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. Form 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.

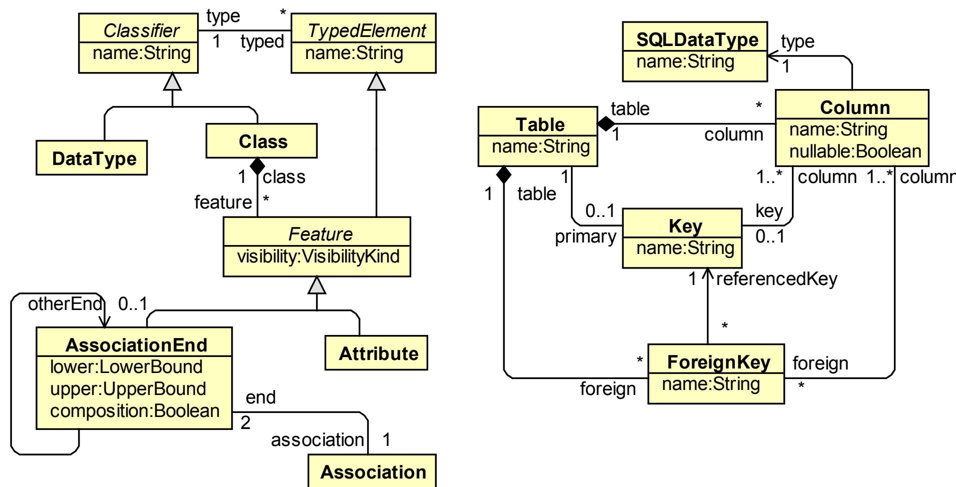

Fig. 1 Source and target metamodels of the example

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.

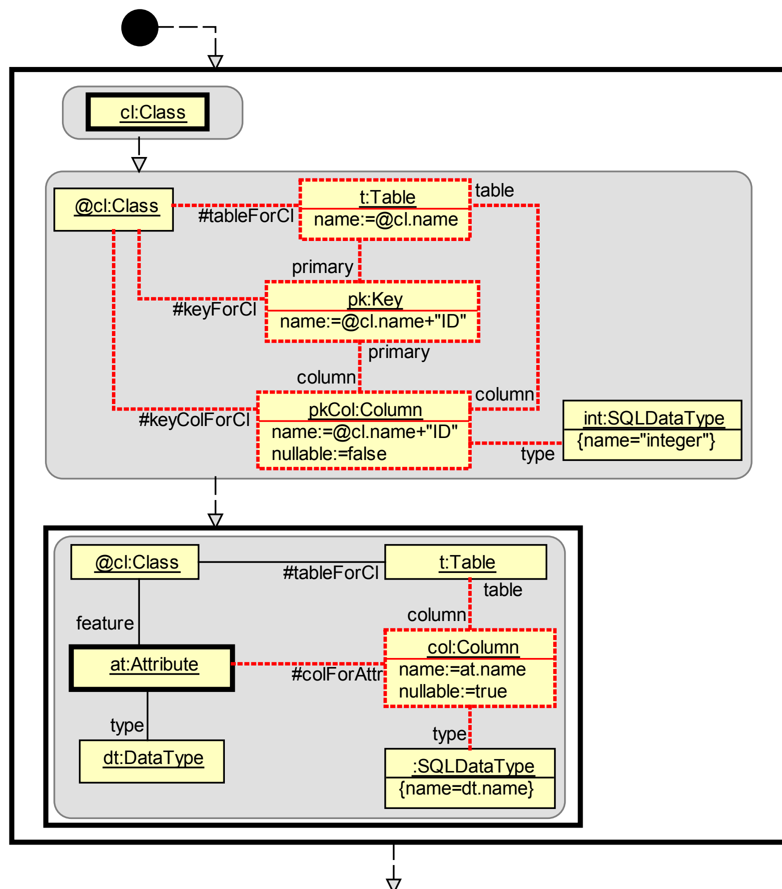

Fig. 2 The first step of the transformation

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.

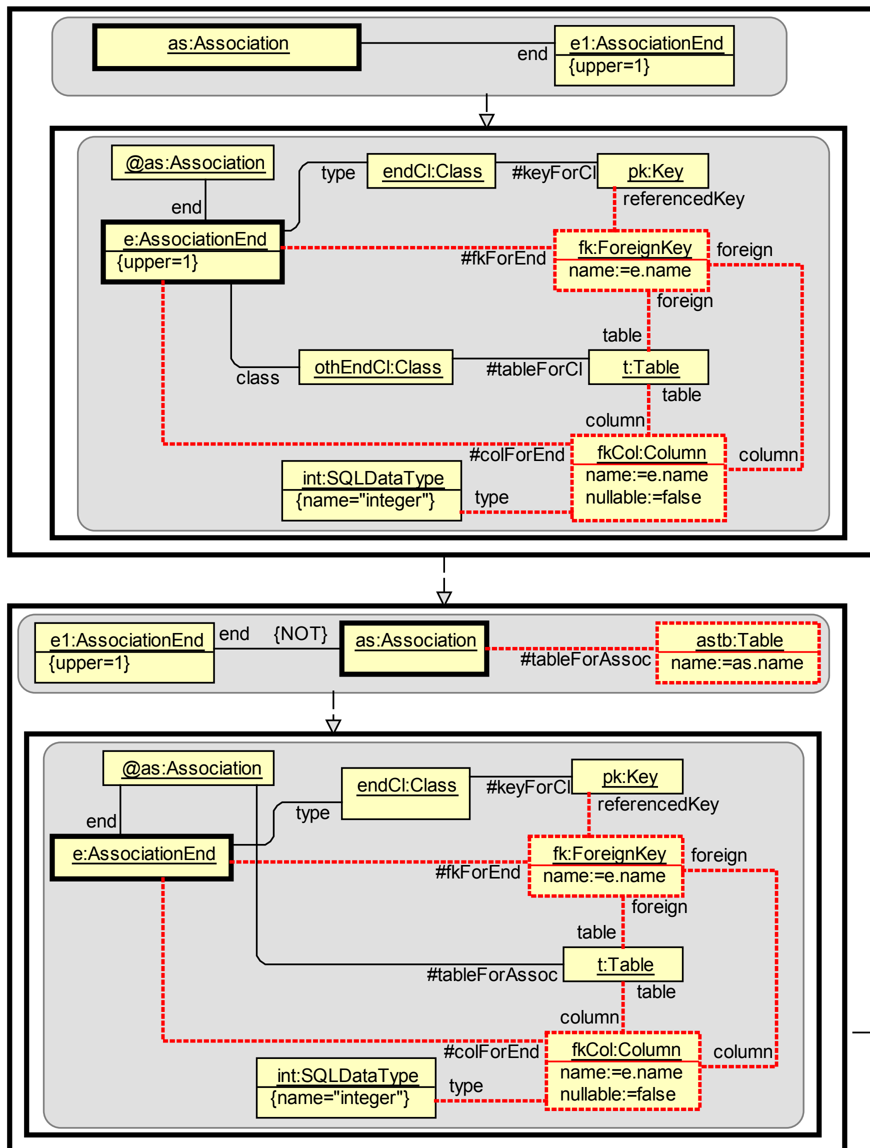

Fig. 3 The second step of the transformation

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.