Electronic Communications of the EASST
Volume 4 (2006)

Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)

Incremental Graph Pattern Matching: Data Structures and Initial Experiments

Gergely Varró, Dániel Varró, Andy Schürr

15 pages

Guest Editors: Gabor Karsai, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Incremental Graph Pattern Matching: Data Structures and Initial Experiments

Gergely Varró[1], Dániel Varró[2], Andy Schürr[3]

[1] gervarro@cs.bme.hu,
Department of Computer Science and Information Theory
Budapest University of Technology and Economics, Hungary

[2] varro@mit.bme.hu,
Department of Measurement and Information Systems
Budapest University of Technology and Economics, Hungary

[3] andy.schuerr@es.tu-darmstadt.de,
Real-Time Systems Lab
Technical University of Darmstadt, Germany

Abstract: Despite the large variety of existing graph transformation tools, the implementation of their pattern matching engine typically follows the same principle. First a matching occurrence of the left-hand side of the graph transformation rule is searched by some graph pattern matching algorithm. Then potential negative application conditions are checked that might eliminate the previous occurrence. However, when a new transformation step is started, all the information on previous matchings is lost, and the complex graph pattern matching phase is restarted from scratch each time. In the paper, we present the foundational data structures and initial experiments for an incremental graph pattern matching engine which keeps track of existing matchings in an incremental way to reduce the execution time of graph pattern matching.

Keywords: Incremental graph transformation, model transformation

1 Introduction

Despite the large variety of existing graph transformation tools, the implementation of their graph transformation engine typically follows the same principle. First a matching occurrence of the left-hand side (LHS) of the graph transformation rule is being found by some graph pattern matching algorithm based on constraint satisfaction (like in AGG [ERT99]) or local searches driven by search plans (PROGRES [Zün96], Dörr's approach [Dör95], FUJABA [FNTZ98]). Then negative application conditions (NAC) 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.

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 critical in the overall performance of a graph transformation engine. However, as the information on a previous match is lost when

a new transformation step is initiated, the complex and expensive graph pattern matching phase is restarted from scratch each time.

Our previous experiments based on benchmarking for graph transformation [VSV05] and practical experience in model-based tool integration based on triple graph grammars [KS06] have clearly demonstrated that traditional non-incremental pattern matching can be a performance bottleneck.

Some basic incremental approaches have already been successfully applied in various graph transformation engines (see Section 6 for a summary) to provide partial support for typical model transformation problems. However, PROGRES [SWZ99] only treated attributes in an incremental way, while the Rete-based approach of [BGT91] lacked the support for negative application conditions and inheritance.

In the current paper, we propose initial concepts (including a common representation for models, metamodels and graph patterns in Section 2), data structures (Section 3) and experiments for incremental graph pattern matching. In a preprocessing phase, all complete matchings (and also non-extensible partial matchings) of a rule are collected and stored explicitly in a matching tree according to a given search plan. This matching tree is updated incrementally triggered by the modifications of the instance graph. Negative application conditions are handled uniformly by storing all matchings of the corresponding patterns. Furthermore, as the main conceptual novelty of the paper, we introduce a notification mechanism by maintaining registries for quickly identifying those partial matchings, which are candidates for extension or removal when an edge is inserted to or deleted from the model.

While the detailed discussion of the algorithms [VVS] is out of scope for the paper, we demonstrate the incremental operation on an example in Section 4 and compare the performance of the incremental approach to Fujaba using the object-relational mapping as graph transformation benchmark in Section 5. Finally, some related work is reviewed in Section 6, while Section 7 concludes our paper.

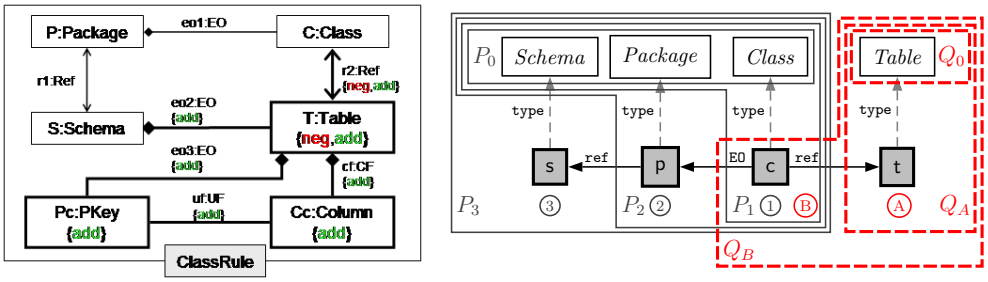
2 A Common Representation for Models and Patterns

First we introduce a uniform representation for models, metamodels and graph patterns informally, using the standard CWM variant [PCTM02] of the object-relation mapping as a running example. This transformation was captured by a set of graph transformation rules in [VSV05].

Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformation. A graph transformation rule contains a left-hand side graph LHS, a right-hand side graph RHS, and (one or more) negative application condition graphs NAC connected to LHS.

The *application* of a rule to a *host (instance) model M* replaces a matching of the LHS in M by an image of the RHS. The most critical step of graph transformation is graph pattern matching, i.e., to find such a matching of the LHS pattern in M which is not invalidated by a matching of the negative application condition graph NAC, which prohibits the presence of certain nodes and edges.

Example. A graph transformation rule ClassRule which transforms an (unmapped) UML class C resided in a UML package P into a relational database table T in the corresponding schema S



(a) ClassRule (b) Its tool independent representation

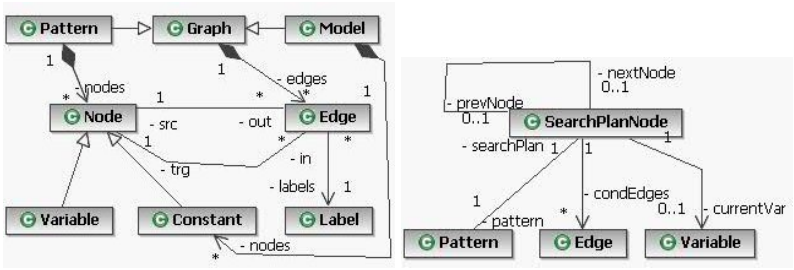
Figure 1: Tool-independent representation of precondition patterns of GT rules

is depicted in Figure 1(a) using the compact Fujaba representation [FNTZ98].

2.1 A Graph Representation for Models and Patterns

In the paper, we use a common, tool independent graph-based framework for representing instance models and graph patterns of rules in a uniform way by using an edge-labelled directed graph. In case of a *pattern* P, a node is either a *constant* (denoted by white boxes in figures) or a *variable* (marked by grey boxes in figures), while a *model* consists of constant nodes only. Inheritance can be handled in this representation by linking all types of a node (i.e., its direct class and all the supertypes) by type edges. A metamodel of our graph representation is presented in Figure 2(a).

A negative application condition [HHT96] is a graph morphism, which maps the LHS pattern to a *NAC pattern*. A *reduced NAC pattern* Q is such a subgraph of a NAC pattern that has (i) the minimum number of nodes (called as *shared nodes*), and (ii) no edges in common with the corresponding LHS pattern P. A *precondition pattern* consists of the LHS pattern, the reduced NAC pattern, and the mapping between them. In the paper, we only use reduced NAC patterns[1] to ensure that the common edges of LHS and NAC patterns are tested only once during pattern matching.



(a) Models and patterns (b) Search plans

Figure 2: Metamodel for models, patterns and search plans

[1] Note that we also omit the word *reduced* in the following.

Example. Figure 4(c) presents a graph representation of an *instance model*. Both the classes of the metamodel (such as *Package*, *Schema*, etc.) and the objects of the instance model (such as *p*, *s*, etc.) uniformly appear as constant nodes. Instance-of relation between nodes is also represented by a dashed `type` edge. Other edge labels (like `EO`) are defined by the metamodel associations.

Figure 1(b) presents the representation of the precondition of the GT rule ClassRule (depicted in Figure 1(a)). The LHS pattern (shown by P_3[2]) has three variables for model-level elements (s, p, c), three constants for metamodel-level elements (*Schema*, *Package*, *Class*), three `type` edges, one `ref` edge, and one `EO` edge. Similarly, the (reduced) NAC pattern (shown by Q_B) consists of variables c, t, the constant *Table*, one `ref` edge and one `type` edge. Furthermore, variable c is a shared node, thus it is contained by both the LHS and NAC patterns.

2.2 Graph Pattern Matching and Search Plans

During graph pattern matching, each variable of a graph pattern is bound to a constant node in the model such that this *matching* (binding) is consistent with edge labels, and source and target nodes of the model. A *matching for a precondition pattern* is a matching for its LHS pattern, provided that no matchings should exist for its NAC pattern.

A *search plan* for a pattern prescribes an order of variables in which they are mapped during pattern matching. In the following, we suppose that a search plan already exists for each pattern, and v_k will denote the kth variable of a pattern according to the corresponding, fixed search plan. A (simplified) metamodel of search plans is depicted in Figure 2(b).

The *kth subpattern* P_k is a subgraph of P where nodes consist of all constants and the first k variables (i.e., $v_1, \ldots, v_k$) of pattern P according to a corresponding search plan, and edges consist of all edges of pattern P whose source and target nodes are both included in the selected set of nodes. *Incoming (outgoing) condition edges* of the kth subpattern P_k are the edges leading into (out of) variable v_k. Without loss of generality, in the following, we consistently use n to denote the number of variables in a (complete) pattern P_n. Consequently, a pattern P_n with n variables has n+1 subpatterns (i.e., $P_0, \ldots, P_n$). A *partial matching for (complete) pattern* P_n is a matching for subpattern P_k. A *maximal partial matching* is a non-extensible partial matching, i.e., pattern P_{k+1} cannot be matched.

Example. For instance, a matching of the LHS pattern P_3 of Figure 1(b) in model Figure 4(e) is: c = *c1*, p = *p*, s = *s*. A matching of the NAC pattern (see Q_B in Figure 1(b)) in model Figure 4(g) is: c = *c1*, t = *t1*.

We define a search plan for the LHS pattern by fixing orders on variables (1) c, (2) p, (3) s. A search plan for the NAC pattern is (A) t, (B) c.[3] The position of a variable in a fixed order is denoted by a numbered circle in Figure 1(b). Search plans are generated independently of each other in the current version of the pattern matching engine.

Based on these search plans, subpatterns of LHS are shown by areas P_0, P_1, P_2, P_3 surrounded by solid (grey) borders in Figure 1(b). Subpatterns of NAC are Q_0, Q_A, Q_B, drawn by dashed (red) borders. Note that P_0 and Q_0 denote the empty matchings for the LHS and the NAC, respectively.

[2] The purpose of P_is and Q_is will be explained later in Section 2.2.

[3] Search plans of the current example have been selected manually for presentation purposes.

The EO edge connecting c to p is an incoming condition edge of pattern P_2, while the type edge connecting p to *Package* represents an outgoing condition edge, since they are edges of pattern P_2, and they lead into and out of the second variable (p) of the corresponding search plan of the LHS pattern.

3 Data Structures for Incremental Pattern Matching

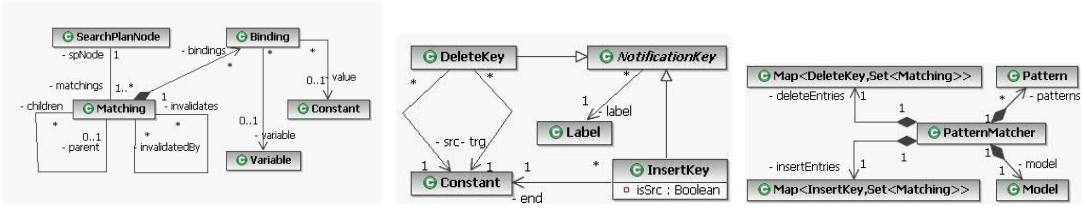
In this section, we present the data structures needed for the efficient storage of partial matchings. Class diagrams depicting the different aspects of data structures being used by the incremental pattern matching engine are shown in Figure 3.

Matching and Matching Tree. A Matching (denoted by a numbered circle in Figure 4) represents a partial matching for a pattern. It contains a set of Bindings. Each binding defines a mapping of a Variable to a Constant.

For each pattern P_n, a *matching tree* is maintained, which consists of matchings being organized into a tree structure along parent-child edges (depicted by dashed arcs in Figure 4). The *root* of the tree denotes the empty matching for the corresponding pattern, i.e., when none of the variables have been bound. Each *level* of the tree (denoted by light grey areas in Figure 4) contains matchings for a subpattern of pattern P_n. The mapping of subpatterns to tree levels is guided by the search plan having been fixed for the pattern. A *tree node* in level k (i.e., having distance k from the root) represents a matching of the kth subpattern being specified by the search plan. Each *leaf* represents a maximal partial matching for the pattern. By supposing that the pattern P_n has n variables, each leaf in (the deepest possible) level n represents a complete matching of the pattern.

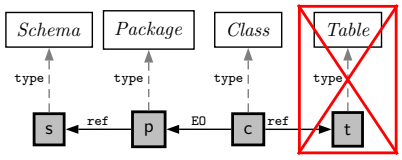
Example. Sample models of Figs. 4(c), 4(e), and 4(g) and the corresponding data structure contents are presented in Figs. 4(d), 4(f), and 4(h), respectively. Figs. 4(d), 4(f), and 4(h) show matching trees in their top-right corner, they depict binding arrays at the bottom, while notification arrays are presented in their left part.

Figure 4(d) contains two matching trees representing the partial matchings of the LHS pattern and the NAC pattern, respectively. Matchings 1 and 2 denote empty matchings. Matching 3 is located on the first tree level of the LHS pattern, thus, it is a matching for subpattern P_1, which contains a single binding that maps variable c to constant *c1*. Matching 3 is a child of matching 1, as the latter can be extended by the mapping of variable c.

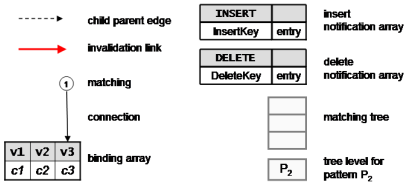


(a) Matchings (b) Event processing (c) Pattern matcher

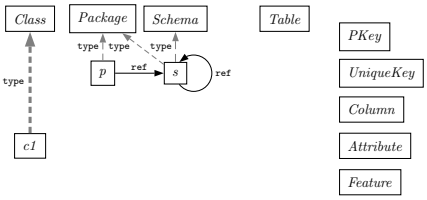
Figure 3: Data structures of the incremental pattern matching engine



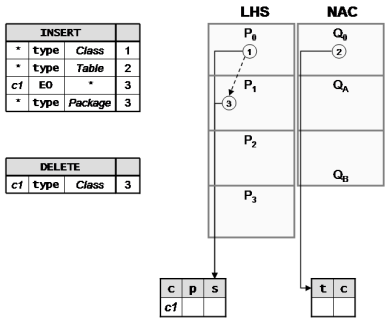
(a) Precondition pattern for ClassRule



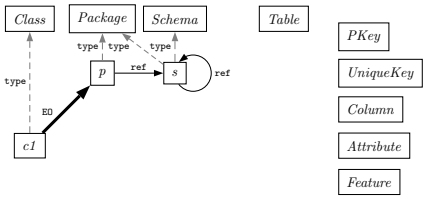
(b) Notational guide for data structures



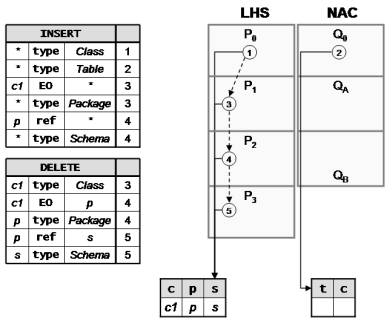
(c) Model 1



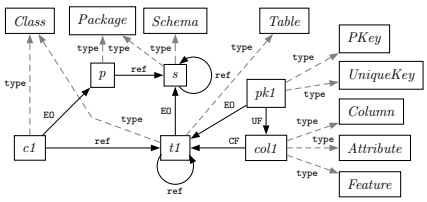
(d) Data structure contents for Model 1



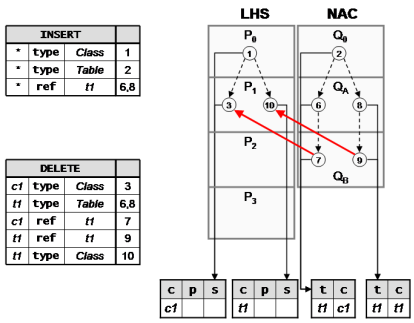
(e) Model 2



(f) Data structure contents for Model 2



(g) Model 3



(h) Data structure contents for Model 3

Figure 4: Sample models and the corresponding data structures

In the context of Figure 4(d), matching 3 is a maximal partial matching as it cannot be further extended, due to the lack of outgoing EO edges leading out of *c1*. On the other hand, matching 3, is not a maximal partial matching in Figure 4(f) as it can be extended e.g., by mappings p to *p* and s to *s* to get matching 5. This means a complete matching for the LHS pattern as matching 5 is located on the lowest tree level P_3.

Binding Arrays. Matchings are physically stored as one-dimensional binding arrays, which are indexed by the variables. An entry in a binding array stores variable–constant pairs in the corresponding matching. When one matching is an ancestor of another one, their binding arrays can be shared in order to reduce memory consumption as the ancestor matching contains a subset of the bindings of the descendant matching. Consequently, for each pattern P_n with n variables, a binding array `match[n]` of size n is used. In figures, binding arrays are connected to matchings by solid black lines.

Example. Since the LHS pattern has 3 variables, matchings of the LHS tree refer to binding arrays having 3 entries as it is shown e.g., in the lower part of Figure 4(f). Each column of the binding array of the LHS matching tree represents a binding of variables (upper row) to constants (lower row). Note that memory consumption can be reduced by sharing binding arrays among a matching and any of its ancestors in the matching tree. E.g., the array that contains mappings c to *c1*, p to *p* and s to *s* can be shared by matchings 1, 3, 4, and 5, as they only consist of the first 0, 1, 2, and 3 bindings of the array, respectively. In spite of the fact that a completely filled binding array is assigned to matching 3 in Figure 4(f), this matching only makes use of the single mapping c to *c1* in the algorithms.

Invalidation Edges. Invalidation edges, which are denoted by thick (red) arcs, represent the invalidation of partial matchings of a LHS caused by complete matchings of a NAC.

Example. The red invalidation edge of Figure 4(h) connecting matchings 7 to 3 means that matching 7 is a complete matching for the NAC pattern, which invalidates matching 3 as both map the shared variable c to the same constant *c1*. As long as matching 3 is invalidated (as shown by the incoming invalidation edge), it cannot be part of a complete matching for the LHS pattern, which fact is marked by the empty subtree rooted at matching 3.

Notification Arrays. Since the transformation engine sends notifications on model changes, notification related data structures (shown in Figure 3(b)) are also needed. The incremental pattern matching engine has a single `INSERT` and a single `DELETE` notification array consisting of notification entries.

- An *entry in the insert notification array* is a pair consisting of an `InsertKey` (with `label`, `end` and attribute `isSrc`) and a list of `Matchings` to be notified. If an edge `e` with label `e.lab` connecting `e.src` to `e.trg` is added to the model, then `Matchings` of such insert notification array entries are notified whose `InsertKeys` are of the form `[e.src,e.lab,*]` and `[*,e.lab,e.trg]`. Based on the notation of Figure 3(b), these `InsertKeys` correspond to the `end=e.src`, `label=e.lab`, `isSrc=true` and `end=e.trg`, `label=e.lab`, `isSrc=false` settings, respectively.

- An *entry in the delete notification array* is a pair consisting of a `DeleteKey` and a list of `Matchings` to be notified. If an edge `e` with label `e.lab` connecting `e.src` to `e.trg` is removed from the model, then `Matchings` of such delete notification array entry is notified whose `DeleteKey` is of the form `[e.src,e.lab,e.trg]`.

Example. Sample notification arrays are presented e.g., in the left part of Figure 4(d). The `INSERT` notification array has 4 entries of which the first is triggered by the `InsertKey [*,type,Class]` and refers to matching `1`. This entry means that matching `1` has to be notified, when a `type` edge leading to *Class* is inserted into the model. Similarly, the first entry in the `DELETE` notification array means that matching `3` must be notified, if the `type` edge connecting *c1* to *Class* is deleted.

Query Index Structure. A *query index structure* (not shown in figures) is also defined for each precondition pattern to speed-up the queries of complete matchings initiated by the GT tool that use the services of the incremental pattern matching approach.

4 Incremental Operations on an Example

During the incremental operation phase, the matching tree is maintained by four main methods of class `Matching`.

1. The `insert()` method is responsible for the possible extension of the current partial matching for proper subpattern P_k to create a new partial matching for subpattern P_{k+1}.
2. The `validate()` method is responsible for the recursive extension of insert operations to all (larger) subpatterns.
3. The `delete()` method removes the whole matching subtree rooted at the current matching for subpattern P_k.
4. The `invalidate()` method is responsible for the recursive deletion of all children matchings of the current matching.

These methods are called by the pattern matching engine when *edge modification events* arrive from the model repository.

- *Insert edge notification.* If an edge `e` with label `e.lab` connecting constants `e.src` to `e.trg` is added to the model, then the `insert()` method of class `Matching` is invoked (i) with parameter `e.trg` on every matching as defined by entry `INSERT[e.src, e.lab,*]`, and (ii) with parameter `e.src` on every matching as defined by the entry `INSERT[*,e.lab,e.trg]`.
- *Delete edge notification.* If an edge `e` with label `e.lab` connecting constants `e.src` to `e.trg` is removed from the model, then `delete()` method of class `Matching` is invoked on every matching being notified by entry `DELETE[e.src,e.lab,e.trg]`.

Due to space restrictions, we only exemplify the process by using our running example of Figure 4. (The details of the algorithms can be found in [VVS].) Let us suppose that a class *c1* is added to package *p* in the model by user interaction initiated by the system designer. The pattern matching engine is notified about this activity in two steps. First a notification arrives about the insertion of a `type` edge connecting `c1` to *Class* (see Figure 4(c)) followed by the insertion of an `EO` edge connecting *c1* to *p* (see Figure 4(e)). Modifications are denoted by thick lines.

Step 1. At the insertion of a `type` edge connecting `c1` to *Class*, the pattern matching engine looks up entries retrieved by insert keys `[c1,type,*]` and `[*,type,Class]`.

The latter entry triggers the possible extension of the empty matching `1` by mapping variable `c` to constant *c1* by invoking the `insert()` method on matching `1` with parameter *c1*. As this binding is a matching for pattern P_1, (i) a new matching `3` is created and added to the (matching) tree as a child of matching `1`, and (ii) the binding `c` to *c1* is recorded.

Then matching `3` is inserted into the delete notification array with delete key `[c1,type, Class]`. This means that whenever the `type` edge from *c1* to *Class* (i.e., the edge that has been just added) is removed, this matching should be deleted.

Effects of adding a new matching to the tree are recursively extended to find matchings for larger subpatterns by calling `validate`. Matching `3` can be further extended (as shown by corresponding new entries being added to the insert notification array pointing to matching `3`), whenever an edge with label `EO` leading out of *c1* or with label `type` leading to *Package* is added to the model *in the future*.

As also the *current content of the model* may extend matching `3`, we initiate the possible extensions of this matching by checking the existence of at least the `EO` edges leading out of *c1*.[4] As no such edges exist in our example, the algorithm terminates with the matching tree presented in Figure 4(d).

Step 2. When `EO` edge connecting *c1* to *p* is inserted (as shown by the thick line of Figure 4(e)), matching `3` is first extended to a new matching `4` by mapping variable `p` to constant *p* and by executing a sequence of `insert()` and `validate()` method calls as shown in Figure 5.

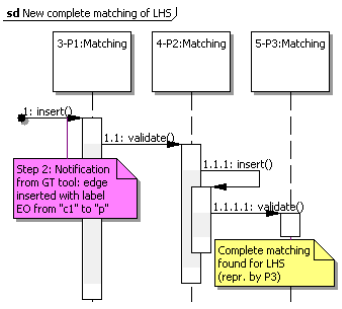


Figure 5: Sequence diagram showing edge insertion into the LHS pattern

[4] Note that the insert key generation and the possible further extension of matching `3` are guided by the condition edges of the one larger subpattern P_2.

This time, matching extension is propagated to another new matching 5 by assigning s to *s* by invoking the `insert` method on matching 4 with parameter *s*, as the current model already contained `ref` and `type` edges connecting *p* to *s* and *s* to *Schema*, respectively.

In addition, both new matchings are appropriately registered in both the insert and delete notification arrays, and the binding array is updated accordingly. The corresponding matching tree is shown in Figure 4(f).

At this point, matching 5 represents a complete matching for the LHS pattern, so the GT rule ClassRule can be applied.

Step 3. The result of applying the GT rule ClassRule on matching 5 can be observed in Figure 4(g) after the insertion of some 13 edges, processed one by one by the pattern matching engine.

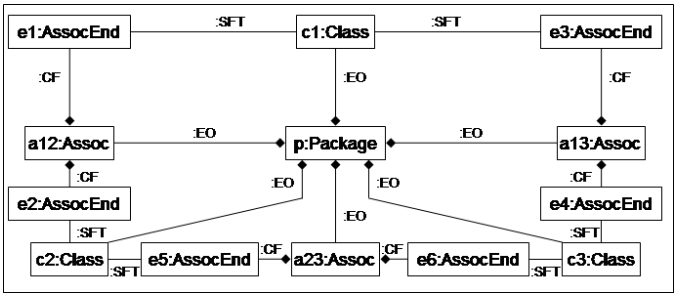
Let us suppose that the new `ref` edge between *c1* and *t1* is processed first, which is followed by the insertion of of `type` edge connecting *t1* to *Table*. The first edge causes no modifications in data structures as no appropriate insert keys appear in the insert notification array.

At the second edge insertion, matching 2 is notified by invoking its `insert` method with parameter *t1*, which creates matchings 6 and 7. As the latter is a complete matching of the NAC pattern Q_B, matching 3 must be invalidated by deleting all its descendant matchings in the tree. When all the 13 edges are added, the data structure will reflect the situation in Figure 4(h).

5 Experimental Evaluation

In order to assess the performance of our incremental approach, we performed measurements on the object-relational mapping benchmark example [VFV06]. As a reference for the measurements, we selected Fujaba [FNTZ98] as it is among the fastest non-incremental GT tools.

By using the terminology of [VSV05], graph transformation rules, the initial model and the transformation sequence have to be fixed up to numerical parameters in order to fully specify a test set.



(a) Initial model of the test case for the $N = 3$ case

	Class #	Model size #	TS length #	Fujaba match msec	Fujaba update msec	Incremental match msec	Incremental update msec
classRule	10	1342	146	0.201	0.479	0.026	5.439
	30	12422	1336	0.287	0.052	0.023	56.116
	50	34702	3726	0.171	0.012	0.021	221.955
	100	139402	14951	0.278	0.011	0.042	2067.462
assocRule	10	1342	146	0.937	0.148	0.019	1.665
	30	12422	1336	2.488	0.101	0.032	4.510
	50	34702	3726	3.371	0.032	0.022	6.849
	100	139402	14951	11.959	0.030	0.039	26.684
assocEndRule	10	1342	146	0.875	0.107	0.043	0.592
	30	12422	1336	3.896	0.045	0.016	1.108
	50	34702	3726	5.975	0.025	0.023	1.948
	100	139402	14951	24.057	0.028	0.068	9.353

(b) Experimental results

Figure 6: Initial model and measurement results

The structure of the initial model is presented in Figure 6(a) for the $N = 3$ case. The model has a single Package that contains *N* classes, which is the only numerical parameter of the test set. An

Association and 2 AssociationEnds are added to the model for each pair of Classes, thus initially, we have $N(N-1)/2$ Associations and $N(N-1)$ AssociationEnds. Associations are also contained by the single Package as expressed by the corresponding links of type EO. Each AssociationEnd is connected to a corresponding Association and Class by a CF and SFT link, respectively.

The object-relational mapping can be specified by 4 graph transformation rules, which describe how to generate the relational database equivalents of Packages, Associations, Classes, and AssociationEnds, respectively. (Due to space restrictions, the exact benchmark specification is omitted from the paper. The reader is referred to [VFV06].) The transformation sequence consists of the application of these rules on each UML entity in the order specified above.

Measurements were performed on a 1500 MHz Pentium machine with 768 MB RAM. A Linux kernel of version 2.6.7 served as an underlying operating system. The time results are shown in Figure 6(b).

The head of a row shows the name of the rule on which the average is calculated. (Note that a rule is executed several times in a run.) The second column (Class) depicts the number of classes in the run, which is, in turn, the runtime parameter N for the test case. The third and fourth columns show the concrete values for the model size (meaning the number of model nodes and edges) and the transformation sequence length, respectively. Heads of the remaining columns unambiguously identify the approach having been used. Values in match and update columns depict the average times needed for a single execution of a rule in the pattern matching and updating phase, respectively. Execution times were measured on a microsecond scale, but a millisecond scale is used in Figure 6(b) for presentation purposes.

Our experiments can be summarized as follows.

- In accordance with our assumptions, the incremental engine executes pattern matching in constant time even in case of large models, while the traditional engine shows significant increase when the LHS of the pattern is large as in case of assocEndRule.

- Incremental techniques by their nature suffer time increase in the updating phase due to (i) the bookkeeping overhead caused by the additional data structures, and (ii) the fact that even the insertion of a single edge may generate (or delete) a significant amount of matchings. Its detrimental performance effects are reported in the updating phase of classRule, when also the matchings of the other rules have to be refreshed. On the other hand, the traditional engine executes the update phase in constant time as it can be expected.

- By taking into account both phases in the analysis, it may be stated that the incremental strategy provides a competitive alternative for traditional engines as the total execution times of the incremental approach are of the same order of magnitude in case of the frequently applied rules (i.e., assocRule and assocEndRule).

- The benefits of the incremental approach are the most remarkable (i) when rules have complex LHS graphs as the pattern matching of Fujaba gets slow in this case and (ii) when the dependency between rules is weak as this leads to fast updates in incremental engines.

As a consequence, we may draw that the incremental approach is a primary candidate for graph transformation tools where (i) complex transformation rules are used and (ii) where all

matchings of a rule have to be accessed rapidly, which is a typical case for analysis/verification tools.

6 Related Work

Incremental updating techniques have been widely used in different fields of computer science. Now we give a brief overview on incremental techniques that could be used for graph transformation.

Rete networks. [BGT91] proposed an incremental graph pattern matching technique based on the idea of Rete networks [For82], which stems from rule-based expert systems. In their approach, a network of nodes is built at compile time from the LHS graph to support incremental operation. Each node performs simple tests on the entities (i.e., nodes, edges, partial matchings) arriving to its input(s). If the test succeeds, the node groups entities into compound ones, which are then put into its output. On the top level of the network, there are nodes with a single input that let such objects and links of a given type to pass that have just been inserted to or removed from the model. On intermediate levels, network nodes with two inputs appear, each representing a subpattern of the LHS graph. These nodes try to build matchings for the subpattern from the smaller matchings located at the inputs of the node. On the lowest level, the network has terminal nodes, which do not have outputs. They represent the entire LHS pattern. Entities reaching the terminals represent complete matchings for the LHS.

The technique of [BGT91] shows the closest correspondance to our approach, as matching levels can be considered as nodes in the Rete network. However, it is not a one-to-one mapping as one matching level in our approach corresponds to several Rete nodes. As a consequence, Rete-based solutions have more bookkeeping overhead as they store information at the inputs of nodes in local memories and they use more nodes.

Two significant consequences can be drawn from this similarity. (i) All techniques (e.g., the handling of common parts of different LHS patterns at the same network node [MB00]) that have already been invented for Rete-based solutions are also applicable to our approach. (ii) The idea of notification arrays can speed-up traditional Rete-based approaches used in a graph transformation context as these arrays help identifying those partial matchings that may participate in the extension of the matching. Thus, it is subject to our future investigations.

PROGRES. The PROGRES [SWZ99] graph transformation tool supports an incremental technique called attribute updates [Hud87]. At compile-time, an evaluation order of pattern variables is fixed by a dependency graph. At run-time, a bit vector having a width that is equal to the number of pattern variables, is maintained for each model node expressing if a variable can be mapped to a given node.

When model nodes are deleted, some validity bits are set to false according to the dependency graph denoting the termination of possible partial matchings. In this sense, PROGRES (just like our approach) performs immediate invalidation of partial matchings. On the other hand, validation of partial matchings are only computed on request (i.e., when a matching for the LHS is requested), which is a disadvantage of the incremental attribute updating algorithm.

As an advantage, PROGRES has a low-level bookkeeping overhead (i.e., some extra bits for model nodes), and the index structures maintained for partial matchings are also smaller.

View updates. In relational databases, materialized views, which explicitly store their content on the disk, can be updated by incremental techniques. Counting and DRed algorithms [GMS93] first calculate the delta (i.e., the modifications) for the view by using the initial contents of the view and base tables and the deltas of base tables. Then the calculated deltas are performed on the view.

In contrast to our approach, view updating algorithms are more flexible as they use a run-time evaluation order for delta calculation, and they can provide both lazy and eager style updates being specified when a view is created.

[VFV06] proposed an approach for representing graph pattern matching in relational databases in form of views. Although some initial research (reported in [VV04]) has been done for incremental pattern matching in relational databases, this solution has been completely thrown away as it suffers from the inadequate support of incremental algorithms by the underlying databases and the strong restrictions being posed on the structures of the select query that defines the view.

7 Conclusion

In the current paper, we proposed data structures for incremental graph pattern matching where all matchings (and non-extensible partial matchings) of a rule are stored explicitly in a matching tree. This matching tree is updated incrementally triggered by the modifications of the instance graph. Negative application conditions are handled uniformly by storing all matchings of the corresponding patterns. As the main added value of the paper, we introduced a notification mechanism by maintaining additional registries for quickly identifying those partial matchings, which are candidates for extension or removal, and thus, which have to be notified when an edge is inserted to or deleted from the model.

Limitations. We have also identified certain limitations of the presented algorithms. First of all, the efficiency of the incremental pattern matching engine highly depends on the selection of search plans as even a single edge insertion (or deletion), which affect matchings located at upper levels of the tree (i.e., near to its root) may trigger computation intensive operations. As a consequence, further investigations on creating good search plans for the incremental pattern matching engine have to be carried out.

Our current solution provides a suboptimal solution, when patterns contain a large number of loop edges. This is related to the fact that our approach currently stores only the matchings of the nodes but not the edges (i.e., edges do not have identifiers), which assumption can be relaxed in the future.

At first glance, it can be strange that NACs are handled independently of the LHS (i.e., all matchings of the NAC are calculated). The goal of our approach is to support the reusability of patterns when the same pattern can be used once in the LHS and once as a NAC, or the same NAC is a negative condition for multiple LHSs (as in VIATRA2 [BV06]).

Future work. In the order of importance, the following tasks would appear on our todo list for the future: (i) investigation on the applicability of Rete-networks in our incremental approach, (ii) generation of search plans that are optimized for incremental pattern matching, (iii) the optimal handling of bulk inserts, which may significantly accelerate the initialization phase, (iv) the

implementation of the pattern merger and optimizer module to be able to share matchings across matching trees, and (v) the incremental handling of path expressions.

Acknowledgements: This work was partially carried out during the visit of the first author to TU Darmstadt (Germany), and it was partially funded by the SegraVis RTN. The first and the second author had partial support from the SENSORIA European IP (IST-3-016004). The second author was also partially supported by the J. Bolyai Scholarship.

Bibliography

[BGT91] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammar Based on the RETE-Matching Algorithm. In *Proc. Graph Grammars and Their Application to Computer Science and Biology*. LNCS 532, pp. 174–189. 1991.

[BV06] A. Balogh, D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proc. of the 21st ACM Symposium on Applied Computing*. Pp. 1280–1287. ACM Press, Dijon, France, April 2006.

[Dör95] H. Dörr. *Efficient Graph Rewriting and Its Implementation*. LNCS 922. Springer-Verlag, 1995.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. *In [EEKR99]*. Chapter The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels and Rozenberg (eds.), *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation*. LNCS 1764, pp. 296–309. Springer Verlag, 1998.

[For82] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence* 19:17–37, 1982.

[GMS93] A. Gupta, I. S. Mumick, V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD Proceedings*. Pp. 157–166. Washington, D.C., USA, 1993.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* 26(3/4):287–313, 1996.

[Hud87] S. E. Hudson. Incremental Attribute Evaluation: an Algorithm for Lazy Evaluation in Graphs. Technical report 87-20, University of Arizona, 1987.

[KS06] A. Königs, A. Schürr. MDI - A Rule Based Multi-document and Tool Integration Approach. *Software and Systems Modeling*, 2006. To appear.

[MB00] B. T. Messmer, H. Bunke. Efficient Subgraph Isomorphism Detection: A Decomposition Approach. *IEEE Transactions on Knowledge and Data Engineering* 12(2):307–323, 2000.

[PCTM02] J. Poole, D. Chang, D. Tolbert, D. Mellor. *Common Warehouse Metamodel*. John Wiley & Sons, Inc., 2002.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. *In [EEKR99]*. Chapter The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.

[VFV06] G. Varró, K. Friedl, D. Varró. Implementing a Graph Transformation Engine in Relational Databases. *Software and Systems Modeling* 5(3):313–341, September 2006.

[VSV05] G. Varró, A. Schürr, D. Varró. Benchmarking for Graph Transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 79–88. IEEE Computer Society Press, Dallas, Texas, USA, September 2005.

[VV04] G. Varró, D. Varró. Graph Transformation with Incremental Updates. In Heckel (ed.), *Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2004)*. ENTCS 109, pp. 71–83. Elsevier, Barcelona, Spain, December 2004.

[VVS] G. Varró, D. Varró, A. Schürr. Incremental Graph Pattern Matching. http://www.cs.bme.hu/~gervarro/publication/IncrementalEngine.pdf.

[Zün96] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*. LNCS 1073, pp. 454–468. Springer-Verlag, 1996.