Simple and Efficient Implementation of Pattern Matching in MOLA Tool

Audris Kalnins, Edgars Celms, Agris Sostaks
University of Latvia, IMCS
29 Raina boulevard, Riga, Latvia
{Audris.Kalnins, Edgars.Celms}@mii.lu.lv, agree@os.lv

Abstract - **One of crucial problems for model transformation implementations is an efficient implementation of pattern matching. The paper addresses this problem for MOLA Tool implementing the model transformation language MOLA. Another goal has been to keep the implementation as simple as possible. The paper presents one possible solution to the combined problem where an SQL database with fixed schema is used as the MOLA runtime repository. A natural coding is selected where a MOLA pattern match can be mapped to a single non-standard self-join SQL query. The paper shows that a sufficient matching efficiency can be obtained this way. The generated queries are analyzed from the table join order point of view and it is shown that the default query optimization for the MySQL database can find an order close to optimal. This analysis and performed experiments are used to conclude that at this moment MySQL is the most fit for MOLA implementation among "free" relational databases. In addition, benchmark tests based on a simple natural model transformation problem are used to estimate efficiency of the selected implementation architecture and to compare MOLA Tool to the popular graph transformation tool AGG. Benchmark tests confirm the efficiency of the current MOLA Tool implementation and applicability of MOLA language to MDD-specific tasks.**

I. INTRODUCTION

Nearly all of model transformation languages use the pattern matching as the main functional element for defining how the source model components must be transformed to the target model. So does the transformation language MOLA analyzed in this paper. When a transformation language is implemented, the implementation of pattern matching typically is the most demanding component to implement and also the key factor determining the implementation efficiency.

This issue has been analyzed theoretically in various contexts. For MOLA, authors of this paper have already shown that a very efficient pattern matching implementation is possible in principle [1], however this implementation would require significant effort to build and therefore is appropriate only for an industrial tool. For other transformation languages, the most thorough analysis has been performed for the GReAT language [2].

In this paper, the problem appears in another setting. An academic model transformation tool supporting MOLA has been built using limited resources, and for this tool both simple and sufficiently efficient implementation has been required.

Another related problem is the choice of runtime repository, since the pattern matching is very intimately related to repository access mechanisms. A standard choice, used in most academic model transformation tools [3,4,5] and some industrial ones [6,7] too, is a metamodel based repository, such as Eclipse EMF [8], MDR [9] or similar ones. These repositories typically have a low level universal API for retrieving class instances. This solution would make the implementation of pattern matching and other language features significantly more complicated.

Several possible solutions for these two related problems in the context of MOLA tool have been analyzed. The final decision, which is described in this paper, occurred to be rather non-typical for model transformation tools – the best kind of repository would be a relational database with fixed schema – tables coding the metamodel and model in the most natural way. The central idea of this implementation is that a MOLA pattern match operation can be implemented by a single SQL query. And this query is easy to generate from the pattern definition.

The only remaining problem is whether such a rather non-standard query (using multiple self-joins) can be processed efficiently by database engines. Analysis in the paper shows that not all engines perform efficiently enough, but there are freely available ones which can do this, currently the best one is MySQL. These results are in concordance with other papers analyzing usability of SQL for pattern matching [10,11] (however, a completely different database structure is used there and the experiment setting is also different).

The paper describes the solution used in MOLA tool. After a brief reminder of MOLA language and an overview of the MOLA tool architecture, the core of the tool – the MOLA virtual machine (VM) is defined (section 5). The most appropriate database structure and the mapping of a pattern to an SQL query are described in detail in section 6. Section 7 analyzes the performance issues of generated queries, especially the table join order. Section 8 contains a benchmark test, which compares the transformation of simplified UML class diagram to simplified OWL diagram implemented both in MOLA Tool and the popular graph transformation tool AGG [12] on various model sizes. The results confirm the efficiency of MOLA implementation and its practical usability.

II. WHAT IS MOLA

MOLA is a graphical model transformation language developed at the University of Latvia [1,13,14,15,16,17,18]. Its main distinguishing feature is the use of simple procedural control structures governing the order in which pattern matching rules are applied to the source model. Due to the large number of papers on MOLA language (the most important ones are [13,15,16,18]) we do not repeat the language description in this paper. In addition, there is a web site http://mola.mii.lu.lv/ devoted to MOLA where all these papers and a formal description of MOLA are available. Just to clarify the terminology, we very briefly remind here the main elements of MOLA.

Source and target metamodels are combined in one class diagram, where the added mapping associations link the corresponding classes in source and target metamodels, these associations are used for traceability and transformation structuring.

The MOLA transformation program consists of one or more **MOLA diagrams** (one of which is the main). A MOLA diagram is a sequence of graphical **statements**, linked by arrows. 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. Elements specify instances of which classes must be matched. The loop variable is also a special kind of element, it is distinguished by having a bold-lined rectangle. In addition, the pattern contains **links** (metamodel associations) – a pattern actually corresponds to a metamodel fragment. Pattern elements may have attribute constraints – simple OCL expressions. 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, attribute constraints are true on these instances and required links are present. Loops may be nested to any depth. There are two types of **FOREACH** loop – *fixed* (the scope of matched class instances does not change – pattern matching is performed only once) and *not fixed* (the scope of matched instances can change and pattern matching must be performed after each iteration to see changes in the appropriate instance scope). The loop variable (and other element instances) from an upper level loop can be referenced by means of the reference symbol – an element with @ prefixed to its name. There is also the **WHILE** loop in MOLA, which is less used and not analyzed in this paper. Another important statement in MOLA is **rule** (also a grey rounded rectangle) – a statement consisting of pattern and actions. These actions can be element or association building actions (denoted by red dotted lines) and delete actions (denoted by dashed lines). An attribute value of an element can be set by means of attribute assignments. A rule is executed once (or not at all if pattern fails) – thus it plays the role of if-statement too. A subprogram is called by means of a call statement (possibly, with parameters – element references). An example of MOLA program is given in section 8.

III. PRECISE SEMANTICS OF PATTERNS IN MOLA

The general semantics of pattern matching is quite similar in all pattern-based transformation (QVT) languages, including the emerging OMG standard [19], nevertheless there are some specific features in any language. A pattern specifies instances of which metaclasses must be selected, how they must be linked by association instances and what attribute-based constraints for the instances must be satisfied. Certainly, there may be several occurrences of the same metaclass in a pattern, then instances must be matched accordingly. A **match** for a **pattern** is a **set** of source model elements – **instances** of metamodel **classes** and **associations**, each of which is associated to the corresponding pattern element and satisfies constraints in the pattern.

The subtleties in pattern matching for different QVT languages lie in the fact which matches must be found. In MOLA patterns are used for loop heads and rules. For a **FOREACH-loop** head **all** matches must be found which contain distinct instances of the loop variable. For all other pattern elements it is irrelevant namely which of the valid instances is selected (in some other QVT languages the semantics is more complicated here). In other words, there is an implicit existential quantifier placed on all elements, except the loop variable (and the reference elements for which the choice is already fixed). Another subtlety for (*not fixed*) loops in MOLA is that the source instance set from which distinct instances of the loop variable are selected may be replenished during the loop execution. For **rules** only **one** valid match is required (any of them if there are several), or the fact that there is none. It must be mentioned (see [1] for more details), that in a semantically correct MOLA program typically there is no much indeterminism during the pattern match – these seemingly "free" pattern elements actually are uniquely determined by the selected loop variable instance. Another positive aspect of match semantics in MOLA is that it facilitates match efficiency – not all instance combinations must be searched.

These simplest kinds of patterns are called **positive** patterns. Another kind of patterns in MOLA are the **negative** ones, which contain the **NOT** constraint on some **class** elements (as it is in [19], earlier MOLA versions [13,14] had NOT constraints only on pattern links). An element with NOT constraint expresses the fact that there must be no instance of the given class satisfying the attribute constraints and linked by the specified associations to the other (positive) pattern elements. An association linking two negative elements in a pattern is considered senseless in MOLA. Though sufficient for most transformations, these syntax features for patterns are not formally complete – an arbitrary universal quantifier on properties of an instance set cannot be expressed this way. Therefore one more element – the **NOT-region** (a rectangle with NOT tag containing some other pattern elements) must be introduced. A NOT-region expresses the fact that there must be no instance set for the pattern elements inside the region, which satisfy the "inside" conditions and are linked by associations crossing the region border to other positive elements. NOT-regions may be nested, but no association may link two NOT-regions. Since NOT-regions are not frequently used, the current implementation does not support them.

IV. MOLA Implementation Overview

The current version of MOLA tool has been developed with mainly academic goals – to test the MOLA usability, teach the use of MDD for software system development and perform some real life experiments. This has influenced some of the design requirements, though with easy usability as one of the goals and sufficient efficiency the tool has confirmed its potential as an industrial tool too.

Similarly to many MDD environments, MOLA environment consists of two major parts: MOLA **Transformation Definition Environment (TDE)** and **Transformation Execution Environment (TEE)**. TDE is completely related to the metalevel M2 according to MOF terminology, while TEE is at M1 level. TDE is used by expert users, which define new model transformations in MOLA for the adopted MDD technology or modify the existing ones from a transformation library to better suit the needs of a specific project. TEE is intended for mass usage by software developers applying the chosen MDD technology and transforming their models from one step to another. One of versions of TEE is a MOLA plug-in for the UML tool RSA.

The main component of **MOLA TEE** is the **MOLA Virtual machine (VM)** (interpreter), which actually performs the transformation of the source model to the target model.

A more detailed overview of MOLA environment architecture is given in [20].

V. Basic Principles of MOLA Virtual Machine

As it was already mentioned in the introduction, the goal of this research is to provide a simple and sufficiently efficient implementation of MOLA. The key factor in reaching this goal is an appropriate implementation of MOLA VM, since the implementation cost and efficiency of all the service components is nearly the same for all considered solutions to MOLA VM. And in turn, a crucial point of MOLA VM implementation is an appropriate repository and execution environment for pattern matching. This is due to the fact that the implementation of control structures and executable actions in MOLA (due to their procedural nature) is very straightforward in all cases. It should be noted that the choice of repository and execution environment are closely linked ones, thus the rest of the paper actually will be devoted to these issues.

Typically model transformation languages are implemented on metamodel based repositories, the most typical of which is Eclipse EMF [8]. Several experimental model transformation tools have been built using EMF as a repository [3,4,5]. The EMF API in Java provides the most basic actions for building a pattern matcher. Netbeans MDR [9] has somewhat similar characteristics and is used in [6].

The authors of this paper have already shown [1] that a very efficient MOLA pattern matching implementation is possible on such a basis. However, the available low level operations in these APIs (even lower level than analyzed in [1]) make the implementation sufficiently complicated. Therefore another solution was considered – to a what degree an SQL database can be used as a repository for pattern matching. On the one hand, the repository structure must match closely enough to EMOF [21] – similarly as EMF does. On the other hand, the desire was to use the powerful capabilities of SQL Select for a simple high level implementation of pattern matching. Such a solution was found, which is described in the next section. The only remaining concern was performance issues – whether the query optimization in SQL databases can at least be not very far from the optimal performance described in [1].

VI. Implementing Patterns by Natural SQL Queries

MOLA VM operates with models – MOF level M1. However, for each model element its metaclass must be known – for pattern matching or any other MOLA action. Therefore MOLA VM has to know the complete metamodel (M2 level) for the transformation. The metamodeling facilities in MOLA are approximately those of EMOF[21]. The most natural way is to store the metamodel in tables which correspond to EMOF metamodel classes. However, due to efficiency reasons, the “plain old class metamodel” containing Classes, Associations and Attributes (but not Properties as association ends) occurred to be more convenient to be coded by the corresponding SQL tables (see the left column of Fig. 1). It can be easily seen, that in fact it is equivalent to EMOF, therefore MOLA compiler can easily store the metamodel in these tables. In addition, there are tables for identifying metamodels and models themselves.

The storage of model elements – instances of metamodel classes, associations and attributes is completely straightforward in the corresponding three tables (see the right column of Fig. 1). The MOLA program is also naturally stored in tables according to the MOLA metamodel, but since we here are mainly concerned with pattern matching, this coding is not so important for the paper. The only fact to be mentioned here is that the MOLA compiler for each program element (loop, rule, pattern class element, pattern link etc.) generates a unique identifier. This fixed database schema is much easier to implement than the metamodel-specific one used in [10].

Now we will show how a MOLA pattern can be naturally mapped to an SQL Select statement. The idea is that each class element in the pattern corresponds to an occurrence of the table `class_inst` (actually an alias of it) in the `From` clause. Similarly, each pattern link corresponds to an alias of the `asoc_inst` table in the `From` clause. Then the `Where` clause is formed. Firstly, each pattern element (i.e., the corresponding alias of `class_inst`) must mandatory have the specified class, i.e., its `meta_class_id` column must have the given value (metamodel elements are fixed during MOLA execution). Similarly it is for links (association instances) in the pattern. A more non-trivial part of the `Where` clause must specify that each link does link the substituted by additional aliases of `attr_inst` in the `From` clause, in addition, the transformed expression must be added to the `Where` clause. Currently all MOLA expressions have direct counterparts in SQL.

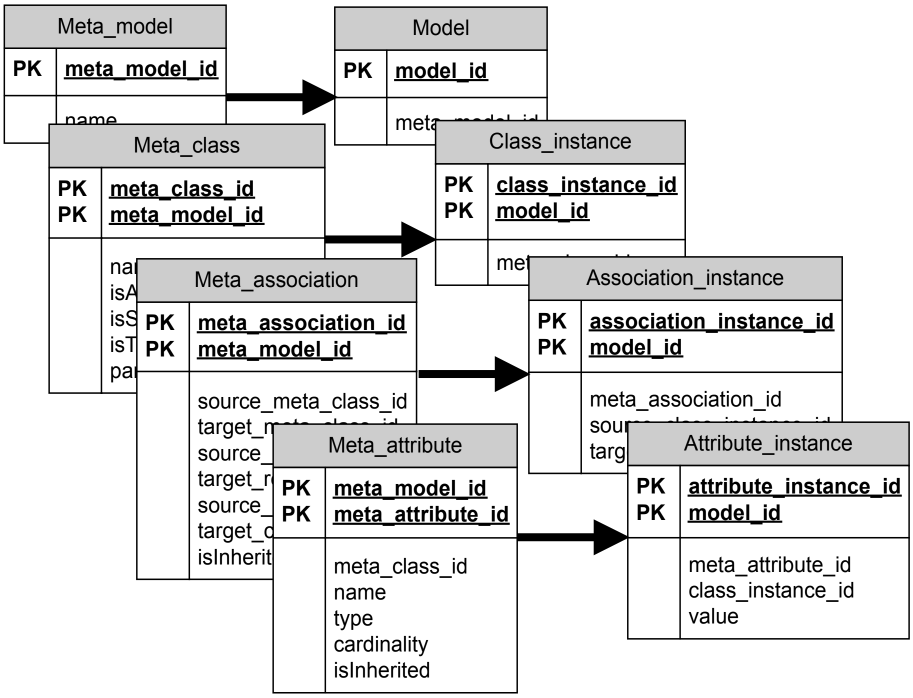

Fig. 1. SQL Tables for storing metamodels and models.

relevant instances, i.e., `src_class_inst_id` is equal to the `class_inst_id` of the corresponding (association source) alias of `class_inst`, similarly for the `trg_class_inst_id`. For reference elements (`@p:Package` in Fig. 2) it must be specified, that their `class_inst_id` has the given value (reference elements always correspond to a fixed instance in MOLA). The most complicated part in the `Where` clause are the attribute constraints, which already are Boolean expressions. However, the simple attribute names used in MOLA constraints must be

Fig. 2 illustrates the generation of an SQL query from a pattern. The pattern is a very simple one – a FOREACH loop head containing the loop variable (of type `Class`, with a constraint) and a reference (to the instance of `Package`) linked by the `package` link. Lines illustrate the described above mapping graphically, the color coding (or levels of gray in the black-and-white version) shows which parts of the query were obtained from one pattern element. The alias names are generated from the pattern element identifiers built by the MOLA compiler and therefore are unreadable.

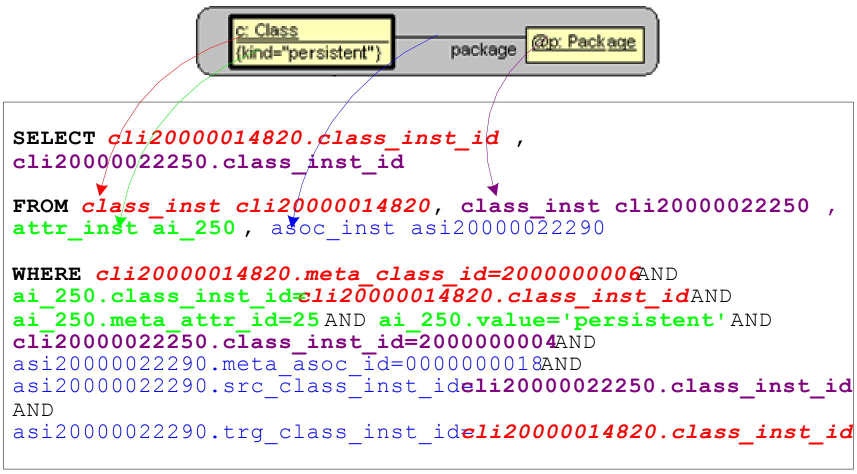

Fig. 2. Generation of an SQL query from a pattern.

The result of the query (a virtual table) is defined in such a way that each row represents (identifiers of) class instances forming a valid match.

Now it can be easily seen, that the built SQL query indeed expresses the pattern match semantics, which for the given example asserts that instances of the metaclass `Class` must be sought, which have the link `package` to the fixed instance of `Package` and which have the given value of the attribute `kind`. Since the pattern is inside a FOREACH-loop, all such instances (all matches returned by the query in this simple case) must be processed. A similar argument applies to any MOLA pattern. To cope with the fact that MOLA loops which are *not fixed* can replenish the instance set used for the match, actually for loop patterns the query is re-executed after each iteration, with instances of the loop variable already used being fixed in a special list. For MOLA loops which are *fixed* for loop patterns the query is executed only once, because all matches are returned by the built SQL query.

Thus the simplicity of the pattern mapping to SQL query has been shown, it remains to show that this SQL Select can easily be built by the MOLA VM (actually it is a sort of "JIT-compiling"). It is being done in several steps. First, the class elements of the pattern are picked up and for each of them an element in the `Select` list and in the `From` list (the table `class_inst` with a new alias) is added, with the MOLA compiler-generated unique element identifier used as the alias name. In addition, a term in the `Where` condition is added, which specifies that the instance must be of the relevant class (or that the instance is the given one for reference elements). Then in a similar manner each link of the pattern is processed. Here the term added to the `Where` part is more complicated, it has to state both that the link's association is the relevant one and that the endpoints are the corresponding class instances. The latter fact is easily to state due to the fact that the MOLA compiler has documented this via references to the relevant element identifiers and namely these identifiers are used as aliases for the element selection. Then pattern constraints are processed, each adding to the `From` part (the required attribute instance) and to the `Where` part (the expression itself). Currently simple OCL expressions having a direct counterpart in SQL and some simple OCL set expressions are supported, but this repertoire will be extended.

Finally, some remarks on the negative patterns. A negative part can be added as a `NOT EXISTS` subquery to the `Where` condition. In the case of a NOT-element, the subquery has just one alias of the `class_inst` in the `From` list plus aliases for the links connecting the element with the positive part of the pattern. The `Where` part of the subquery is generated similarly as for positive patterns. If the negative part is a NOT-region, all elements of this region (plus connecting links) are placed in the subquery.

VII. Database Performance Issues

In this section we analyze the performance of the generated queries in several databases, which are relevant for MOLA tool. A query generated from a pattern is somewhat special in the sense that it is a so-called **self-join** – aliases of the tables `class_inst` and `asoc_inst` are repeated in the `From` clause as many times as there are elements and links in the pattern respectively. Large self-join queries are non-typical for standard database applications and therefore may be processed by some engines not so optimally.

The first natural choice for an experimental tool was the open source database MySQL, currently the version 5.0.12. The first intuitive performance evaluations were also encouraging, but it was clear that a more thorough analysis of query optimization is required.

Since the authors have shown [1] that pattern matching in MOLA can be performed very efficiently as a sequence of small queries on a reasonable model repository (and the database schema described in this paper is such), it is clear that potentially the generated "large" queries can also be executed efficiently. Since the performance of a join type SQL query is mostly dependent on the join order of tables in `WHERE` part [22], the right order in which the tables in a complicated self-join are joined must be found that is equivalent to the sequence of small queries.

Let us explain the situation in detail on an example (Fig. 3). This example is a fragment of the MOLA transformation transforming a class model to OWL notation [23] (used as a benchmark in section 8), namely, the FOREACH loop head is shown, which generates an OWL object property for each UML association instance (for classes the corresponding OWL Classes are already built). It was shown in [1], that for simple cases such as in Fig. 3, the optimal order is to start from the loop variable (the element `as:BinaryAssociation`, all instances of which must be tested anyway), and to proceed along the paths leading away from the loop variable. In the example there are two such paths – one leading via the link `targetEnd` to `objEnd:Property` and further, and another one starting with the link `sourceEnd`. Even without seeing the

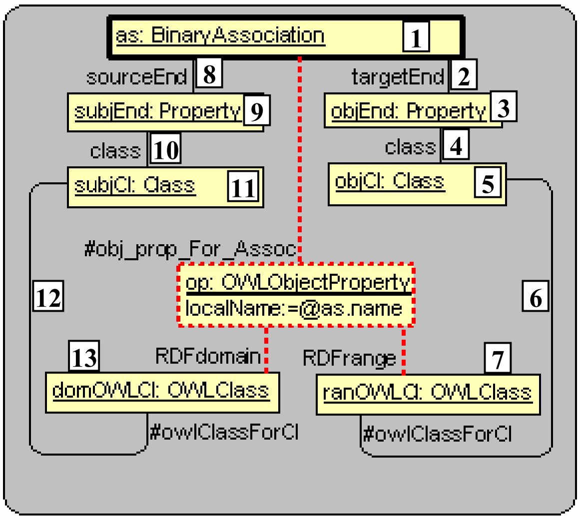

Fig. 3. Optimal pattern matching order.

metamodel, it is clear that in a valid class model this is an optimal order – a UML binary association has just one `targetEnd` (i.e., just one row in the table `asoc_inst`, where the join condition is true), which in turn is followed by just one `objEnd` (one row in `class_inst`) and so on. Fig. 3 illustrates this order by numeric tags.

Certainly, there are other optimal orders – any of the paths could be traversed first, and the paths can be traversed "intermittently". Similar easy-to-be-explained optimal join orders exist for more complicated patterns, where paths may have "cross-links" and where reference (fixed) elements exist (see more in [1]).

The generated query corresponding to this pattern is shown in Fig. 4.

```
ELECT cli20000020780.class_inst_id , cli20000020970.class_inst_id ,
cli20000021040.class_inst_id , cli20000021110.class_inst_id ,
cli20000021180.class_inst_id , cli20000021260.class_inst_id ,
cli20000021330.class_inst_id
ROM class_inst cli20000020780 , class_inst cli20000020970 , class_inst cli20000021040 ,
class_inst cli20000021110 , class_inst cli20000021180 , class_inst cli20000021260 ,
class_inst cli20000021330 , asoc_inst asi20000021080 , asoc_inst asi20000021150 ,
asoc_inst asi20000021300 , asoc_inst asi20000021400 , asoc_inst asi20000021700 ,
asoc_inst asi20000021760
WHERE cli20000020780.meta_class_id=2000001847 AND
cli20000020780.meta_model_id=0000000000 AND cli20000020780.model_id=0 AND
cli20000020970.meta_class_id=2000001790 AND
cli20000020970.meta_model_id=0000000000 AND cli20000020970.model_id=0 AND
cli20000021040.meta_class_id=2000001721 AND
cli20000021040.meta_model_id=0000000000 AND cli20000021040.model_id=0 AND
cli20000021110.meta_class_id=2000001723 AND
cli20000021110.meta_model_id=0000000000 AND cli20000021110.model_id=0 AND
cli20000021180.meta_class_id=2000001790 AND
cli20000021180.meta_model_id=0000000000 AND cli20000021180.model_id=0 AND
cli20000021260.meta_class_id=2000001721 AND
cli20000021260.meta_model_id=0000000000 AND cli20000021260.model_id=0 AND
cli20000021330.meta_class_id=2000001723 AND
cli20000021330.meta_model_id=0000000000 AND cli20000021330.model_id=0 AND
asi20000021080.meta_asoc_id=2000001835 AND
asi20000021080.meta_model_id=0000000000 AND
asi20000021080.src_class_inst_id=cli20000021040.class_inst_id AND
asi20000021080.trg_class_inst_id=cli20000020970.class_inst_id AND
asi20000021080.model_id=0 AND asi20000021150.meta_asoc_id=2000001725 AND
asi20000021150.meta_model_id=0000000000 AND
asi20000021150.src_class_inst_id=cli20000021040.class_inst_id AND
asi20000021150.trg_class_inst_id=cli20000021110.class_inst_id AND
asi20000021150.model_id=0 AND asi20000021300.meta_asoc_id=2000001835 AND
asi20000021300.meta_model_id=0000000000 AND
asi20000021300.src_class_inst_id=cli20000021260.class_inst_id AND
asi20000021300.trg_class_inst_id=cli20000021180.class_inst_id AND
asi20000021300.model_id=0 AND asi20000021400.meta_asoc_id=2000001725 AND
asi20000021400.meta_model_id=0000000000 AND
asi20000021400.src_class_inst_id=cli20000021260.class_inst_id AND
asi20000021400.trg_class_inst_id=cli20000021330.class_inst_id AND
asi20000021400.model_id=0 AND asi20000021700.meta_asoc_id=2000001858 AND
asi20000021700.meta_model_id=0000000000 AND
asi20000021700.src_class_inst_id=cli20000020780.class_inst_id AND
asi20000021700.trg_class_inst_id=cli20000020970.class_inst_id AND
asi20000021700.model_id=0 AND asi20000021760.meta_asoc_id=2000001852 AND
asi20000021760.meta_model_id=0000000000 AND
asi20000021760.src_class_inst_id=cli20000020780.class_inst_id AND
asi20000021760.trg_class_inst_id=cli20000021180.class_inst_id AND
asi20000021760.model_id=0
```

Fig. 4. Generated query example.

Further, it was to be found, how close the MySQL query execution plans are to an optimum, and at what expenses such a plan is found. Fortunately, MySQL has the `Explain` statement [24], which reveals some details of the execution plan. Fig. 5 shows the join order of query shown in Fig. 4, exposed by the `Explain` statement. Actually, two experiments are merged there – one with order tags in squares has been performed on a small source model (29 rows in `class_inst`, 39 rows in `asoc_inst`). Another one has been performed on a large source model (725 rows in `class_inst`, 975 rows in `asoc_inst`), the join order (where different from the first one) is shown in circles. For the large model the join order is equivalent to the optimal one, only another starting point has been selected, and paths are traversed intermittently. For the small one the deviation is larger, but also not critical.

However, if the number of elements and links in a pattern is increased, the query execution time also increases. The query (discussed above) having a pattern with 7 elements and 6 links executes in 200ms on a model with 3000 class instances and 4000 links, a query with 8 elements and 7 links in 600 ms on the same model, 9 elements and 8 links in 3200ms, but 10 elements and 9 links in 43000ms that is a significant jump. There are only few papers on MySQL optimization [25,26], and they do not explain the optimization of the specific self-join queries used in MOLA pattern matching. Another observation should be mentioned – the `Explain` statement [24] execution itself requires nearly as much time as the query execution, so we can assert that MySQL query optimization in case of large self-join queries is not optimal – it itself is too time consuming. Thus we have to rely on our "black box" experiments, which say that MySQL optimization is acceptable when there are limits on the pattern size (no more than 8 elements), but the query execution time increases too much for larger patterns, to make sense in using this RDBMS for pattern matching.

Thus the current version of MySQL can be used for MOLA runtime repository, but with restrictions on MOLA transformation patterns. The hope is for versions to come (the current version performs better than those tested earlier), but next versions could only raise the limit for pattern size – not remove this restriction completely.

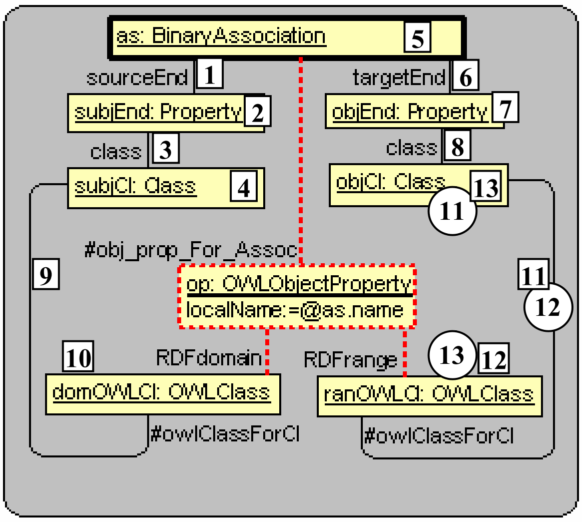

Fig. 5. MySQL query plan (table join order).

Due to the mentioned above problem other alternatives were sought. Possible alternatives are MSDE 2000 [27] – the free "small" version of MS SQL 2000 server, PostgreSQL [28] – another popular open source RDBMS, MSSQL Server 2005 Express [29] - – the free "small" version of MS SQL 2005 server. Similar performance experiments on large queries have been performed with these engines too. Single pattern query execution times for these alternatives were significantly better (Microsoft products) or similar

(PostgreSQL). The join order was nearly optimal. It can be concluded from available references [30] that both MS SQL and MSDE use instance data for query optimization in a more sophisticated way. However, experiments show that execution of a complete transformation is much slower than by using MySQL. MySQL was faster by an order of magnitude. It seems that MSDE 2000 and MSSQL Server 2005 Express engines have major problems with completing large sequences of SQL queries, because of built-in features such as workload governor [31] in MSDE 2000, that decreases the server performance.

An alternative approach would be to enforce the optimal join order manually, since MySQL has such possibilities. Unfortunately, these features are vendor-specific extensions of SQL. In addition, finding of this order during query generation is a significant part of implementing the pattern via "small queries" and therefore much more complicated.

VIII. BENCHMARK RESULTS

The previous section demonstrated that usage of MySQL database server as model repository and pattern matching engine has proven to be sufficient. To estimate MOLA Tool performance the experiments have been done.

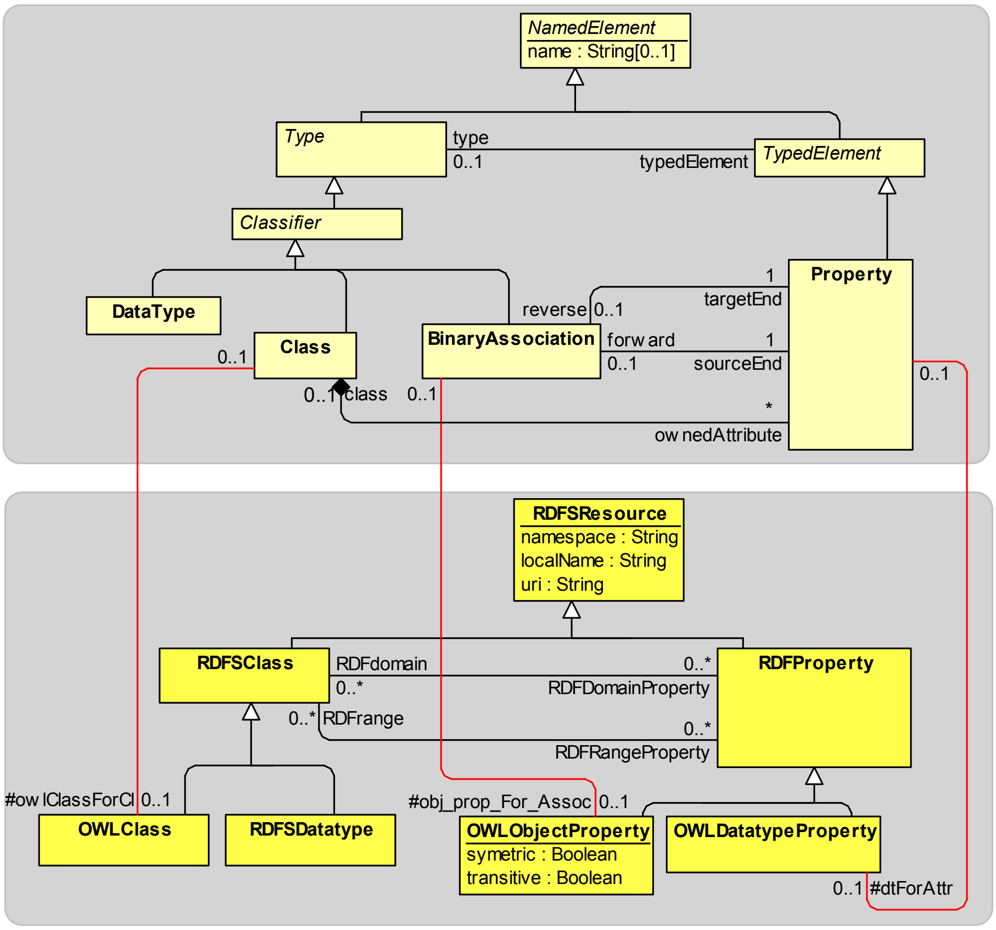

Fig. 6. Metamodels of UML Class Diagram and OWL Diagram.

Thus, MySQL is a satisfactory implementation for MOLA runtime repository if the pattern size does not exceed 8-9 elements (actually, only the “free” pattern elements count – those which are class elements, but not references or parameters, in Fig. 5 all pattern elements are free). The existing experience of using MOLA tool on some nearly real life examples has confirmed this. The transformation execution times in these examples testify that apparently close-to-optimal join order was used by MySQL in most cases. Nearly all patterns in these examples were below the size limit. In practice it is also possible to bypass the limit by decomposing a pattern into several smaller ones (actually, even without sacrificing the transformation readability).

A simple task and appropriate model transformation tool for comparison have been chosen. The choice - AGG[12] is a popular graph transformation language, that uses pattern constructs similar to MOLA, only explicit NAC’s (negative application conditions) must be added. AGG rules have no explicit control structures, but in simple cases MOLA control structures can be adequately emulated by AGG rule layering. AGG has already been used for benchmark testing [10], thus allowing us to ensure certain correctness of the experiment. The transformation was executed on both MOLA Tool and AGG for models of various size and complete execution times were measured. Both MOLA Tool and AGG were used with configurations recommended by developers.

The example transforms simplified UML class diagram to simplified OWL diagram. Metamodels are shown in Fig. 6.

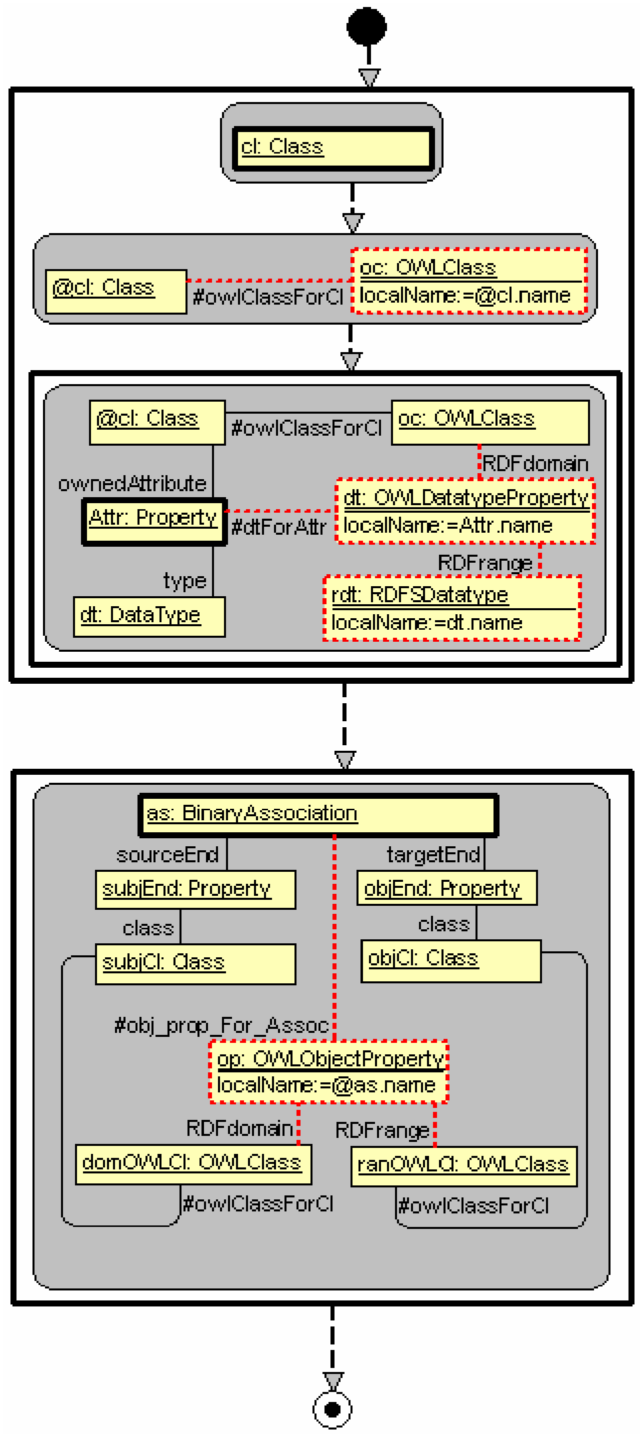

Fig. 7. Transformation UML Class Diagram to OWL Diagram

The transformation creates an OWLClass instance for every Class instance and OWLDataTypeProperty for every Property which is an owned attribute of the Class. This task is done using nested loops. The first *fixed* **foreach** loop iterates through all Class instances and the nested *fixed* **foreach** loop iterates through appropriate Property instances. The third *fixed* **foreach** loop creates OWLDataTypeProperty for each BinaryAssociation (Fig. 7). Though this transformation is very simple it is a typical representative of MDD tasks where frequently a model has to be transformed to a semantically equivalent one in another notation.

The transformation was executed on a hyper-threaded Intel Pentium4 3GHz, 1 GB RAM Windows XP workstation. No additional performance tuning was done to MySQL database server or operating system configuration. Identical models of various sizes were prepared for MOLA Tool and AGG. The first column of Table 1 contains model data size N – the number of class instances in the model. Second and third columns contain complete transformation time for MOLA and AGG measured in seconds.

Both MOLA Tool and AGG showed sufficient performance on models with size below N=175. MOLA Tool execution time grows nearly linearly up to model size N=3500, but starts to grow faster above this value. Thus the current MOLA Tool implementation performs well in this range, but real examples could be also larger – there are ontologies containing more than 5000 OWL Classes. Real transformations are also more complicated. AGG has problems similar to MOLA Tool, but both tools are usable for tasks they are designed for.

The main relational database engine feature, which enables fast search is table indexing [30]. The MOLA Tool uses table indexes in the most appropriate way, apparently this ensures the nearly linear time growth for queries.

The reason for faster complete transformation time growth for large N lies in the fact that the model size grows while transformation is being executed.

TABLE I.
BENCHMARK RESULTS.

	Transformation ExecutionTime (s)	
Model size (N)	**MOLA**	**AGG**
42	1	4
56	1	6
70	2	9
84	3	14
175	5	62
400	10	334
1050	19	8280
1750	36	
3500	65	
17500	1781	

A proportional to N number of insert and update operations must be done in this MOLA program and each operation time grows due to the need of refreshing indexes (but indexes are crucial for fast pattern matching). A similar problem is the main reason for AGG slowdown, even to a larger degree, as it is shown in [10]. For real MDD tasks it is typical that a new model must be built of size proportional to the source model. Thus not only the pattern match time influences the performance, but still it seems to be the key factor.

IX. CONCLUSIONS

In the paper the main principles and solutions used in the MOLA virtual machine have been described. It is shown that both simple and sufficiently efficient implementation of pattern matching via SQL queries has been built. Thus this is a viable solution at least for an experimental tool (what MOLA tool currently is). Several model transformations supporting real MDD style development (automated use of Hibernate persistence framework in Java – a plug-in for the RSA tool, conversion of UML activity diagrams to BPMN notation and other) have been built and tested on examples of realistic size. In none of these examples the "natural" pattern size in MOLA programs exceeded 8 – the critical value up to which the given MOLA implementation is efficient. These experiments and benchmark tests described in the paper have shown that the implemented MOLA VM performs satisfactorily and MOLA is a suitable transformation language for typical MDD tasks – transforming a UML model to another one closer to the system implementation. However, for an industrial usage of MOLA a special in-memory repository and a compiler/interpreter that implements the principles described in [1] could be required. The main reason could be the desire to get rid of any limits on pattern size; also the general performance for large models is expected to be better.

Certainly, these results obtained for MOLA implementation have value also for other transformation languages, where the pattern match semantics is similar.

REFERENCES

[1] A. Kalnins, J. Barzdins, E. Celms. "Efficiency Problems in MOLA Implementation". 19th International Conference, OOPSLA'2004 (Workshop "Best Practices for Model-Driven Software Development"), Vancouver, Canada, October 2004. URL:http://www.softmetaware.com/oopsla2004/mdsd-workshop.html

[2] 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

[3] ATL. URL: http://www.sciences.univ-nantes.fr/lina/atl/

[4] Tefkat.URL: http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/

[5] MTF. URL: http://www.alphaworks.ibm.com/tech/mtf

[6] ArcStyler. URL: http://www.interactive-objects.com/

[7] AndroMDA http://www.andromda.org/

[8] UML 2.0 Eclipse EMF. URL: http://www.eclipse.org/uml2/

[9] Metadata Repository (MDR). URL: http://mdr.netbeans.org/

[10] G. Varro, A. Schurr, D. Varro "Benchmarking for Graph Transformation" Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing 2005 (VL/HCC 05), Dallas, Texas, USA, September 2005, IEEE Press, pp 79-88.

[11] G. Varro, K. Friedl, D. Varro "Graph Transformations in Relational Databases" Proceedings of GraBaTs 2004: International Workshop on Graph Based Tools, Rome, Italy, 2004, Elsevier.

[12] AGG - The Attributed Graph Grammar System. URL: http://tfs.cs.tu-berlin.de/agg/

[13] A.Kalnins, J. Barzdins, E.Celms. "Model Transformation Language MOLA." - LNCS, Springer, v. 3599, 2005. Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linkoping, Sweden, June 10-11, 2004. Rev. Sel. Papers, pp. 62-76.

[14] Kalnins A., Barzdins J., Celms E. "Model Transformation Language MOLA: Extended Patterns". Selected papers from the 6th International Baltic Conference DB&IS'2004, IOS Press, FAIA vol. 118, 2005, pp. 169-184.

[15] A. Kalnins, J. Barzdins, E. Celms. "Basics of Model Transformation Language MOLA". ECOOP 2004 (Workshop on Model Transformation and execution in the context of MDA), Oslo, Norway, June 14-18, 2004. URL: http://heim.ifi.uio.no/~janoa/wmdd2004/papers/

[16] A. Kalnins, J. Barzdins, E. Celms. "MOLA Language: Methodology Sketch". Proceedings of EWMDA-2, Canterbury, England, 2004. pp.194-203.

[17] E. Celms, A. Kalnins, L. Lace. "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.

[18] A. Kalnins, E. Celms, A. Sostaks. „Model Transformation Approach Based on MOLA". ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML '2005). (MoDELS/UML'05 Workshop: Model Transformations in Practice (MTIP)) , Montego Bay, Jamaica, October 2 -7, 2005, p. 25.

[19] OMG, MOF 2.0 Query/View/Transformation Specification. URL: http://www.omg.org/docs/ptc/05-10-02.pdf

[20] A.Kalnins, E.Celms, A.Sostaks, "Tool support for MOLA", Fourth International Conference on Generative Programming and Component Engineering (GPCE'05), Proceedings of the Workshop on Graph and Model Transformation (GraMoT), Tallinn, Estonia, September 2005, pp. 162-173 (preliminary version).

[21] OMG, Meta Object Facility (MOF) Core Specification. URL: http://www.omg.org/docs/formal/06-01-01.pdf

[22] C. J. Date, "An Introduction to Database Systems", Chapter 17, Optimisation, Addison-Wesley, 7th Edition, 2000

[23] W3C. "Web Ontology Language (OWL)". URL: http://www.w3.org/2004/OWL/

[24] MySQL Reference Manual. URL: http://dev.mysql.com/doc/mysql/en/index.html

[25] T. Katchaounov. "An Overview of the MySQL 5.0 Query Optimizer". The MySQL Users Conference, 2005. URL:http://conferences.oreillynet.com/presentations/mysql05/timour_update.pdf

[26] P. Dubois. "MySQL", Chapter 4, Query Optimization. Sams, 3rd Edition, 2005.

[27] Microsoft SQL Server 2000 Desktop Engine (MSDE 2000). URL: http://www.microsoft.com/sql/msde/default.asp

[28] PostgreSQL - Open Source Database Server. URL: http://www.postgresql.org/

[29] Microsoft SQL Server 2005 Express Edition URL: http://www.microsoft.com/sql/editions/express/default.mspx

[30] R. Elmasri, S. Navathe. "Fundamentals of Database Systems", Chapter 18, Query Processing and Optimisation, Addison-Wesley, 3rd Edition, 2000.

[31] The SQL Server 2000 Workload Governor. URL: http://msdn.microsoft.com/library/default.asp?url=/library/enus/architec/8_ar_sa2_0ciq.asp