# Practical Issues in the Implementation of Graph Pattern Matching

GERGELY VARRÓ*

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2., Budapest, Hungary
gervarro@cs.bme.hu

**Abstract:** Since graph pattern matching is an NP-complete subtask of graph transformation, its computer-based implementations have to employ heuristics for calculating matchings in an acceptable amount of time on real-world application domains. This paper surveys the notions and techniques used during the automatic generation of application-specific heuristics that may be integrated into an extensible, general sense pattern matching algorithm. In this sense, the overview includes (i) the concept of search plans which are compact representations of heuristics, (ii) their most widespread cost functions, and (iii) the corresponding algorithms that generate low cost search plans.

**Keywords: graph transformation, graph pattern matching, heuristics, search plan**

## 1 Introduction

Graph transformation (GT) [5,9] can be considered as a generalization of Chomsky grammars for graphs. In this sense, it has rules for specifying the manipulation of graph based models in a visual, declarative and formal way, which makes this approach especially useful in computer engineering, which typically employs such models for designing systems. In the process of graph transformation, first a matching occurrence of the left-hand side (LHS) of the GT rule is sought in the model by graph pattern matching. Then the selected part is replaced with the right-hand side (RHS) of the rule. No fast algorithms can be guaranteed to exist for graph transformation as it is NP-hard in general due to its pattern matching task, which leads to the subgraph isomorphism problem, for which NP-completeness has been proven [1].

Mathematicians typically focus on NP-complete problems only until (i) algorithms for the problems are invented and (ii) upper and lower bounds of time- and space-complexity are determined. In computer engineering, when software applications are developed, which aim at providing a solution for an NP-complete problem, the implementation details of the algorithms become at least as important, since users claim to have a good solution in an acceptable amount of time. This demand obviously appreciates the role of heuristics, which are often used to improve the runtime performance of the implemented tool by building application domain specific knowledge in traditional algorithms.

In the current paper, our goal is to give insight into the practical issues of graph pattern matching. More specifically, we focus on how graph-based notations and algorithms can help developing heuristics that improve the runtime performance of pattern matching.

*Research is supported by grant No. 67651 of the Hungarian National Science Fund (OTKA).

After a introduction to graph transformation (Sec. 2), in Sec. 3.1 an overview is given on a general sense, pattern matching algorithm, whose variants have been implemented in several graph transformation tools. Then in Sec. 3.2, we present a graph-based representation of the search space that is being traversed by the pattern matching algorithm. Sec. 3.3 introduces the concept of search plans, which provide a compact way to describe the search space *before* its actual traversal. Furthermore, as the main novelty of the paper, in Sec. 3.4 we survey (i) the most widely used cost functions being defined for search plans, and (ii) algorithms that generate minimum-cost search plans, which can be used to improve the process of pattern matching. Section 4 concludes the paper.

# 2 Graph transformation

I briefly introduce the basic concepts of graph transformation, which include the definition of models, rules, and rule applications.

## 2.1 Basic definitions

A **label** (written with typewriter font in examples and figures) is a basic concept in the application domain that expresses a binary relation among entities. The set of labels (denoted by $\mathfrak{L}$) is fixed in advance by collecting them from the application domain.

**Definition 1** *Given a set of labels $\mathfrak{L}$, an **edge-labelled directed graph** $G = (V_G, E_G, src_G, trg_G, lab_G)$ is a 5-tuple, where $V_G$ and $E_G$ denote nodes and edges of the graph, respectively. Functions $src_G : E_G \to V_G$ and $trg_G : E_G \to V_G$ map edges to their source and target node, respectively. Labelling function $lab_G : E_G \to \mathfrak{L}$ assigns labels to edges.*

In the following, we use the term **graph** for denoting an edge-labelled directed graph.

**Definition 2** *A **morphism $\varphi$ from graph $G$ to graph $H$** (denoted by $\varphi_{(G,H)}$) consists of a node mapping function $\varphi_V : V_G \to V_H$ and an edge mapping function $\varphi_E : E_G \to E_H$, for which the following properties hold:*

- ***Preservation of source nodes:** The source node $src_G(e)$ of each edge $e$ is mapped to the source node $src_H(\varphi_E(e))$ of its image $\varphi_E(e)$. Formally, $\forall e \in E_G : \varphi_V(src_G(e)) = src_H(\varphi_E(e))$.*
- ***Preservation of target nodes:** The target node $trg_G(e)$ of each edge $e$ is mapped to the target node $trg_H(\varphi_E(e))$ of its image $\varphi_E(e)$. Formally, $\forall e \in E_G : \varphi_V(trg_G(e)) = trg_H(\varphi_E(e))$.*
- ***Preservation of labels:** The label of each edge $e$ is preserved by the edge mapping function $\varphi_E$. Formally, $\forall e \in E_G : lab_G(e) = lab_H(\varphi_E(e))$.*

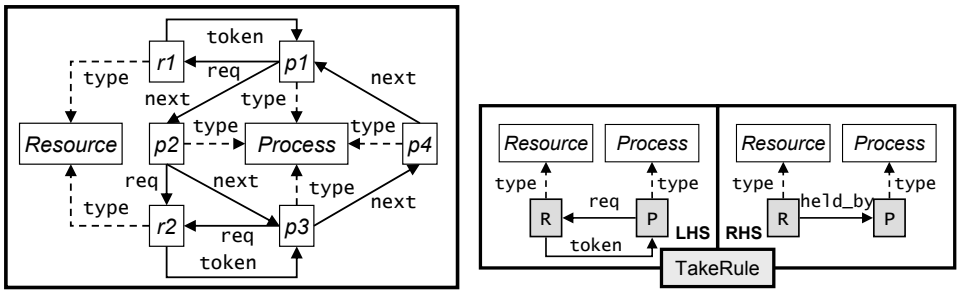
## 2.2 Models and rules

Graph transformation [5,9] provides a pattern and rule based manipulation of graph-based models. Each rule application transforms a graph by replacing a part of it by another graph.

A **model** is a graph that is being manipulated during graph transformation. In the paper, by following the naming conventions of the Unified Modeling Language [11], we use terms **objects** and **links** for the nodes and edges of the model, respectively.

**Example 3** *In order to illustrate the basic terms and concepts of graph transformation, a simplified version of the distributed mutual exclusion algorithm of [7] has been selected as a running example.*

*A sample model in this application domain is depicted in Fig. 1(a).*




(a) A sample model (b) Graph transformation rule TakeRule

Figure 1: Basic constructs of graph transformation

*The model describes a system consisting of 4* Processes *(namely* p1*,* p2*,* p3*,* p4*) and 2* Resources *(*r1 *and* r2*). In this application domain, explicit* `type` *edges are used for representing instantiation. E.g., the* `type` *edge connecting* p1 *to* Process *expresses that* p1 *is a* Process*. Processes* p1*,* p2*,* p3*, and* p4 *constitute a ring along the* `next` *edges. Additionally, a* `token` *edge connects each resource to a corresponding process, which denotes the access right for a process to a given resource. Such edges lead out of resources* r1 *and* r2 *to processes* p1 *and* p3*, respectively. A process can issue a request for a resource by setting a* `req` *edge between the corresponding nodes. In the model, process* p1 *has already asked for resource* r1*, and resource* r2 *has been requested by processes* p2 *and* p3*.*

A **graph transformation (GT) rule** $r$ consists of a left–hand side graph $LHS$, a right–hand side graph $RHS$ and an injective partial morphism $p : LHS \rightarrow RHS$.

In practical graph transformation scenarios, some nodes of the rules are fixed to objects even in the specification by an initial (partial) morphism. In the following, we use the term **constant** (denoted by white boxes in figures and by slanted fonts in texts) for (i) an object in the model, (ii) a node of an $LHS$ graph that has been mapped to an object by the initial morphism, or (iii) a node of an $RHS$ graph that has a constant origin in its corresponding $LHS$. All the other nodes are called **variables** and they are denoted by grey boxes in figures and by sans serif fonts in texts. In this sense, a variable can be (i) a node of an $LHS$ that has not been mapped by the initial morphism, (ii) a node of an $RHS$ without an origin in its corresponding $LHS$, or (iii) a node of the $RHS$ whose origin is a variable in the $LHS$. As a summary, variables can be considered as such nodes of a GT rule, to which an object is assigned during rule application.

**Example 4** *GT rule* TakeRule *is presented in Fig. 1(b). On both sides of the rule, nodes* P *and* R *are variables, while nodes* Process *and* Resource *are constants. The rule expresses that if process* P *has a* `token` *for resource* R*, and it has already issued a request on the same resource (as expressed by the* `req` *edge), then process* P *may get access to resource* R *(as shown by the* `held_by` *edge of the* $RHS$*).*

## 2.3 Rule application

The application of a rule replaces a matching of the $LHS$ in the model by an image of the $RHS$. Rule application consists of two well-separable phases, which are discussed in details in the following.

**Pattern matching.** A **matching for a graph in a model** is an injective total morphism from the graph to the model. In the **pattern matching** phase, a matching for the $LHS$ of a rule is being sought in the model. In the following, the term **pattern** is used as a synonym for the $LHS$ graph of a rule.

Theoretically, pattern matching phase always starts from an empty initial partial matching. However, in practice-oriented graph transformation scenarios, the initial partial matching typically prescribes a rule application context by some mappings, which can be determined a priori for each rule. These are always mappings of the constant nodes of the $LHS$.

Since the mappings of variables in themselves completely determine the partial matching in turn, we omit mappings of edges and constant nodes from matching specifications in the rest of the paper. In this sense, a matching can now be considered as a set of **mappings** of the form (Variable, *Constant*), while pattern matching can be interpreted as a process, in which an object is assigned to each variable of the $LHS$. In tabular representations of partial matchings (e.g., as later in Fig. 3), each column denotes a mapping with a variable and a constant in the upper and the lower row, respectively.

**Example 5** *In our running example, a matching for the $LHS$ of the GT rule TakeRule can be found in the model of Fig. 1(a), when variables P and R are mapped to process p1 and resource r1, respectively. Note that we get another matching solution by mapping the variables to process p3 and to resource r2, respectively.*

**Updating.** At this point, a matching for the pattern has already been calculated, and it designates that part of the model that is going to be modified in the updating phase.

The **updating phase** can be divided into a deletion and an insertion phase.

- In the **deletion phase**, we delete (i) all objects (together with their dangling (i.e., incident) edges), which are assigned to nodes appearing only in the $LHS$, and (ii) all links, which are assigned to edges appearing only in the $LHS$.
- In the **insertion phase**, a new object is added to the model for each node of $RHS$ that is not contained by $LHS$, and similarly, a new link is added to the model for each edge of $RHS$ that cannot be found in the $LHS$ graph. Note that in the latter case source and target objects already exist in the model.

**Example 6** *When the GT rule TakeRule is applied on the matching, which maps variable P to process p1 and variable R to resource r1, the updating phase consists of (i) the deletion of the* `token` *and* `req` *links going between constants p1 and r1, and (ii) the insertion of a* `held_by` *link from r1 to p1.*

Since graph pattern matching leads to the subgraph isomorphism problem, which is NP-complete in general [1], this phase has the strongest influence on the overall performance of graph transformation. As a consequence, only the algorithmic details of pattern matching are discussed in the rest of the paper.

# 3 Pattern matching implementation

Practice oriented considerations of pattern matching are discussed in this section by first introducing a general sense algorithm into which heuristics can be integrated. These heuristics are defined by search plans, which are produced automatically by algorithms that aim at finding a search plan with minimum-cost. This generation process is also presented in this section.

### 3.1 A pattern matching algorithm

As a result of intensive research, several general sense, pattern matching algorithms [3, 12] have been developed for the last decades. In Fig. 2 we present a skeleton that demonstrates the typical structure of these algorithms.

```
 1: PROCEDURE match(k, m)
 2: if m is a total morphism from LHS to model M then
 3:    RETURN m
 4: else
 5:    Compute the set of mapping candidates P(m)
 6:    for all (n, o) ∈ P(m) do
 7:       if check(m, n, o) then
 8:          Compute the morphism m' obtained by adding (n, o) to m
 9:          CALL match(k + 1, m')
10:       end if
11:    end for
12:    Restore data structures
13: end if
```

Figure 2: A skeletal pattern matching algorithm

The pattern matching algorithm consists of a single method `match()` which gets the recursion level $k$ and a morphism $m$ as its inputs (line 1). Procedure `match()` is initially invoked with the initial partial matching, which contains those mappings that are denoted by the constant nodes of the $LHS$. If morphism $m$ represents a complete matching, then it can be returned as a solution (lines 2–3). If morphism $m$ is not yet total (lines 5–13), then attempts are made to extend the morphism. For this reason, a set of mapping candidates $P(m)$ is computed (line 5), and then each candidate $(\mathsf{n}, o)$, which represents the mapping of variable $\mathsf{n}$ to object $o$, is checked by also using the mappings stored by morphism $m$ (lines 6–7). If a mapping candidate passes this test, then it is added to morphism $m$ resulting in a one larger morphism $m'$ (line 8), which is later used as the second input when invoking procedure `match()` recursively in line 9.

Algorithm implementations typically differ from each other in the technique of the computation and the checking of mapping candidates in lines 5 and 7, respectively. In order to be able to analyze the algorithm variants, we need an appropriate description of the search space being traversed by the algorithm of Fig. 2 during pattern matching.

### 3.2 Search space tree

**Search space tree (SST)** is a tree with the following structure. The root of the tree is on the 0th level and it is the initial partial matching (denoted by an empty table) from which pattern matching starts. A node on the $k$th level of the search space tree represents a partial matching, which consists of (i) the mappings defined by the parent node on the $(k-1)$th level, and (ii) a mapping candidate being generated by line 5 at the $k$th level of recursion during the execution of procedure `match()`. Consequently, if a pattern has $l$ variables, then the search space tree has at most $l+1$ levels, and only nodes on the $l$th level may denote complete matchings for the pattern.

**Example 7** *A sample search space tree is presented in Fig. 3. The root of the tree represents the initial partial matching, which contains only mappings of constant nodes. Note that these mappings are not*

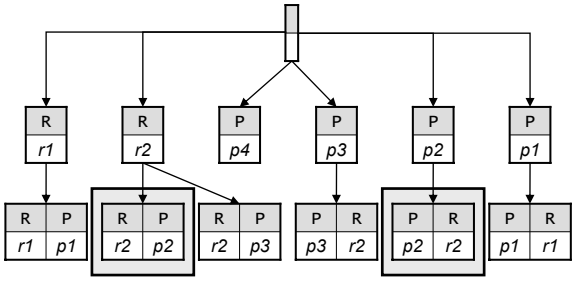



Figure 3: A sample search space tree

*depicted in the figure. On the first level, there are 6 partial matchings, each consisting of a mapping of exactly one variable. Variable R can be mapped to resources r1 and r2, while variable P can be matched to processes p1, p2, p3, and p4. The second level also has 6 nodes of which only 3 represent different matchings as e.g., both framed nodes map variable R to resource r2 and variable P to process p2. It should also be emphasized that the framed matchings are not complete as they have to be filtered out in line 7 by the checking of mapping candidates as no* `token` *edges go from resource r2 to process p2.*

This example has clearly demonstrated that if no restrictions are posed on the mapping candidate selection, then even a single matching with $n$ mappings may cause search space explosion as a consequence of the fact that the matching can be reached on $n!$ paths in the SST. Thus, it is obvious that search space can be reduced by introducing variable ordering restrictions at mapping candidate selection.

Since different variable orderings can produce search space trees with significantly different size, the problem of finding a correct order for variables, which also minimizes the SST, constitutes a critical part in the process of pattern matching as the size of (i.e., the number of nodes in) the search space tree directly affects the runtime performance of a graph transformation engine.

### 3.3 Search plans

Graph transformation tools use clever heuristics being based on knowledge collected from the application domain for determining and improving a variable ordering, which is usually specified in the form of search plans. In the current paper, the search plan notation of [13] is used, but several further sophisticated representations [6, 14] exist as well.

A **search graph** is a graph with non-negative numeric weights as labels. It is generated for each pattern by the following algorithm. Each constant (variable) node of the pattern is mapped to a constant (variable) node in the search graph. Each edge of the pattern is mapped to *a pair of edges* in the search graph that connect the corresponding end nodes in both directions expressing bidirectional navigability. A non-negative numeric weight is assigned to each edge of the search graph. Due to space restrictions, considerations and techniques [6, 8, 13, 14] for weight assignment are not discussed here.

A **search forest** is a spanning forest of the search graph, in which each tree is rooted at a constant node. Consequently, each variable node should be reachable on a directed path from a constant node. Edges of a search forest are denoted by thick lines in Figs. 4(a) and 4(c).

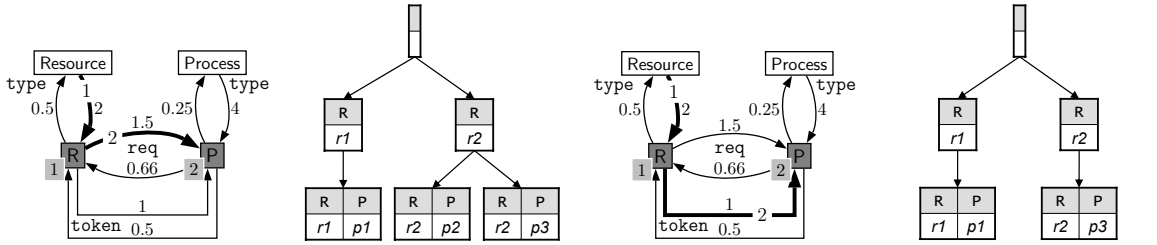
A **search plan** is one possible traversal of a search forest. A traversal defines a sequence in which edges are traversed. The position of a given edge in this sequence is marked by increasing integer numbers written *on* the thick edges in Figs. 4(a) and 4(c). Based on a search plan, a total order of

variables can be specified by assigning the number on the tree edge leading to each variable as its position in the order. This position information is depicted by a number in a light grey box located at the lower left corner of the corresponding variable in Figs. 4(a) and 4(c).

When method `match()` is executed on the $k$th level of recursion, we may specify the corresponding mapping candidate generation (line 5) and filtering (line 7) strategies from the search plan as follows. As a precondition, we may suppose that mappings are available for all the variables whose position is smaller than $k$ according to the order defined by the search plan.

- **Mapping candidate generation.** If the $k$th edge of the search plan represents a navigation direction for an edge of the pattern with label `lab` that connects node $src$ to variable trg in this specific direction, then a mapping (trg, *obj*) is added to mapping candidates $P(m)$ for each object *obj* that can be reached on a `lab` link from the object being assigned to pattern node $src$ by partial matching $m$. If the $k$th edge represents a reverse navigation, then mapping candidates for variable src are generated by navigating `lab` links in an opposite direction from objects being assigned to pattern node $trg$.

- **Mapping candidate filtering.** If the $k$th search plan node $src$ is connected to an already mapped node $trg$ (i.e., whose position is smaller than $k$) by an unselected pair of edges that originate from the same pattern edge with label `lab`, then for each such pair, the existence of a `lab` link between the images of $src$ and $trg$ has to be checked to maintain the consistency of matching candidates.




(a) First search plan

(b) SST of the execution of Fig. 4(a)

(c) Second search plan

(d) SST of the execution of Fig. 4(c)

Figure 4: Search plans and corresponding search space trees

**Example 8** *Two sample search plans and the corresponding search space trees of their execution are depicted in Fig. 4. As shown by the first thick edges, search plans of Fig. 4(a) and 4(c) both prescribe to start with the enumeration of resources, which can be accessed from the object* Resource *by navigating along* `type` *links in outgoing direction. At this step, resources* r1 *and* r2 *are enumerated as represented by the first level of search space trees of Figs. 4(b) and 4(d).*

*The search plan of Fig. 4(a) continues with traversing* `req` *links in backward direction from the already selected resources to produce matchings, in which variable* P *is also mapped to an appropriate process from the model. In this case, 3 matchings can be generated as shown by the second level of Fig. 4(b). These matchings are then filtered by checking whether the newly chosen process is connected to object* Process *via a* `type` *link and to the already selected resource via a* `token` *link. Since no* `token` *links exist between resource* r2 *and process* p2*, the corresponding matching is thrown away. In*

*contrast to Fig. 4(a), the search plan of Fig. 4(c) traverses along* `token` *links as its second step, which results only in 2 matchings as shown by Fig. 4(d). Existence checks for* `req` *and* `type` *links are still executed for the search plan of Fig. 4(c), but no matchings are filtered out. It should be noted that the execution of the second search plan requires a smaller number of matchings to be traversed, thus it can be considered a better search plan.*

A cost is assigned to each search plan to assess the size of search space tree, which would be traversed during the execution of the search plan. If this cost is in strong correlation with the size of the search space tree, then the execution of the minimum-cost search plan yields the fastest algorithm variant for pattern matching. This fact calls the attention to the importance of search plan generation algorithms that aim at finding a minimum-cost search plan in a given search graph. Note that an overall speed-up can only be achieved, if the acceleration of pattern matching caused by the execution of a better search plan can compensate the extra time spent on generating the plan. This requires search plan generation algorithms to run fast.

### 3.4 Search plan generation

Now we survey the most frequent measures being used for characterizing search plans, and the corresponding search plan generation algorithms. In order to have a uniform notation, let $w_k$ denote the weight of the $k$th edge according to the order defined by the search plan. (Note that this edge is used for calculating mapping candidates on the $k$th level of recursion.) Let us further suppose that the search plan consists of $l$ tree edges.

**Sum of weights.** In the first alternative, the cost of a search plan is defined by the sum of weights of the edges that comprise the search plan. Formally, $w_\Sigma(P) = \sum_{k=1}^{l} w_k$. The main strength of the approach is that the Chu-Liu / Edmonds algorithm [2, 4] can quickly generate minimum-cost search forests as the algorithm has a time complexity $O(ne)$ [10], and search graphs have at most a few dozen nodes and edges in a typical application scenario. On the other hand, the cost of a forest is completely insensible to the different orderings of its tree edges, thus, sum-based cost functions provide a poor estimate for the size of the search space tree, which means that even the minimum-cost search plan does not necessarily lead to a fast pattern matching process.

**Product of weights.** The implementation of the GT tool being presented in [6] uses the product of weights as a cost function ($w_\Pi(P) = \prod_{k=1}^{l} w_k$). By taking the logarithm of the cost, we get the sum of the logarithms of weights, which makes this approach similar to the above-mentioned technique both in the applied search plan generation algorithm and in its advantages and disadvantages. This cost function gives a better estimate for the size of the search space tree, but it is still highly insensible to the different orderings of the search forest edges.

**A more complex function.** [13] proposed to calculate the cost function as $w(P) = \sum_{i=1}^{l} \prod_{k=1}^{i} w_k$, which is a correct estimation for the size of the search space tree, if weights of search graph edges denote branching factors, which are collected from the actual model on which graph transformation is performed. The main drawback of this technique is the lack of algorithms, which could find a minimum-cost search plan according to this special cost function. As a consequence, implementations still use the Chu-Liu / Edmonds algorithm as it is fast and gives a search plan with acceptably low cost.

# 4 Conclusion

In the current paper, I surveyed several implementation related issues of graph pattern matching including (i) a general sense pattern matching algorithm description, (ii) a graph-based framework for representing the search space being traversed by this algorithm, and (iii) the concept of search plans by which application-specific heuristics can be compactly described. As main added values of the paper, I gave an introduction to some widely used search plan cost functions, which denote the quality of heuristics by estimating the size of the corresponding search space, and I discussed the related algorithms that generate minimum-cost search plans.

In the future, I plan to generalize the set of search plan operations in order to be able to handle advanced graph transformation related constructs such as alternate, negated and recursive patterns. Another research direction is to develop an algorithm, which can generate minimum-cost search plans for the complex cost function of Sec. 3.4.

# References


[1] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[2] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[3] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, October 2004.

[4] Jack Edmonds. Optimum branchings. *Journal Research of the National Bureau of Standards*, pages 233–240, 1967.

[5] 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.

[6] Rubino Geiß, Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. of the 3rd International Conference on Graph Transformation*, pages 383–397, Natal, Brazil, September 2006. Springer Verlag.

[7] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *Fundamental Approaches to Software Engineering: First International Conference, FASE 1998*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.

[8] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic search plans for matching advanced graph patterns. In *Proc. of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques*, 2007. Accepted paper.

[9] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[10] Alexander Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer Verlag, 2003.

[11] U2-Partners. *UML: Infrastructure v. 2.0 (Third revised proposal)*, January 2003. `http://www.u2-partners.org/artifacts.htm`.

[12] J. R. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[13] 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.

[14] 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.