Graph Transformations in Relational Databases

Gergely Varró¹, Katalin Friedl¹, and Dániel Varró²

¹ Budapest University of Technology and Economics
Department of Computer Science and Information Theory
H-1117 Magyar tudósok körútja 2, Budapest, Hungary
{gervarro,friedl}@cs.bme.hu

² Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok körútja 2, Budapest, Hungary
varro@mit.bme.hu

Abstract. We present a novel approach to implement a graph transformation engine based on standard relational database management systems (RDBMSs). The essence of the approach is to store not only the instance graphs in relational database tables but also keep track explicitly of all the matchings of graph transformation rules in database views. Furthermore, the model manipulation prescribed by the application of a graph transformation rule is also implemented using elementary data manipulation statements (such as insert, delete, update). In this respect, the rich modeling paradigm of graph transformation can be directly integrated into off-the-shelf relational databases.

Keywords: Tool support, Pattern matching, Rule- and knowledge-based systems, Relational databases

1 Introduction

Graph transformation and relational databases. Relational database management systems (RDBMSs) that serve as the storage medium for business critical data for any large companies are probably the most successful products of software engineering. A crucial factor in this success is the close synergy between theory and practice: SQL, the standard data definition, manipulation and query language is built upon precise mathematical foundations.

Graph transformation [18] has proved its maturity for describing model queries and manipulations on a very high abstraction level. During the past years, intensive research has been focusing on how graph transformation could be adapted as a visual query and data manipulation language for databases. The following list is merely a brief selection of some main results in the field.

– Andries and Engels propose in [1] a hybrid (visual and textual) query language based upon graph transformation.
– In [8, 9], Jahnke and Zündorf propose the use of triple graph grammars [19] for database re-engineering of legacy systems in their Varlet framework.
GRAS [10, 12] is a graph-oriented database management system developed at the University of Aachen, which served as the underlying database for the PROGRES [20] graph transformation tool.

It is common in all these approaches that they investigate how graph transformation can contribute to database management systems and tasks. However, it is also worth examining how the mature theory and practice of RDBMSs can potentially contribute to the paradigm of graph transformation.

**Objectives.** In the current paper, we follow this second direction. More precisely, we report on the development of a graph transformation engine (not a full-featured graph transformation tool) which is implemented on an open, off-the-shelf relational database (namely, PostgreSQL [13]).

The essence of the approach is to store not only the models (instance graphs) in relational database tables but also keep track of all the matchings of graph transformation rules in database views. Furthermore, the model manipulation prescribed by the application of a graph transformation rule is also implemented using elementary data manipulation statements (such as INSERT, DELETE, UPDATE).

The main potential advantage of our approach is related to the MDA philosophy of model transformations (see [2]) where one can distinguish between platform-independent transformations (PIT) and platform-specific transformations (PST). PITs are very general specification techniques for model transformations, which can be simulated, tested and/or verified on a high-level of abstraction. Graph transformation is an ideal candidate for this purpose. Afterwards PITs are mapped automatically into PSTs, which provide implementations for the PIT in a specific technological space (such as XSLT, Java, or CWM). In fact, since the CWM standard [15] plays a central role in the MDA framework, relational databases are a highly relevant technological space for the final implementation of a model transformation.

However, a critical question is how the performance of a graph transformation engine based upon a relational database scales up for large models or long transformation sequences. After examining the performance of our prototype implementation on various problems and comparing it to the transformation engine of AGG [6], the popular graph transformation tool, we claim that such an implementation is a promising alternative.

**Structure of the paper.** 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 relational databases. In Sec. 3, which is the main part of the paper, we sketch how to encode graph transformation rules into SQL queries and operations. The experimental evaluation of our prototype graph transformation engine is provided in Sec. 4 on different examples (including a comparison with AGG for various cases). Finally, our conclusions are in Section 5.
2 Graph transformation and databases

2.1 Metamodels and models

The metamodel (MM) describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called classes (C). A class may have attributes (Attr) that define some kind of properties of the specific class. Inheritance may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Finally, associations (Assoc) define connections between classes (edge types between node types).

In the MOF terminology [16], a metamodel is defined visually in a UML class diagram notation. For instance, the left part of Fig. 1 is an extract from the metamodel of UML (which itself is a modeling language) defining the previous constructs of classes, associations and attributes. Furthermore, the right part of Fig. 1 is an extract from the CWM standard [15] that defines the main notions of relational database schemata.

The instance model (M) (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 (O) and links (L), respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as slots (S) in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation where instances of the superclass are required.

2.2 Graph transformation

Graph transformation [5,18] 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 \( r = (\text{LHS}, \text{AC}, \text{RHS}, \text{Asgn}, \text{NAC}) \) contains a left–hand side graph LHS with optional attribute conditions AC, a right–hand
side graph $RHS$ with optional attribute assignments $Asgn$, and negative application condition graphs $NAC$ (depicted by crosses). The $LHS$ and the $NAC$ graphs are together called the precondition $PRE$ of the rule. Sample graph transformation rules will be presented later in Fig. 2.

The application of $r$ to an host (instance) model ($M$) replaces a matching (or occurrence) ($occ$) of the $LHS$ in $M$ by an image of the $RHS$. This is performed by (i) finding a matching of $LHS$ in $M$, (ii) checking the negative application conditions $NAC$ (which prohibit the presence of certain objects and links) and the attribute conditions $AC$ (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 $LHS$) and further refreshing attributes defined by attribute assignments $Asgn$ obtaining the derived model ($M'$). A graph transformation is a sequence of rule applications from an initial model ($M_I$).

**Graph pattern matching** Typically, the most critical phase of a graph transformation step is graph pattern matching, i.e. to find a single (or all) occurrence(s) of a given $LHS$ graph in a host model $M$. This problem can be reduced to the subgraph isomorphism problem, which is known to be NP-complete. Despite this theoretical limit, existing graph transformation tools behave relatively well for larger graphs of practical problems.

Graph pattern matching techniques of these tools can be grouped into two main categories.

- Algorithms based on constraint satisfaction (such as [11] in AGG [6], VIATRA [21]) interpret the graph elements of the pattern to be found as variables which should be instantiated by fulfilling the constraints imposed by the elements of the instance model. Our implementation also falls into this category.
- Algorithms based on local searches start from matching a single node and extending the matching step-by-step by neighboring nodes and edges. The graph pattern matching algorithm of PROGRES (with search plans [22]), Dörr’s approach [4], and the object-oriented solution in FUJABA [14] fall in this category.

### 2.3 An overview on relational databases

In this section a short overview about relational database objects and operations is given.

- A column (attribute) $A$ is basically a named set with values conforming to a specific datatype.
- A tuple $r$ is an element from the set $A_1 \times \cdots \times A_m$.
- A table (relation) $R(A_1, \ldots, A_m)$ is a relation over $m$-ary tuples, defined formally as $R \subseteq A_1 \times \cdots \times A_m$. If a tuple $r$ is in relation $R$ (formally $r \in R$) then it appears as a row in the corresponding database table.
- Columns and tables can be renamed locally in a query by using the `AS` keyword.
- A projection of table $R$ to columns $B_1, \ldots, B_n$ (denoted by $\pi_{B_1 \ldots B_n}(R)$) is when we drop all the columns of table $R$ except for $B_i$s. Formally,

$$\pi_{B_1 \ldots B_n}(R) = \{(x_{B_1}, \ldots, x_{B_n}) \mid (x_1, \ldots, x_m) \in R \}.$$ 

In SQL terms projection is implemented using the `SELECT` construct

```
SELECT B_1, ..., B_n FROM R.
```

- Filtering (denoted by $\sigma_F(R)$) involves a single table and collects the rows of $R$ where formula $F$ holds. The formal definition of filtering is

$$\sigma_F(R) = \{(x_1, \ldots, x_m) \mid (x_1, \ldots, x_m) \in R \land F(x_1, \ldots, x_m)\}.$$ 

In this paper, only atoms of type $A = B$ (two column names in equality relation) and the logical and operator will be used in $F$. Filtering operation can be expressed in SQL using the `WHERE` condition (with $F$ as the parameter).

- A cross join $R \times S$ makes a Cartesian product of base tables. A row is in the result table, if its first $m$ values correspond to a row in $R$ and its last $n$ values forms a row in $S$. Its formal definition is:

$$R \times S = \{(x_1, \ldots, x_m, y_1, \ldots, y_n) \mid (x_1, \ldots, x_m) \in R \land (y_1, \ldots, y_n) \in S\}.$$ 

Cross join operation in SQL can be formulated as `SELECT * FROM R, S`.

- The inner join operation (denoted by $R \bowtie F S$) is a selection from the Cartesian product, i.e. a cross join $R \times S$ filtered by some formula $F$. SQL notation of the inner join operation is `SELECT * FROM R, S WHERE R.A=S.B` where $A$ and $B$ are those common columns in tables $R$ and $S$, respectively on which inner join is executed. The equality relation in the `WHERE` condition is the filtering formula $F$ in this case.

- The left outer join of $R$ and $S$ (denoted by $R \bowtie L S$)
  1. contains all the rows of $R \bowtie S$, and
  2. additionally contains all such rows of $R$, for which there are no rows in $S$ satisfying $F$. These rows are filled with `NULL` values for the columns of $S$.

The formal definition of left outer join is

$$R \bowtie L S = (R \bowtie S) \cup \{(x, \text{NULL}) \mid x \in R \land \nexists y \in S \text{ for which } F \text{ holds}\}.$$ 

A sample left outer join is `SELECT * FROM R LEFT JOIN S ON R.A=S.B`.

- A view $V$ is a derived table (relation) with a separate name. It can be defined with a full featured `SELECT` query where we may use all the aforementioned constructs and operations. Most database applications only store the defining query for a view definition (without explicitly storing the tuples belonging to the relation) and always execute the query when the view is used.
3 Graph transformation in relational databases

We present how a graph transformation engine can be implemented using a relational database. First, we create a appropriate database schema based on the metamodel, then the database representation of the model is generated (Sec. 3.1). Afterwards, the pattern matching phase of rule application is implemented using database queries (Sec. 3.2), finally data manipulation is handled (in Sec. 3.3).

3.1 Mapping models and metamodels to database tables

Instance models representing the system under design are stored in database tables. A standard way to define the schema of the database is driven by the metamodel and it can be implemented using the transformation rules of Fig. 2.

- SchemaR: We assign a schema $s$ for each package $p$ in the metamodel.
- ClassR: Each class $c$ is mapped to a table $t$ with a single column $id$. This column will store the identifiers of objects of the specific class.
- AssociationR: We assign a table $t$ for each association $rel$ that appears in the metamodel. This table has only a single column $id$ when it is created. This column contains the identifiers of links.
- AssocEndR: We add a column $relid$ to the table $t_{rel}$ for each association end $ae$. Furthermore, a new foreign key $rel2dir$ constraint that connects columns $cid$ and $relid$ should also be added to table $t_{rel}$.
- GeneralizationR: If a subclass $sub$ is inherited from a superclass $sup$, then table $t_{sub}$ that corresponds to the subclass should be extended by a foreign key constraint $subsup$ that links primary key columns $subid$ and $supid$. This means that all identifiers appearing in the subclass table should also appear in the superclass table as well.
- AttributeR: Each attribute $a$ of class $c$ is mapped to a column $col$ in the corresponding table $t$. Column $col$ will store the value of the given attribute.

3.2 Graph pattern matching in databases

Views for rule graphs ($LHS$, $NAC$). We propose to calculate the matching patterns of a graph transformation rule by using views (i.e. a SELECT query), which contain all the successful matchings of the rule. More specifically, we introduce separate views for each $LHS$, $NAC$, and $PRE$ graphs (which is a combination of an $LHS$ and several $NAC$s) for each rule.

Example 1. We introduce the essence of this approach by an example listing the view generated for the $LHS$ and $NAC$ graph of the ClassR rule (see Fig. 2).

```sql
CREATE VIEW classR_lhs AS
SELECT c.id AS c, eo1.id AS eo1, p.id AS p, r1.id AS r1, s.id AS s
FROM Class AS c, EO AS eo1, Package AS p, Ref AS r1, Schema AS s
WHERE p.id=eo1.src AND c.id=eo1.trg AND -- for edge eo1
    p.id=r1.src AND s.id=r1.trg; -- for reference edge r1
```
Fig. 2. Model transformation rules for the object-relational mapping
CREATE VIEW classR_nac AS
SELECT c.id AS c, refn.id AS refn, tn.id AS tn
FROM Class AS c, Ref AS refn, Table AS tn
WHERE c.id=refn.src AND tn.id=refn.trg -- for reference edge refn

A row in the result view should correspond to a successful matching of the graph pattern.

We can make the following observations:

– There are as many columns in the result view as the number of graph objects (i.e. nodes and edges) in the corresponding rule graph (which means five columns in Example 1 including c.id AS c).
– The type of each graph object (i.e. each column) in the result view corresponds to a specific database table (see e.g. Class AS c).
– Valid rows in the result view should be source and target preserving for all edges in the rule graph. For instance, condition p.id=eo1.src AND c.id=eo1.trg expresses that the source node of eo1 is p and the target node of eo1 is c.

The general structure of a query for a rule graph has the following syntax.\(^1\)

CREATE VIEW graph.name AS
SELECT go\(_1\).id AS go\(_1\), ..., go\(_n\).id AS go\(_n\)
FROM go\(_1\).type AS go\(_1\), ..., go\(_n\).type AS go\(_n\)
WHERE edge_constraints AND
  injectivity_constraints AND
  attribute_conditions

The filtering condition is constituted of three different types of constraints.

– **Edge constraints** express the adjacency of nodes and edges (as discussed above). For each edge, we add a subformula \(n_1.id=e.src\) AND \(n_2.id=e.trg\) where \(n_1\) is the source node and \(n_2\) is the target node of edge \(e\) (in the rule graph).
– **Injectivity constraints** are defined for all pairs of LHS graph objects of the same type (or, more precisely, that have common supertypes). For each pair \(go_1\) and \(go_2\), we add a subformula of the form \(go_1.id<>go_2.id\). In fact, this constraint can be easily weakened to follow the identification condition of the double pushout approach [3].
– An **attribute condition** restricts the values of attribute \(attr\) stored at a graph object \(go\) and it is a subformula of the form \(go.attr \theta\ value\) where \(\theta\) is a relational operator.

\(^1\) The disturbingly overloaded use of \(go_i\) is only an SQL hack, basically \(go_i\) always corresponds to one graph object in the rule graph.
**Left joins for preconditions of rules.** When the view for the \textit{PRE} graph is generated, views of all its positive and negative application conditions are available. If the \textit{PRE} graph does not have any negative application conditions then the view defined for its \textit{LHS} graph can be used directly. If the \textit{PRE} graph has at least one \textit{NAC} graph, the corresponding view definition has the following syntax:

```
CREATE VIEW rule.name AS
SELECT lhs.name.*
FROM lhs
LEFT JOIN nac1 ON lhs.c1 = nac1.c1 AND ... AND lhs.cn = nac1.cn
...
LEFT JOIN nack ON lhs.c1 = nack.c1 AND ... AND lhs.cn = nack.cn
WHERE
nac1.c1 IS NULL AND ... AND nac1.cn IS NULL AND
...
nack.c1 IS NULL AND ... AND nack.cn IS NULL
```

Informally, each \textit{NAC} is left outer joined to the \textit{LHS} graph one by one. The morphism between the \textit{LHS} and a \textit{NAC} graph (in other terms, the shared graph objects) is translated into a join condition of type \(lhs.c_i = nac_j.c_i\) (where \(c_i\) refers to the same graph object). Furthermore, for a successful matching we require that the corresponding columns of \textit{NAC}(s) are filled with \textit{null} values. This means that there are no possible extensions of a matching of the \textit{LHS} that is also a matching of (any) \textit{NAC}.

**Example 2.** To continue our running example, we present the view definition for the \textit{PRE} graph of the \textit{ClassR} rule.

```
CREATE VIEW classR AS
SELECT lhs.*
FROM classR_lhs AS lhs LEFT JOIN classR_nac AS nac ON lhs.c=nac.c
WHERE nac.c IS NULL
```

Finally, all successful matchings of a rule can be enumerated as \textbf{SELECT * FROM rule.view}, where a single matching is a row in the corresponding view.

### 3.3 Graph manipulation in relational databases

We propose that operations in the graph manipulation phase can be implemented by issuing several data manipulation commands (\texttt{INSERT}, \texttt{DELETE}, and \texttt{UPDATE}) in a single transaction block. The transaction block is needed to ensure that a graph transformation step is atomic, i.e., either all commands or none of them are executed to result in a consistent model after rule application.

**Deletions.** Let \(go\) be a graph object in \textit{LHS}\textbackslash\textit{RHS} which prescribes the deletion of the successfully matched model element \(me\). The removal of \(me\) is implemented with a delete command of the form:

```
DELETE FROM go.type WHERE go.id = me.
```
As a single model element may appear in different tables (according to the inheritance hierarchy), a `DELETE` command should be executed on each supertype of `go.type`. Fortunately, by using foreign key constraints of the DBMS, it is sufficient to remove an element from root table (i.e. the table representing a common root in the type hierarchy). Therefore, the real delete command is

\[
\text{DELETE FROM root WHERE root.id = me.}
\]

If the deletion of a node `go` is prescribed by a rule then all dangling edges (i.e. all incident edges) should be removed as well. In this case operations like `DELETE FROM t WHERE t.src = me.id OR t.trg = me.id` have to be executed on any table `t` that corresponds to an edge type. However, this deletion is obtained automatically by using the previous foreign key constructs.

**Insertions.** Let `go` be a graph object in `RHS\LHS` which prescribes the creation of a model element `me`. The creation of `me` is implemented by `INSERT` statements in the following way.

- If `go` is a node, then we execute a sequence of `INSERTS` of the form
  
  \[
  \text{INSERT INTO go.type}_i (id) VALUES (me.id) \quad \text{where each type } type_i \text{ is processed in a top-down way according to the inheritance hierarchy starting from the root table (to fulfill the restrictions imposed by foreign keys).}
  \]

- If `go` is an edge, then a series of `INSERTS` of the form
  
  \[
  \text{INSERT INTO go.type}_i (id,src,trg) VALUES (me.id,me.src,me.trg) \quad \text{is executed where each type } type_i \text{ is processed in a top-down way according to the inheritance hierarchy starting from the root table.}
  \]

**Attribute updates.** Let `go` be a graph object in `RHS` mapped to the model element `me` with an attribute assignment `attr := val` which prescribes the update of attribute `attr` to contain the value `val`. Attribute updates are implemented with SQL statements of the form:

\[
\text{UPDATE go.type}_x \text{ SET attr = val WHERE go.id = me.id.}
\]

**Example 3.** We continue our sample graph transformation rule `ClassR` with the model manipulation parts. Note that any text with a postfix `.newid` denotes the identifier of the object that is added to the model. Postfixes of the form `.id` denote values that originate from the pattern matching phase.

\[
\begin{align*}
\text{INSERT INTO root (id) VALUES (t.newid)} \\
\text{INSERT INTO Table (id) VALUES (t.newid)} \\
\text{INSERT INTO root (id) VALUES (tid.newid)} \\
\text{INSERT INTO Column (id) VALUES (tid.newid)} \\
\text{INSERT INTO root (id) VALUES (tpk.newid)} \\
\text{INSERT INTO PrimaryKey (id) VALUES (tpk.newid)} \\
\text{INSERT INTO root (id) VALUES (eo2.newid)} \\
\text{INSERT INTO EO (id,src,trg) VALUES (eo2.newid,s.id,t.newid)} \\
\text{INSERT INTO root (id) VALUES (eo3.newid)} \\
\text{INSERT INTO EO (id,src,trg) VALUES (eo3.newid,t.newid,tpk.newid)} \\
\text{INSERT INTO root (id) VALUES (eo4.newid)} \\
\text{INSERT INTO EO (id,src,trg) VALUES (eo4.newid,t.newid,tid.newid)}
\end{align*}
\]
4 Experimental results

In order to assess the performance of our graph transformation engine, tests were performed on a 300 MHz Pentium machine with 64 MB RAM. We used PostgreSQL as the underlying relational database. No additional optimization techniques were applied in our engine, so all optimization features were provided by PostgreSQL by default.

All the execution times appearing in subsequent tables are measured in milliseconds. A value in a cell shows the time needed to execute a pattern matching or manipulation phase, but it is always an average value calculated from several runs.

**Practical example. (Obj2DB)** We have already introduced the problem of transforming an object model into a relational database schema (see Section 3.1). This problem domain is not a full-fledged case study, but formulates a practical problem to have some initial experiments. Its main characteristics can be sketched as (i) having six graph transformation rules (as depicted in Fig. 2), (ii) each of which consisting of at least 1 and at most 19 graph objects in the LHS graph, (iii) The number of graph objects in RHS graphs is between 3 and 24, (iv) Each rule contains a small NAC graph, (v) Each LHS graph is connected, (vi) The length of transformation sequences is limited by the number of graph objects appearing in the source part of the initial model.

Small examples showed that our approach has an acceptable runtime on (at least) one typical practical problem domain (see the upper (Obj2DB) part of Table 1). (Rule and model size denote the number of nodes and edges appearing in the LHS and in the instance model, respectively.)

**Large patterns. (Comb)** A special case study has been prepared for measuring pattern matching execution times when the LHS graph is complex (with respect to patterns used in typical software engineering applications). In this example, we had a grid structure of size $N \times N$. Nodes are connected by horizontal and vertical directed edges (see Fig. 3).

The pattern has a comb-like structure, which means that it has $2M$ nodes organized into two rows and $M$ columns. The order of magnitude of parameters $N$ and $M$ was about $10^1$ and $10^2$.

There were two different kinds of instance graphs used for our tests. One type contains many matchings and the other contains none.

The middle (Comb) part of Figure 1 summarizes our experiments. Values in rows marked by 1st match denote execution times that were needed to calculate
the first matching (or to point out that no matching exists). Execution times needed for the calculation of any subsequent matchings were of the same order of magnitude, so in those rows only average execution times are presented for a single pattern matching phase.

Our observations of the case study are the following. (i) The time to find the first matching is quite long, while subsequent matchings are generated in a few milliseconds. So it can be stated that all the matchings are available at the time when the first matching has been calculated. (ii) The response time becomes about 3 or 4 times smaller, if there are no matching patterns.

This case study provided the first comparison of AGG and our engine. When comparing the times needed for the calculation of additional matchings, we learned that our engine gave significantly better results. Since our approach

---

**Table 1.** Experimental results

<table>
<thead>
<tr>
<th>Obj2DB</th>
<th>Pattern Model</th>
<th>TS DB</th>
<th>DB DB</th>
<th>Obj2DB size</th>
<th>size length</th>
<th>DB match msec</th>
<th>DB update msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>26</td>
<td>&lt; 200</td>
<td>13</td>
<td>45</td>
<td>59</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comb</th>
<th>Grid size Comb width</th>
<th>Pattern Model</th>
<th>TS DB</th>
<th>DB DB</th>
<th>Obj2DB size</th>
<th>size length</th>
<th>AGG match msec</th>
<th>AGG update msec</th>
<th>DB match msec</th>
<th>DB update msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st match</td>
<td>100 10</td>
<td>39 29900 9009</td>
<td>47114</td>
<td>116183</td>
<td>67706</td>
<td>17950</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next match</td>
<td>100 10</td>
<td>39 29900 9009</td>
<td>304</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1st match</td>
<td>79 29800 8019</td>
<td>exceeded the limits of manual editing of rules</td>
<td>125297</td>
<td>37902</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next match</td>
<td>79 29800 8019</td>
<td>and we ran into problems with automatic rule generation</td>
<td>17106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>first match</td>
<td>50 50</td>
<td>exceeded the limits of manual editing of rules</td>
<td>567610</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutex (short TS)</th>
<th>Proc. Has NAC size Pattern Model</th>
<th>TS DB</th>
<th>AGG match msec</th>
<th>AGG update msec</th>
<th>DB match msec</th>
<th>DB update msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>mount 5 - 1</td>
<td>17 24</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>req 5 * 5</td>
<td>17 24</td>
<td>165</td>
<td>38</td>
<td>28</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>give 5 - 5</td>
<td>17 24</td>
<td>35</td>
<td>15</td>
<td>15</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>mount 100 - 1</td>
<td>302 499</td>
<td>12</td>
<td>11</td>
<td>5</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>req 100 * 5</td>
<td>302 499</td>
<td>120</td>
<td>19</td>
<td>27</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>give 100 - 5</td>
<td>302 499</td>
<td>32</td>
<td>66</td>
<td>13</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>req 100 - 5</td>
<td>302 499</td>
<td>476</td>
<td>238</td>
<td>266</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>large model 1000 - 1</td>
<td>3002 4999</td>
<td>668</td>
<td>425</td>
<td>264</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutex (long TS)</th>
<th>Proc Cycles Iteration Model</th>
<th>TS DB</th>
<th>AGG match msec</th>
<th>AGG update msec</th>
<th>DB match msec</th>
<th>DB update msec</th>
</tr>
</thead>
<tbody>
<tr>
<td>small model 4 100 2/100</td>
<td>21 2500</td>
<td>34</td>
<td>18</td>
<td>17</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>large model 100 1 1/11</td>
<td>5001 80001</td>
<td>956</td>
<td>977</td>
<td>565</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Fig. 3. Tests on large patterns

calculates all matchings together, only a simple loop is needed to traverse the result set. In contrast, AGG uses backtracking, which necessitates further calculations for the next matching. However, we surprisingly experienced that our approach was six times faster in the case when no matchings exist. We found no explanation for this experience.

Tool comparison. (Mutex) The last set of tests provides a comparison between AGG and our prototype graph transformation engine. The problem domain was a distributed mutual exclusion algorithm, where processes try to access shared resources. The full specification can be found in [7]. The metamodel (type graph) contains two nodes and six edges. The whole algorithm can be described with 13 simple graph transformation rules. (The most complex rule has 4 nodes and 3 edges.)

The initial instance graph in tests for short transformation sequences only contained two process nodes and two edges linking the process nodes in both directions. Let \( N \) denote the maximum number of processes appearing in the instance model during a specific test. The transformation sequence in itself consisted of the execution of \( 5N - 1 \) graph transformation rules. The largest instance graph that appears during the rule application phase has \( N+1 \) nodes and \( 2N+1 \) edges. \( N \) was chosen to 5, 100, and 1000 in our different experiments resulting in models of size 17, 302, and 3002, respectively.

For the tests of long transformation sequences we modified two rules (namely \( \text{req} \) and \( \text{rel} \)) in order to restrict their applicability in certain situations and to get a deterministic transformation sequence. The initial model in this case consisted of \( 2N \) nodes (\( N \) processes and \( N \) resources) and \( 2N \) edges. \( 6N+1 \) rules were collected into a basic execution unit that was executed \( R \) times in our experiments. This basic execution unit contained all the rules that did not modify the number of processes and resources. During the execution of a basic unit the instance graph had exactly \( 2N \) nodes and at most \( 3N+1 \) edges. Our small model could be characterized by choosing \( N \) to 4 and \( R \) to 100, respectively, resulting in a model of size 21 and a transformation sequence of length 2500. In
order to get a large model (of size 5001) and a long transformation sequence (of length 60001) we chose parameters $N$ and $R$ to 1000 and to 1, respectively.

Our test results are presented in the bottom (Mutex) part of Fig. 1.

As a summary, our experiments show that

- the length of the updating phase in AGG is monotonously increasing for small models, while our engine updates tables in constant time, and
- for large models and long transformation sequences our engine is twice faster in the pattern matching phase, and significantly faster in the updating phase.

Further experiments. A conceptually similar approach has been implemented in the CheckVML tool, which aims at verifying graph transformation systems by compiling them into off-the-shelf model checking tools. CheckVML experiments (reported in [17]) also demonstrated the technical feasibility of a relational database implementation. In fact, the performance of the translator has been significantly increased in contrast to the previous, pure Java implementation.

5 Conclusion

In the paper, we proposed a new graph transformation engine based on off-the-shelf relational databases. After sketching the main concepts of our approach, we carried out several benchmarks to evaluate our prototype implementation by testing its practical limits and comparing it to the transformation engine of the AGG tool [6].

The main conclusion that can be drawn from our experiments is that relational databases provide a promising candidate as an implementation framework for graph transformation engines. However, since the comparison with other tools that we managed to carry out is still somewhat limited, further experiments are planned to place the performance of our approach in the overall “rankings” of graph transformation engines.

Finally, our approach enables to integrate graph transformation as a specification paradigm into traditional RDBMSs, which is a crucial area for further research and development in our view.

References