Efficiency Problems in MOLA Implementation

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

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

1. Introduction

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

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

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

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

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

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

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

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

2. Brief Overview of MOLA

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

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

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

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

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

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

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

3. Basic Principles of MOLA Implementation

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

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

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

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

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

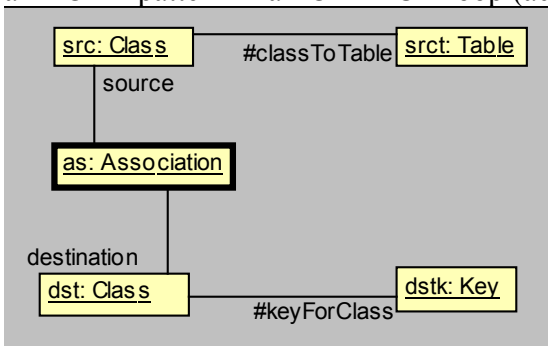

Fig. 1. Pattern example

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

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

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

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

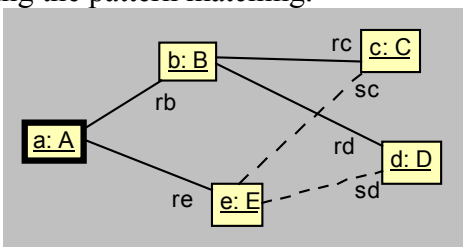

Fig. 2. Another pattern example

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

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

The virtual machine must contain one main operation for patterns:

getPatternElement(int i) .

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

getPatternRoot( ) .

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

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

getNext(metaClass mcl).

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

The operation most used for implementing pattern matching is

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

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

initializeGetNextByLink(association assoc, instance sourceInst, metaclass mcl)

Two more auxiliary operations are:

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

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

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

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

getNextFromSet(metaClass mcl, set instSet) and

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

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

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

4. Pattern Matching in MOLA

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

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

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

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

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

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

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

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

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

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

5. Example

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

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

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

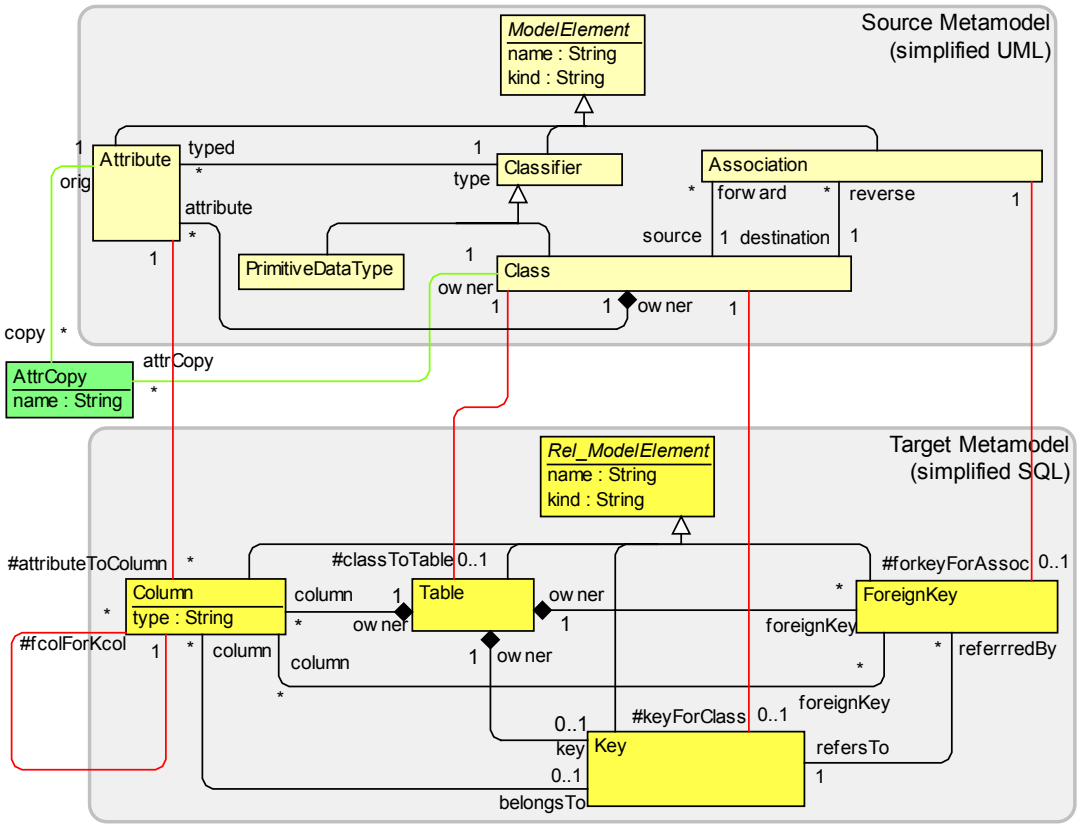

Fig. 3. Combined Metamodel

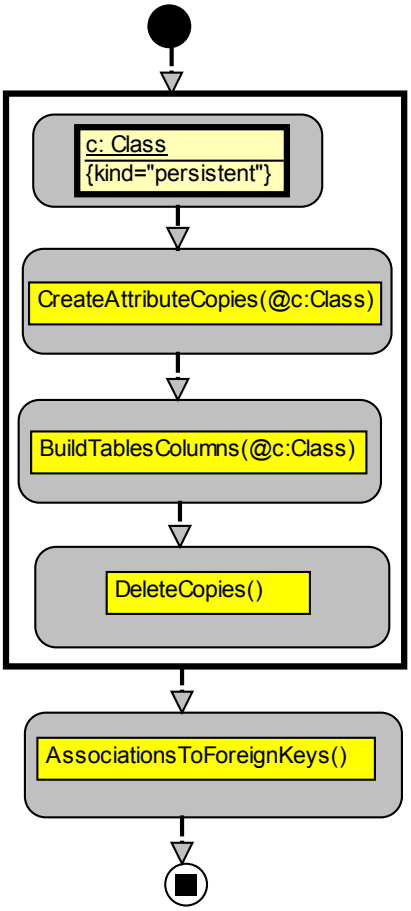

Fig. 4. Main Program

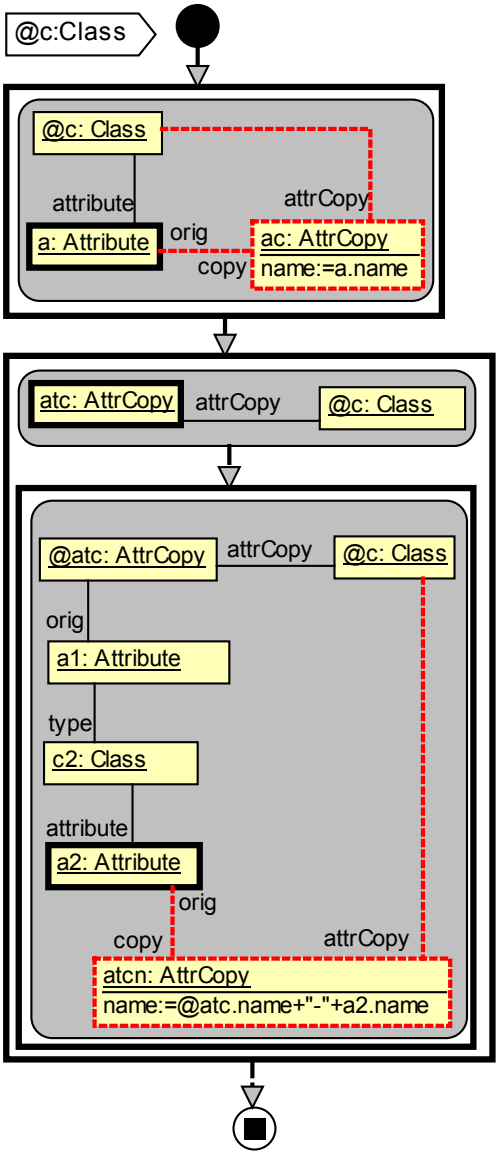

Fig. 5. Subprogram CreateAttributeCopies

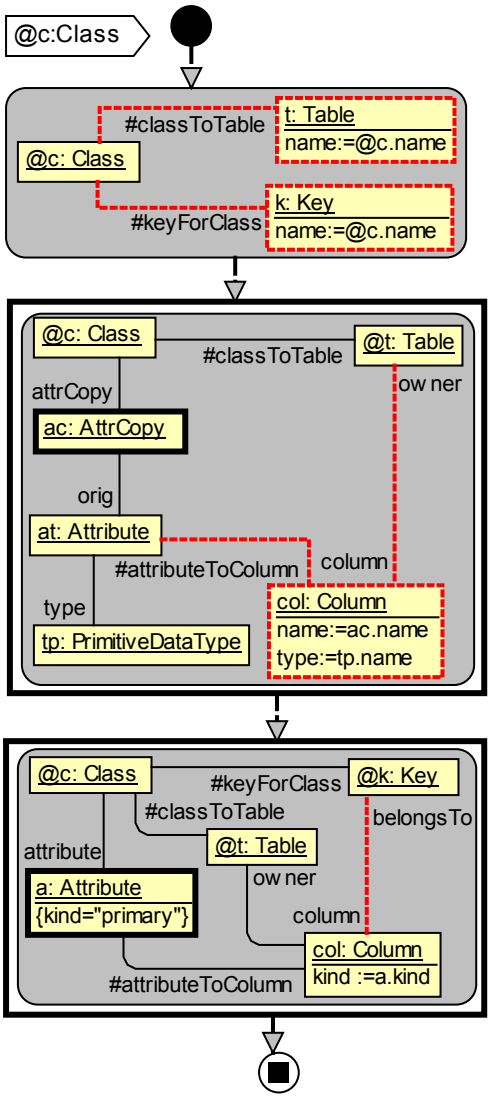

Fig. 6. Subprogram BuildTablesColumns

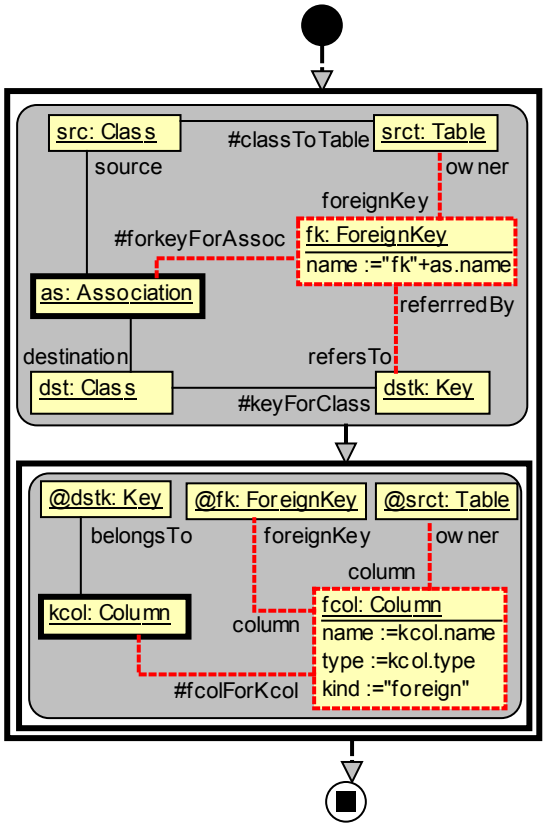

Fig. 7. Subprogram AssociationsToForeignKeys

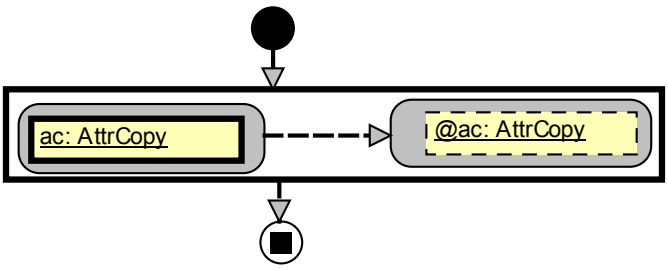

Fig. 8. Subprogram DeleteCopies

6. Performance evaluation for the example

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

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

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

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

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

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

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

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

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

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

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

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

7. Conclusions

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

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

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

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

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

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

References

[1] QVT-Merge Group. MOF 2.0 QVT RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, http://www.omg.org/cgi-bin/doc?ad/2004-04-01

[2] Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, http://www.omg.org/cgi-bin/doc?ad/2003-08-07

[3] Willink E.D. A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX. Workshop on Metamodelling for MDA, University of York, England, 24-25 November 2003

[4] Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003

[5] Bézivin J., Dupé G., Jouault F., et al. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. 2nd OOPSLA Workshop on Generative Techniques in Context of MDA, Anaheim, California, 2003.

[6] Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDAFA 2004, University of Linkoping, Sweden, 2004, pp.14-28. (see also http://melnais.mii.lu.lv/audris/MOLA_MDAFA.pdf)

[7] Kalnins A., Barzdins J., Celms E. Basics of Model Transformation Language MOLA. Proceedings of WMDD 2004, Oslo, 2004, http://heim.ifi.uio.no/~janoa/wmdd2004/papers/kalnis.pdf

[8] Kalnins A., Barzdins J., Celms E. MOLA Language: Methodology Sketch. To be published in proceedings of EWMDA-2, Canterbury, England, 2004. (see also http://melnais.mii.lu.lv/audris/EWMDA_MOLA2.pdf)

[9] A. Schürr. PROGRES for Beginners. Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.

[10] Vizhanyo A., Agrawal A., Shi F. Towards Generation of High-performance Transformations. Generative Programming and Component Engineering, (accepted), Vancouver, Canada, 2004.

[11] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anaheim, California, 2003.