Implementing an EJB3-Specific Graph Transformation Plugin by Using Database Independent Queries

Gergely Varró [1,2]

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

Abstract

The current paper presents a novel approach to implement a graph transformation engine as an EJB3-specific plugin by using EJB QL queries for pattern matching. The essence of the approach is to create an EJB QL query for the precondition of each graph transformation rule. Pattern matching and updating phases of a rule application are implemented in a public method of a stateless session bean by executing the prepared EJB QL query and by manipulating persistent objects, respectively.

Key words: graph transformation, EJB 3.0, EJB QL queries.

1 Introduction

Nowadays, the immense role of model transformation concepts and tools is unquestionable for the success of model-driven systems development. Model transformation approaches should support cost and time efficient specification, design, execution, validation and maintenance of manipulations within and between modeling languages. As different phases of transformation design have conflicting requirements, their optimal solution also necessitates different approaches.

In a recent paper [3], we proposed to separate the design of model transformations from their *execution* by generating stand-alone plugins for the EJB 3.0 platform from platform-independent specifications of transformations given by a combination of graph transformation and abstract state machine rules.

Based on the observations of several studies [10,3,14], it may be stated that (i) graph pattern matching is the critical part in graph transformation, and (ii)

[1] This work was partially supported by the SENSORIA European project (IST-3-016004).
[2] Email: gervarro@cs.bme.hu

very large models can only be handled by EJB3-based plugins with an underlying database as pure Java solutions run out of memory. In the paper, we examine the generation of EJB3-based graph transformation plugins.

In addition to handle very large models, EJB3-based graph transformation plugins have further advantages including (i) the transparent access to system models via a traditional Java interface by hiding the underlying relational database where these models are physically stored, (ii) the atomic execution of graph transformation rules by using the transaction handling mechanism of the application server, (iii) the integration of graph transformation into existing business applications via a standard business logic interface defined by EJB3.

The current implementation of EJB3-based graph transformation plugins (also reported in [3]) performs the computation intensive graph pattern matching on business-level objects. This solution suffers from unnecessary memory handling operations as all objects being traversed must be loaded into the application server at least once, even if the pattern has only a couple of successful matchings.

Our current aim is to improve the performance of pattern matching in graph transformation plugins by using the query support of EJB3. In this case, queries are executed in the underlying relational database, and only those business-level objects are loaded into the application server, which effectively participate in at least one successful matching.

EJB3 provides two declarative languages (the Standard Query Language (SQL) [12] and the EJB Query Language (EJB QL) [11]) for specifying queries. Since the underlying relational databases typically use different dialects of SQL, an approach that uses database dependent SQL queries for describing graph transformation like the one presented in [13] would not be portable.

In order to provide a portable solution, the current paper proposes a novel approach to implement an EJB3-specific graph transformation plugin by using EJB QL queries for pattern matching. The essence of the approach is to create an EJB QL query for the precondition of each graph transformation rule by using *search plans* [17], which have been calculated by some sophisticated algorithms [17,6,16] for the LHS and NAC patterns of the rule in a preprocessing phase. Pattern matching and updating phases of a rule application are implemented in a public method of a stateless session bean by executing the prepared query and by manipulating persistent objects, respectively.

In contrast to [3], the main novelty of this paper is *the usage of database independent EJB QL queries for pattern matching*. Consequently, in the current paper, we only focus on graph pattern matching techniques (as in Sec. 4.2) and completely omit the handling of the updating phase.

The main advantages of the proposed approach include (i) the ability to handle large models in contrast to pure in-memory solutions; (ii) portability and database independence contrary to pure SQL-based approaches; and (iii) reduced memory consumption in the application server compared to other EJB3-based solutions.

The rest of the paper is structured as follows. Section 2 provides a brief introduction to models and metamodels, graph transformation and the main concepts of search plans. Sec. 3 gives an overview on the EJB3 platform and on the syntax of its query language. In Sec. 4, which is the main part of the paper, we sketch how to encode preconditions of graph transformation rules into EJB QL queries. Finally, some related work is reviewed in Sec. 5, while Sec. 6 concludes our paper.

2 Model manipulation by graph transformation

We first briefly introduce the main notions of metamodels and models, and then show how these models can be manipulated by using graph transformation.

2.1 Metamodels and models

In order to present the concepts of models, metamodels and transformations, a standard object-relational mapping (see e.g. [12]) will be used throughout this paper as a running example, which generates a relational database schema from a UML class diagram.

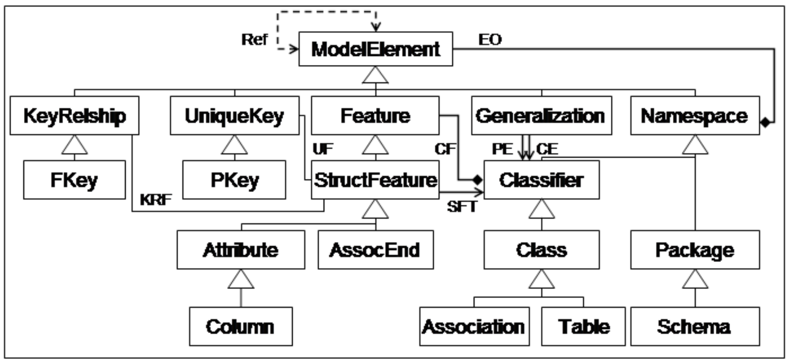


Fig. 1. An extended metamodel for the object-relational mapping

The *metamodel* describes the abstract syntax of a modeling language, which can be formally represented by a type graph. The metamodels of UML class diagrams and relational database schemas (following the CWM standard [9]) are depicted in Fig. 1. Nodes (e.g. Schema, Table) of the type graph are called *classes*. *Associations* like EO, CF, SFT, KRF and UF define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. We consider the most typical multiplicity constraints, which are (i) the at most one (denoted by arrows or diamonds), and (ii) the arbitrary (denoted by line ends without arrows and diamonds). Furthermore, we use one-to-one reference edges (denoted by bidirectional dashed lines in instance models) connecting source and target model nodes. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may specify further associations. Note that the CWM standard derives database notions like tables, columns, etc. from UML notions by inheritance (see Fig. 1). Finally, we assume without the loss of generality that multiple inheritance is not allowed and both ends of associations are navigable.

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

Example 2.1 A well-formed instance model of this domain (going to be shown in Fig. 2(a)) has a single package p that contains two classes (c1 and c2) and the association a. Association a connects class c1 to c2 via association ends ae1 and ae2, respectively. Package p is mapped to a corresponding schema s in the database. Additionally, a table with a single primary key column has already been added to schema s for each content (i.e., c1, c2, and a) of package p.

2.2 Graph transformation

Graph transformation [4] provides a pattern and rule based manipulation of graph models. Each rule application transforms a graph by replacing a part of it by another graph.

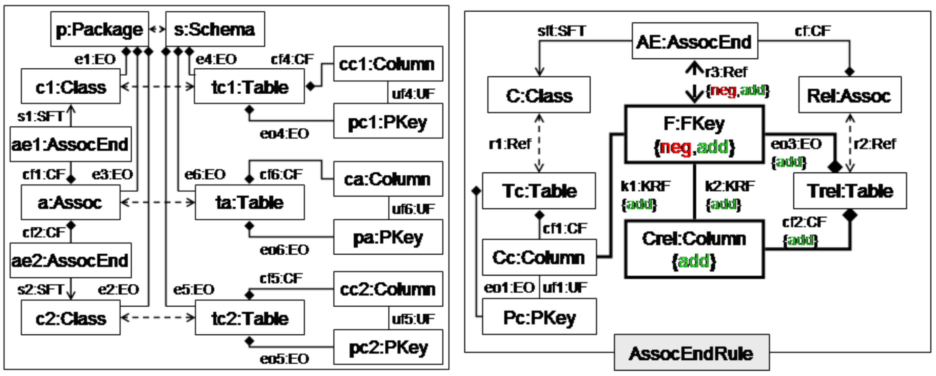
A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left–hand side graph pattern LHS, a right–hand side graph pattern RHS, and negative application condition graph pattern NAC [7]. The LHS and the NAC patterns are together called the precondition PRE.

In the paper, we use the graphical representation initially introduced in [5] where the union of these graphs is presented. Elements to be deleted are marked by the del keyword, elements to be created are labelled by the new, while elements in the NAC graph are denoted by the neg keyword.

The *application* of r to an *instance model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example 2.2 A single graph transformation rule (AssocEndRule in Fig. 2(b)) is selected as an example for the paper, which handles association ends.

The rule is applicable, if a table Tc with a primary key column Cc already exists for the class C representing the type of the association end AE, and moreover, there is a database table Trel that corresponds to the association Rel whose end is currently processed. The application of the rule creates a new column, which



(a) A sample instance model (b) A sample graph transformation rule

Fig. 2. A sample instance model and graph transformation rule

will refer to the already matched column Cc as a foreign key constraint. Graph transformation rules of the entire object relational mapping are presented in [15].

2.3 Search plans

Informally, a search plan defines a sequence of pattern nodes, which can be used at run-time during pattern matching to control the order of traversal for the objects of the instance model. At first, a search graph is constructed by using the LHS and NAC patterns of the rule. This step is followed by the execution of a sophisticated algorithm (e.g. [17,16]) that generates an optimal search plan on the search graph.

A *search graph* is a directed graph with the following structure. (i) Each node of the pattern is mapped to a *pattern node* (denoted by a solid circle) in the search graph. (ii) A *center node* (denoted by a hollow circle) is also added to the graph. (iii) *Iteration edges* are directed edges connecting the center node to every pattern nodes. The selection of one such edge means an iteration over all objects having the same type as the pattern node being located at the target end of the edge. (iv) Each navigable direction of each pattern edge is mapped to a *navigation edge* in the search graph.[3] The selection of one such edge corresponds to a navigation along the pattern edge in the given direction. If the navigation target of the pattern edge has an at-most-one (arbitrary) multiplicity constraint, then the corresponding navigation edge is referred to a *to-one (to-many) navigation edge*, and it is denoted by an arrow with single (double) arrowhead(s).

Starting nodes (denoted by dashed boxes) mark the center node and the set of pattern nodes that are already matched when the pattern matching starts. The remaining (initially unmatched) pattern nodes are called *traversed nodes* as they are processed during pattern matching, when appropriate objects are to be matched.

A *search plan* is a traversal of such spanning trees of the search graph that are rooted at some starting nodes. A traversal defines a sequence in which edges are

[3] Note that for each pattern edge, a pair of navigation edges having their end nodes connected in both directions is created as the pattern edge is navigable in both directions.

traversed. The position of a given edge in this sequence is marked by increasing integers written *on* the thick edges of spanning trees as in the left part of Fig. 3. In the following, we suppose that a search plan is available for each LHS and NAC.

Example 2.3 Search plans for LHS and NAC patterns of the AssocEndRule are shown in the upper and lower left part of Fig. 3, respectively. As matchings for NAC are searched after pattern matching for LHS is completed, shared nodes (i.e., AE) of LHS and NAC can be considered starting nodes in the search graph of NAC.

3 Enterprise Java Beans 3.0

The Java 2 Enterprise Edition (J2EE) platform defines a layered architecture for scalable, distributed application development including several Java standards and APIs. An enterprise application being developed on the J2EE platform consists of Enterprise Java Beans (EJBs) as its most fundamental building blocks representing business data and functionality. An enterprise application is deployed to and executed by an application server, which provides many high-level services (such as transactions, security, persistence, etc.) beyond the execution of applications.

The two types of EJBs used in the current paper are the following.

- *Entity beans* are persistent objects representing business data, which are kept synchronized with an underlying relational database by means of an object-relational mapping. Entity beans are uniquely identified by their primary key and they can be in relationship with other entity beans referring to each other by direct references (many-to-one or one-to-one relationships) or typed collections (many-to-many or one-to-many relationships).
- *Session beans* implement the business functionality of the application. They can be considered as simple collections of business methods. As our approach does not require any transformation related information to be stored, we use stateless session beans.

EJB Query Language.

An application server has an entity manager unit, which provides operations (i) for creating and removing persistent entity instances, (ii) for finding entities by their primary key, and (iii) for querying over entities.

Queries can be specified in the declarative, object-oriented EJB Query Language (EJB QL) [11]. Due to space limitations, only the structure of the SELECT statement is presented in the current paper, which has the following structure.

```
SELECT select_clause
FROM from_clause
WHERE where_clause
```

The SELECT *clause* denotes the result of the query by a comma separated list of identification variables. An *identification variable* is a variable that can refer to a single instance of a particular entity bean class.

The FROM *clause* designates the domain of the whole SELECT statement by a comma separated list of *identification variable declarations* of the form $type$ `AS` new_var. The $type$ of an identification variable new_var can be defined explicitly by using the name of an entity bean class, or implicitly by navigating along links of type assoc from an already declared variable old_var. In the latter case, the target class of assoc defines the type of identification variable new_var. Navigation is defined by path expressions old_var`.assoc` and `IN(`old_var`.assoc)`, if the navigation returns a single value and a collection, respectively.

The optional WHERE *clause* is a Boolean expression, and it filters out those results of the query that do not satisfy this expression. A *Boolean expression* is the conjunction (logical AND) of Boolean valued factors, which may test (i) the non-existence of results for a well-formed subquery (`NOT EXISTS (`$subquery$`)`), (ii) the equality of simple factors (sf_1=sf_2), and the (iii) inequality of simple factors (sf_1<>sf_2). A *simple factor* can be a constant, or a navigation operation (denoted by var`.id`) to access the identifier id of an identification variable var.

4 Graph transformation on EJB3 platform

Now we discuss how to generate an EJB3-specific graph transformation plugin, which follows the single pushout [10] approach with injective matchings.

4.1 Mapping metamodels and models to EJB3 entity bean classes and instances

Based on the metamodel, we generate entity bean classes by using the standard object-relational mapping of [11], which can be summarized as follows. (i) A class of the metamodel is mapped to an entity bean class. (ii) The inheritance relations between classes are maintained accordingly. (iii) Each association end with an at most one (arbitrary) multiplicity constraint is mapped to a Java attribute (collection) and two corresponding property accessor (i.e., a getter and a setter) methods in the entity bean class that represents the metamodel class being located at the opposite end of the association. (iv) A Java attribute `id` representing the unique identifier and its two corresponding property accessor methods are added to each entity bean class that does not have a superclass.

Example 4.1 The skeleton of the entity bean class representing a StructuralFeature is as follows.

```
@Entity
public class StructuralFeature extends Feature {
    private Classifier sft;
    private Collection<UniqueKey> uf = new ArrayList<UniqueKey>();
    private Collection<KeyRelationship> krf = new ArrayList<KeyRelationship>();

    @ManyToOne
    public Classifier getSFT() { return sft; }
    public void setSFT(Classifier sft) { this.sft = sft; }

    @ManyToMany(mappedBy="uf")
    public Collection<UniqueKey> getUF() { return uf; }
    public void setUF(Collection<UniqueKey> uf) { this.uf = uf; }
    @ManyToMany(mappedBy="krf")
    public Collection<KeyRelationship> getKRF() { return krf; }
    public void setKRF(Collection<KeyRelationship> krf) { this.krf = krf; }
}
```

As StructuralFeature is a subclass of Feature, the identifier attribute `id` has not been created. According to the metamodel of Fig. 1, the StructuralFeature class has three incident edges. Consequently, the generated code has three attributes and six accessor methods.

Instance models representing the system under design are stored in an underlying database of the application server. By using entity beans, objects of the instance model can be created, accessed and manipulated exactly the same way as traditional (plain old) Java objects with the single exception that these objects have to be explicitly persisted by calling the `persist()` method of the entity manager.

4.2 Graph pattern matching on EJB platform

By using search plans of LHS and embedded NAC patterns, we construct and execute a single SELECT EJB QL query that calculates and retrieves all the successful matchings of the precondition of a rule.

The general form of the query is as follows:

```
SELECT $node_1$, ..., $node_N$
FROM $traversed\_nodes$
WHERE $type\_checking\_constraints$ AND $check\_edge\_constraints$
  AND $injectivity\_constraints$ AND $NAC\_constraints$
```

A *traversed node* is an identification variable being declared in the FROM clause of the EJB QL query, which represents a pattern node being processed during the traversal of the search plan.

If a traversed node is reached by navigation in the FROM clause of an EJB QL query, then the type of this traversed node may be an ancestor of the type prescribed by the pattern node itself. This yields a situation where the traversed node possibly has a larger set of matching objects than it is allowed by the type restriction set up by the pattern node. In order to resolve this situation, an additional traversed node is declared for representing the same pattern node and a *type checking constraint* is defined to narrow the set of matching objects for this pattern node.

Traversed nodes declarations and type checking constraints are generated during search plan traversal, which processes search plan edges in increasing order.

Processing iteration edges. If an iteration edge with a target node trg is being processed, then an expression $type_{trg}$ `AS` trg is added to the end of the `FROM` clause where $type_{trg}$ is the type of the pattern node trg.

Processing to-one navigation edges. If a to-one navigation edge of type `assoc` connecting node src to trg is being processed, then expressions src`.assoc AS` trg`_sup` and $type_{trg}$ `AS` trg are appended to the end of the `FROM` clause, and a subformula trg`_sup.id =` trg`.id` is also added as a type checking constraint.

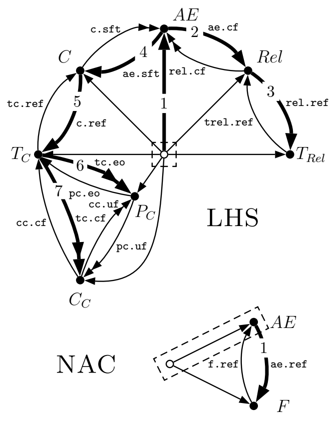
Processing to-many navigation edges. If a to-many navigation edge of type assoc connecting node src to trg is being processed, then terms `IN(`src`.assoc) AS` trg`_sup`, and $type_{trg}$ `AS` trg are appended to the `FROM` clause, and a subformula trg`_sup.id =` trg`.id` is also added as a type checking constraint.

An *edge checking constraint* expresses a restriction, which is caused by a pattern edge that has not been processed at all during the traversal of the search plan. For each pair of unnumbered navigation edges connecting nodes src and trg in both directions, we append a subformula src`.assoc.id=`trg`.id` or trg `MEMBER OF` src`.assoc` to the `WHERE` condition by using a logical AND operator for affixing, if src.`assoc` represents a to-one or a to-many navigation edge, respectively.

Injectivity constraints are defined for such pairs of pattern nodes where one member has a type that conforms to a supertype of the other. For each such pair $node_i$ and $node_j$, we add a subformula of the form $node_i$`.id <>` $node_j$`.id`.

NAC constraints express restrictions formulated by NAC patterns that are embedded into the pattern being processed. For each embedded NAC pattern, we add a constraint of the form `NOT EXISTS (`$subquery$`)`, where $subquery$ is the EJB QL query that is going to be generated for the embedded NAC pattern. Note that the `NOT EXISTS` constraint will be evaluated to true if and only if the subquery, which would list the successful matchings of the NAC pattern has no rows.

Example 4.2 To continue our running example, we present the SELECT statement (right part of Fig. 3) that is generated for the search plans of the LHS and NAC pattern of the AssocEndRule (as depicted in the upper and lower left corner of Fig. 3, respectively).



```
SELECT ae,rel,trel,c,tc,pc,cc
FROM AssocEnd AS ae,                    -- 1 (iter)
   ae.cf AS rel_sup, Association AS rel,-- 2 (one)
   rel.ref AS trel_sup, Table AS trel,  -- 3 (one)
   ae.sft AS c_sup, Class AS c,         -- 4 (one)
   c.ref AS tc_sup, Table AS tc,        -- 5 (one)
   IN(tc.eo) AS pc_sup, PKey AS pc,     -- 6 (many)
   IN(tc.cf) AS cc_sup, Column AS cc    -- 7 (many)
WHERE -- type checking constraints
   rel_sup.id=rel.id AND trel_sup.id=trel.id
   AND c_sup.id=c.id AND tc_sup.id=tc.id
   AND pc_sup.id=pc.id AND cc_sup.id=cc.id
   -- edge checking constraint
   -- (unprocessed edges between cc and pc)
   AND cc MEMBER OF pc.uf
   -- injectivity constraints
   AND c.id<>rel.id AND c.id<>tc.id
   AND c.id<>trel.id AND tc.id<>trel.id
   -- NAC constraint
   AND NOT EXISTS (
      SELECT ae,f
      FROM ae.ref AS f_sup, FKey AS f   -- 1 (one)
      WHERE f_sup.id=f.id
   )
```

Fig. 3. Search plans generated for the LHS and the NAC of AssocEndRule and the corresponding EJB QL query

Lines 1–12 of the query are generated during the traversal of the search plan of LHS, when its edges are processed in increasing order as shown by the comments at the ends of lines. (Expressions in parentheses denote the search plan edge processing method being used.) As neither edges between Cc and Pc are processed by the traversal, a corresponding edge checking constraint (lines 13–15) is added to the query. Metamodel classes Association and Table are subclasses of class Class, so C cannot be mapped to the same object as association Rel and tables Tc and TRel, and moreover, matchings for tables Tc and TRel must also differ as expressed by lines 16–18. The query for the NAC pattern (lines 19–24) is processed similarly with the single exception that AE now counts as a starting node as a matching for node AE has already been found.

On the implementation level, we map each graph transformation rule to a public method of the stateless session bean representing the whole graph transformation system. One such method first executes the prepared EJB QL query, then retrieves objects and links needed in the updating phase from the result list, and finally, it manipulates persistent objects. The handling of the updating phase is not mentioned in the current paper as we use the technique presented in [3].

Due to the similarity of the syntax and semantics of SQL and EJB QL queries, the proof for the correctness of the code generation algorithm would be similar to the one presented in [13]. The termination of the algorithm is guaranteed by the finiteness of nodes and edges in the precondition of graph transformation rules.

5 Related Work

Search plans are a widely used technique to *control the order of traversal for the objects of instance models* in algorithms that perform local search for pattern matching meaning that a partial matching is extended step-by-step by neighbouring objects and links. Here we shortly review the four most advanced approaches using *local search with search plans*.

- Fujaba [8] has a token graph based search plan definition [6], which uses a static model for defining the costs of basic operations (i.e., tokens). The optimization of search plans is guided by several well-established rules of thumb.
- PROGRES [17] uses a very sophisticated cost model for defining costs of basic operations of operation graphs, which are similar to search graphs in the current paper. The compiled version of PROGRES generates search plan by a greedy algorithm performed on the operation graph.
- The pattern matching engine of GReAT [1] employs a breadth-first traversal strategy starting from a set of nodes that are initially matched. GReAT also uses simple rules of thumb like Fujaba for search plan generation.
- The compiled version of VIATRA2 [2] employs model-sensitive search plans [16], which are calculated by greedy algorithms performed on search graphs containing statistical data collected from typical instance models.

In contrast to the above-mentioned methods, our approach uses search plans on a syntactic level for the generation of EJB QL queries. As search plans have been optimized in a preprocessing phase, the generated queries give optimal solution for pattern matching on a database independent level. Depending on the features and configuration possibilities of the underlying database, the user may either enforce the same execution order on the database level, or allow its alteration to exploit further database-specific optimization techniques.

6 Conclusion and Future Work

In the current paper, we proposed an EJB3-based graph transformation plugin, which uses queries specified in the declarative EJB QL language for pattern matching. This approach additionally provides a promising, object-oriented and database independent alternative of pure SQL based pattern matching solutions [13].

The essence of the technique is to formulate an EJB QL query and also to generate explicit Java code from search plans for the precondition of each graph transformation rule. The execution of the prepared query and the manipulation of persistent objects implement the pattern matching and the updating phases of graph transformation rule application on the EJB3 platform, respectively.

Our previous experiments [3,13] show that due to the same technology and the underlying relational database, this approach (just like previous EJB3-based graph transformation plugins) is able to handle models having more than 1 million elements for a performance penalty of an order of magnitude (compared to a pure Java solution) in case of smaller models. Based on these experiments, our expectation for the current approach is a slightly better run-time performance, and noticeably reduced memory consumption in the application server compared to solutions, which use pure SQL for specifying queries. As a natural limitation of the approach, it is worth to emphasize the trade-off between portability and run-time performance when database-specific query optimizations are switched on and off. In the future, we plan to carry out experiments to confirm our expectations on both the run-time performance and memory consumption aspects of our approach.

Acknowledgements. The author is very grateful to Dániel Varró for reading initial versions of the paper and giving valuable feedback.

References

[1] Agrawal, A., G. Karsai and F. Shi, *Graph transformations on domain-specific models*, Technical Report ISIS-03-403, Institute for Software Integrated Systems, Vanderbilt University (2003).

[2] Balogh, A. and D. Varró, *Advanced model transformation language constructs in the VIATRA2 framework*, in: *Proc. of the 21st ACM Symposium on Applied Computing* (2006), pp. 1280–1287.

[3] Balogh, A., G. Varró, D. Varró and A. Pataricza, *Generation of platform-specific model transformation plugins for EJB 3.0*, in: *Proc. of the 21st ACM Symposium on Applied Computing*, Dijon, France, 2006, pp. 1288–1295.

[4] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools," World Scientific, 1999.

[5] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the Unified Modeling Language*, in: G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, LNCS **1764** (1998), pp. 296–309.

[6] Geiger, L., C. Schneider and C. Reckord, *Template- and modelbased code generation for MDA-tools*, in: *Proc. of the 3rd International Fujaba Days*, 2005, pp. 57–62.

[7] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1996), pp. 287–313.

[8] Nickel, U., J. Niere and A. Zündorf, *The FUJABA environment*, in: *The 22nd International Conference on Software Engineering (ICSE)* (2000), pp. 742–745.

[9] Poole, J., D. Chang, D. Tolbert and D. Mellor, "Common Warehouse Metamodel," John Wiley & Sons, Inc., 2002.

[10] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations," World Scientific, 1997.

[11] Sun Microsystems, "JSR 220: Enterprise JavaBeans, Version 3.0," Early draft 2 edition (2005), `http://java.sun.com/products/ejb/docs.html`.

[12] Ullman, J. D., J. Widom and H. Garcia-Molina, "Database Systems: The Complete Book," Prentice Hall, 2001.

[13] Varró, G., K. Friedl and D. Varró, *Implementing a graph transformation engine in relational databases*, Journal on Software and Systems Modeling (2006), in press.

[14] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, in: *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing* (2005), pp. 79–88.

[15] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (2005), `http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf`.

[16] Varró, G., D. Varró and K. Friedl, *Adaptive graph pattern matching for model transformations using model-sensitive search plans*, in: G. Karsai and G. Taentzer, editors, *Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05)*, ENTCS **152**, Tallinn, Estonia, 2005, pp. 191–205.

[17] Zündorf, A., *Graph pattern-matching in PROGRES*, in: *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, LNCS **1073** (1996), pp. 454–468.