Graph Transformation in Relational Databases

Gergely Varró [1] Katalin Friedl [2]

Department of Computer Science and Information Theory

Dániel Varró [3]

Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

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 create database views for each rules and to handle pattern matching by inner join operations while negative application conditions by left outer join operations. 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).

Key words: Tool support, Graph transformation, Pattern matching, Relational databases

1 Introduction

Relational database management systems (RDBMSs) that serve as the storage medium for business critical data for 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 [4] 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.

[1] Email: gervarro@cs.bme.hu
[2] Email: friedl@cs.bme.hu
[3] Email: varro@mit.bme.hu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- Andries and Engels propose in [1] a hybrid (visual and textual) query language based upon graph transformation.
- In [9], Jahnke and Zündorf propose the use of triple graph grammars [16] for database re-engineering of legacy systems in their Varlet framework.
- GRAS [10] is a graph-oriented database management system developed at the University of Aachen, which served as the underlying database for the PROGRES [18] graph transformation tool. A recent version of the GRAS database (namely GRAS/GXL [2]) aims to define an interface that provides access to RDBMSs for graph based tools (e.g., PROGRES).

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.

In the current paper, we follow this second direction. More precisely, we report on the development of a graph transformation engine, which uses an open, off-the-shelf relational database (namely, PostgreSQL [12]) as a backend, and it provides an interface to existing tools that serve as frontends in the architecture.

The essence of the approach is to create database views for each rules and to handle graph pattern matching by inner join operations while negative application conditions by left outer join operations 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).

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 two popular transformation engines (AGG [5] and PROGRES [18]), we claim that such an implementation is a promising alternative.

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 both AGG and PROGRES 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 binary connections between classes (edge types between node types).

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.

Example 2.1 A distributed mutual exclusion algorithm whose full specification can be found in [8] will serve as a running example throughout the paper. *Processes* try to access shared *resources* in this domain. One requirement from the algorithm is to allow access to each resource by at most one process at a time. This is achieved by using a token ring, which consists of processes connected by edges of type *next*. In the consecutive phases of the algorithm, (i) a process may issue a *request* on a resource, (ii) the resource may eventually be *held by* a process and finally a process may *release* the resource. The right to access a resource is modeled by a *token*. The algorithm also contains a deadlock detection procedure, which has to track the processes that are *blocked*.

The metamodel (type graph) of the problem domain and two instance models are depicted in the left and right parts of Fig. 1, respectively.

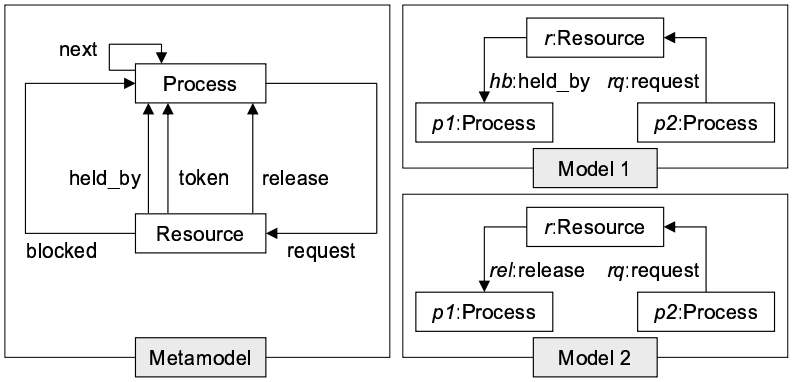

Fig. 1. Metamodel and sample instance models for the problem domain

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* $r = (LHS, RHS, NAC)$ contains a left–hand side graph LHS, a right–hand side graph RHS, 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.

Example 2.2 The distributed mutual exclusion algorithm can be described with 13 simple graph transformation rules. (The most complex rule has 4 nodes and 3 edges.) A sample transformation rule describing how to release a resource is presented in Fig. 2.

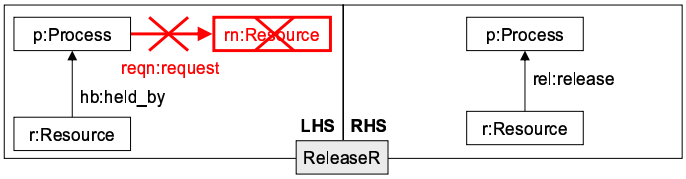

Fig. 2. A sample transformation rule (ReleaseR)

This rule is applicable if there is a resource that is held by a process, which does not have any request issued on any resources. Model 1 of Fig. 1 presents a sample situation where this rule is applicable, since resource r is held by process p1 and p1 does not have any other relationship with resources.

In this specific case, rule application means that the selected resource is to be released by the process, which results in an instance model (Model 2) presented in the lower right part of Fig. 1.

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) (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.

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. Pattern matching techniques of existing graph transformation tools can be grouped into two main categories. For further comparison of graph transformation approaches see [17].

- Algorithms based on *constraint satisfaction* (such as [11] in AGG [5], VIATRA [20]) 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 sophisticated search plans [21]), Dörr's approach [3], and the object-oriented solution in FUJABA [13] fall in this category.

Our experiments in Sec. 4 will show that algorithms based on constraint satisfaction have better performance in general, if interpreted graph transformation engines are under observation. However, it is obvious that a compiled approach gives better results than an interpreted engine. The comparison of constraint satisfaction and local search based algorithms in case of compiled engines is to be performed in the future.

3 Graph transformation in relational databases

We present how a graph transformation engine (following the single pushout [15] approach with injective matchings) 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. 3.3–3.4), finally data manipulation is handled (in Sec. 3.5).

Due to space restrictions, we assume the reader's familiarity with elementary concepts of relational databases concepts. These issues are presented, e.g., in [19].

3.1 Mapping models and metamodels to database tables

Instance models representing the system under design are stored in database tables. We used a standard mapping (for more details see [19, 14]) to generate the schema of the database from the metamodel.

- Each class is mapped to a table with a single column. This column will store the identifiers of objects of the specific class.
- We assign a table for each association that appears in the metamodel. This table has three columns that contain the identifiers of links, source nodes and target nodes, respectively. Foreign keys should additionally be defined for the last two columns. These keys refer to identifier columns in source and target node tables, respectively.
- If a subclass is inherited from a superclass, then table that corresponds to the subclass should be extended by a foreign key constraint that links primary key columns of the two tables. This means that all identifiers appearing in the subclass table should also appear in the superclass table as well.

Example 3.1 The database representation of the instance model Model 1 is depicted in the upper part of Fig. 3. The meaning of the lower part of Fig. 3 will be discussed later.

3.2 Inner joins and left outer joins: An overview

We give a short overview on the most crucial concepts of RDBMSs that we build on in the sequel, namely, views, inner joins and left outer joins. The formal treatment of these concepts can be found in [19].

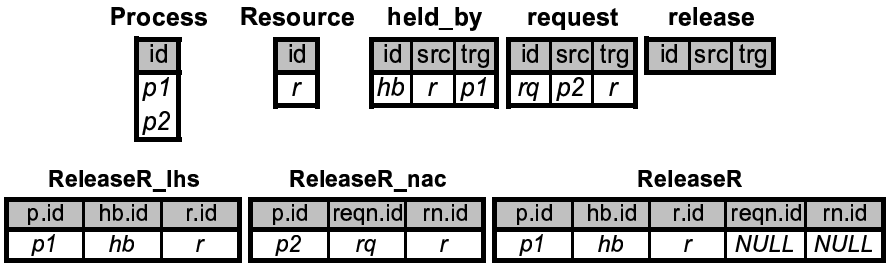

Fig. 3. Database representation

The *inner join of tables R and S* (denoted by $R \overset{F}{\bowtie} S$) 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 filtering formula F is the equality relation in the WHERE condition in this case.

The *left outer join of R and S (denoted by $R \overset{F}{\ltimes} S$)* (i) contains all the rows of $R \overset{F}{\bowtie} S$, and (ii) additionally it 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. 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.

3.3 Views for rule graphs (LHS and 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 NACs) for each rule.

Example 3.2 We introduce the essence of this approach by an example listing the view generated for the LHS and NAC graph of the ReleaseR rule (see Fig. 2).

```
CREATE VIEW ReleaseR_lhs AS              -- an LHS view
SELECT p.id AS p, hb.id AS hb, r.id AS r -- with 3 columns
FROM Process AS p, Held_by AS hb, Resource AS r
WHERE r.id=hb.src AND p.id=hb.trg      -- for held_by edge hb

CREATE VIEW ReleaseR_nac AS
SELECT p.id AS p, reqn.id AS reqn, rn.id AS rn
FROM Process AS p, Request AS reqn, Resource AS rn
WHERE p.id=reqn.src AND rn.id=reqn.trg -- for request edge reqn
```

We can make some observations related to the structure and content of the result view. (i) The view contains as many columns as the number of graph objects (i.e. nodes and edges) appearing in the corresponding rule graph (which means three

columns in Example 3.2 including `p.id AS p`). (ii) The type of each graph object (i.e. each column) corresponds to a specific database table (see e.g. `Process AS p`). (iii) Valid rows should be source and target preserving for all edges in the rule graph. For instance, condition `r.id=hb.src AND p.id=hb.trg` expresses that the source node of `hb` is `r` and the target node of `hb` is `p`. (iv) A row should correspond to a successful matching of the graph pattern.

The general structure of a query for a rule graph has the following syntax. [4]

```
CREATE VIEW graph.name AS
SELECT go1.id AS go1, ..., gon.id AS gon
FROM go1.type AS go1, ..., gon.type AS gon
WHERE edge_constraints AND  injectivity_constraints
```

Edge constraints express the adjacency of nodes and edges. 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`.

3.4 Left joins for preconditions of rules.

When the view for the PRE graph is generated, views of all its positive and negative application conditions are available. If the PRE graph does not have any negative application conditions then the view defined for its LHS graph can be used directly. If the PRE graph has at least one 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 NAC is left outer joined to the LHS graph one by one. The morphism between the LHS and a 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 related graph object). Furthermore, for a successful matching we require that the corresponding columns of NAC(s) are filled with NULL values. This means that there are no possible extensions of a matching of the LHS that is also a matching of (any) NAC graph.

[4] 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.

Example 3.3 To continue our running example, we present the view definition for the PRE graph of the ReleaseR rule.

```
CREATE VIEW ReleaseR AS
SELECT lhs.*
FROM ReleaseR_lhs AS lhs LEFT JOIN ReleaseR_nac AS nac
ON lhs.p=nac.p
WHERE nac.p IS NULL
```

The lower part of Fig. 3 shows the contents of views that have been defined for the LHS, the NAC and the PRE part of rule ReleaseR, respectively.

Finally, all successful matchings of a rule can be enumerated as `SELECT * FROM rule.view`, where a single matching is a row in the corresponding view. Storing all the matches of a rule can be extremely useful for model transformations where no conflicts occur within (a well-designed set of) graph transformation rules, thus the rule can be applied in parallel to independent matches. However, for our experiments (in Sec. 4), we did not use this possibility in order to compare the real efficiency of use of the relational databases.

3.5 Graph manipulation in relational databases

We propose that operations in the graph manipulation phase can be implemented by issuing several data manipulation commands (INSERT and DELETE) 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. If go is a graph object in $LHS \backslash RHS$ prescribing the deletion of the successfully matched model element me then the removal of me is implemented with a DELETE command: "`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 "`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 of the form "`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. If go is a graph object in $RHS \backslash LHS$ prescribing the creation of a model element me, then 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 "`INSERT INTO` $go.type_i$ `(id) VALUES (`$me.id$`)`" where each type $type_i$ is pro-

cessed 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 "`INSERT INTO` $go.type_i$ `(id,src,trg) VALUES (`$me.id$`,`$me.src$`,`$me.trg$`)`" is executed where each type $type_i$ is processed again in a top-down way according to the inheritance hierarchy starting from the root table.

Example 3.4 We continue our sample graph transformation rule ReleaseR 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.

```
DELETE FROM Held_by WHERE id = hb.id
DELETE FROM root WHERE id = hb.id
INSERT INTO root (id) VALUES (rel.newid)
INSERT INTO Release (id,src,trg) VALUES (rel.newid,r.id,p.id)
```

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. A Linux kernel of version 2.4.18 served PostgreSQL that we used as the underlying relational database. No additional optimization techniques were applied in our engine, so all optimization features were provided by PostgreSQL by default.

During the execution of tests on AGG, we switched off the GUI, so rule applications were guided by a JAVA program. In contrast, we used the standard interpreter with the underlying GRAS database as a running environment for the PROGRES tests and in addition, the Prolog-style cuts in the specification to make the execution deterministic. This way, we were doing programmed graph rewriting in each case for batch transformations. Furthermore, we threw away the generated DB views after each step to obtain a worst-case performance assessment for our transformation engine.

Figure 4 shows the execution times of the three test sets (having different characteristics) carried out on our mutual exclusion example. Values in avg columns are average times needed for a single rule application, while sum columns denote the execution time of the whole transformation sequence.

Short transformation sequences. Initial instance graphs in this test set 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.

Mutex (short TS)	Proc. #	Model size #	TS length #	AGG avg msec	AGG sum msec	Progres avg msec	Progres sum msec	DB avg msec	DB sum msec
small	5	17	24	105	2512	125	3000	48	1150
medium	100	302	499	110	55047	1042	520000	35	17459
large	1000	3002	4999	409	timeout	timeout	timeout	140	700419
Mutex* (long TS)	**Proc. #**	**Model size #**	**TS length #**	**AGG avg msec**	**AGG sum msec**	**Progres avg msec**	**Progres sum msec**	**DB avg msec**	**DB sum msec**
small	4	21	2500	145	362811	97	242000	34	84951
large	1000	5001	60001	1952	timeout	920	timeout	257	1544621
Mutex' (for all)	**Proc. #**	**Model size #**	**TS length #**	**AGG avg msec**	**AGG sum msec**	**Progres avg msec**	**Progres sum msec**	**DB avg msec**	**DB sum msec**
	10	50	40	78	3111	100	4000	18	723
	30	150	120	74	8926	225	27000	33	3909
	50	250	200	83	16680	345	69000	37	7332
	100	500	400	128	51047	657	263000	38	15000
	200	1000	800	315	251706	1294	1035000	51	40581

Fig. 4. Experimental results

Long transformation sequences. For this test set, we modified two rules (namely, *req* and *rel* of [8]) in order to restrict their applicability in certain situations and to get a deterministic transformation sequence. The initial model 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 several 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.

Few matches on large models. The third test sequence consisted of $4N$ rule applications that were organized into four blocks. One such block corresponded to a specific rule that could be applied concurrently on N different processes. Each rule application (i) disabled the execution of the same rule on the same process, (ii) it left unchanged the enabledness of the same rule on other processes, and finally, (iii) it enabled the execution of the following rule on the same process. This test sequence produced models of size $5N$, which were 50, 150, 250, 500, and 1000 in the concrete runs. (Rule *req* had to be slightly modified again to ensure the appropriate behavior.)

Naturally, we carried out additional test cases (on different examples) to compare these tools which were not presented in the current paper due to space considerations. Our experiments can be summarized as follows.

- PROGRES had good performance in cases, when the number of matches was relatively large compared to the model size. However, if only several matches existed in a large model, then its backtracking strategy caused a heavy decrease in the runtime performance.
- In case of large models, the update strategy of AGG consumes at least as much

time as the pattern matching phase itself which is quite unexpected since the DB engine performed updates (to base tables) in constant time.

- Our graph transformation engine based on a *compilation* to relational databases fulfilled our minimum goal, namely, to significantly outperform *interpreted* approaches like AGG or PROGRES. However, for a real assessment, we need to compare our DB approach to other compiled graph transformation engines such as Fujaba [13] which is still to be done.

Although all our examples consisted of only a relatively small number of rules (less than 20), we believe the performance of our database approach is not drastically decreased in case of typical software engineering applications (e.g. hundreds of rather small rules) since the calculation of a view requires to join only few tables at a time and it is independent of the number of rules.

5 Conclusion and Future Work

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 test cases to evaluate our prototype implementation by comparing it to the transformation engines of the AGG [5] and PROGRES [18] tools.

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. We call attention to the fact that our promising experimental results were obtained using a worst-case assessment method i.e. by recalculating the views of the next rule to be applied from scratch which is still highly inefficient, especially, for model transformations with a large number of independent matches of the same rule.

Further optimizations are required if we aim at incremental transformations in the future. Despite the fact that incremental updating techniques are subject to research in many fields (e.g. database view recalculation [7], expert systems [6]), there are still only a few RDBMSs that implement incremental view updating even with strong restrictions. PostgreSQL does not support this feature at all, which was the main reason for recalculating the views from scratch in each step.

References

[1] Andries, M. and G. Engels, *A hybrid query language for the extended entity relationship model*, Journal of Visual Languages and Computing **8** (1997).

[2] Böhlen, B., *Specific graph models and their mappings to a common model*, in: *Proc of the 2nd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, LNCS **3062** (2003), pp. 45–60.

[3] Dörr, H., "Efficient Graph Rewriting and Its Implementation," LNCS **922**, Springer-Verlag, 1995.

[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] Ermel, C., M. Rudolf and G. Taentzer, "In [4]," World Scientific, 1999 pp. 551–603.

[6] Forgy, C. L., *RETE: A fast algorithm for the many pattern/many object match problem*, Artificial Intelligence **19** (1982), pp. 17–37.

[7] Gupta, A. and I. S. Mumick, editors, "Materialized Views: Techniques, Implementations, and Applications," MIT Press, 1999.

[8] Heckel, R., *Compositional verification of reactive systems specified by graph transformation*, in: E. Astesiano, editor, *Fundamental Approaches to Software Engineering: First International Conference, FASE*, LNCS **1382** (1998), pp. 138–153.

[9] Jahnke, J. H., W. Schäfer, J. P. Wadsack and A. Zündorf, *Supporting iterations in exploratory database reengineering processes*, Science of Computer Programming **45** (2002), pp. 99–136.

[10] Kiesel, N., A. Schürr and B. Westfechtel, *GRAS, a graph-oriented database system for (software) engineering applications*, in: J. Lee, Reid, editor, *Proc. CASE '93, 6th Int. Conf. on Computer-Aided Software Engineering* (1993), pp. 272–286.

[11] Larrosa, J. and G. Valiente, *Constraint satisfaction algorithms for graph pattern matching*, Mathematical Structures in Computer Science **12** (2002), pp. 403–422.

[12] Momjian, B., "PostgreSQL: Introduction and Concepts," Addison-Wesley, 2000.

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

[14] Ramakrishnan, R. and J. Gehrke, "Database Management Systems," McGraw-Hill, 2002, 3rd edition.

[15] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformations: Volume 1: Foundations," World Scientific, 1997.

[16] Schürr, A., *Specification of graph translators with triple graph grammars*, Technical report, RWTH Aachen, Fachgruppe Informatik, Germany (1994).

[17] Schürr, A., "In [15]," World Scientific, 1997 pp. 479–546.

[18] Schürr, A., A. Winter and A. Zündorf, "In [4]," World Scientific, 1999 .

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

[20] Varró, D., G. Varró and A. Pataricza, *Designing the automatic transformation of visual languages*, Science of Computer Programming **44** (2002), pp. 205–227.

[21] 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.