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 been 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.

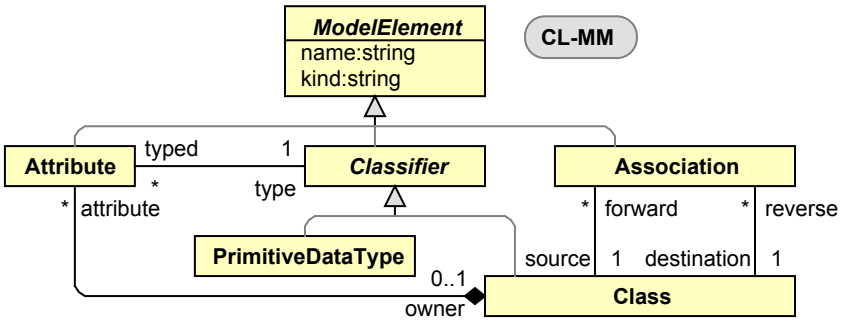

Figure 1. Source metamodel for simplified class diagram.

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.

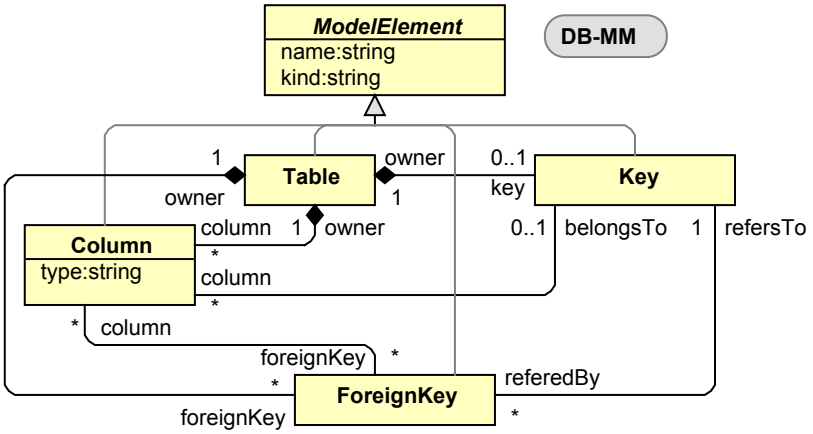

Figure 2. Target metamodel for database definition.

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.

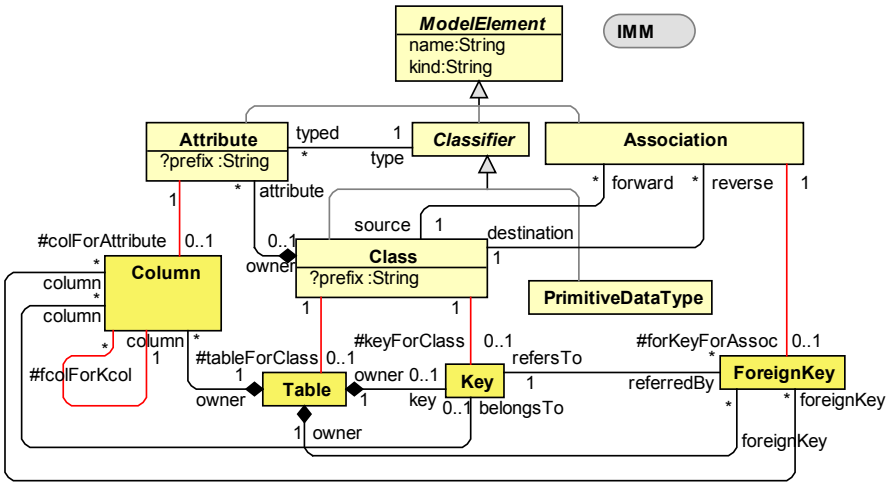

Figure 3. Intermediate metamodel for the transformation.

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*).

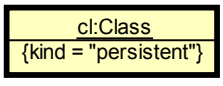

Figure 4. Simplest pattern example.

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.

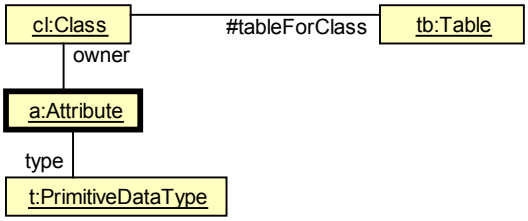

Figure 5. Associations in a pattern.

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.

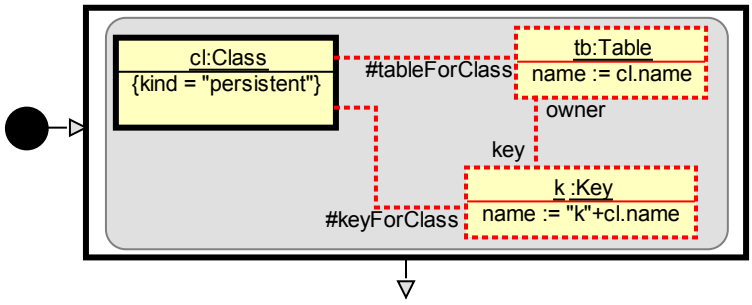

Figure 6. Simple statement in MOLA.

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.

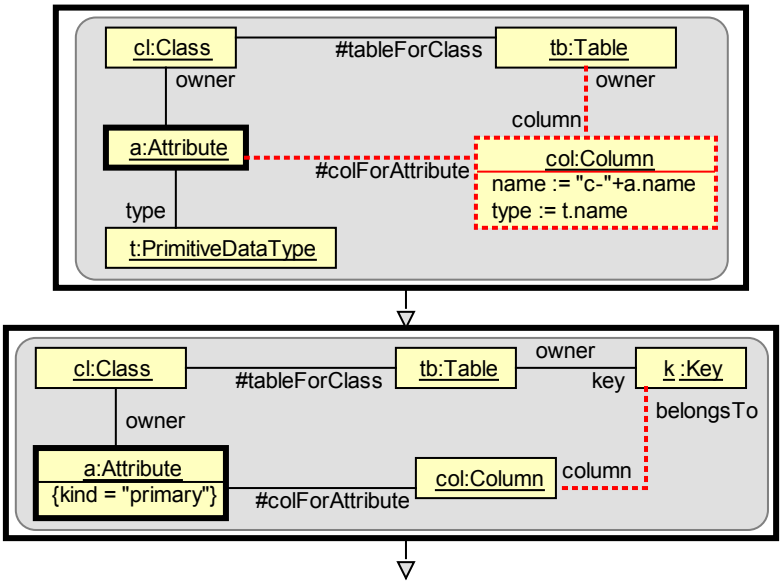

Figure 7. Statements transforming primitive attributes to columns and associating key columns.

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.

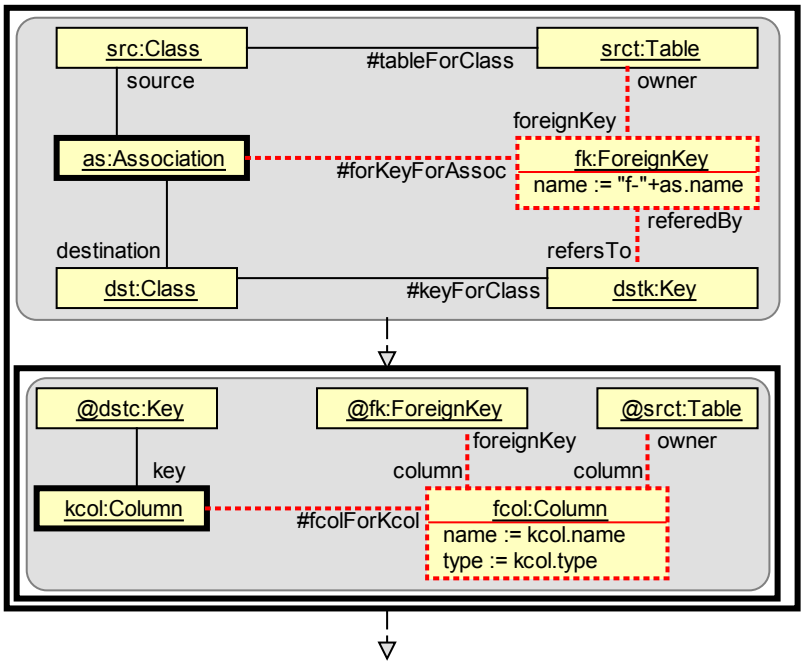

Fig. 8 Statement processing source model associations.

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 **multiobject** 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.

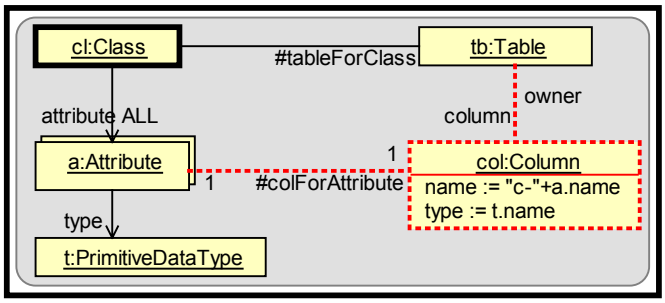

Figure 9. Rule with cardinality constraint.

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.

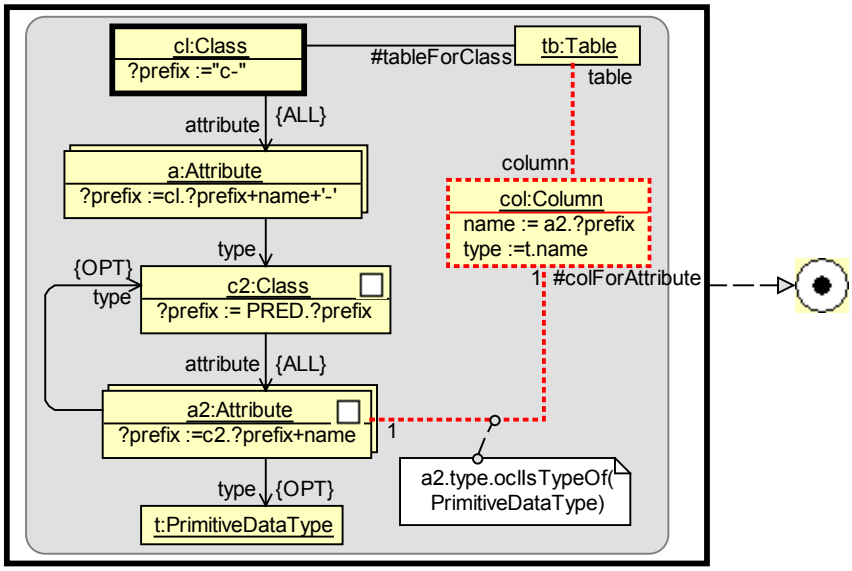

Figure 10. Rule with looping pattern.

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 <=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., Jackobson 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.