Incremental Graph Pattern Matching

Gergely Varró*and Dániel Varróand Andy Schürr*****
*Department of Computer Science and Information Theory
Budapest University of Technology and Economics
**Department of Measurement and Information Systems
Budapest University of Technology and Economics
***Real-Time Systems Lab
Technical University of Darmstadt

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 (LHS) of the graph transformation rule is being found by some sophisticated graph pattern matching algorithm based on constraint satisfaction (like [LV02] in AGG [ERT99]) or local searches driven by search plans (PROGRES [Zün96], Dörr's approach [Dör95], FUJABA [FNTZ98], VIATRA2 [VVF05]). Then potential 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, 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. 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 Sec. 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 foundational data structures, algorithms, and experiments for incremental graph pattern matching where all complete matchings (and also non-extensible partial matchings) of a rule are 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, we keep track if a matching of the negative condition pattern invalidates the matching of the positive pat-

tern. 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.

Our aim in this paper is to propose data structures and algorithms in a general way independent of existing graph transformation tools, while the adaptations to such GT tools are subject of future plans.

Architectural Overview In Figure 1, an architectural overview is provided on the envisaged workflow of an incremental pattern matching engine. Note that a main driver of this architecture is to allow easy adaptation to existing GT engines.

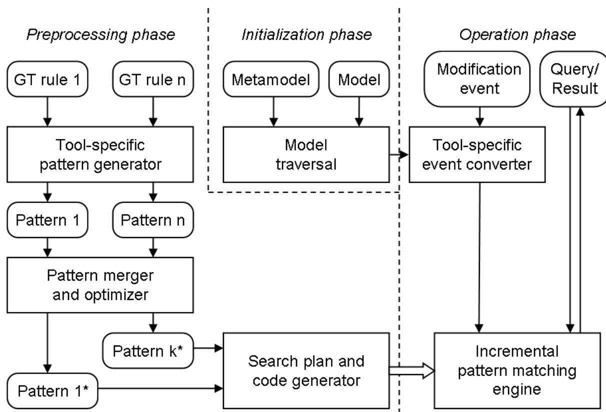


Figure 1: Architectural overview of incremental pattern matching

Preprocessing. In a preprocessing phase, patterns are first extracted from graph transformation rules (based upon the LHS and NAC of the rules). Since these patterns may be overlapping, this initial set of graph patterns can be optimized to normalized to minimize between patterns. Afterwards, search plans are derived for the optimized pattern set, and template-based code generation is applied to implement the matching tree tailored to the actual GT rules.

Initialization. In the initialization phase, the matching tree is constructed based upon a given initial model and its metamodel. While this initialization step can be time consuming, this is only performed once, prior to the actual transformations.

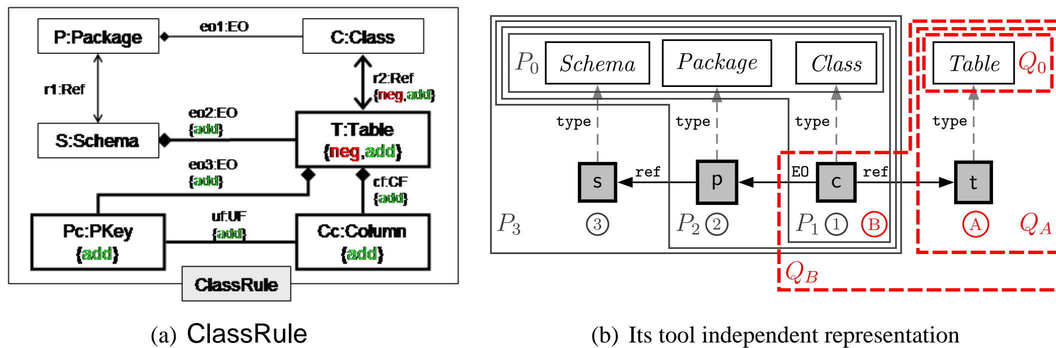
Operation. In the operation phase (which is the main focus of the current paper), the incremental pattern matching engine listens to the notifications sent by the GT engine on model modifications, and keeps track of the changes in the matching tree. As a consequence, pattern matching queries coming from the GT engine are executed in constant time.

2 Tool independent model and pattern representation

First we introduce a uniform and tool-independent 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].

2.1 Informal introduction

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



(a) ClassRule (b) Its tool independent representation

Figure 2: Tool-independent representation of precondition patterns of GT rules

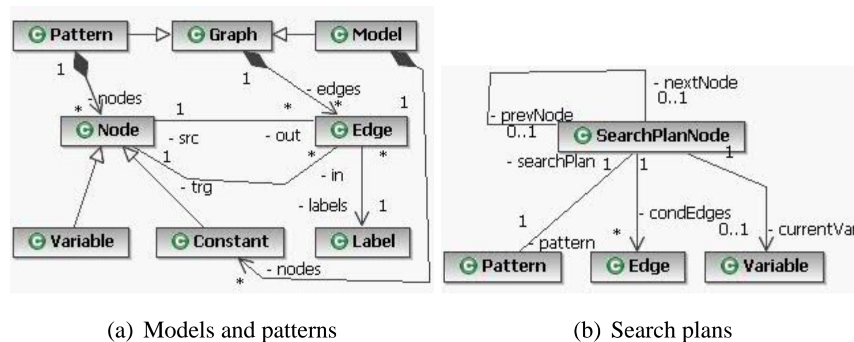
condition graphs NAC connected to LHS.

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 is depicted in Fig. 2(a) using the compact Fujaba representation [FNTZ98].

2.2 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. Both models and patterns are described by directed labelled graphs where a node is further either a constant or a variable. A metamodel of our graph representation is presented in Fig. 3(a).



(a) Models and patterns (b) Search plans

Figure 3: Metamodel for models, patterns and search plans

Example. Figure 5(c) presents a tool independent 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*, *c1*, etc.) uniformly appear as constant nodes. Traditional instance-of relation between nodes is also represented by edges using dashed (light grey) edges with label `type`. Other edge labels (like `EO`, and `ref`) are defined by the associations of the metamodel.

Figure 2(b) presents the tool independent representation of the precondition of the graph transformation rule ClassRule (depicted in Fig. 2(a)). The LHS pattern (shown by P_3[1]) 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*, 1 `ref` edge and 1 `type` edge.

Note that in our graph representation, LHS and NAC patterns share nodes minimally required as interfaces between the two graphs. For instance, variable c is a shared node, thus it is contained by both patterns.

Definitions. Formally, an *(edge-)labelled directed graph* $G = (N_G, E_G, src_G, trg_G, l_G)$ consists of a set of nodes $N_G = V_G \cup C_G$ (where V_G are variables and C_G are constants with $V_G \cap C_G = \emptyset$), a set of edges E_G, and a label morphism $l_G : l_G : E_G \to \mathcal{E}_n$, a source morphism $src_G : E_G \to V_G$ and a target morphism $trg_G : E_G \to V_G$.

A *model* M is a labelled directed graph consisting of only constant nodes (i.e., $V_M = \emptyset$). Note that inheritance can be handled in this representation by multiple outgoing type edges from a model node to all (type-consistent) metamodel nodes.

A *pattern* P is a labelled directed graph. Traditionally, a negative application condition (nac) [HHT96] is treated as a graph morphism, which maps the LHS pattern P to a *NAC pattern* N, formally, $nac : P \to N$. A *reduced NAC pattern* Q is a subgraph of NAC pattern N, which is derived by keeping exactly those edges of N (together with their source and target nodes), where at least its source or target node is in $N \setminus P$. *Shared nodes* S are such nodes of reduced NAC pattern Q that are contained by both patterns P and Q. A *precondition pattern* $\mathsf{PRE} = (P, N, nac)$ consists of the LHS pattern P, the NAC pattern N, and the mapping nac between them. In the paper, we only use reduced NAC patterns to ensure that the common edges of P and N are tested only once during pattern matching. Note that we also omit the word *reduced* in the following.

2.3 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 binding (matching) is consistent with edge labels, and source and target nodes of the target model. A subpattern is a subgraph of a graph pattern. A (complete) matching of subpattern is a partial matching of the entire pattern.

A search plan for a pattern prescribes an order in which pattern variables are to be mapped during pattern matching. At each step, the match of the kth subpattern is extended to a match of the $k+1$th subpattern by binding the next variable. A (simplified) metamodel of search plans is depicted in Fig. 3(b).

Example. For instance, a matching of the LHS pattern (see P_3 in Fig. 2(b)) in model Fig. 5(e) is: C =

[1]The purpose of P_is and Q_is will be explained later in Sec. 2.3.

c1, P = *p*, S = *s*. A matching of the NAC pattern (see Q_2 in Fig. 2(b)) in model Fig. 5(g) is: C = *c1*, T = *t*.

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.

Based on these search plans, subpatterns of LHS are shown by areas (P_0, P_1, P_2, P_3 with solid (grey) borders in Fig. 2(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.

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

Definitions. A *matching* m *for a pattern* P *in a model* M (denoted by m^P) is a label preserving total graph morphism $m^P : P \rightarrow M$, which means that (i) each variable of P should be mapped to a constant of M, (ii) each constant of P should be mapped to the same constant in M, and (iii) for each edge e of pattern P with label $l(e)$, there should exist an edge $m(e)$ with label $l(e)$ in model M, such that the matching is source and target consistent (i.e., $m(src(e)) = src(m(e))$ and $m(trg(e)) = trg(m(e))$). A *matching for a precondition pattern* PRE *in a model* M is a matching for its LHS pattern, provided that no matchings should exist for its NAC pattern.

A *search plan* π_P *for pattern* P is an ordering of variables V_P of pattern P, in which they are to be mapped during pattern matching. In the following, we suppose that a search plan already exists for each pattern, and the notation v_k will denote the kth variable of a pattern P according to the corresponding, fixed search plan π_P.

Given a search plan π_P for pattern P, the *kth subpattern* P_k is a subgraph of P where nodes $N_k = C \cup V_k$ consist of all constants and the first k variables $V_k = \bigcup_{1 \leq i \leq k} \{ v_i \}$ of pattern P, 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 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.

3 Data Structures for Incremental Pattern Matching

In this section, we present the data structures needed for the efficient storage of partial matchings. Algorithms of the incremental pattern matching engine, which operate on these data structures are discussed later in Sec. 4.

Class diagrams depicting the different aspects of data structures being used by the incremental pattern matching engine are shown in Fig. 4.

Matching and matching tree. A `Matching` (denoted by a numbered circle in Fig. 5) represents a partial matching for a pattern. It contains a set of `Bindings`. Each binding defines a mapping of a `Variable` to a `Constant`.

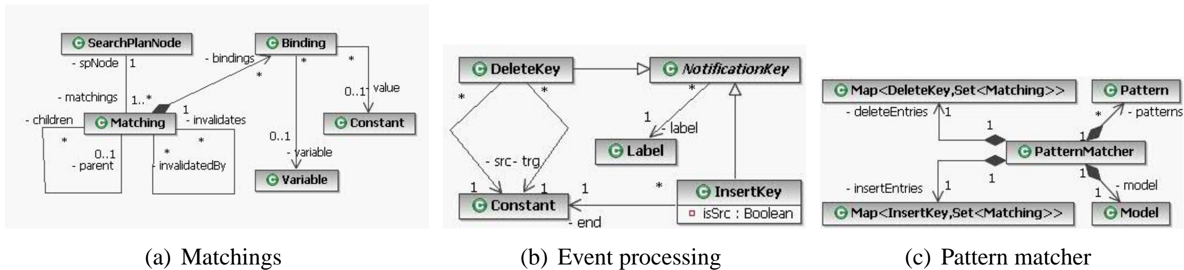
For each pattern P_n, a *matching tree* is maintained, which consists of matchings being organized into a tree structure along `parent-child` edges (depicted by dashed arcs in Fig. 5). The *root* of the tree denotes the empty matching for the corresponding pattern, i.e., when none of the variables have been bound. Each *level* of the tree (denoted by light grey areas in Fig. 5) 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 π. 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. 5(c), 5(e), and 5(g) and the corresponding data structure contents are presented in Figs. 5(d), 5(f), and 5(h), respectively. Figs. 5(d), 5(f), and 5(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.

Fig. 5(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.

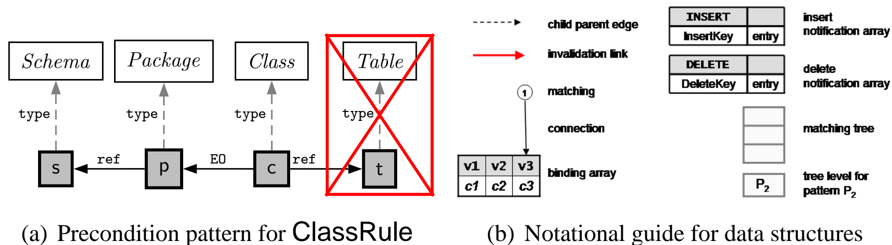
In the context of Fig. 5(d), matching 3 is a maximal partial matching as it cannot be further extended, due to the lack of outgoing `EO` edges leading out of *c1*. On the other hand, matching 3, is not a maximal partial matching in Fig. 5(f) as it can be extended e.g., by mappings p to *p* and s to *s* 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 descen-

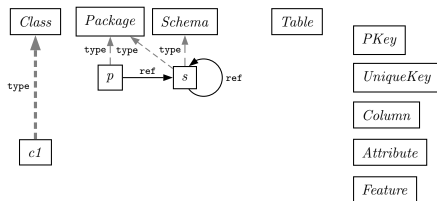


(a) Matchings (b) Event processing (c) Pattern matcher

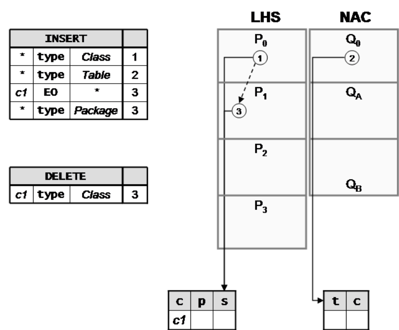
Figure 4: Data structures of the incremental pattern matching engine



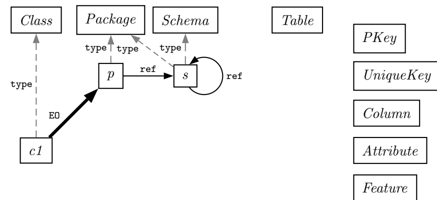
(a) Precondition pattern for ClassRule (b) Notational guide for data structures



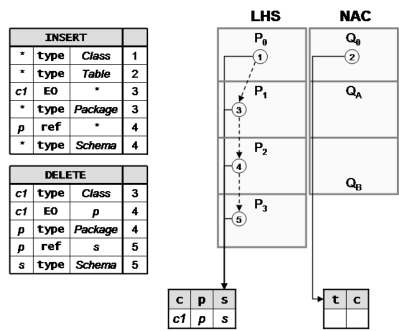
(c) Model 1



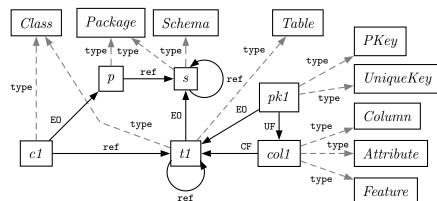
(d) Data structure contents for Model 1



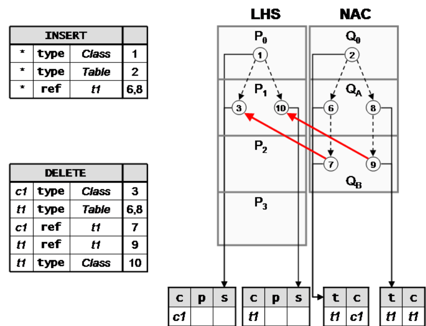
(e) Model 2



(f) Data structure contents for Model 2



(g) Model 3



(h) Data structure contents for Model 3

Figure 5: Sample models and the corresponding data structures

dant 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 Fig. 5(f). Each column of the binding array of the LHS matching tree represents a binding, which shows the constant (in the lower row) to which the variable (in the upper row) has been mapped. Note that 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.

Invalidation edges. Invalidation edges represent the invalidation of partial matchings of a LHS caused by complete matchings of a NAC. In the following, we simply use thick (red) arcs for denoting invalidation.

Example. The red invalidation edge of Fig. 5(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 graph transformation engine sends notifications on model changes, notification related data structures (shown in Fig. 4(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]`. We use notations `[end,label,*]` and `[*,label,end]` for cases when `end` denotes the source (`isSrc=true`) and target (`isSrc=false`) end of an edge with label `label`, 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 Fig. 5(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 Operations for Incremental Pattern Matching

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 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]`.

4.1 Incremental operations on an example

Prior to the detailed discussion of the algorithms, we first exemplify the process by using our running example of Fig. 5. Let us suppose that a class *c1* is added to package *p* in the model by user interaction initiated by the system designer. The tool-independent representation of the model is notified about that in two steps. First a notification arrives about the insertion of a `type` edge connecting c1 to *Class* (see Fig. 5(c)) followed by the insertion of an `EO` edge connecting *c1* to *p* (see Fig. 5(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 added to 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`. To record the fact that 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*, matching 3 can be further extended, corresponding new entries are added to the insert notification array pointing to matching 3.

As also the *current content of the model* may extend matching 3, we initiate the possible extensions of this matching by the `propagate` method, which checks the existence of at least the `EO` edges leading out of *c1*.[2] As no such edges exist in our example, the algorithm terminates with the matching tree presented in Fig. 5(d).

Step 2. When `EO` edge connecting *c1* to *p* is inserted (as shown by the thick line of Fig. 5(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 Fig. 6.

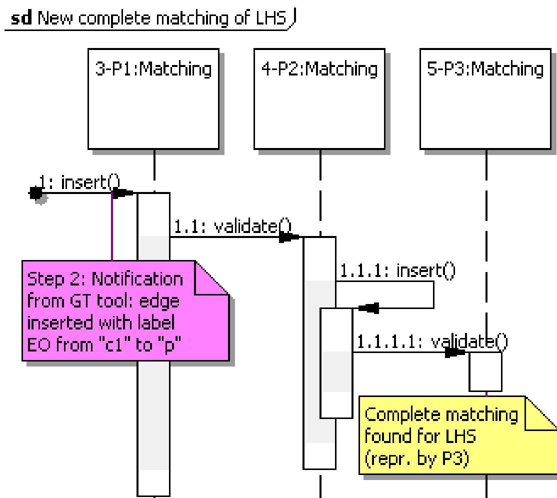


Figure 6: Sequence diagram showing edge insertion into the LHS pattern

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 Fig. 5(f).

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

[2]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.

Step 3. The result of applying the GT rule ClassRule on matching 5 can be observed in Fig. 5(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 Fig. 5(h).

4.2 Insert method

The insert method (shown by Alg. 1) is responsible for the possible extension of the current partial matching for proper subpattern P_k to compute a new partial matching for subpattern P_{k+1}. If the current matching represents a complete matching for pattern P_n, then the method immediately terminates as matchings of pattern P_n can never be further extended.

Algorithm 1 The `insert()` method of class `Matching`

```
public void insert(Constant c) {
  // If the current matching is NOT a complete matching
  if (this.spNode.nextNode != null) {
    // If all condition edges of the next SP node can be matched
    if (checkExistenceOfEdges(c)) {
       // Create a new matching
       Matching newM = new Matching();
       // Copy current matchings to the new matching
       newM.copyMatchings(this, c);
       // New delete entries for matchings of condition edges
       newM.addDeleteEntries();
       if (newM.invalidatedBy.isEmpty()) {
          // Extend the new matching if not invalidated by NAC
          newM.validate();
       } } } }
```

- The insert method is invoked with a constant `c`, which is supposed to be the mapping of the next variable v_{k+1} belonging to search plan in a new *potential matching*, which also contains all mappings defined by the current matching for all variables of subpattern P_k.

- We first check the mappings of the edges for the potential matching. Since the current matching already specifies a graph morphism, we know that all edges of subpattern P_k have been correctly mapped, thus, only mappings of incoming and outgoing condition edges of subpattern P_{k+1} defined by the potential matching are required to be checked by the `checkExistenceOfEdges` method.

- If all edge mappings are correct (and the `checkExistenceOfEdges` returns true), the potential matching can be considered as a new matching for subpattern P_{k+1}. As such, a new matching is created. Then by invoking `copyMatchings` on the new matching (i) mappings of the current matching are cloned, (ii) variable v_{k+1} is bound to `c`, and (iii) the new matching is inserted into the matching tree as a child of the current matching.
- The new matching is added to the delete notification array at all locations defined by the mappings of incoming and outgoing condition edges of subpattern P_{k+1}.
- If the new matching is being invalidated by any complete matchings of any NAC patterns Q_m, then the insert method terminates.
- Otherwise, the `validate()` method is invoked on the new matching trying to recursively extend this matching.

4.3 Validate method

The validate method (shown in Alg. 2) is responsible for the recursive extension of insert operations. It is invoked either (i) when a new matching has been inserted into the matching tree and its further extensions have to be checked (see Alg. 1), or (ii) when extensions of the current matching possibly become valid due to the removal of a complete matching of an embedded NAC pattern (by the `invalidate()` method).

Algorithm 2 The `validate()` method of class `Matching`

```
public void validate() {
  if (this.spNode.nextNode == null) {
    if (this.spNode.pattern.negOf == null) {
      // If this is a COMPLETE matching of a LHS pattern
      // Add to a set of valid matchings of the pattern
      this.spNode.pattern.matchings.add(this);
    } else {
      // If this is a COMPLETE matching of a NAC pattern
      for (Matching m: findInvalidatedMatchings())
        m.invalidate();
    }
  } else {
    // If this is NOT a complete matching
    // Add insert entries
    addInsertEntries();
    // Propagate it to find a matching of the next variable
    propagateInsert();
  }
}
```

- If `this` is a complete matching for a LHS pattern P_n, then the current matching is inserted into the query index structure `this.spNode.pattern.matchings` to be accessed by the GT tool.

- If `this` is a complete matching for a NAC pattern Q_m, then all partial matchings `m` of the LHS pattern that map the shared variable to the same constant as the current matching (which is returned by `findInvalidatedMatchings`) have to be invalidated.

- For each incoming condition edge `e` of the *one larger* subpattern P_{k+1} with label `e.lab` connecting node `e.src` to next variable, the current matching is added to the insert notification array at location `[m[e.src],e.lab,*]` by the `addInsertEntries` method invoked. Similarly, for each outgoing condition edge `e`, the same method adds the current matching to `INSERT[*,e.lab,m[e.trg]]`.

- Insertion is attempted to be propagated to a matching for subpattern P_{k+1}. In this sense, an arbitrary (incoming or outgoing) condition edge `e` is selected from subpattern P_{k+1}. If an outgoing (incoming) condition edge has been chosen, then we lookup all label-preserved model edges `mEdge` leading out of (to) the matched target (source) node `m[e.trg]` (`m[e.src]`) of condition edge `e`, and try to extend the current matching by mapping the next variable `this.spNode.nextNode` to the source (target) node of all chosen model edges `mEdge`, which is represented by the invocation of the `insert` method with constant `mEdge.src`.

4.4 Delete and invalidate methods

Delete and invalidate methods implement the inverse operation of insert and validate methods, respectively.

The `delete()` method removes the whole subtree rooted at the current matching by (i) removing all matchings of the subtree from the notification arrays and the query index structure, and (ii) erasing all "dangling" invalidation links.

The `invalidate()` method deletes the whole subtree *excluding* the current matching, thus, it starts recursive deletion at its children. Another difference is that in case of validate method, the current matching is only removed from the insert notification array, and it remains in the delete notification array.

If the current matching is a complete matching of an LHS then the invalidate method removes the matching from the query index structure. If it is a complete matching of a NAC pattern, then it (re-)validates all matchings invalidated previously by the current matching. On the implementation level, delete and invalidate methods mutually invoke each other, while descending in the tree for recursive matching removal. The Java code for the `delete()` and `invalidate()` methods (as well as all auxiliary methods) are listed in Appendix A.

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.

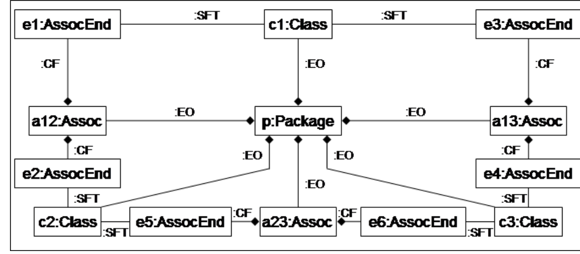


Figure 7: Initial model of the test case for the $N = 3$ case

The structure of the initial model is presented in Fig. 7 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 Table 1.

	Class	Model	TS	Fujaba		Incremental	
		size	length	match	update	match	update
	#	#	#	msec	msec	msec	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

Table 1: Experimental results

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 Table 1 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 a fast updating phase 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), the index structures maintained for partial matchings (i.e., a set of bit vectors) 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 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 and algorithms 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.

References

[BGT91] Horst Bunke, Thomas Glauser, and 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*, volume 532 of *LNCS*, pages 174–189, 1991.

[BV06] András Balogh and Dániel Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Proc. of the 21st ACM Symposium on Applied Computing*, pages 1280–1287, Dijon, France, April 2006. ACM Press.

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

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[ERT99] Claudia Ermel, Michel Rudolf, and Gabriele Taentzer. *In [EEKR99]*, chapter The AGG-Approach: Language and Tool Environment, pages 551–603. World Scientific, 1999.

[FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In Gregor Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, 1998.

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

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Proceedings*, pages 157–166, Washington, D.C., USA, 1993.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[Hud87] Scott E. Hudson. Incremental attribute evaluation: an algorithm for lazy evaluation in graphs. Technical Report 87-20, University of Arizona, 1987.

[KS06] Alexander Königs and Andy Schürr. MDI - a rule based multi-document and tool integration approach. *Journal of Software and Systems Modelling*, 2006. To appear.

[LV02] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.

[MB00] Bruno T. Messmer and Horst Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[PCTM02] John Poole, Dan Chang, Douglas Tolbert, and David Mellor. *Common Warehouse Metamodel*. John Wiley & Sons, Inc., 2002.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. *In [EEKR99]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

[VFV06] Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a graph transformation engine in relational databases. *Journal on Software and Systems Modeling*, 2006. in press.

[VSV05] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Computer Society Press.

[VV04] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In Reiko Heckel, editor, *Proc. of the 4th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2004)*, volume 109 of *ENTCS*, pages 71–83, Barcelona, Spain, December 2004. Elsevier.

[VVF05] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Gabor Karsai and Gabriele Taentzer, editors, *Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05)*, volume 152 of *ENTCS*, pages 191–205, Tallinn, Estonia, September 2005.

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

A Additional Algorithms

```
public void delete() {
  // Remove current matching from parent matching
  this.parent.children.remove(this);
  this.parent = null;
  removeDeleteEntries();
  for (Matching m: this.invalidatedBy) {
    m.invalidates.remove(this);
  }
  this.invalidatedBy.clear();
  invalidate();
}
```

```
public void invalidate() {
  if (this.spNode.nextNode == null) {
    if (this.spNode.pattern.negOf == null) {
      // If this is a COMPLETE matching of a LHS pattern
      // Remove this from valid matchings of the pattern
      this.spNode.pattern.matchings.remove(this);
    } else {
      // If this is a COMPLETE matching of a NAC pattern
      for (Matching m: this.invalidates)
        m.validate();
    }
  } else {
    // If this is NOT a complete matching
    // Remove insert entries
    removeInsertEntries();
    propagateDelete();
  }
}
```

```
public void copyMatchings(Matching currM, Constant c) {
  this.spNode = currM.spNode.nextNode;
  // If the current matching has no submatchings
  if (currM.children.isEmpty()) {
    // The new match is set to the current match
    this.match = currM.match;
  }
  else {
    // The new match is clone from the current match
    this.match = (HashMap<Variable,Constant>)
        currM.match.clone();
  }
  this.match.put(this.spNode.currVar, c);
  // Set new matching as child of the current matching
  this.parent = currM;
  currM.children.add(this);
}
```

```
public void propagateInsert() {
  // Assert that there are unmatched nodes in the search plan
  assert this.spNode.nextNode != null;
  // Select the next node in the search plan
  SearchPlanNode spNext = this.spNode.nextNode;
  // Select an arbitrary (incoming or outgoing) condition edge
  // of the current variable of the SP node
  Edge e = spNext.condEdges.iterator().next();
  // If the pattern edge is an outgoing condition edge
  if (e.src == spNext.currVar) {
    // We lookup the matched target node
    Constant mTrg = match.get(e.trg);
    // For each incoming edge leading to the matched trg node
    for (Edge mEdge: mTrg.in) {
      // which is label-preserving
      if (mEdge.label == e.label) {
        // Extend the matching by mapping the current variable
        // to the source node
        insert((Constant) mEdge.src);
      }
    }
  }
  // If the pattern edge is an incoming condition edge
  else if (e.trg == spNext.currVar) {
    // We lookup the matched source node
    Constant mSrc = match.get(e.src);
    // For each outgoing edge leaving the matched src node
    for (Edge mEdge: mSrc.out) {
      // which is label-preserving
      if (mEdge.label == e.label) {
        // Extend the matching for the current variable
        insert((Constant) mEdge.trg);
      }
    }
  }
}
```

```
public void manipulateInsertEntries(int op) {
  // Assert that there are unmatched nodes in the search plan
  assert this.spNode.nextNode != null;
  // For each condition edge at the NEXT level
  // connected to an already matched node at the CURRENT level
  SearchPlanNode spNext = this.spNode.nextNode;
  for (Edge e: spNext.condEdges) {
    InsertKey key = null;
    if (e.src == spNext.currVar) {
      // We lookup the matched target node
      Constant mTrg = match.get(e.trg);
      // A new insert key is created: [*, e.lab, m[e.trg]]
      key = new InsertKey(mTrg, e.label, InsertKey.TRG);
    }
    else if (e.trg == spNext.currVar) {
```

```
      // We lookup the matched source node
      Constant mSrc = match.get(e.src);
      // A new insert key is created: [m[e.src], e.lab, *]
      key = new InsertKey(mSrc, e.label, InsertKey.SRC);
    }
    if (op == ADD_ENTRY)
      // A new insert entry is created with key
      PatternMatcher.insertEntries.get(key).add(this);
    else if (op == REMOVE_ENTRY)
      // An existing insert entry with key is removed
      PatternMatcher.insertEntries.get(key).remove(this);
  }
}
```

```
public boolean checkExistenceOfEdges(Constant c) {
  // Assert that there are unmatched nodes in the search plan
  assert this.spNode.nextNode != null;
  // Select the next node in the search plan
  SearchPlanNode spNext = this.spNode.nextNode;
  // For all (incoming or outgoing) condition edge of
  // current variable of the SP node
  for (Edge e: spNext.condEdges) {
    // If the pattern edge is an outgoing condition edge
    if (e.src == spNext.currVar) {
      // We lookup the matched target node
      Constant mTrg = match.get(e.trg);
      // For each incoming edge leading to the matched trg node
      for (Edge mEdge: mTrg.in) {
        // which is label-preserving
        if (!(mEdge.label == e.label && mEdge.src == c)) {
          return false;
        }
      }
    }
    // If the pattern edge is an incoming condition edge
    else if (e.trg == spNext.currVar) {
      // We lookup the matched source node
      Constant mSrc = match.get(e.src);
      // For each outgoing edge leaving the matched src node
      for (Edge mEdge: mSrc.out) {
        // which is label-preserving
        if (!(mEdge.label == e.label && mEdge.trg == c)) {
          return false;
        }
      }
    } }
  return true;
}
```