Available online at www.sciencedirect.com
SCIENCE DIRECT
ELSEVIER
Electronic Notes in Theoretical Computer Science 109 (2004) 71–83
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Graph Transformation with Incremental Updates

Gergely Varró [1]

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

Dániel Varró [2]

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

Abstract

We propose an efficient implementation technique for graph transformation systems based on *incremental updates*. The essence of the technique is to keep track of all possible matchings of graph transformation rules in database tables, and update these tables incrementally to exploit the fact that rules typically perform only local modifications to models.

Keywords: graph transformation, graph pattern matching, relational databases.

1 Introduction

Despite the large variety of existing graph transformation tools, the implementation of their graph transformation engine typically follows the same principle. In this respect, first a matching occurrence of the left-hand side of the graph transformation rule is being found by some sophisticated graph pattern matching algorithm. Then potential negative application conditions

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

doi:10.1016/j.entcs.2004.02.057

are checked that might eliminate the previous occurrence. Finally, the engine performs some local modifications to add or remove graph elements to the matching pattern, and the entire process starts all over again.

Since graph pattern matching leads to the subgraph isomorphism problem that is known to be NP-complete in general, this step is considered to be the most crucial in the overall performance of a graph transformation engine. The diversity of tools is thus mainly characterized by the different strategies used for the graph pattern matching step. These strategies can be grouped into two main categories.

- Algorithms based on *constraint satisfaction* (such as [9] in AGG [5], VIATRA [12]) 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.
- 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 [13]), Dörr's approach [3], and the object-oriented solution in FUJABA [6] fall in this category.

In the current paper, we argue that the efficiency of graph transformation is not necessarily equal to the efficiency of graph pattern matching, especially for long transformation sequences. In fact, any implementation of a graph transformation engine is not optimal, if all the information on previous match is lost when a new transformation step is started. Thus we restart the complex and expensive graph pattern matching phase from scratch each time.

Several solutions already exist for reducing the overhead of finding matches for LHS of rules as implemented in PROGRES [13]: (i) applying a graph transformation to all matches in the graph as one graph rewriting step (pseudo-parallel graph transformation), (ii) using incrementally computed derived attributes and relationships in LHS, and (iii) using rule parameters in graph transformations to pass computed knowledge about possible LHS matches from one rule to the next one.

In the paper, we propose a technique based on *incremental updates* which, in itself, is not a new idea, but provides a new philosophy for implementing efficient graph transformation engines.

After many years of research, different techniques based on this idea have evolved and by now they are widely accepted and successfully used in several types of applications (e.g., expert systems, relational databases).

- In the area of rule-based expert systems, the Rete-algorithm (for more details see [7]) uses the idea of incremental pattern matching for facts. First a

dataflow network is constructed based on the condition (*if*) parts of rules, which is basically a directed acyclic graph of a special structure. Initially, this network is fed by basic facts through its input channels. Compound facts are constituted of more elementary facts, thus they are the inputs of internal nodes in the network. If a fact reaches a terminal node, then the rule related to this specific node becomes applicable and assignments modifying the set of basic facts may be executed (according to the *then* part). Since every node keeps a record of its input facts, only modifications of these facts have to be tracked at each step.

- In the area of relational databases, views may be updated incrementally. A database view is a query on a database that computes a relation whose value is not stored explicitly in the database, but it appears to the users of the database as if it were. However, in a group of methods, which is called by view materialization approach, the view is explicitly maintained as stored relation [8]. Every time a base relation changes, the views that depend on it may need to be re-computed.

The main idea of incremental updates in graph transformation systems is to keep track of all possible matchings of graph transformation rules in database tables to make the graph pattern matching step very fast. Afterwards when a rule is applied we update these tables for all locations it is required. Since graph transformation typically manipulates only a small fragment of the instance model, incremental updates require minor changes to these tables. Naturally, the initialization of the tables needs some considerable amount of pre-processing prior to the transformation, but the subsequent transformation process itself becomes much faster.

In this way, significant speed-up can be expected in complex transformations which consist of long sequences and manipulates on huge instance models. Furthermore, an even more significant gain can be achieved for the parallel execution of independent transformation steps, since each matching is stored explicitly for all the rules. Model transformations between two modeling languages typically have this property as the target model has to be constructed from scratch by applying almost exclusively non-deleting rules.

In the current paper, we discuss our initial experiments in mapping models and metamodels into an off-the-shelf relational database to implement the incremental update technique for the dining philosophers problem. Note that the integration of database and graph transformation techniques has a long tradition (see e.g., [1,13]), but these approaches use graph-oriented databases in contrast to relational ones (as in our case).

2 Mapping models and metamodels to database tables

First we informally discuss a (relatively standard) mapping of models and metamodels into relational database tables.

The *metamodel* 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*. A class may have *attributes* 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* define connections between classes.

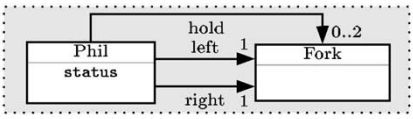
The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* 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.

In our approach, instance models are stored in database tables. A possible way to define the schema of the database can be driven by the metamodel.

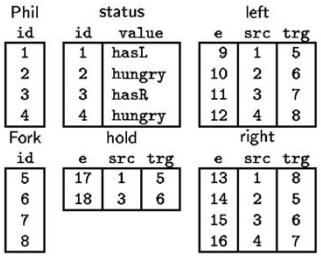
- Each class is mapped to a table with a single column (`class(I)`). This column will store the identifiers of objects of the specific class.
- We assign a table for each association that appears in the metamodel. This table has three columns (`assoc(I,S,T)`), which contain identifiers for the link, and its source and target objects, respectively.
- Each attribute is mapped to a table with two columns (`attr(I,V)`) storing the object identifier and the attribute value, respectively.
- If a subclass is inherited from a superclass, then two tables have to be constructed as if they were two independent classes. However, all identifiers appearing in the subclass table should also appear in the superclass table as well.

Tables that are created by this mapping will be referred to as *base tables*.

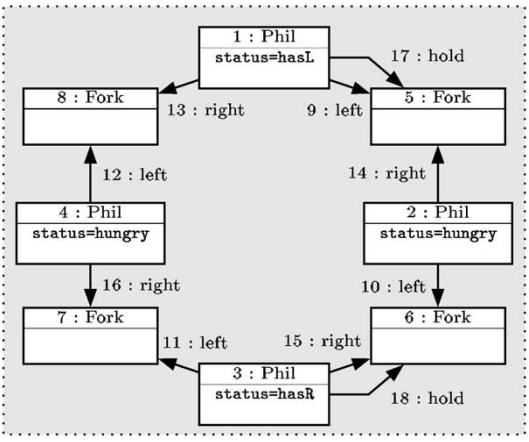
Example 2.1 In order to present our concepts, the dining philosophers problem will be used throughout this paper as a running example. There are *philosophers* sitting around a table, each having a *left* and a *right* fork. *Forks* are placed on the table between two neighboring philosophers, so forks are shared resources. Philosophers may grab their left and right forks and may *hold* them in their hands. Philosophers also have a *status* attribute. This



(a) Metamodel



(b) Database tables



(c) Instance model

Fig. 1. Dining philosophers

modeling domain is depicted in the metamodel of Fig. 1(a).

A well-formed instance model of this domain (shown in Fig. 1(c)) has 4 objects of class Phil (with status attributes initialized to different values) and 4 objects of class Fork. The model additionally has 4 links of type left, 4 links of type right, and 2 links of type hold. Each link leads from an object of type Phil to an object of type Fork. The equivalent database representation of the instance model is depicted in Fig. 1(b).

3 Graph transformation in relational databases

Graph transformation [10,4] provides a pattern and rule based manipulation of graph-based 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 LHS, a right–hand side graph RHS, and negative application condition graph NAC. The LHS and the NAC graphs are together called the precondition PRE of the rule.

The *application* of r to an *host (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.

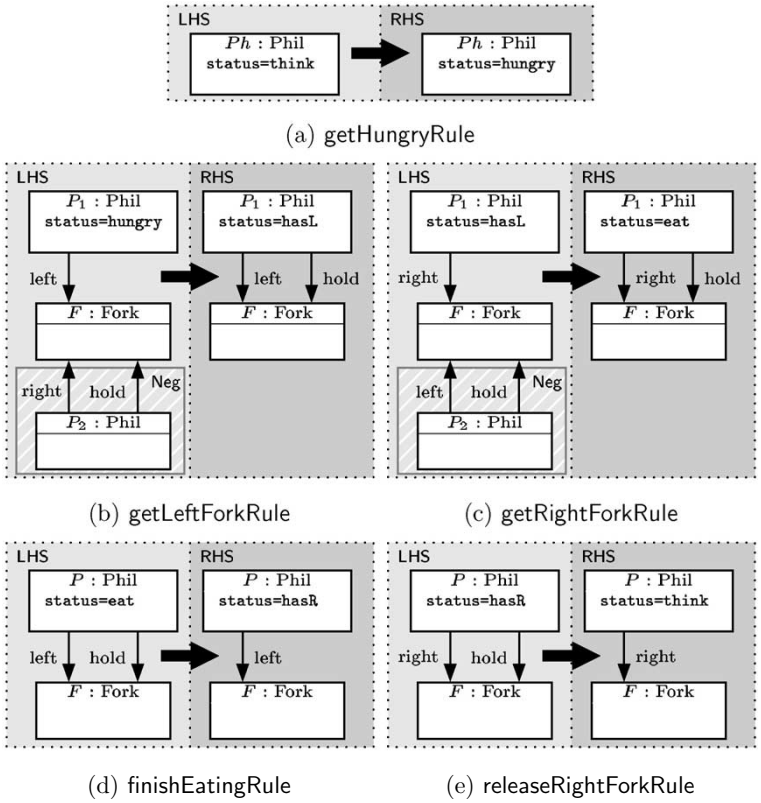


Fig. 2. Graph transformation rules for dining philosophers

Example 3.1 Our running example has five graph transformation rules, of which one (namely the getRightForkRule depicted in Fig. 2(c)) is selected for demonstrating the concepts of graph transformation.

This rule can be applied when a philosopher already holds his left fork in his hands, and now he tries to grab his right fork, if it is not held by another philosopher yet. This latter statement is expressed by the NAC. The RHS of the getRightForkRule shows that the philosopher will grab his right fork as the result of rule application. On rule level, the LHS has two nodes and an edge of type right that leads between these nodes, while the NAC graph (marked by the striped area) implicitly contains node F as well to impose a well-formed graph structure.

3.1 Graph pattern matching in databases

We state that graph pattern matching can be interpreted distinctly for the LHS and NAC graphs as *inner join* operations (denoted by $R \overset{F}{\bowtie} S$) on the corresponding database tables. Afterwards, the precondition (PRE) where the LHS graph is constrained by a NAC can be expressed by *left outer join* operations (denoted by $R \overset{F}{\ltimes} S$).

Inner join operations are basically selections from the Cartesian product $(R \times S)$ using some formula F for filtering. For the paper, only atoms of type $A = B$ (two column names in equality relation) are considered which can be connected by the logical AND operator to construct formulae. *Left outer join* operation contains all the rows of $R \overset{F}{\bowtie} S$, and additionally, it also contains all rows of R for which no row of S exists that satisfies F. Rows of the latter type are filled with NULL values in all columns originating only from S. The formal treatment of inner and left outer joins can be found in [11].

A successful matching of the LHS (or the NAC) graph is a row in a table obtained as the inner join of the corresponding base tables. The joint precondition of the rule is constituted from the left outer join of the previous LHS table and NAC table (or tables). A successful matching of the precondition is a row in the joint PRE table where the columns originating only from the NAC table have NULL values.

This technique allows to map the same objects to LHS nodes, which can be forbidden by the so-called identification condition [2], which can be implemented by additional filtering formulae of type $A \neq B$.

Example 3.2 For demonstration, we define (in Fig. 3) all the potential matchings of getRightForkRule found in the instance model of Fig. 1(c).

The matching of the LHS can be determined by three inner join operations (relating tables Fork, Phil, right and status) followed by a selection with formula `status.value='hasL'` as it is presented in the upper part of Fig. 3.

Submodels that match the NAC pattern can be collected into a table by three inner join operations (relating tables Fork, Phil, left and hold along with the corresponding attributes) as it is presented in the middle part of Fig. 3.

Finally, submodels that match the whole PRE of getRightForkRule (see the lower part of Fig. 3) can be determined by the left outer join of the LHS and the NAC tables. In this case, the columns originating from the NAC table are filled with NULL values in the single result row to show that the row is a successful matching (the fork selected by the LHS is permitted by the NAC).

This construction necessitates to introduce new *auxiliary* tables in addition to base tables. First, we map each NAC of each rule to a new table containing

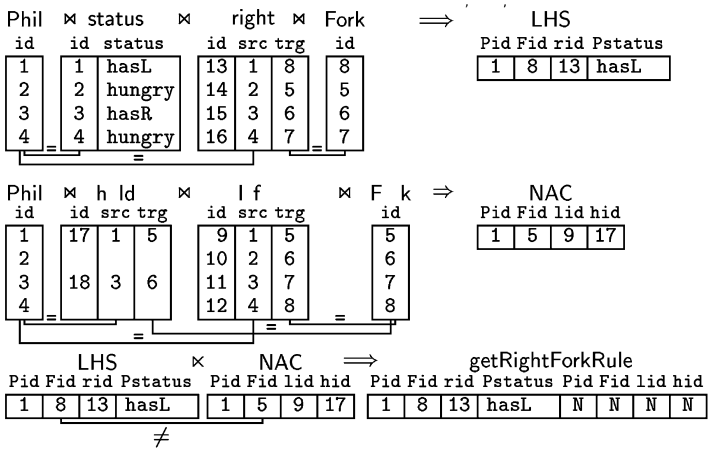


Fig. 3. Pattern matching queries

as many columns as the number of graph elements in the NAC. New auxiliary tables are also created for preconditions having as many columns as the number of graph elements the LHS and the NAC graph have altogether.

Furthermore, in order to easily demonstrate the effects of attribute assignments in incremental updates, we introduce an auxiliary table for each value an attribute may have. For instance, it means five new tables for the status attribute of philosophers (see tables $\mathsf{Phil}_{\mathsf{think}}, \ldots, \mathsf{Phil}_{\mathsf{hasR}}$ in Fig. 4).

These tables can be initialized in a preprocessing phase by applying the previous join operations in order to store all the potential matchings of each graph transformation rule. Since a join operation is a complex task, our next goal is to avoid re-executing it during the graph transformation process by the concept of incremental updates.

3.2 Modifications with incremental updates

After the pattern matching phase, we identified all the submodels of the host model that can be matched to the LHS graph but not to the NAC graph of a rule r, meaning that r is applicable to the host model. Then executing r on the instance model has the following effects.

Deletion If there are graph elements in the LHS that are not present in the RHS, then the images of these elements have to be deleted from the model. Deletion may affect several base tables either because of (i) deleting dangling links (edges) together with the corresponding object, or (ii) as a consequence of inheritance, which means that if an object is deleted from the superclass table, then it also has to be deleted from the subclass tables.

Insertion If there are graph elements that can be found in the RHS, but are not present in the LHS, then new model elements have to be added to the base tables. Note that inheritance may lead to several INSERT operations, when adding a single model element.

Attribute update If the LHS accesses an attribute (in an attribute condition) and the RHS contains assignment for the same attribute, then the corresponding attribute has to be updated.

Since several auxiliary tables were introduced for storing the potential matchings of rules, insert and delete operations should explicitly handle such tables as well. The goal of the *incremental update* technique is to determine how to propagate the effects of modifying a base table to other auxiliary tables.

For this purpose, we introduce the notion of a *dependency graph*. Each (base or auxiliary) table becomes a node in this dependency graph, while the (directed) edges denote the update dependencies between the tables. More specifically, we identify *positive and negative dependencies* between tables.

- In case of a *positive dependency*, an INSERT(DELETE) operation in a source table (defined by the source of the dependency edge) implies one or more INSERT (DELETE) operations in the dependent target table (defined by the target of the dependency edge). In graph transformation terms, this means that the graph defined by the source table is a subgraph of the pattern defined by the target table. A typical example for positive dependency is the dependency between the precondition table of a rule and the base tables constituting the precondition table (such as the base hold table and the auxiliary releaseRightForkRule table).
- A *negative dependency* denotes the handling of negative conditions thus it always leads from a NAC table to a PRE table.
  - When deleting a row from the NAC table then the entire precondition is weakened at the specific location. Therefore the matching of the PRE is not forbidden any more at the corresponding location, thus the corresponding columns in PRE should be changed to NULL to denote that.
  - When adding a row to the NAC table then the entire precondition is strengthened at the specific location. Therefore a corresponding matching of the PRE is forbidden, thus the corresponding columns in PRE should be changed from NULL to the values of the related NEG table.

The dependency graph of a graph transformation system can be defined at compile-time. In fact, for practical applications, we only have to include dependencies of dynamic model elements, i.e., those that can be modified by at least one rule.

Since a formal definition of positive and negative dependencies is out of scope for the current paper due to space limitations, we only give a demonstrative example to capture the essence of incremental updates.

Example 3.3 Figure 4 exemplifies the effects of applying releaseRightForkRule on the instance model of Fig. 1(c) using incremental updates.

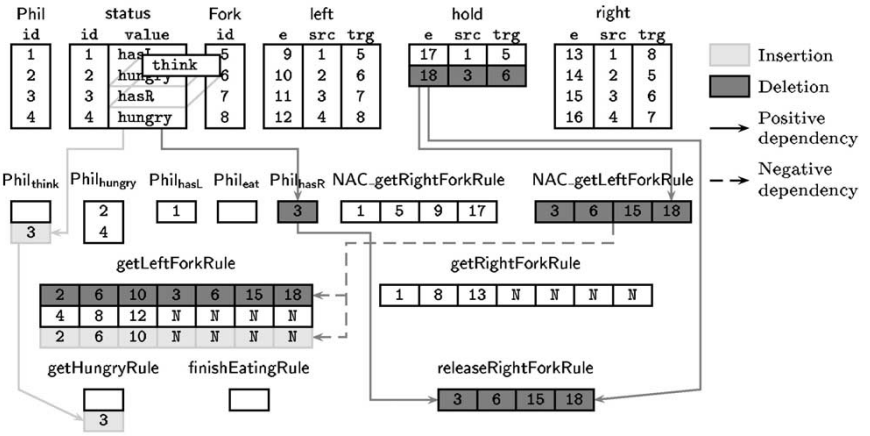


Fig. 4. Applying releaseRightForkRule in incremental mode

ReleaseRightForkRule stores only a single submodel consisting of elements 3, 6, 15 and 18 that matches the precondition. Therefore, the rule is applicable, and it prescribes the update of status attribute for philosopher 3 from hasR to think and the deletion of the hold link 18 leading from philosopher 3 to fork 6.

Since the table status is split along the possible values into auxiliary tables ($\mathsf{Phil}_{\mathsf{think}}, \ldots, \mathsf{Phil}_{\mathsf{hasR}}$), the update operation on the status attribute equals to removing philosopher 3 from $\mathsf{Phil}_{\mathsf{hasR}}$ and adding it to $\mathsf{Phil}_{\mathsf{think}}$.

As releaseRightForkRule is positively dependent on $\mathsf{Phil}_{\mathsf{hasR}}$, all rows containing philosopher 3 should be removed from the releaseRightForkRule table as well. Due to the positive dependency between tables $\mathsf{Phil}_{\mathsf{think}}$ and getHungryRule, a new row containing philosopher 3 is added to the latter table.

Furthermore, tables NAC_getLeftForkRule and releaseRightForkRule are positively dependent on hold, therefore the removal of the hold link 18 implies the deletion of the corresponding rows containing 18 in both tables. As a result, releaseRightForkRule no longer becomes applicable, but the negative condition NAC_getLeftForkRule is also weakened in the meantime.

Finally, as getLeftForkRule is negatively dependent on NAC_getLeftRightForkRule, the row containing the hold link 18 is removed, and a new row on the same matching LHS pattern (i.e., philosopher 2, fork 6 and the left link 10 between them) is added that is filled with NULL values to denote that the

matching is no longer invalidated by the negative condition.

4 Practical evaluation and conclusions

The dining philosophers example has been implemented and tested using a relational database for storing graphs to assess the performance of our incremental update technique. The relational database used for our experiments was MySQL running on a 300 MHz Pentium machine with 64 MB RAM.

An instance model consisting of 10^5 philosophers was set up also containing the same amount of Forks, left, right and hold relations. Philosophers were in different states depending on their positions around the table. Our test consisted of applying each rule once both in *from-scratch (*FS*)* mode (when tables were re-generated after each step thus auxiliary tables were non-existent) and in *incremental (*INC*)* mode. Our observations can be summarized as follows:

(i) Initialization of tables took more time with an overall factor of 1.25 in INC mode compared to the FS approach (81.84 sec in INC mode vs. 65.31 sec in FS mode). This result meets our expectations since we have to initialize the auxiliary tables as well in INC mode.

(ii) Pattern matching without considering negative application conditions is faster in INC mode with a total factor of 7.9 (2.15 sec in INC mode vs. 16.93 sec in FS mode). If negative application conditions are also considered as a part of the pattern matching phase, then the factor significantly (with a factor of 56) increases in favor of the INC method (122 sec in FS mode and 2.15 in INC mode).

(iii) The average cost of manipulations (insert and delete operations) on tables in a single transformation step was 3 times as much in INC mode as in FS mode (0.66 sec vs. 0.23 sec). This is not surprising since the consistency of auxiliary tables has to be guaranteed as well in INC mode thus more tables should be accessed.

As a summary, the overall execution time of the entire transformation process (consisting of 5 rule applications) without the initialization phase was 22 times faster in INC mode with negative conditions and 3.3 times faster without considering negative conditions. Together with the initialization phase, there was still a factor of 2 in the favor of the INC mode.

As the main conclusion of the paper, our initial experiments demonstrated that a graph transformation engine based on incremental updates is extremely efficient when (i) the instance model is large, (ii) all possible matchings of rules should be made available (iii) long transformation sequences are executed, and (iv) many rules contain negative application conditions.

5 Future Work

Our plans for the near future can be outlined in the following directions:

- Unfortunately, MySQL is not a perfect choice as an underlining relational database for graph transformation, since its present version does not support views. The use of relational databases that offer support for defining views may result in more simple queries to be executed on the database level.
- The Rete-algorithm gives an orthogonal solution for incremental updates, since it does not use relational database for data storage. Our plan is to make an implementation of a graph transformation tool, which is only based on Rete-networks.
- Experiments should also be extended, since our initial experiments have only covered a small subset of graph transformation problems. A comparison of short and long rule application sequences, and problems having many graph transformation rules are in our future plans.
- Finally, it is also worth checking whether the incremental approach can be combined with other optimization strategies implemented in existing graph transformation tools.

Acknowledgement

The authors are grateful for the valuable comments of the anonymous reviewers.

References

[1] Andries, M., "Graph Rewrite Systems and Visual Database Languages," Ph.D. thesis, Leiden University, The Netherlands (1996).

[2] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, "In [10]," World Scientific, 1997 pp. 163–245.

[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 of Graph Grammars and Computing by Graph Transformation. Vol. 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] 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).

[7] Forgy, C. L., *RETE: A fast algorithm for the many pattern/many object match problem*, Artificial Intelligence (1982).

[8] Gupta, A. and I. S. Mumick, *Maintenance of materialized views: Problems, techniques and applications*, IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing (1995).

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

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

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

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

[13] 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.