MOLA Language: Methodology Sketch

Audris Kalnins, Janis Barzdins, Edgars Celms

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

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

1. Introduction

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

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

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

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

2. Brief Overview of MOLA

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

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

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

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

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

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

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

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

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

3. The Benchmark Example

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

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

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

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

4. MOLA Solution

4.1. Building the Workspace Metamodel

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

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

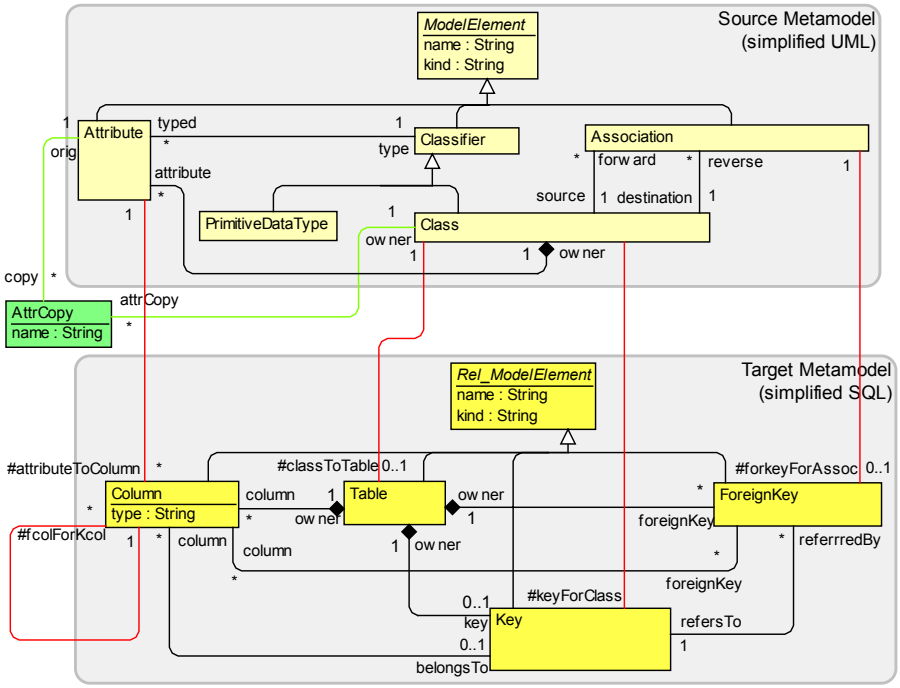

Fig. 1. The workspace metamodel

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

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

4.2. MOLA Program Implementing the Transformation

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

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

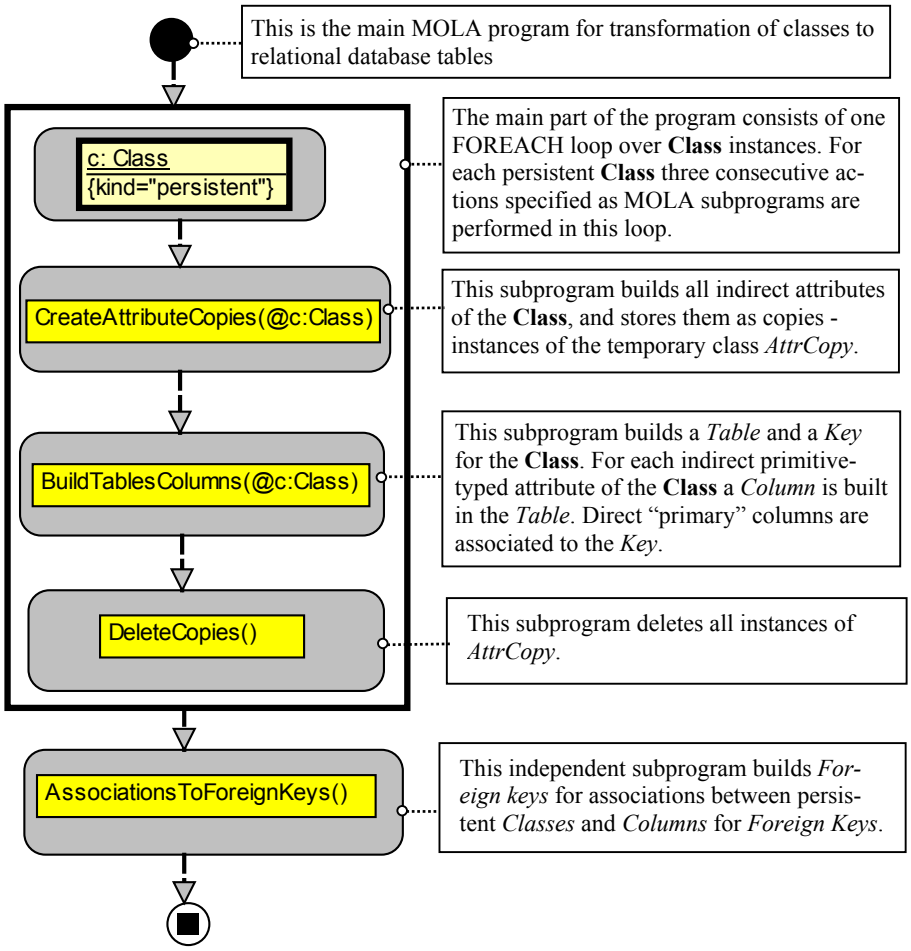

Fig. 2. The main diagram of the transformation

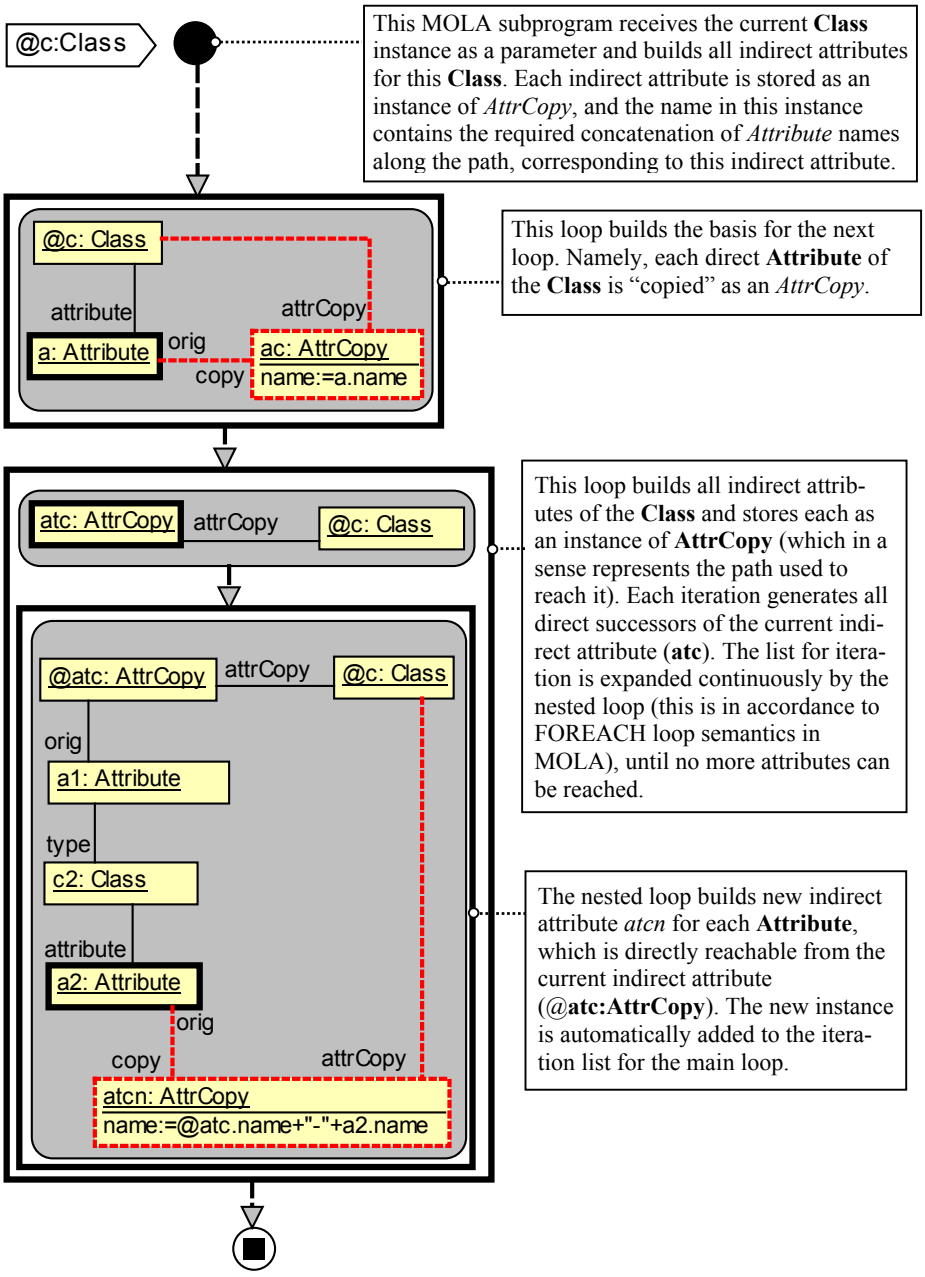

Fig. 3. Subprogram *CreateAttributeCopies*

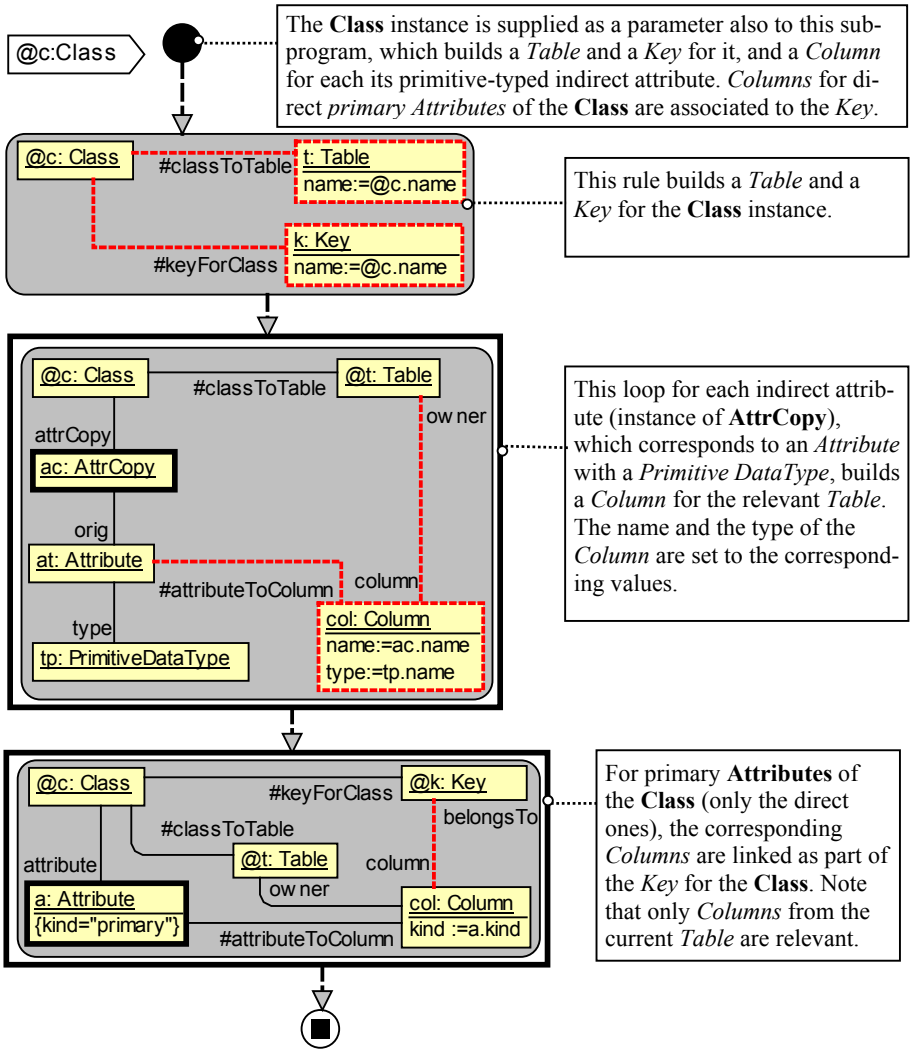

Fig. 4. Subprogram *BuildTablesColumns*

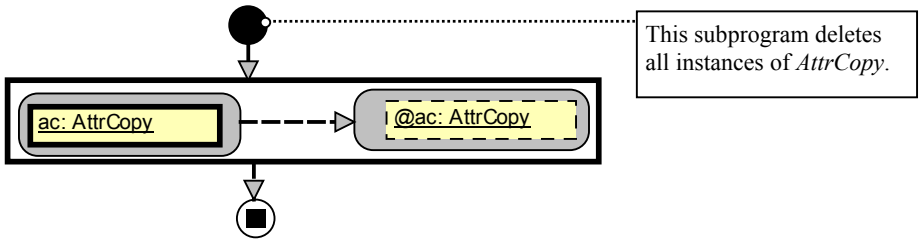

Fig. 5. Subprogram *DeleteCopies*

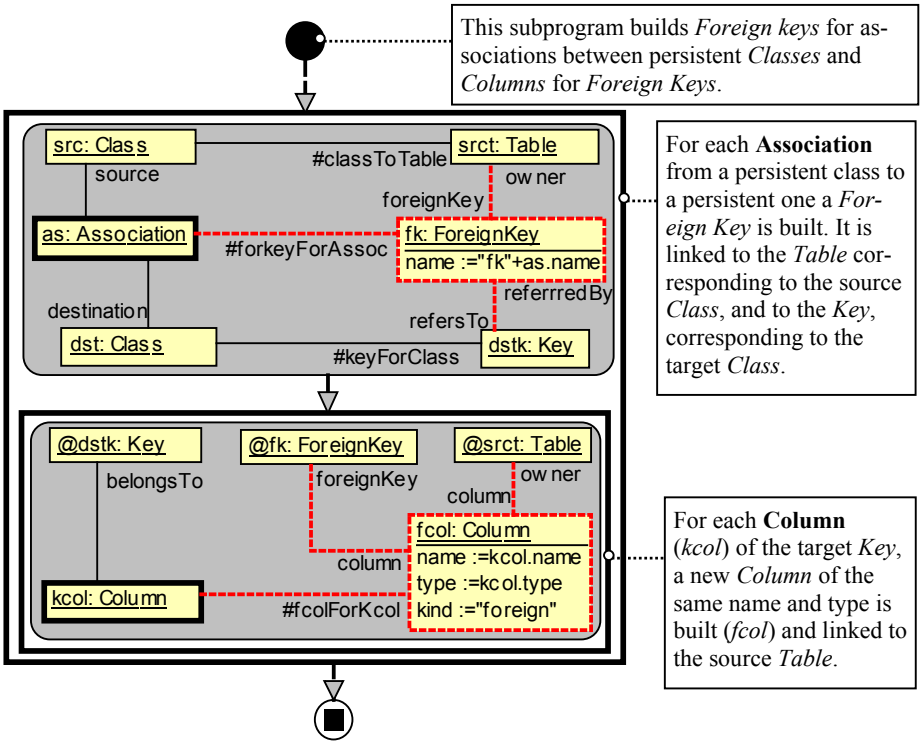

Fig. 6. Subprogram *AssociationsToForeignKeys*

5. Some Remarks on MOLA Methodology

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

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

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

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

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

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

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

6. Conclusions

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

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

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

References

1. QVT-Merge Group. MOF 2.0 QVT RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, http://www.omg.org/cgi-bin/doc?ad/2004-04-01
2. Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, http://www.omg.org/cgi-bin/doc?ad/2003-08-07
3. Willink E.D. A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX. Workshop on Metamodelling for MDA, University of York, England, 24-25 November 2003
4. Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003
5. Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDAFA 2004, University of Linkoping, Sweden, 2004, pp.14-28. (see also http://melnais.mii.lu.lv/audris/MOLA_MDAFA.pdf)
6. Kalnins A., Barzdins J., Celms E. Basics of Model Transformation Language MOLA. Proceedings of WMDD 2004, Oslo, 2004, http://heim.ifi.uio.no/~janoa/wmdd2004/papers/kalnis.pdf