

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 most advanced proposal seems to be [3]. 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].

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].

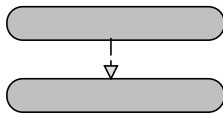
It should be mentioned that an early pioneer in the area (well before the MDA era) is the PROGRES language [9]. This semi-graphical language offered pattern-based graph rewrite rules applicable to “models” described by schemas (actually, metamod-els). The execution of rules is governed by the traditional 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.

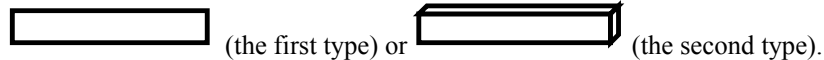
This paper proposes a new transformation language **MOLA (MOdel transformation Language)**. The prime goal of MOLA is to provide an easy readable graphical transformation language by combining traditional structured programming in a graphical form (a sort of “structured flowcharts”) with rules based 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

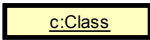
This section presents a brief overview of basic constructs of MOLA. The MOLA language is a procedural graphical language, with control structures taken from traditional structured programming. The elements specific to model transformations can easily be combined with traditional language elements such 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.

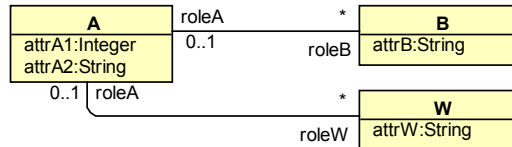


Fig.1. Metamodel for the toy example

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.

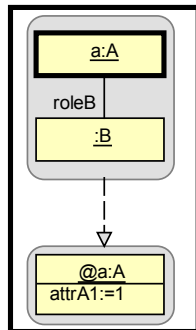


Fig.2. Program finding *A*'s linked to a *B*

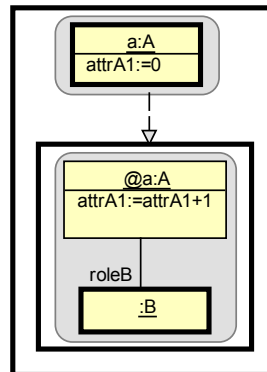


Fig.3. Program counting *B*'s linked to an *A*

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 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*.

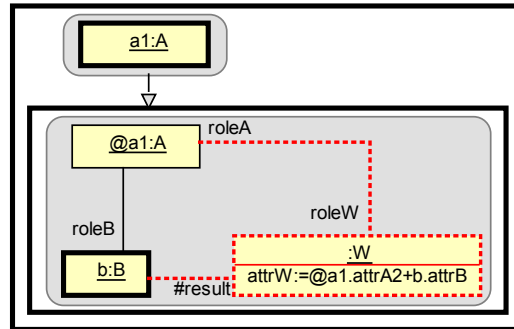


Fig.4. Program building *W* for each *B*

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** (not specified in the metamodel), which is typically used in MDA-related transformations for setting the context for next subordinate transformations and for tracing instances between models (therefore it normally links elements from different metamodels). Role names of mapping associations are prefixed by the # character in MOLA. Certainly, in this trivial example we could do well without this association.

Two more basic 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.

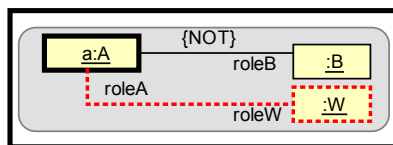


Fig.5. NOT constraint

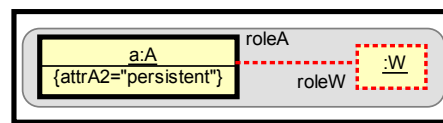


Fig.6. Attribute constraint

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 is that from OCL (explicit qualified references to other instances in the pattern could also be used).

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 defined in

the metamodel, instances may have “temporary” computed attributes which can be used as variables for storing values during the computation. These temporary attributes are defined and cleared by means of special statements. Similarly, there may be temporary associations. 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!).

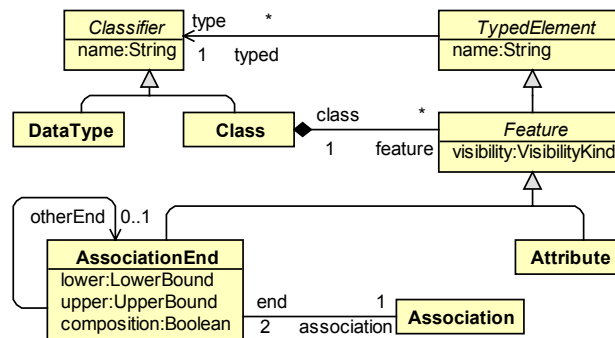


Fig.7. Simplified class metamodel

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.

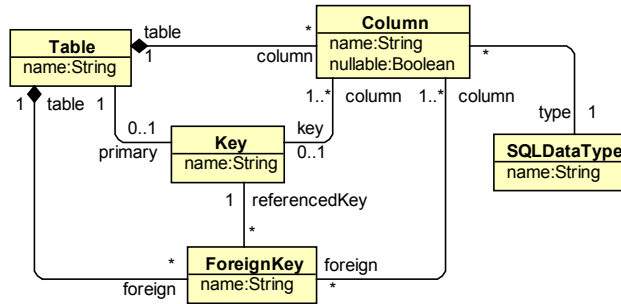


Fig.8. Simplified relational database metamodel

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

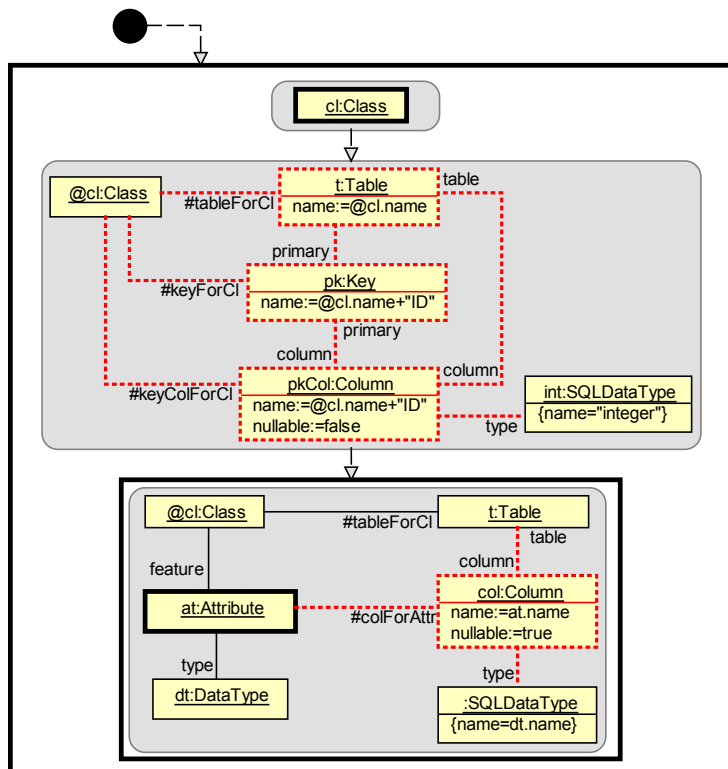


Fig.9. Class to database transformation (part 1)

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

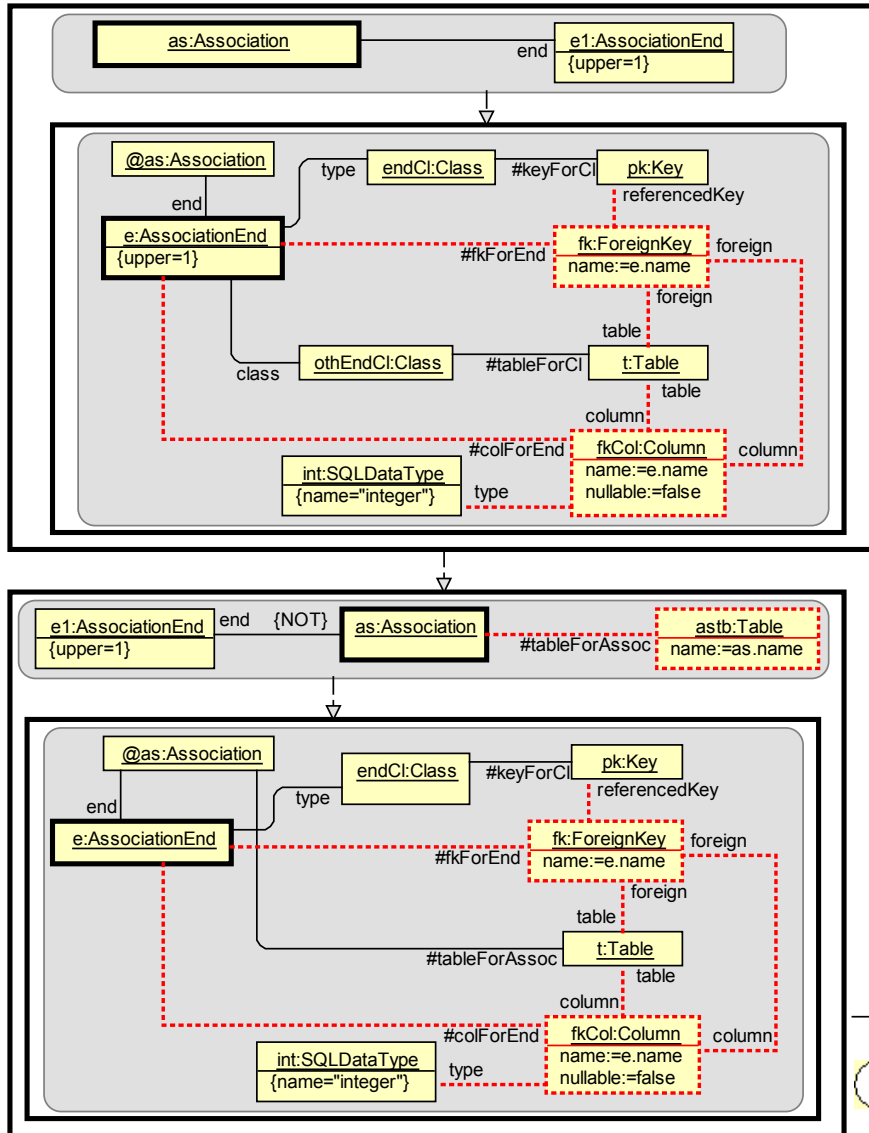


Fig.10. Class to database transformation (part 2)

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 in 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. 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).

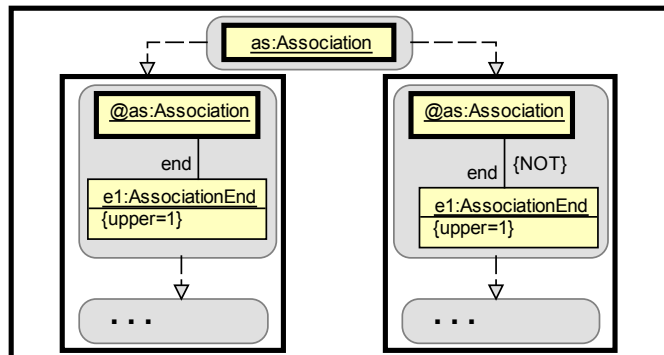


Fig.11. Loop with an if-then-else statement

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. However, there is no great need in this since the whole transformation actually fits in one A4 page.

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 some variations).

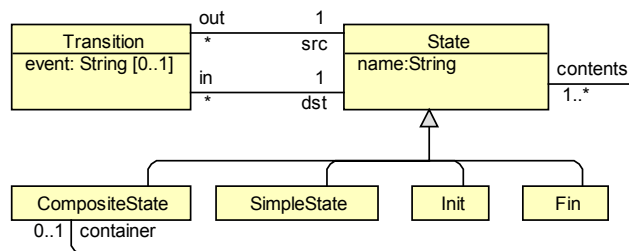


Fig.12. Metamodel of hierarchical statechart

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 topmost composite state (i.e., 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 an update – source and target models coincide.

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 apply – 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.

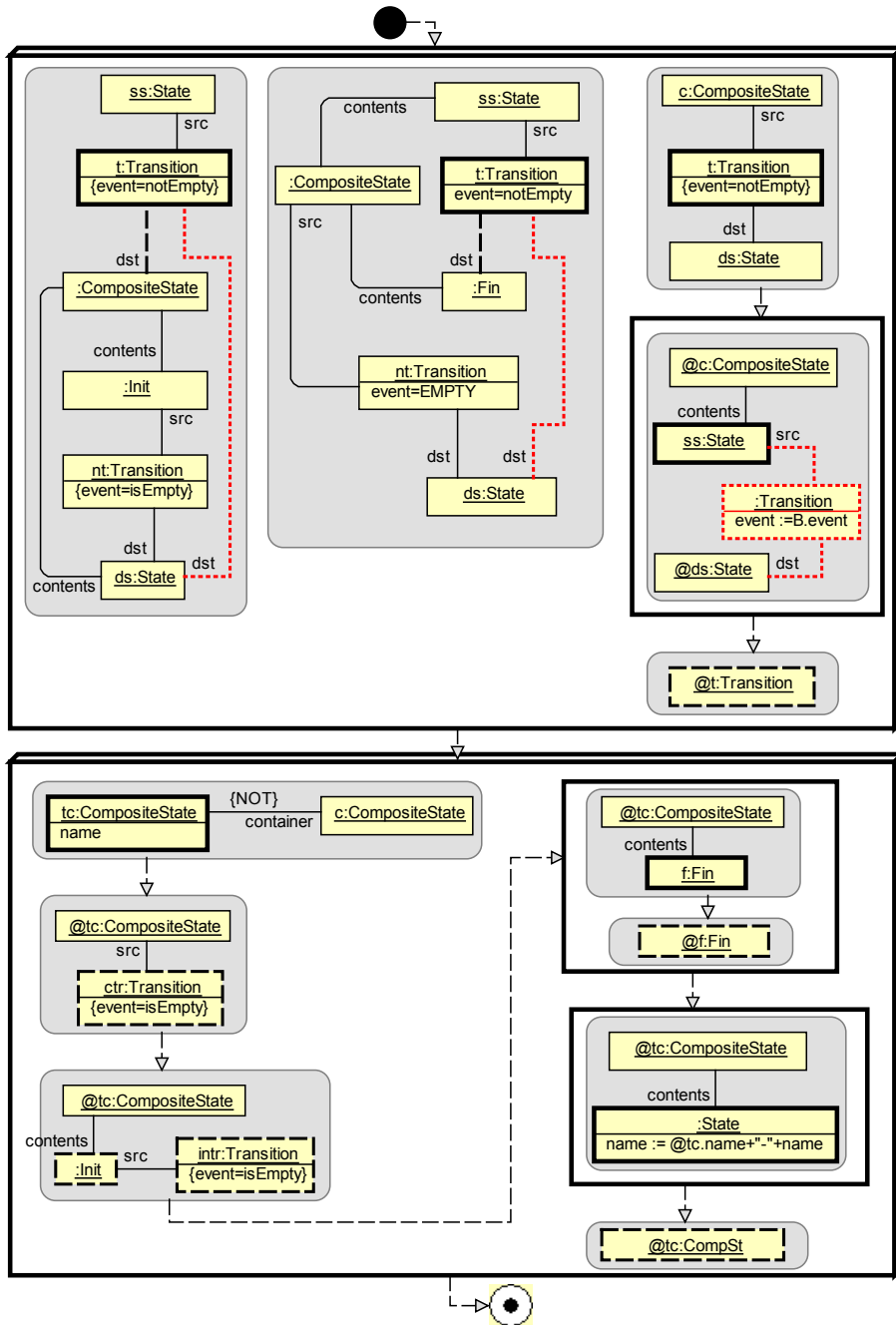


Fig.13. Statechart flattening

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.

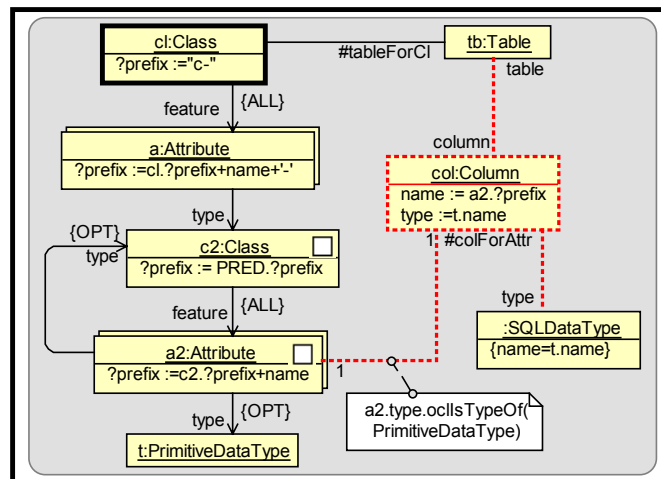


Fig.14. Transitive closure by extended pattern

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. Currently all the experiments with MOLA, including pictures for this paper, are performed by means of the modeling tool GRADE [13, 14], in the development of which authors have participated.

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, Singapur, 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>