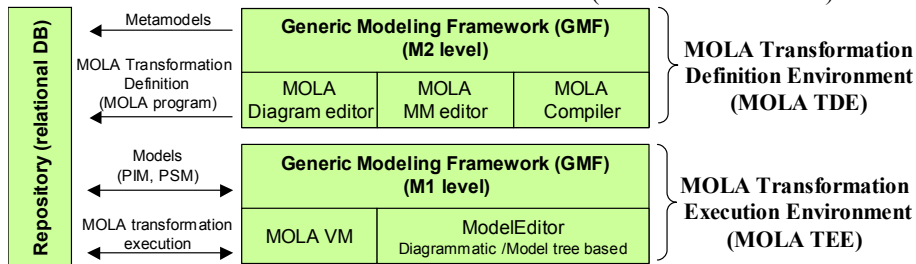


Fig. 1. MOLA tool schema.

4 The mandatory example in MOLA

In this section we provide the MOLA solution for the mandatory model transformation example. The example is taken literally as specified in the workshop call for papers (CFP) [9]. However, the lately added to FAQ comment that subclasses of persistent classes do not add new elements to the primary key is not used – we permit primary attributes to be merged up to the persistent class. All diagrams of the proposed MOLA solution are shown in Fig. 2 – 12.

4.1 Metamodel of the example

Fig. 2 shows the metamodel of the example. In MOLA source and target metamodels (if different) must be combined in one class diagram. The upper region in Fig. 2 is the source metamodel (simplified UML) and the lower one is the target (simplified SQL). The regions are just graphical comments. All black associations are the original ones.

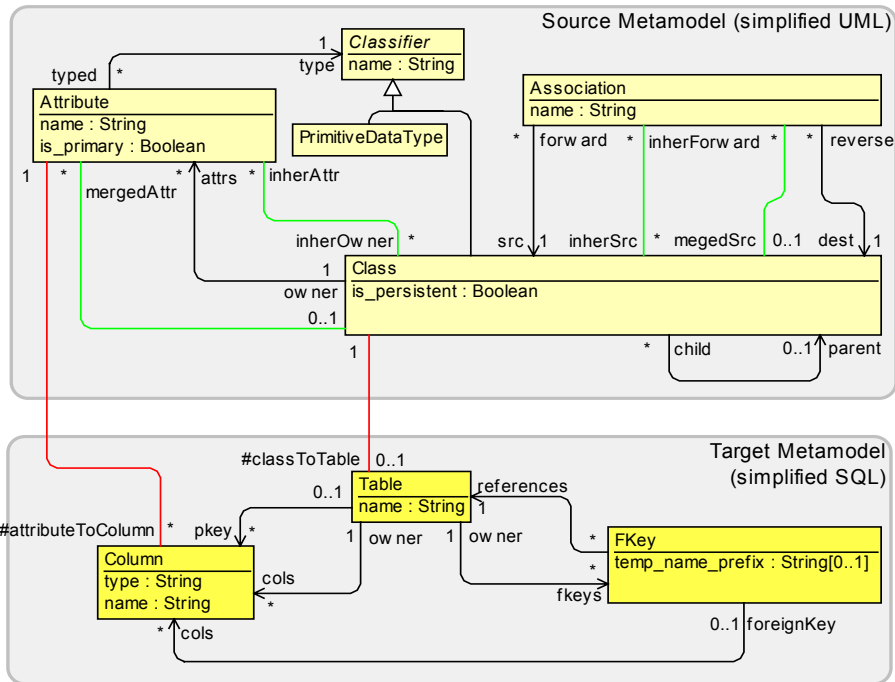


Fig. 2. The combined source and target metamodel in MOLA.

MOLA uses a slightly simplified EMOF syntax for metamodels. Association multiplicities must be explicit in MOLA, therefore the default ones have been added. Some role names for non-navigable ends also have been added (they are not mandatory for transformations, but ease the instance management in MOLA environment).

Associations in colors other than black have a special meaning in MOLA. The green ones are temporary – they are not present in the source model, but built by MOLA programs to store some intermediate relations. They are not also included in the resulting model. The red ones are the mapping associations, typically they link classes in source metamodel to target ones. They are built by MOLA programs, and their role is similar to relations, e.g., in QVT-Merge language – to transfer the results of high level transformations to subordinated ones and to facilitate the definition of inverse transformations (they are retained in the resulting model).

Fig. 2 contains two intermediate relations between `Class` and `Attribute` and between `Class` and `Association` – they are used to relate all (transitively) inherited elements (according to the standard UML semantics) and all "transitively

merged-up" elements – as specified by the example requirements. See the section 4.2 and 4.3, how their use makes the transformations more readable. There are also two mapping associations – from Class to Table and from Attribute to Column. They serve as a "backbone" for defining the correspondence between the source and target models, e.g., it is very convenient to find easy, whether a table for a class has been built and namely which. A temporary attribute `temp_name_prefix` is also added to `Fkey` class (certainly, with multiplicity 0..1) – to store a temporary string. Actually, the role of all these additional metamodel elements is clearly visible when transformations themselves are discussed, and normally they are added "on the fly" during the transformation program design.

4.2 The main program of transformation

Now the transformation itself as a set of MOLA programs is being described. We start with the description of the main program, where the main principles of the proposed solution can be seen. Fig. 3 shows the main MOLA program.

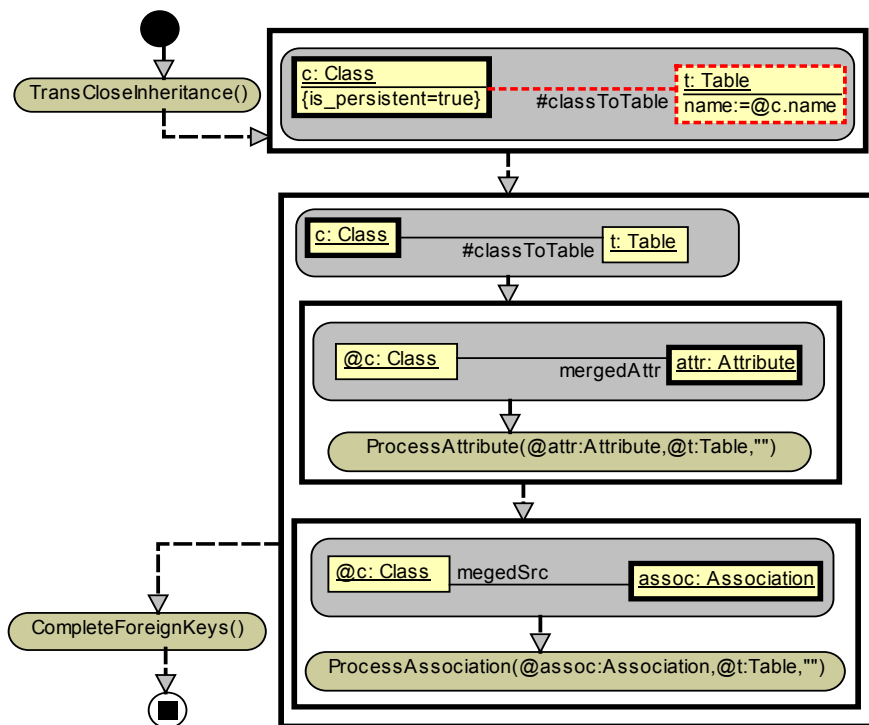


Fig. 3. The main MOLA program of the transformation.

We start with some comments on the transformation algorithm. Inheritance-related items 6 and 7 of the requirements specification [9], together with the specified precondition on inheritance (persistent classes are topmost parents), suggest that it would be convenient to process transitively the inheritance as the first step. More

precisely, for non-persistent classes the traditional UML inheritance semantics must be applied, while for persistent classes the "transitive merge up" semantics must be used. The results of this transitive closure for a non-persistent class can be stored by means of temporary associations `inherAttr` (to all inherited attributes – including the direct ones) or `inherSrc` (to exiting associations), and `mergedAttr/mergedSrc` for persistent classes respectively. Namely this inheritance processing is performed in the subprogram `TransCloseInheritance`. In all the follow-up activities the appropriate temporary associations are used instead of the original ones (`attrs` or `src`). It should be noted that many "classical" UML tools (including Rose by IBM Rational) process the inheritance namely this way – you can always see all inherited attributes/associations of a class directly.

Now the comments on the MOLA program are given. We remind that MOLA control flows have some similarity to UML activity diagram – the same Start/End symbols are used. After the subprogram call for inheritance processing, the first `FOREACH` loop starts. This loop builds an equally named table for each persistent class – note the simple pattern consisting only of the loop variable (`c:Class`) itself (with the attribute constraint expressing the persistence). An assignment expression in MOLA can contain attributes from all elements in the same loop head (or rule), prefixed by the element name. In addition to the `Table` instance, an instance of the mapping association is also built.

The next loop actually again iterates over all persistent classes, but it has a different pattern – formally, loop over all `Class` instances which have a link to a `Table` instance (which is the same since such a link and instance have been built in the previous loop). The reason why we use the other pattern now is that we want to reference both the class (`@c:Class`) and its table (`@t:Table`) in the loop body. And in turn, we couldn't insert all the actions in this loop body into the first loop – we want to build also foreign keys (in the nested subprograms), which reference another table, and during the first loop it could happen that the target table is not yet built.

The body of this loop does the main job in the whole transformation. At the top level, it consists of two nested loops – for each merged up `Attribute` (i.e., having the temporary `mergedAttr` link to the current `Class` instance) invoke the `ProcessAttribute` subprogram with appropriate parameters and for each merged up exiting `Association` invoke the `ProcessAssociation`. Namely, the use of `mergedAttr` and `mergedSrc` links (built by the `TransCloseInheritance` subprogram) ensures the fulfilment of item 7 in the requirements specification – "the resultant table should contain the merged columns from all of its subclasses". The subprograms `ProcessAttribute` and `ProcessAssociation` are recursive – they invoke themselves (indirectly), thus implementing the recursive definition of names for target columns (and the recursive drill-down as such). The third (string) parameter of these subprograms is the currently cumulated up name prefix – for the top level invocation it is just empty string. The second parameter is the `Table` instance to which the generated `Column` (if any) or `FKKey` must be attached. These subprograms actually implement rules 2, 3, 4, 5 of the requirements specification [9].

When the main job is done, there still remains something to do – foreign keys have no columns. The reason, why we couldn't fill them up "on the fly" again is – an FK

must have columns corresponding to all columns of the referenced PK, and that PK could yet be undefined. So a separate subprogram `CompleteForeignKeys` completes the job.

4.3 The principal subprograms of the transformation

In this section we analyze the principal subprograms of the transformation: `ProcessAttribute`, `ProcessAssociation`, `BuildColumn`, `BuildForeignKey` and `ProcessNonPersistent`, which jointly perform the recursive drill-down of attributes and associations for a class. We start with the `ProcessAttribute` (Fig. 4). It has three parameters – the attribute to be processed, the table to which to add the result and the cumulated name prefix (string).

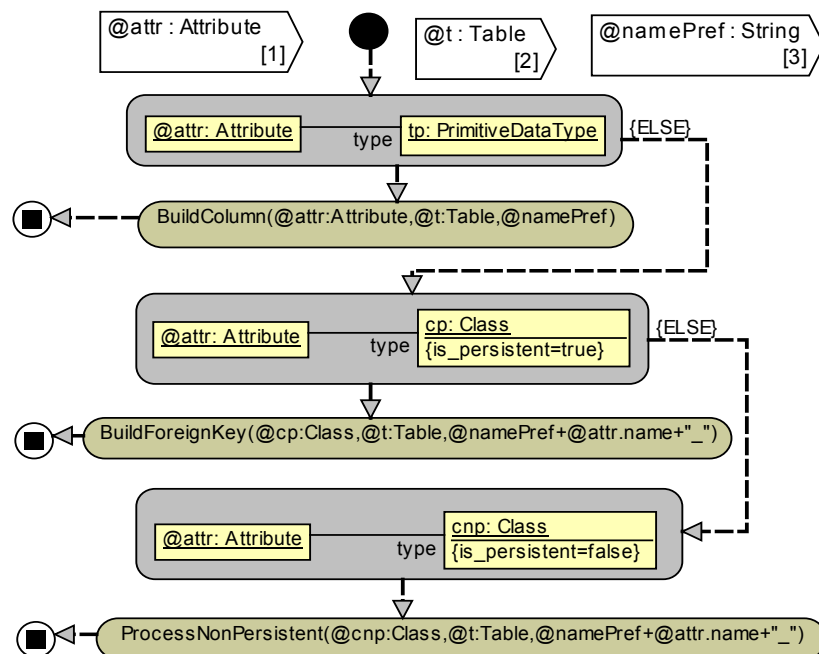


Fig. 4. `ProcessAttribute` subprogram.

This relatively straightforward subprogram implements items 3, 4 and 5 of the specification [9], by invoking the relevant subprograms. It contains no loops, but only rules. The first rule acts as a precondition for the item 3 – "an attribute has a primitive data type", therefore its unmarked (positive) exit leads to `BuildColumn` with appropriate parameters. If the pattern fails (the attribute's type is not primitive) the ELSE exit is taken. Similar graphical if-then-else constructs implement the other two cases (build foreign key if the type is a persistent class, invoke recursive processing of a non-persistent class). In both these cases the name prefix is prolonged – current attribute name added to it.

The ProcessAssociation subprogram (Fig. 5) is quite similar, except that only two cases are possible (there is no direct column generation from an association).

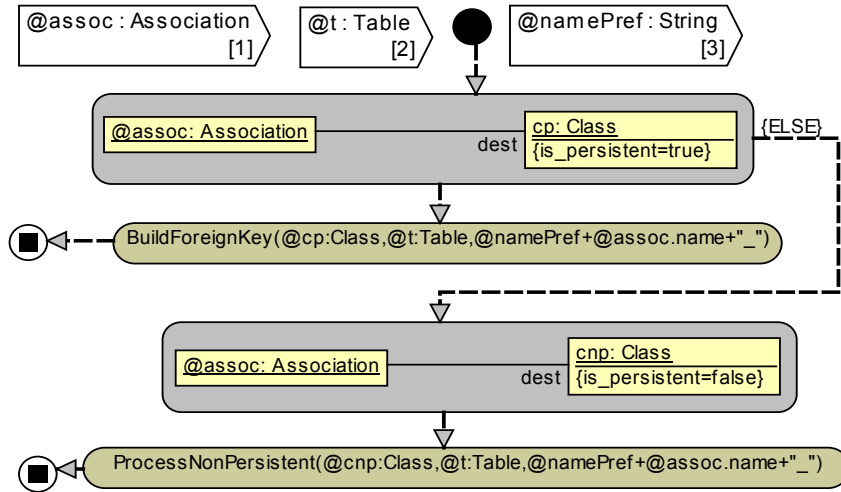


Fig. 5. ProcessAssociation subprogram.

The BuildColumn (Fig. 6) subprogram is also quite simple, it contains only rules for building instances (the ELSE exit of the first rule is semantically impossible; if the pattern does not match for the second rule the default program end is used).

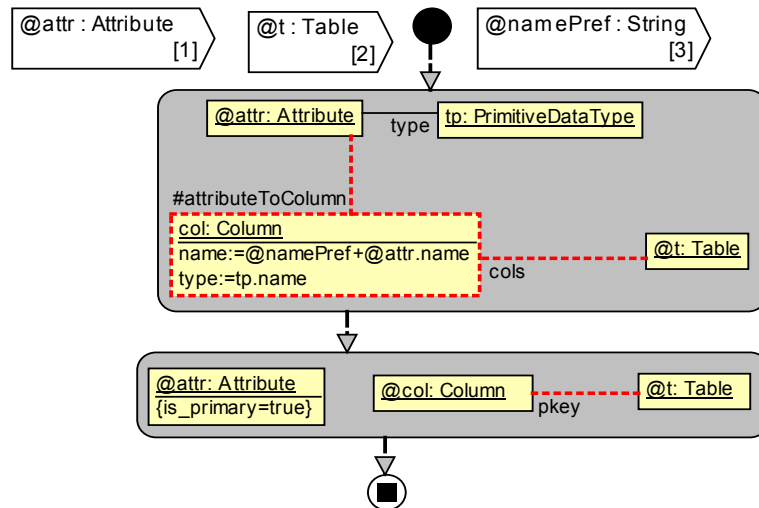


Fig. 6. BuildColumn subprogram.

In addition to building a column (using both the prefix and the current attribute), a primary attribute enforces the column to be included into the PK list.

Similarly, the `BuildForeignKey` subprogram (Fig. 7) contains a rule for building a foreign key, together with its reference to the target (note that the required `dt:Table` instance now exists for sure).

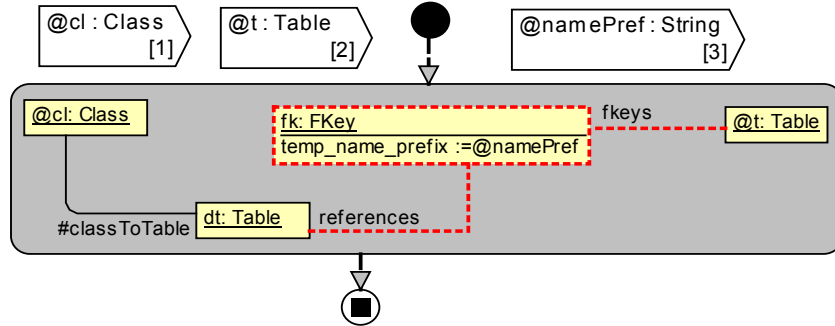


Fig. 7. BuildForeignKey subprogram.

The final subprogram in this set is `ProcessNonPersistent` (Fig.8), which completes the recursion (item 2 in the requirements [9]) for a non-persistent class (by processing all its inherited attributes and exiting associations).

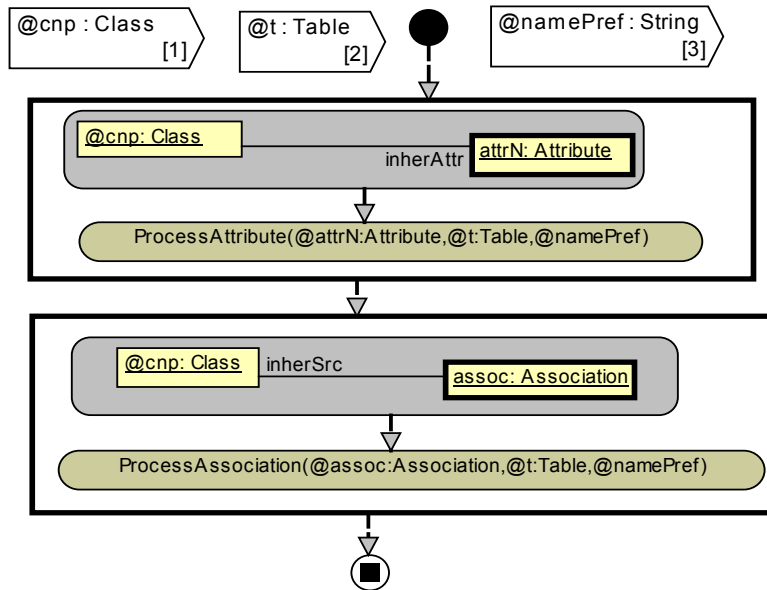


Fig. 8. ProcessNonPersistent subprogram.

4.4 Other subprograms of the transformation

We start with the `TransCloseInheritance` subprogram (Fig. 9), which was already mentioned in 4.2. Its role is extremely simple – for non-persistent classes

perform `ProcessInheritance`, but for persistent – `ProcessMerge` (it was already explained in 4.2, why the specification implies such division). Both these subprograms process parent links recursively, therefore the "initial calls" to them have both parameters set to reference the current class (a class attribute is also an inherited attribute and so on). Alternatively, there could be one loop iterating over all classes, but with an if-then-else in the body.

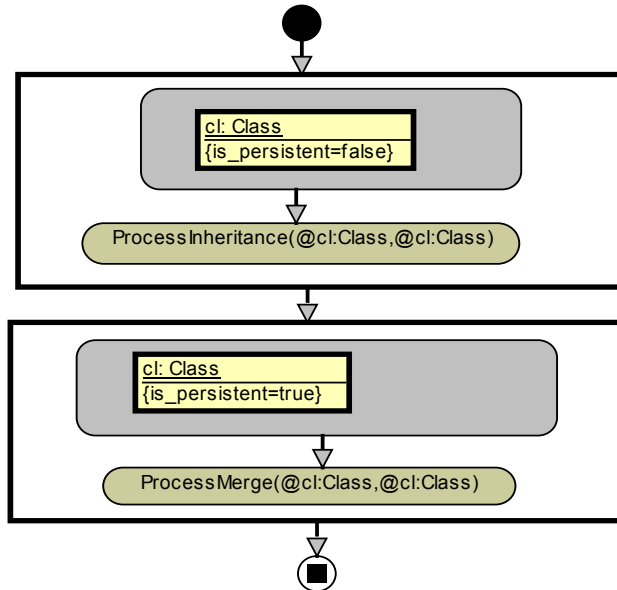


Fig. 9. TransCloseInheritance subprogram.

Subprograms performing the real transitive closure – `ProcessInheritance` (Fig. 10) and `ProcessMerge` (Fig. 11) are very similar – the former iterates up via parent link, the latter – down. However, the difference in closure semantics implies some difference in programs. For inheritance, an attribute must not be inherited if there already is an (inherited) attribute with the same name. This fact is expressed by (the only one in the whole example) NOT constraint in the `attr:Attribute` pattern element – the instance of `attrsup:Attribute` doesn't match, if there is an instance of `Attribute` linked via `inherAttr` to the same `Class` and having a name equal to `attrsup` name.

Since the "up" multiplicity of parent is 0..1, there is no loop involving the recursive call, but just an if-then-else branch.

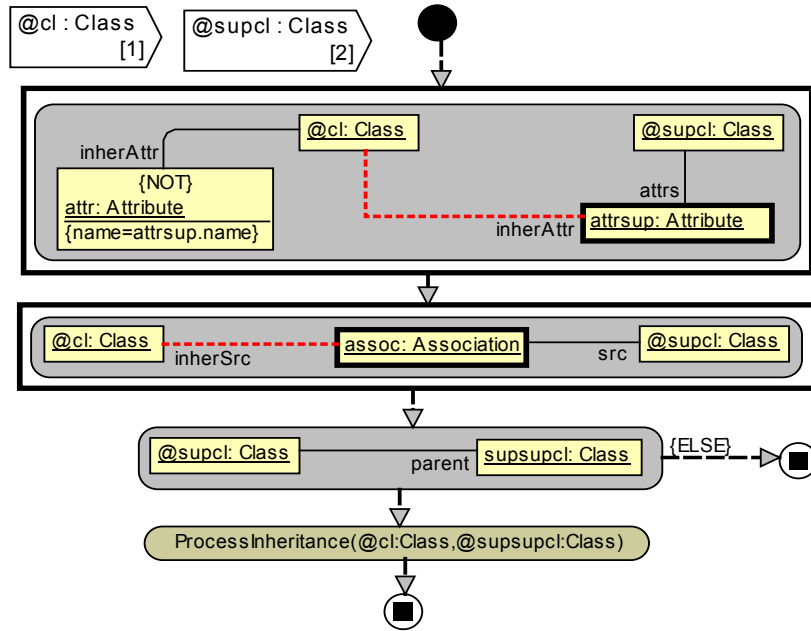


Fig. 10. ProcessInheritance subprogram.

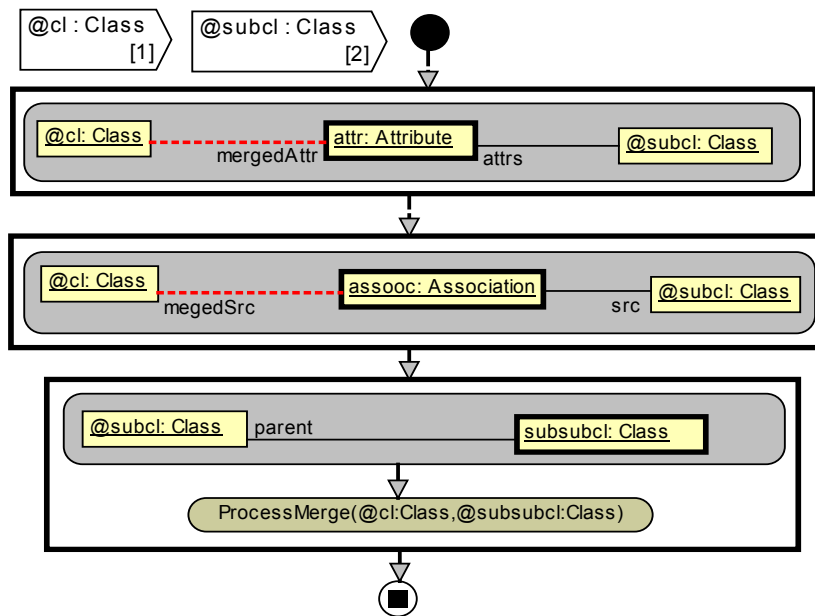


Fig. 11. ProcessMerge subprogram.

The `ProcesMerge` subprogram is simpler – there is no overriding in the merge definition. On the other hand, the "down" multiplicity of the parent link is *, therefore the recursive call is within a loop.

Finally, the `CompleteForeignKeys` subprogram does a simple job – it runs through all foreign keys and for each builds a set of columns (one for each column of the relevant primary key), using the name prefix, temporarily stored in `FKey` by the `BuildForeignKey` subprogram. Then the temporary attribute is cleared.

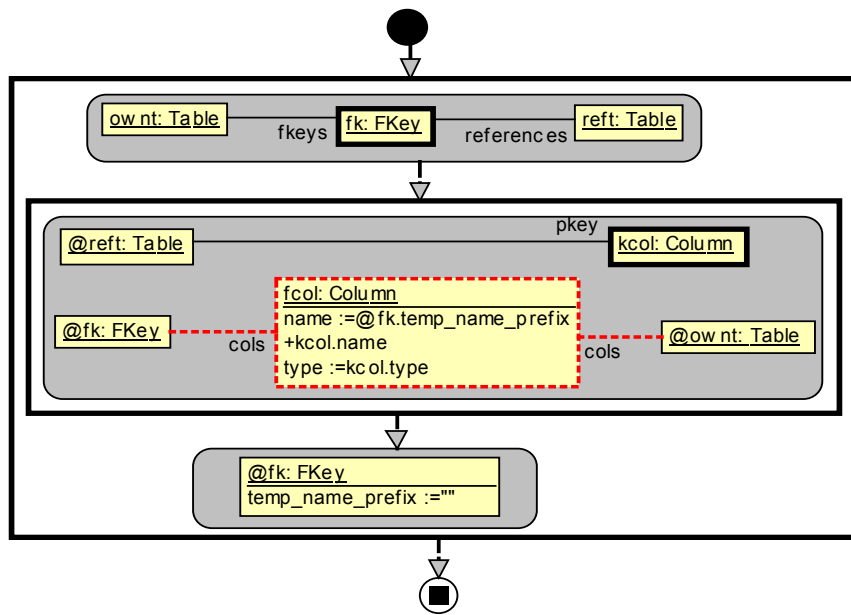


Fig. 12. `CompleteForeignKeys` subprogram.

This completes the mandatory example in MOLA.

4.5 Analysis of the example implementation

Certainly, the same way there is no one absolutely best implementation of `OrderEntry` subsystem for `MySales`, there is no absolutely best implementation of a transformation. Any analysis is subjective to a degree.

Authors themselves consider this implementation of the mandatory example a very nice application of MOLA. It seems to be very readable and clear (readability is subjective too!), no missing feature of the MOLA language has been found. It seems also that a certain optimum has been reached between the use of graphical patterns in loops and rules and purely programmatic constructs (sometimes one can replace another). It should be noted that only relatively recently the role of Recursive Call pattern in MOLA has been fully estimated. Though recursive calls have been permitted from the beginning, early examples of MOLA [11,14] all try to use only pure iteration for a very similar "drill-down" transformation, which makes the

implementation more clumsy. It should be noted, that recursive calls could be used even more deeply – inherited (or merged) attributes or associations could be recursively found each time they are needed for the drill-down, but this was considered to be an overuse of recursion reducing the clarity. Namely therefore the temporary associations completely separating the processing of inheritance and drill-down were introduced. The recently introduced true if-then-else construct also makes the transformation behavior description clearer.

It is nearly impossible to compare textual transformation languages (textual QVT-Merge, ATL, MTL et al) to MOLA – simply each style has its proponents. The textual definitions are, certainly, much shorter but we consider them significantly less readable and consequently, more error prone. It should be noted that this example was intentionally completed without the use of MOLA TEE, using only manual "code inspections". Then it was subjected to proper testing via MOLA TEE, and only one error was found. Taking into account that published textual transformation examples contain bugs frequently enough it seems that more sizeable transformation definitions in MOLA pay off.

A more fair would be comparison to other graphical transformation languages (graphical QVT-Merge, FUJABA SDM, GReAT). Authors have not performed any direct comparisons due to unavailability of respective environments for these languages. Some indirect comparison could be made only to the graphical QVT-Merge, where the latest proposal document [1] contains a unidirectional transformation example, similar to this workshop example (but having some significant differences). An equivalent functionality seems to be definable more compactly in QVT-Merge than in MOLA. But since the only control structure in QVT-Merge governing rules actually is a recursive call (via the Where and When constructs), this notation seems to be much harder to read and understand. This fact was confirmed to a certain degree via experiments involving master students in CS.

So it is up to users to decide which transformation definition facilities are better.

5 Use of MOLA TEE for the example

When a transformation is defined in MOLA (using the MOLA TDE) it can be compiled to check its syntax. However, a proper transformation validation can be done only using source model test examples within the MOLA TEE. Only the GMF-based version (see section 3) can be used for the example, since its metamodel is not part of the standard UML. As it was outlined in section 3, some visual facilities for building source models and viewing the transformed target models must be defined in GMF.

Initially the MOLA metamodel (combined) must be ported into the GMF metamodeling facility. In the case of the simple metamodel for the example (Fig.2) this could be done without any complexities (namely to facilitate the porting some role names were already added to the metamodel).

At first the simplest way of instance visualization – via customized model trees will be demonstrated. This approach is similar to the generated from a (meta) model tree and editor set in Eclipse EMF [18], but is significantly more flexible. For example, we can chose to represent a `Class` instance as a node, which shows the

name, persistence and possible parent (the latter ones with keyword style separators to distinguish, which of the values are present). Then we can specify that child nodes of this node correspond to `Attribute` instances of the class (i.e., accessible via `attrs` link), each node showing the name, type and "primary". Additional node type can be defined for associations, containing name plus source and target class names. Primitive types also must be shown as nodes. In addition, customized object dialogs can be defined for the main metaclasses (here `Class` and `Association`, with attributes as elements inside the `Class` dialog). GMF has also default object dialogs (like property editors in EMF), but they can be not so convenient for use. Fig. 13 shows the example tree in GMF (according to the abovementioned definitions), which corresponds to the input example – Fig. 3 from the workshop CFP. Parent is empty everywhere since there is no inheritance in this example (there is no way to remove the separator if the value is empty).

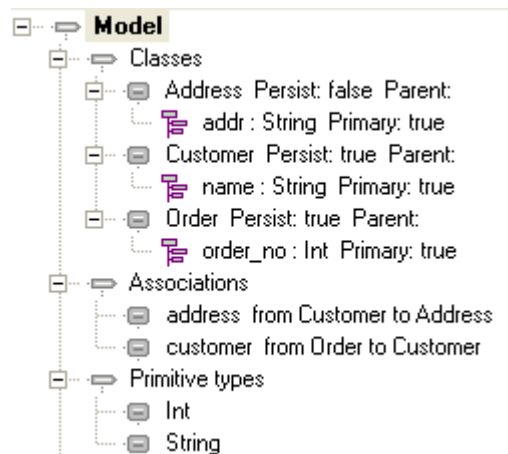


Fig. 13. Input example in GMF.

Similarly, tree nodes for the target model must be defined. Here the sole top level node should be `Table`, showing the name. It has two types of children – columns and foreign keys. `Column` nodes display name, type and whether part of PK. For both table and column nodes it can be shown from which source model elements they were generated (via the mapping associations), visually separated by ":-" string – this is an element of explicit traceability. For foreign key nodes the referenced table may be shown, with included columns as children nodes.

Now it remains to export the instance data (source model) from GMF repository to MOLA runtime repository, start the selected transformation and import back the transformed model to the GMF repository. All these actions have been added as standard services to GMF. Fig. 14 shows what was obtained from the source model in Fig. 13.

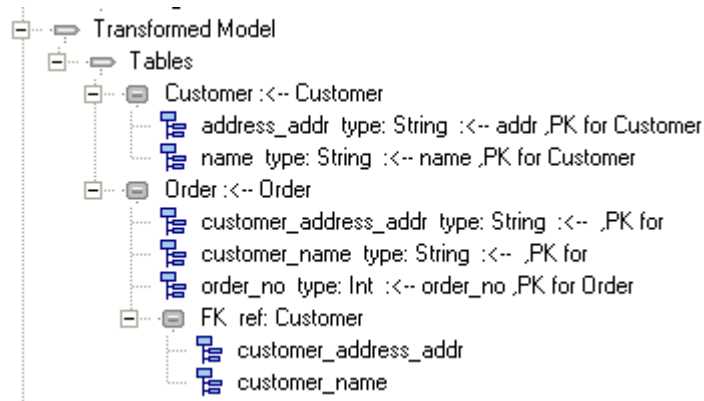


Fig. 14. Transformation results in GMF (obtained from data in Fig. 13).

It can be easily verified, that the results do comply with the Fig. 4 in the CFP [9] (columns which are not PK show the empty ", PK for " separator, columns which are not direct maps of source model attributes, show empty " :<-" string). Namely this way the sole transformation error was detected – the underscore symbol in names initially was placed wrongly.

Certainly, to validate the defined transformation to a certain degree, much more test examples would be needed, e.g., inheritance is not tested at all. Larger examples can be built via this visualization for sure, but we want to demonstrate briefly the other possibility in GMF – present models as custom diagrams. Both the source and target metamodels of the example satisfy "GMF diagramming" requirements, only a special metaclass (representing a "domain diagram") must be added to each. This requires also one "technical subprogram" to be added to the transformation end – the domain diagram instance must also be built automatically. All these "scaffolding activities" in no way affect the original models or transformation. Fig. 15 shows the source model represented as a slightly non-standard class diagram – according to the assumed metamodel. Additional metaattributes (*is_persistent*, *is_primary*) are displayed as tagged values. Definition of this diagram-style presentation is more complicated, it must be specified, e.g., that *Class* maps to an auxiliary metamodel element *ClassSymbol*, which in turn has a rectangular shape and contains three text compartments one of which (for attributes) is a list compartment. Thus a sort of model transformation (domain to presentation) actually is defined in GMF, more details can be found in [17]. The definition result is a "normal" graphical editor for this variation of class diagrams, with standard facilities to be found in diagramming tools. The example in Fig. 15 (built via this editor) is a slightly adapted advanced case study (Fig. 5 in CFP), which was not meant to be used for the strict transformation rules of the mandatory example (therefore the results will be slightly unexpected). The adaptation had to be done to satisfy the preconditions on class models. Nevertheless it is a good test for the transformation – many "use cases" can be observed on it.

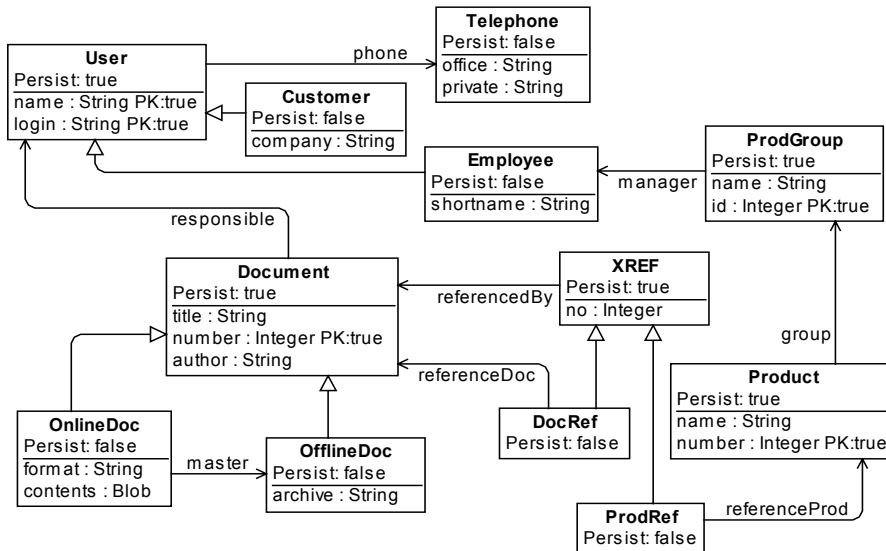


Fig. 15. Complicated input example as a GMF class diagram.

Transformation results frequently also can be displayed as a diagram, in this case an "RDBdiagram" (somewhat similar to Fig. 6 in CFP [9]) is defined. Tables are presented as rectangles showing columns in a list compartment, separate compartments present members of PK and the reference for each of the FKs. The columns included in an FK are shown as a list attached to the line representing this FK (unfortunately, FKs have no names in this transformation). When the transformation is run on the example and the transformed instances imported back into GMF, the diagram itself is displayed automatically via the GMF auto-layout facility.

Fig. 16 shows the result of transformation when applied to the model in Fig. 15. It can be noted that only persistent classes result into tables, but inheritance and drill-down generate a lot of new columns – according to the transformation specification. No transformation program errors were detected in this test, which can be considered as an exhaustive enough (though authors have not tried to apply any formal testing completeness criteria). The only conclusion is that in practice more sophisticated transformations from class models to RDB should be used.

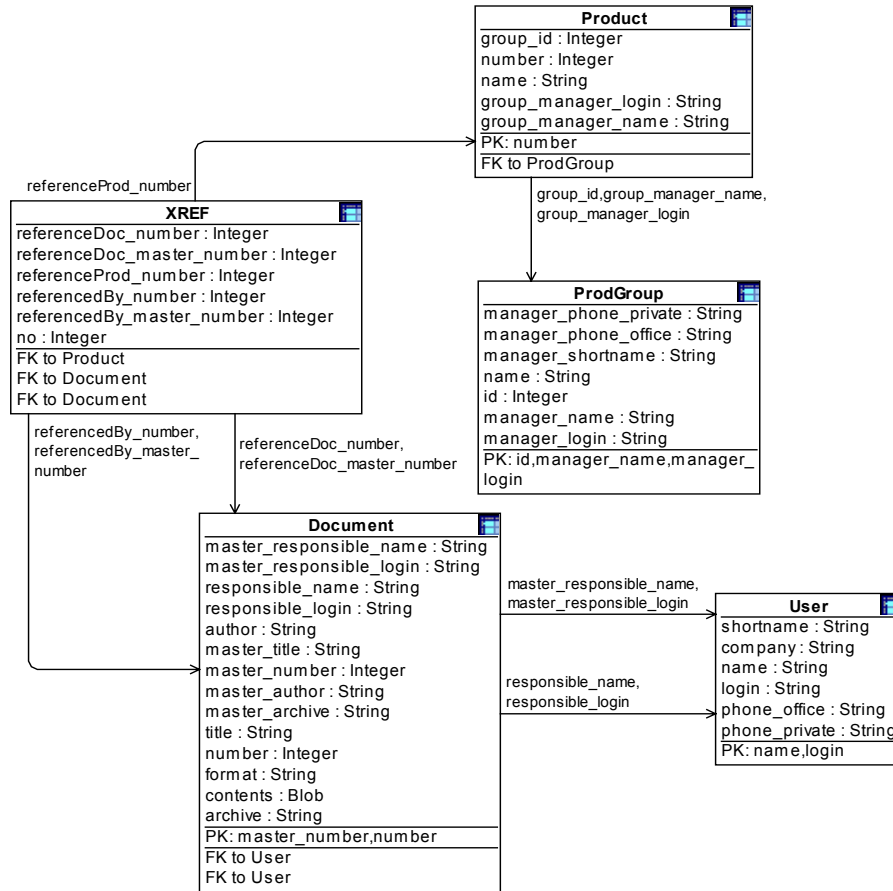


Fig. 16. The transformation result as an RDB diagram.

6 The optional example – nondeterministic FSM to deterministic

In this section we briefly describe one more example – the transformation of a nondeterministic automaton (FSM) to a deterministic one. Automata are assumed to be language recognizers (no output), a nondeterministic one can have many initial states and many final states, a string belongs to the language if there is a path from an initial to a final state marked by this string (empty or lambda moves are not included). Thus a simplest possible definition is assumed. For deterministic automaton the standard language recognizer definition is used. Automata are defined as sets consisting of state, event (=input alphabet element) and transition instances. The classical determinization algorithm is implemented – explore the state powerset (set of all subsets) space, by starting from the "initial set" and trying to expand the reachable set of statesets by applying transitions for all possible events and analyzing

whether a new stateset has been reached by the given event (or it is a copy of existing one). When nothing more can be reached, the reached powerset elements are coded as new states of the deterministic FSM, and new transitions are defined accordingly, as well as the initial state and final states.

Fig. 17 shows the metamodel (source = target), with the `StateSet` class used during the algorithm run. Fig. 18 – 22 show the main MOLA program and subprograms implementing the abovementioned algorithm. Some of the subprograms use additional MOLA elements not used in the main example.

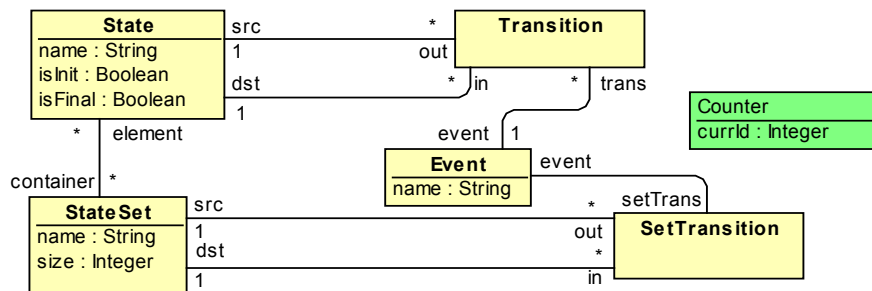


Fig. 17. Metamodel of automaton.

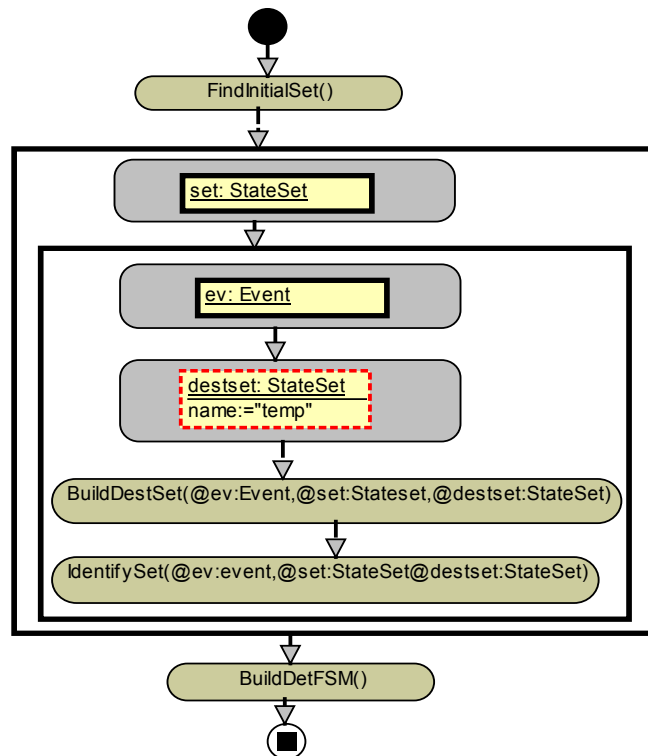


Fig. 18. Main MOLA program for the determinization.

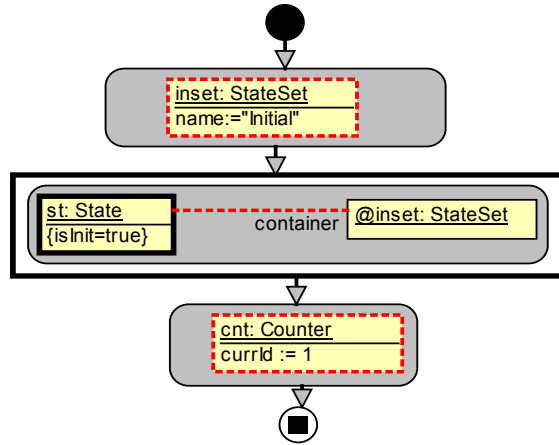


Fig. 19. Subprogram FindInitialSet.

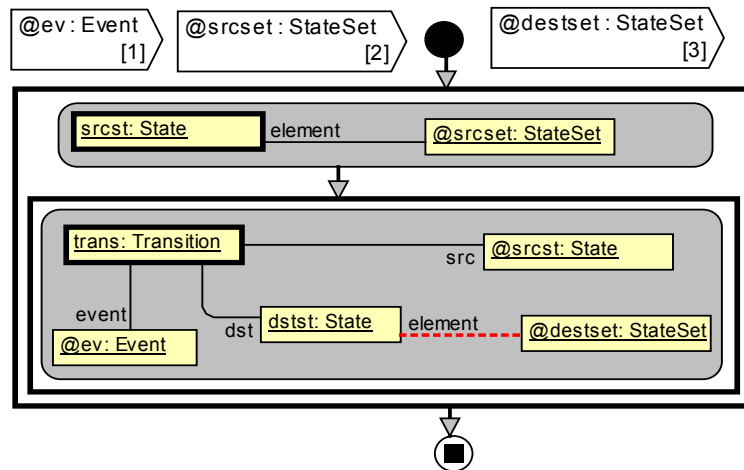


Fig. 20. Subprogram BuildDestSet.

The next subprogram `IdentifySet` uses more complicated OCL expressions in constraints – subexpressions of the form `element_name.role_name`, which denote an instance set (if the multiplicity is *) and elementary OCL operations on sets (here – the set equality). Two special control constructs – explicit *continue* (flow to the loop border) and *return* (flow to end symbol) are used in the first loop.

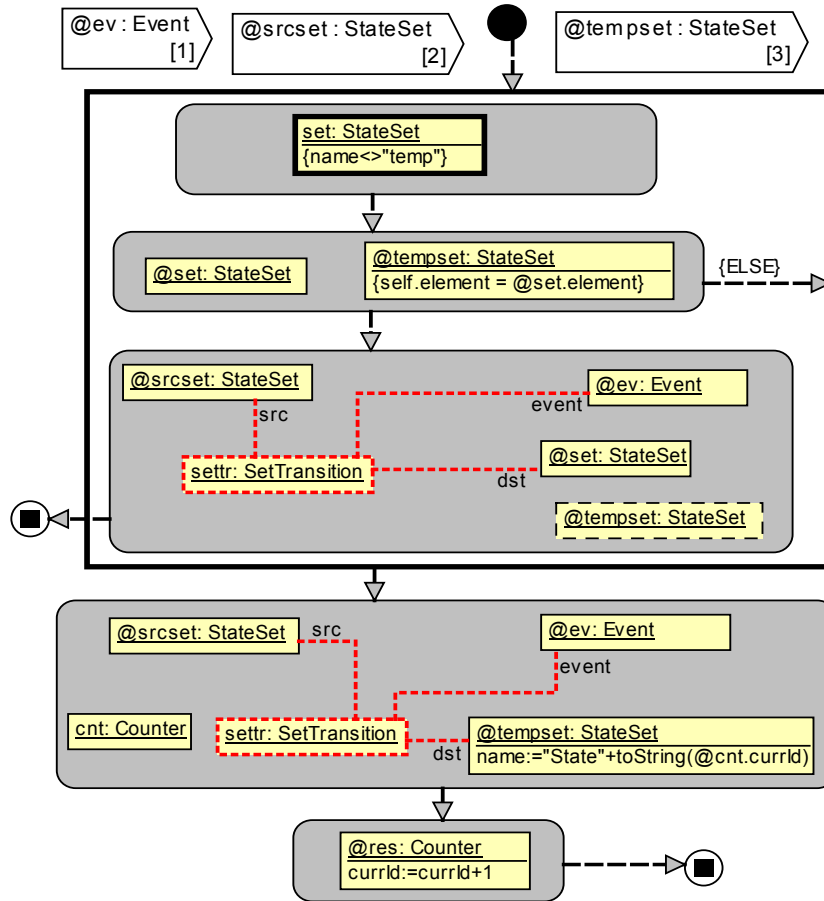


Fig. 21. Subprogram IdentifySet.

The subprogram BuildDetFSM also uses OCL set operations in constraints – notEmpty and the quantifier exists.

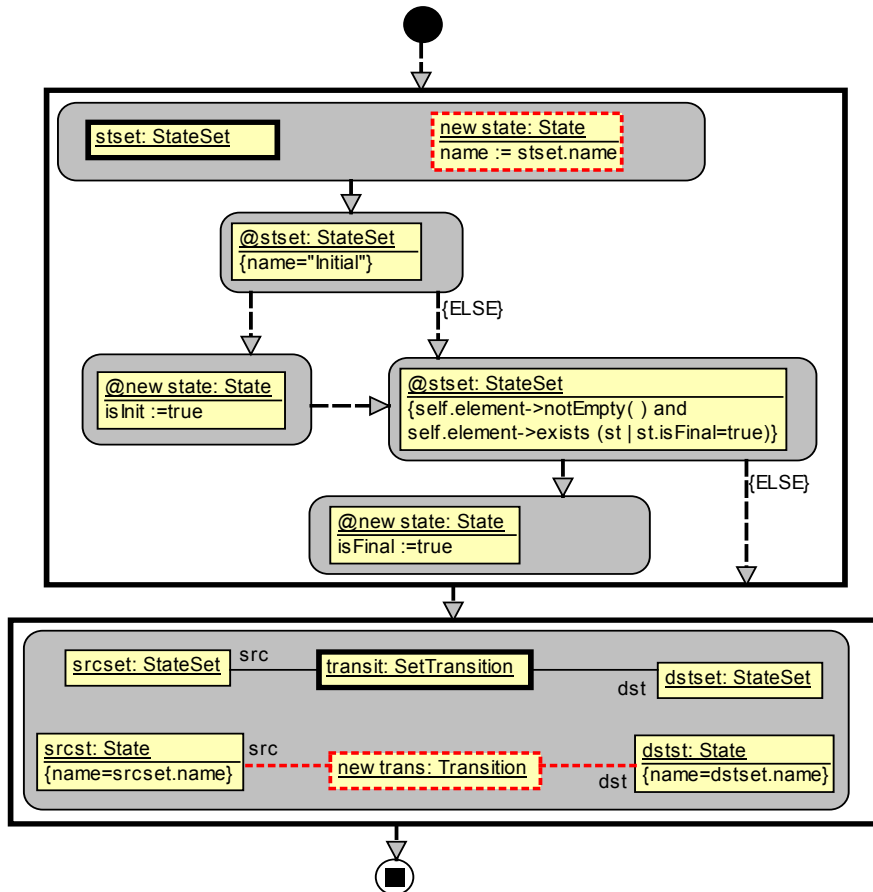


Fig. 22. Subprogram BuildDetFSM.

Authors consider this example also a right balance between the textual and graphical style of transformation specifications. Namely to make the example maximally readable, explicit sets defined via associations from an instance and OCL set operations are used in patterns. Certainly, the example could be specified "100% graphically", using nested loops, but this seems not to be the best choice. The extended use of OCL is not yet implemented in MOLA VM, therefore the example has not been validated in MOLA TEE.

6 Conclusions

The description of the implementation of the mandatory transformation example in MOLA (in section 4) provides, according to the authors' view, a good style of a graphical transformation definition. The increased size of the solution is compensated by a better readability, which in turn ensures that less effort for the transformation

development is required and it is less error prone. The latter fact to a certain degree has been confirmed by a controlled experiment – developing the transformation and only then testing it. The MOLA execution environment, based on GMF, also occurred to be very fit for building test models and executing the transformation on them. Thus the practical transformation validation, using the facilities to build/view models in a graphical form, appeared to be completely satisfactory. The optional example, in turn, demonstrates that graphical pattern definition facilities should not be overused – they can naturally be combined with the use of OCL in MOLA element constraints.

Certainly, there are more problems in practical transformation development. First, the transformation composition is more the tool than language issue – in MOLA environment, for sure, it is possible to apply consecutively several transformations while the model data are in the runtime repository (certainly, if the metamodels are consistent to this). Bidirectional or incremental transformations certainly don't come for free in MOLA because it is an outspokenly procedural language. Reverse or incremental transformations must be developed specially with the goal in mind, but some experiments show that MOLA pattern features are powerful enough to implement the relevant source-target relations easily. It is especially easy if the mapping associations are used adequately for the direct transformation, e.g., it can be easily detected in the example that a new `Table` has been added to the target model which has no link to its `Class`. To sum up, the MOLA language seems to meet all the main transformation technology requirements, certainly, the existing MOLA tool will be extended to meet all the aspects of practical usability.

References

- [1] QVT-Merge. URL: <http://www.omg.org/docs/ad/05-03-02.pdf>
- [2] ATL. URL: <http://www.sciences.univ-nantes.fr/lina/atl/>
- [3] MTF. URL: <http://www.alphaworks.ibm.com/tech/mtf>
- [4] Tefkat. URL: <http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/>
- [5] Tratt L. “The MT model transformation language”. Technical report TR-05-02, Department of Computer Science, King's College London, May 2005.
- [6] Fujaba User Documentation.
URL: <http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>
- [7] 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
- [8] The Attributed Graph Grammar System (AGG). URL: <http://tfs.cs.tu-berlin.de/agg/>
- [9] Model Transformations in Practice Workshop. Call for papers (CFP).
URL: http://sosym.dcs.kcl.ac.uk/events/mtip/long_cfp.pdf
- [10] Graphical Modeling Framework (GMF, Eclipse technology subproject).
URL: <http://www.eclipse.org/gmf/>
- [11] A. Kalnins, J. Barzdins, E. Celms. “Model Transformation Language MOLA”. Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004), Linköping, Sweden, June 10-11, 2004. pp.14-28.
- [12] Kalnins A., Barzdins J., Celms E. “Model Transformation Language MOLA: Extended Patterns”. Selected papers from the 6th International Baltic Conference DB&IS'2004, IOS Press, FAIA vol. 118, 2005, pp. 169-184.

- [13] A. Kalnins, J. Barzdins, E. Celms. "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. URL: <http://heim.ifi.uio.no/~janoa/wmdd2004/papers/>
- [14] A. Kalnins, J. Barzdins, E. Celms. "MOLA Language: Methodology Sketch". Proceedings of EWMDA-2, Canterbury, England, 2004. pp.194-203.
- [15] MOF 2.0 Core Final Adopted Specification.
URL: <http://www.omg.org/docs/ptc/03-10-04.pdf>
- [16] A. Kalnins, E. Celms, A. Sostaks. "Tool support for MOLA". (Preliminary version). GPCE'05. Paper accepted to the workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005
- [17] E. Celms, A. Kalnins, L. Lace. "Diagram definition facilities based on metamodel mappings". Proceedings of the 18th International Conference, OOPSLA'2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23-32.
- [18] UML 2.0 Eclipse EMF. URL: <http://www.eclipse.org/uml2/>
- [19] Microsoft SQL Server 2000 Desktop Engine (MSDE 2000).
URL: <http://www.microsoft.com/sql/msde/default.asp>

Simple and Efficient Implementation of Pattern Matching in MOLA Tool

Audris Kalnins, Edgars Celms, Agris Sostaks
University of Latvia, IMCS
29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Edgars.Celms}@mii.lu.lv, agree@os.lv

Abstract - One of crucial problems for model transformation implementations is an efficient implementation of pattern matching. The paper addresses this problem for MOLA Tool implementing the model transformation language MOLA. Another goal has been to keep the implementation as simple as possible. The paper presents one possible solution to the combined problem where an SQL database with fixed schema is used as the MOLA runtime repository. A natural coding is selected where a MOLA pattern match can be mapped to a single non-standard self-join SQL query. The paper shows that a sufficient matching efficiency can be obtained this way. The generated queries are analyzed from the table join order point of view and it is shown that the default query optimization for the MySQL database can find an order close to optimal. This analysis and performed experiments are used to conclude that at this moment MySQL is the most fit for MOLA implementation among "free" relational databases. In addition, benchmark tests based on a simple natural model transformation problem are used to estimate efficiency of the selected implementation architecture and to compare MOLA Tool to the popular graph transformation tool AGG. Benchmark tests confirm the efficiency of the current MOLA Tool implementation and applicability of MOLA language to MDD-specific tasks.

I. INTRODUCTION

Nearly all of model transformation languages use the pattern matching as the main functional element for defining how the source model components must be transformed to the target model. So does the transformation language MOLA analyzed in this paper. When a transformation language is implemented, the implementation of pattern matching typically is the most demanding component to implement and also the key factor determining the implementation efficiency.

This issue has been analyzed theoretically in various contexts. For MOLA, authors of this paper have already shown that a very efficient pattern matching implementation is possible in principle [1], however this implementation would require significant effort to build and therefore is appropriate only for an industrial tool. For other transformation languages, the most thorough analysis has been performed for the GReAT language [2].

In this paper, the problem appears in another setting. An academic model transformation tool supporting MOLA has been built using limited resources, and for this tool both simple and sufficiently efficient implementation has been required.

Another related problem is the choice of runtime repository, since the pattern matching is very intimately related to repository access mechanisms. A standard choice, used in most academic model transformation tools [3,4,5] and some industrial ones [6,7] too, is a metamodel based repository, such as Eclipse EMF [8], MDR [9] or similar ones. These repositories typically have a low level universal API for retrieving class instances. This solution would make the implementation of pattern matching and other language features significantly more complicated.

Several possible solutions for these two related problems in the context of MOLA tool have been analyzed. The final decision, which is described in this paper, occurred to be rather non-typical for model transformation tools – the best kind of repository would be a relational database with fixed schema – tables coding the metamodel and model in the most natural way. The central idea of this implementation is that a MOLA pattern match operation can be implemented by a single SQL query. And this query is easy to generate from the pattern definition.

The only remaining problem is whether such a rather non-standard query (using multiple self-joins) can be processed efficiently by database engines. Analysis in the paper shows that not all engines perform efficiently enough, but there are freely available ones which can do this, currently the best one is MySQL. These results are in concordance with other papers analyzing usability of SQL for pattern matching [10,11] (however, a completely different database structure is used there and the experiment setting is also different).

The paper describes the solution used in MOLA tool. After a brief reminder of MOLA language and an overview of the MOLA tool architecture, the core of the tool – the MOLA virtual machine (VM) is defined (section 5). The most appropriate database structure and the mapping of a pattern to an SQL query are described in detail in section 6. Section 7 analyzes the performance issues of generated queries, especially the table join order. Section 8 contains a benchmark test, which compares the transformation of simplified UML class diagram to simplified OWL diagram implemented both in MOLA Tool and the popular graph transformation tool AGG [12] on various model sizes. The results confirm the efficiency of MOLA implementation and its practical usability.

II. WHAT IS MOLA

MOLA is a graphical model transformation language developed at the University of Latvia [1,13,14,15,16,17,18]. Its main distinguishing feature is the use of simple procedural control structures governing the order in which pattern matching rules are applied to the source model. Due to the large number of papers on MOLA language (the most important ones are [13,15,16,18]) we do not repeat the language description in this paper. In addition, there is a web site <http://mola.mii.lv/> devoted to MOLA where all these papers and a formal description of MOLA are available. Just to clarify the terminology, we very briefly remind here the main elements of MOLA.

Source and target metamodels are combined in one class diagram, where the added mapping associations link the corresponding classes in source and target metamodels, these associations are used for traceability and transformation structuring.

The MOLA transformation program consists of one or more **MOLA diagrams** (one of which is the main). A MOLA diagram is a sequence of graphical **statements**, linked by arrows. The most used statement type is the **loop statement** – a bold-lined rectangle. Each loop statement has a **loop head** – a special statement (grey rounded rectangle) containing the **loop variable** and the **pattern** – a graphical condition defining which instances of the loop variable must be used for iterations. The pattern contains **elements** – rectangles containing `instance_name:class_name` – the traditional UML instance notation. Elements specify instances of which classes must be matched. The loop variable is also a special kind of element, it is distinguished by having a bold-lined rectangle. In addition, the pattern contains **links** (metamodel associations) – a pattern actually corresponds to a metamodel fragment. Pattern elements may have attribute constraints – simple OCL expressions. The semantics of this loop statement (called the **FOREACH** loop) is natural – the loop is executed once for each instance of the loop variable, where the condition is true – the pattern elements can be matched to existing instances, attribute constraints are true on these instances and required links are present. Loops may be nested to any depth. There are two types of **FOREACH** loop – *fixed* (the scope of matched class instances does not change – pattern matching is performed only once) and *not fixed* (the scope of matched instances can change and pattern matching must be performed after each iteration to see changes in the appropriate instance scope). The loop variable (and other element instances) from an upper level loop can be referenced by means of the reference symbol – an element with @ prefixed to its name. There is also the **WHILE** loop in MOLA, which is less used and not analyzed in this paper. Another important statement in MOLA is **rule** (also a grey rounded rectangle) – a statement consisting of pattern and actions. These actions can be element or association building actions (denoted by red dotted lines) and delete actions (denoted by dashed lines). An attribute value of an element can be set by means of attribute assignments. A rule is executed once (or not at all if pattern fails) – thus it plays the

role of if-statement too. A subprogram is called by means of a call statement (possibly, with parameters – element references). An example of MOLA program is given in section 8.

III. PRECISE SEMANTICS OF PATTERNS IN MOLA

The general semantics of pattern matching is quite similar in all pattern-based transformation (QVT) languages, including the emerging OMG standard [19], nevertheless there are some specific features in any language. A pattern specifies instances of which metaclasses must be selected, how they must be linked by association instances and what attribute-based constraints for the instances must be satisfied. Certainly, there may be several occurrences of the same metaclass in a pattern, then instances must be matched accordingly. A **match** for a **pattern** is a **set** of source model elements – **instances** of metamodel **classes** and **associations**, each of which is associated to the corresponding pattern element and satisfies constraints in the pattern.

The subtleties in pattern matching for different QVT languages lie in the fact which matches must be found. In MOLA patterns are used for loop heads and rules. For a **FOREACH-loop** head **all** matches must be found which contain distinct instances of the loop variable. For all other pattern elements it is irrelevant namely which of the valid instances is selected (in some other QVT languages the semantics is more complicated here). In other words, there is an implicit existential quantifier placed on all elements, except the loop variable (and the reference elements for which the choice is already fixed). Another subtlety for (*not fixed*) loops in MOLA is that the source instance set from which distinct instances of the loop variable are selected may be replenished during the loop execution. For **rules** only **one** valid match is required (any of them if there are several), or the fact that there is none. It must be mentioned (see [1] for more details), that in a semantically correct MOLA program typically there is no much indeterminism during the pattern match – these seemingly "free" pattern elements actually are uniquely determined by the selected loop variable instance. Another positive aspect of match semantics in MOLA is that it facilitates match efficiency – not all instance combinations must be searched.

These simplest kinds of patterns are called **positive** patterns. Another kind of patterns in MOLA are the **negative** ones, which contain the **NOT** constraint on some **class** elements (as it is in [19], earlier MOLA versions [13,14] had NOT constraints only on pattern links). An element with NOT constraint expresses the fact that there must be no instance of the given class satisfying the attribute constraints and linked by the specified associations to the other (positive) pattern elements. An association linking two negative elements in a pattern is considered senseless in MOLA. Though sufficient for most transformations, these syntax features for patterns are not formally complete – an arbitrary universal quantifier on properties of an instance set cannot be expressed this way. Therefore one more element – the **NOT-region** (a rectangle with NOT tag containing some other pattern elements) must

be introduced. A NOT-region expresses the fact that there must be no instance set for the pattern elements inside the region, which satisfy the "inside" conditions and are linked by associations crossing the region border to other positive elements. NOT-regions may be nested, but no association may link two NOT-regions. Since NOT-regions are not frequently used, the current implementation does not support them.

IV. MOLA IMPLEMENTATION OVERVIEW

The current version of MOLA tool has been developed with mainly academic goals – to test the MOLA usability, teach the use of MDD for software system development and perform some real life experiments. This has influenced some of the design requirements, though with easy usability as one of the goals and sufficient efficiency the tool has confirmed its potential as an industrial tool too.

Similarly to many MDD environments, MOLA environment consists of two major parts: **MOLA Transformation Definition Environment (TDE)** and **MOLA 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, which define new model transformations in MOLA for the adopted MDD technology or modify the existing ones from a transformation library to better suit the needs of a specific project. TEE is intended for mass usage by software developers applying the chosen MDD technology and transforming their models from one step to another. One of versions of TEE is a MOLA plug-in for the UML tool RSA.

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.

A more detailed overview of MOLA environment architecture is given in [20].

V. BASIC PRINCIPLES OF MOLA VIRTUAL MACHINE

As it was already mentioned in the introduction, the goal of this research 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 paper 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 [8]. Several experimental model transformation tools have been built using EMF as a repository [3,4,5]. The EMF API in Java provides the most basic actions for building

a pattern matcher. Netbeans MDR [9] has somewhat similar characteristics and is used in [6].

The authors of this paper have already shown [1] 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 [1]) make the implementation sufficiently complicated. Therefore another solution was considered – to a 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 [21] – similarly as EMF does. On the other hand, the desire was to use the powerful capabilities of SQL Select 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 [1].

VI. 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. The metamodeling facilities in MOLA are approximately those of EMOF[21]. 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. 1). 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. 1). 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 for the paper. 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 [10].

Now we will show 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. Then 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. A more non-trivial part of the Where clause must specify that each link does link the

substituted by additional aliases of `attr_inst` in the From clause, in addition, the transformed expression must be added to the Where clause. Currently all MOLA expressions have direct counterparts in SQL.

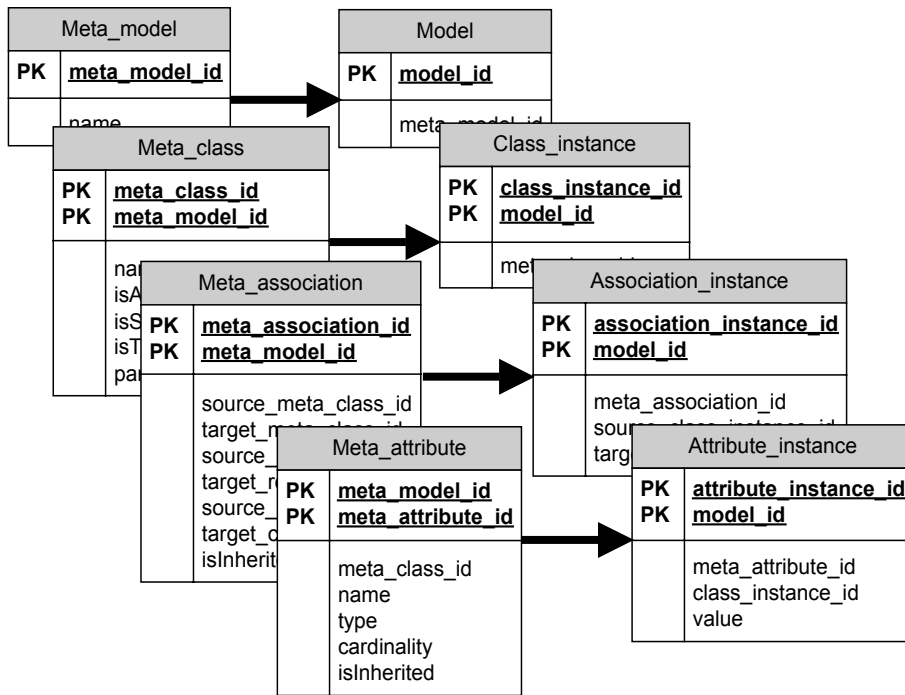


Fig. 1. SQL Tables for storing metamodels and models.

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. 2) 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

Fig. 2 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

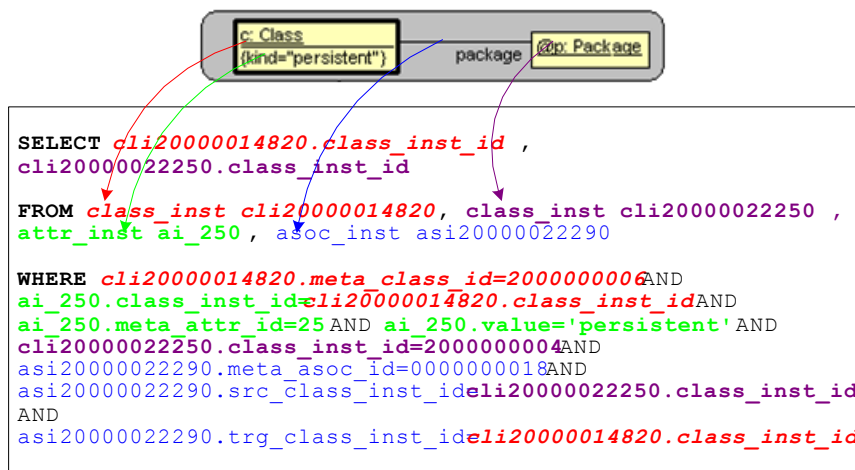


Fig. 2. Generation of an SQL query from a pattern.

constraints, which already are Boolean expressions. However, the simple attribute names used in MOLA constraints must be

by the MOLA compiler and therefore are unreadable.

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. To cope with the fact that MOLA loops which are *not fixed* can replenish the instance set used for the match, actually for loop patterns the query is re-executed after each iteration, with instances of the loop variable already used being fixed in a special list. For MOLA loops which are *fixed* for loop patterns the query is executed only once, because all matches are returned by the built SQL query.

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). Currently simple OCL expressions having a direct counterpart in SQL and some simple OCL set expressions are supported, but this repertoire will be extended.

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. If the negative part is a `NOT`-region, all elements of this region (plus connecting links) are placed in the subquery.

VII. DATABASE PERFORMANCE ISSUES

In this section we analyze the performance of the generated queries in several databases, which are relevant for MOLA

tool. 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, currently the version 5.0.12. The first intuitive performance evaluations were also encouraging, but it was clear that a more thorough analysis of query optimization is required.

Since the authors have shown [1] 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 this paper 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 [22], 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. 3). This example is a fragment of the MOLA transformation transforming a class model to OWL notation [23] (used as a benchmark in section 8), namely, the `FOREACH` loop head 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 [1], that for simple cases such as in Fig. 3, 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`.

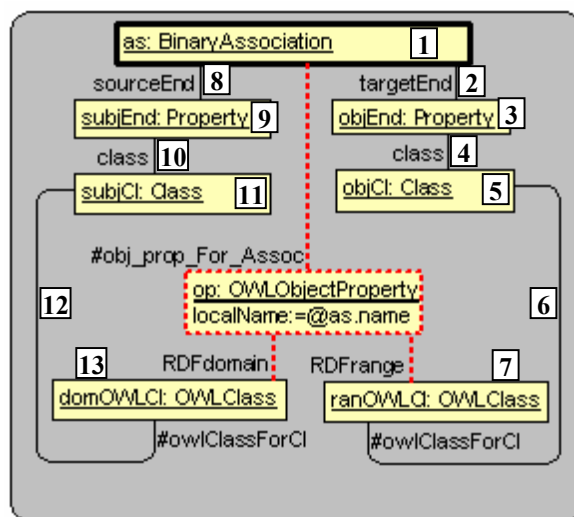


Fig. 3. Optimal pattern matching order.

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. 3 illustrates this order by numeric tags.

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 [1]).

The generated query corresponding to this pattern is shown in Fig. 4.

```

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 ,
cli20000021330.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=2000001858 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=2000001852 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. 4. Generated query example.

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 [24], which reveals some details of the execution plan. Fig. 5 shows the join order of query shown in Fig. 4, 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 [25,26], 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 [24] 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.

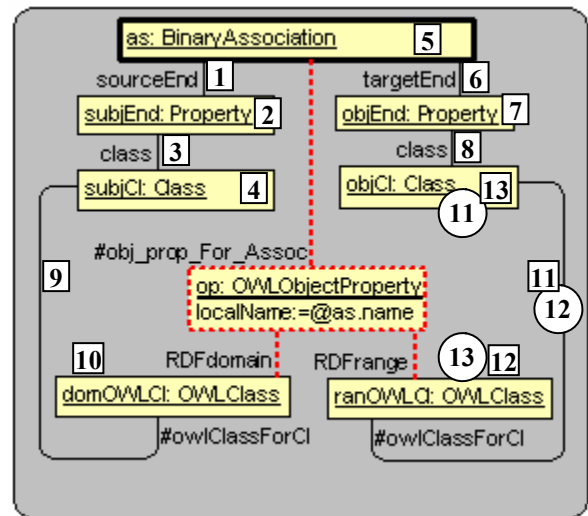


Fig. 5. MySQL query plan (table join order).

Due to the mentioned above problem other alternatives were sought. Possible alternatives are MSDE 2000 [27] – the free "small" version of MS SQL 2000 server, PostgreSQL [28] – another popular open source RDBMS, MSSQL Server 2005 Express [29] - – the free "small" version of MS SQL 2005 server. 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 [30] 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 [31] in MSDE 2000, that decreases the server performance.

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.

VIII. 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

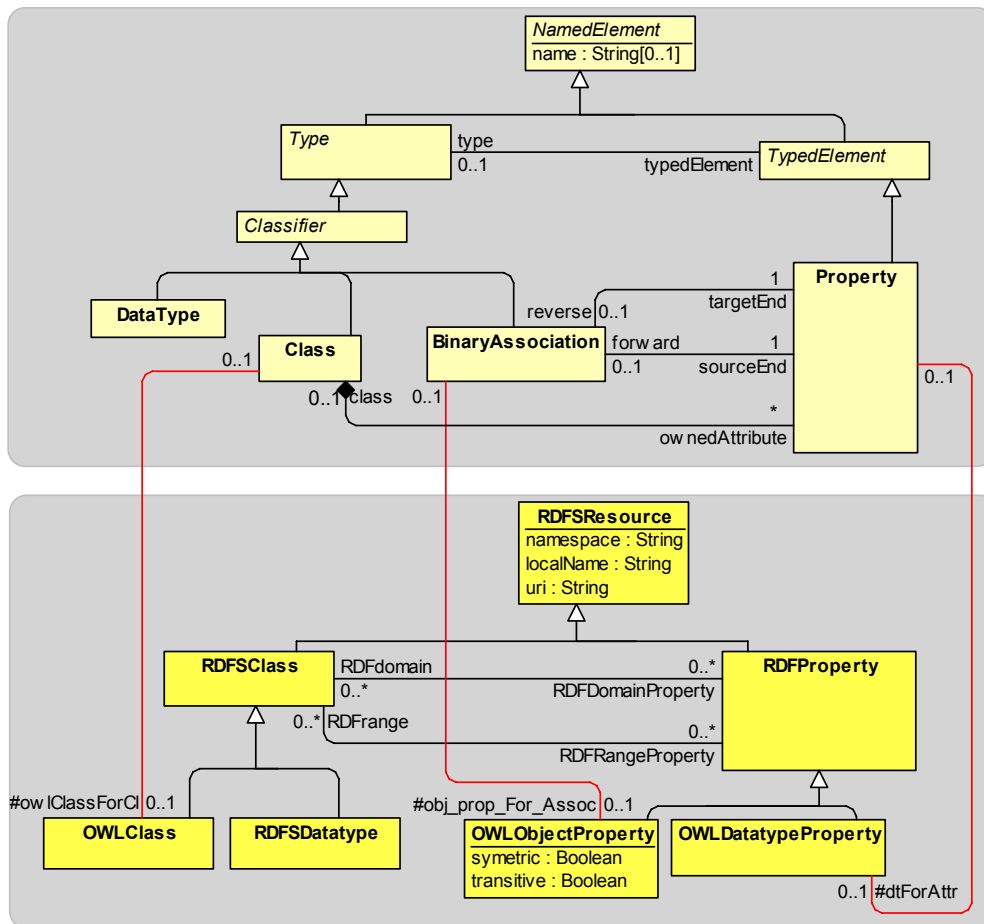


Fig. 6. Metamodels of UML Class Diagram and OWL Diagram.

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. 5 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).

performance the experiments have been done.

A simple task and appropriate model transformation tool for comparison have been chosen. The choice - AGG[12] 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 [10], thus allowing us to ensure 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. 6.

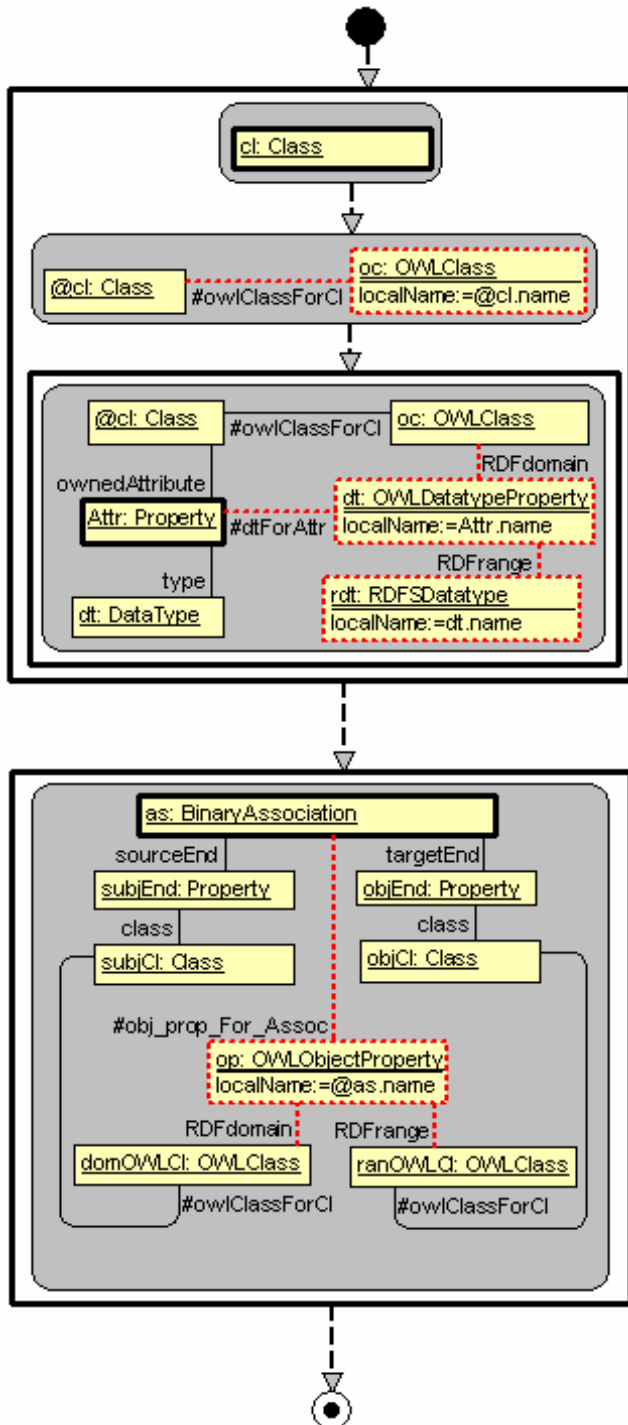


Fig. 7. Transformation UML Class Diagram to OWL Diagram

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 *fixed foreach* loop iterates through all Class instances and the nested *fixed foreach* loop iterates through appropriate Property instances. The third *fixed foreach* loop creates OWLDatatypeProperty for each

BinaryAssociation (Fig. 7). Though this transformation is very simple it is a typical representative of MDD tasks where frequently a model has to be transformed to a semantically equivalent one in another notation.

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.

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 [30]. 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.

TABLE I.
BENCHMARK RESULTS.

| Model size (N) | Transformation ExecutionTime (s) | |
|----------------|----------------------------------|------|
| | 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 | |

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 [10]. For real MDD 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.

IX. CONCLUSIONS

In the paper the main principles and solutions used in the MOLA virtual machine have been described. It is shown that 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 MOLA tool currently is). Several model transformations supporting real MDD 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. 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 MDD 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 [1] 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.

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

REFERENCES

- [1] A. Kalnins, J. Barzdins, E. Celms. “Efficiency Problems in MOLA Implementation”. 19th International Conference, OOPSLA’2004 (Workshop “Best Practices for Model-Driven Software Development”), Vancouver, Canada, October 2004. URL: <http://www.softmetaware.com/oopsla2004/mdsd-workshop.html>
- [2] 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
- [3] ATL. URL: <http://www.sciences.univ-nantes.fr/lina/atl/>
- [4] Tefkat. URL: <http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/>
- [5] MTF. URL: <http://www.alphaworks.ibm.com/tech/mtf>
- [6] ArcStyler. URL: <http://www.interactive-objects.com/>
- [7] AndromDA <http://www.andromda.org/>
- [8] UML 2.0 Eclipse EMF. URL: <http://www.eclipse.org/uml2/>
- [9] Metadata Repository (MDR). URL: <http://mdr.netbeans.org/>
- [10] G. Varro, A. Schurr, D. Varro “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.
- [11] G. Varro, K. Friedl, D. Varro “Graph Transformations in Relational Databases” Proceedings of GraBaTs 2004: International Workshop on Graph Based Tools, Rome, Italy, 2004, Elsevier.
- [12] AGG - The Attributed Graph Grammar System. URL: <http://tfs.cs.tu-berlin.de/agg/>
- [13] A. Kalnins, J. Barzdins, E. Celms. "Model Transformation Language MOLA." - LNCS, Springer, v. 3599, 2005. 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. Rev. Sel. Papers, pp. 62-76.
- [14] Kalnins A., Barzdins J., Celms E. “Model Transformation Language MOLA: Extended Patterns”. Selected papers from the 6th International Baltic Conference DB&IS’2004, IOS Press, FAIA vol. 118, 2005, pp. 169-184.
- [15] A. Kalnins, J. Barzdins, E. Celms. “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. URL: <http://heim.ifi.uio.no/~janoa/wmdd2004/papers/>
- [16] A. Kalnins, J. Barzdins, E. Celms. “MOLA Language: Methodology Sketch”. Proceedings of EWMDDA-2, Canterbury, England, 2004. pp.194-203.
- [17] E. Celms, A. Kalnins, L. Lace. “Diagram definition facilities based on metamodel mappings”. Proceedings of the 18th International Conference, OOPSLA’2003 (Workshop on Domain-Specific Modeling), Anaheim, California, USA, October 2003, pp. 23-32.
- [18] A. Kalnins, E. Celms, A. Sostaks. „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, p. 25.
- [19] OMG, MOF 2.0 Query/View/Transformation Specification. URL: <http://www.omg.org/docs/ptc/05-10-02.pdf>
- [20] A. Kalnins, E. Celms, A. Sostaks, “Tool support for MOLA”, Fourth International Conference on Generative Programming and Component Engineering (GPCE’05), Proceedings of the Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005, pp. 162-173 (preliminary version).
- [21] OMG, Meta Object Facility (MOF) Core Specification. URL: <http://www.omg.org/docs/formal/06-01-01.pdf>
- [22] C. J. Date, “An Introduction to Database Systems”, Chapter 17, Optimisation, Addison-Wesley, 7th Edition, 2000
- [23] W3C. “Web Ontology Language (OWL)”. URL: <http://www.w3.org/2004/OWL/>
- [24] MySQL Reference Manual. URL: <http://dev.mysql.com/doc/mysql/en/index.html>
- [25] T. Katchaounov. “An Overview of the MySQL 5.0 Query Optimizer”. The MySQL Users Conference, 2005. URL: http://conferences.oreillynet.com/presentations/mysql05/timour_update.pdf
- [26] P. Dubois. “MySQL”, Chapter 4, Query Optimization. Sams, 3rd Edition, 2005.
- [27] Microsoft SQL Server 2000 Desktop Engine (MSDE 2000). URL: <http://www.microsoft.com/sql/msde/default.asp>
- [28] PostgreSQL - Open Source Database Server. URL: <http://www.postgresql.org/>
- [29] Microsoft SQL Server 2005 Express Edition URL: <http://www.microsoft.com/sql/editions/express/default.msp>
- [30] R. Elmasri, S. Navathe. “Fundamentals of Database Systems”, Chapter 18, Query Processing and Optimisation, Addison-Wesley, 3rd Edition, 2000.
- [31] The SQL Server 2000 Workload Governor. URL: http://msdn.microsoft.com/library/default.asp?url=/library/enu/architect/8_ar_sa2_0ciq.asp