Electronic Communications of the EASST
Volume X (2007)

Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)

Generic Search Plans for Matching Advanced Graph Patterns

Ákos Horváth, Gergely Varró and Dániel Varró

12 pages

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

Generic Search Plans for Matching Advanced Graph Patterns

Ákos Horváth[1], **Gergely Varró**[2] **and Dániel Varró**[3]

[1] ahorvath@mit.bme.hu
[3] varro@mit.bme.hu
Department of Measurement and Information Systems
Budapest University of Technology and Economics,
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

[2] gervarro@cs.bme.hu
Department of Computer Science and Information Theory
Budapest University of Technology and Economics,
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

Abstract: In the current paper, we present search plans which can guide pattern matching for advanced graph patterns with edge identities, containment constraints, type variables, negative application conditions, attribute conditions, and injectivity constraints. Based on a generic search graph representation, all search plan operations (e.g. checking the existence of an edge, or extending a matching candidate by navigating along an edge) are uniformly represented as special predicates with heuristically assigned costs. Finally, an executable search plan is defined as an appropriate ordering of these predicates. As a main consequence, attribute, injectivity, and negative application conditions can be checked early (but not unnecessarily early) in the pattern matching process to cut off infeasible matching candidates at the right time.

Keywords: graph pattern matching, search plan

1 Introduction

While nowadays model-driven system development is being supported by a wide range of conceptually different *model transformation tools*, nearly all of these tools have to solve a common problem: the efficient query and manipulation of complex graph-like model structures. Tools based on the rule and pattern-based formal paradigm of *graph transformation (GT)* [Roz97, EEKR99] already integrate research results of several decades. In these tools, a matching of the left-hand side (LHS) of a graph transformation rule is being sought by some graph pattern matching algorithm, which might be invalidated by valid matchings of negative application conditions (NAC) [HHT96]. Finally, the engine performs some local modifications to add or remove graph elements to the matching pattern.

Graph pattern matching leads to the subgraph isomorphism problem that is known to be NP-complete in general [Ata99], which means that highly time-consuming computations are expected for the worst-case scenario from theoretical aspects. However, practical model transformation problems rather have a regular and sparse graph structure, which drastically reduces the

execution time of graph pattern matching. In order to provide acceptable performance in real-world application scenarios, graph transformation tools apply sophisticated pattern matching algorithms, which are mostly based on either constraint satisfaction (like AGG [ERT99]) or local searches driven by search plans (like PROGRES [Zün96], Dörr's approach [Dör95], FUJABA [FNTZ98] or GReAT [AKN+06]). As a commonality, all such algorithms have to appropriately order elementary operations (such as navigations and edge existence checks) in advance by using tool-specific heuristics, which later guide the pattern matching process itself.

Research [Zün96, GSR05, GBG+06, VVF05] has been focusing so far on the performance optimal ordering of elementary pattern matching operations like (i) the enumeration of objects and links of a certain type, (ii) the navigation along links of a given type, and (iii) the existence checks for links. On the other hand, the ordering of (iv) attribute, (v) injectivity and (vi) NAC constraint checking operations has been hard wired into the graph transformation engines by using some simple heuristics.

For instance, in case of NAC checking operations, two wiring strategies are known in GT tools. The "*as soon as possible*" (ASAP) style positioning (used by Fujaba) places the NAC checking operation to the first possible location where all its arguments are bound. Intuitively, when the size of the NAC pattern is small compared to the unexplored part of the LHS pattern, a quickly retrieved matching for the NAC may significantly reduce the search space by avoiding the unnecessary traversal of the remaining part of the LHS. On the other hand, when the NAC is large, the corresponding check operation can be time consuming, so a delayed execution may provide better overall performance for the pattern matching of the LHS. This idea is implemented by the "*as late as possible*" (ALAP) strategy (used by PROGRES), which executes NAC checking only when a complete matching for the LHS has been found.

These best engineering practices are acceptable for performing cheap checks like checking attribute and injectivity constraints, but when a single search plan operation represents a complete pattern matching process like in case of checking a NAC or calling native external libraries (as in AGG or VIATRA2), hard-wired positioning may cause performance degradation as (a) it lacks flexibility and extensibility and (b) it ignores the complexity of the actual search plan operation. However, checking NAC is critical to model transformation problems in order to forbid multiple application of a rule on the same matching.

This is a common situation in case of model-specific search plans [VVF05, GBG+06] where the cost of search plan operations depends on the actual graph being transformed. However, even in the case of (traditional) metamodel-specific search plans (like in FUJABA, GReAT or PROGRES), the bindings of input parameters of rules may have a huge impact on the optimal ordering of complex search plan operations. Intuitively, if many input parameters are passed to a rule, the ASAP strategy can be too expensive for complex NACs.

In the current paper, we propose a general framework for uniformly representing a large variety of search plan operations by expressing them as cost-weighted predicates. As an appropriate ordering of these predicates defines an executable search plan, this approach is able to uniformly guide the pattern matching process for advanced graph patterns regardless of how we assign the actual costs to different search plan operations. As a result, better performance is expected, especially, for checking negative application conditions, which avoids the previous problems.

The main practical advantages of our approach are modularity, flexibility, and extensibility. The different phases of pattern matching (e.g. cost assignment, generation of search plans, exe-

cution of search plans etc.) are fully separated and independent, thus they can be adapted to very different graph transformation engines and strategies (metamodel-based vs. model-based search plans). Furthermore, new types of predicates can be introduced easily by assigning appropriate costs without altering the algorithms for search plan generation.

The rest of the paper is structured as follows. First, Section 2 briefly introduces a combined graph-based representation for models and metamodels used in the paper (and in the VIATRA2 framework). Then Section 3 proposes our unified predicate based framework for driving pattern matching processes. Related work is discussed in Section 4, while Section 5 concludes our paper.

2 Background

2.1 Models and Metamodels

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e., abstract syntax) of modeling languages.

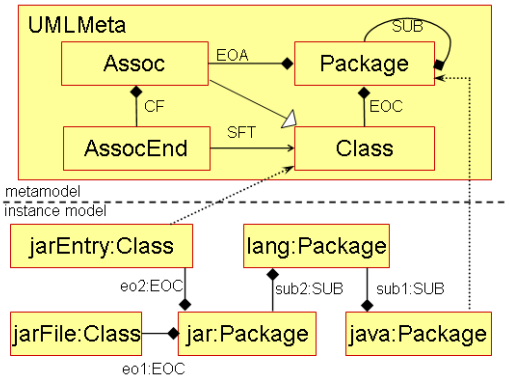
In the paper, we use a unified hierarchical and directed graph representation which stores metamodels and models in a combined model space. Intuitively, the morphisms from instance nodes (and edges) to their respective node (edge) types are stored explicitly in our graph model. This unified graph representation serves as the underlying model of the VIATRA2 framework.

This way, graph nodes (called entities in VIATRA2) uniformly represent MOF packages, classes, or objects on different metalevels, while graph edges with identities (called relations in VIATRA2) denote MOF association ends, attributes, link ends, and slots in a uniform way. As a summary, nodes represent basic concepts of a (modeling) domain, while edges represent the relationships between other model elements. Nodes are arranged into a strict containment hierarchy (to denote model element containment either on the metamodel or model-level).

There are two special relationships between graph elements: the **supertypeOf** (inheritance, generalization) relation represents binary superclass-subclass relationships between nodes or edges, while the **instanceOf** relation represents type-instance relationships (to explicitly represent the meta-levels).

By using explicit **instanceOf** relationship, metamodels and models can be stored in the same model space in a compact way. Furthermore, this allows the use of so-called *generic patterns* in transformation rules (see later in Figure 2), which capture common graph algorithms (e.g. transitive closure, graph searches, etc.) independently of a certain metamodel.

Example 1 *Figure 1 presents the joint representation of a simplified* UML *metamodel and an instance model. The metamodel is contained in the* ***UMLMeta*** *element. Both the classes of the metamodel (such as* ***Package***, ***Assoc***, *etc.) and the objects of the instance model (such as* ***jar***, ***jarEntry***, *etc.) uniformly appear as nodes (entities). Instance-of relation between nodes is also represented by dotted edges (for easier readability not all edges are illustrated). This example illustrates the joint graph representation and also the* VIATRA2 *representation.*




```
//metamodel
entity(UMLMeta){
  entity(Package){
    relation(EOA,Package,Assoc);
    relation(EOC,Package,Class);
    relation(SUB,Package,Package);}
  entity(Class);
  entity(Assoc){
    relation(CF,Assoc,AssocEnd);
    supertypeOf(Assoc,Class);}
  entity(AssocEnd){
   relation(STF,AssocEnd,Class);}
}
//instance model
 Package(java);
 Package.SUB(sub1,java,lang);
 Package(lang);
 Package.SUB(sub2,lang,jar);
 Package(jar);
 Package.EOC(eo1,jar,jarEntry);
 Package.EOC(eo2,jar,jarFile);
 class(jarEntry); class(jarFile);
```

Figure 1: VPM example

2.2 Graph Patterns

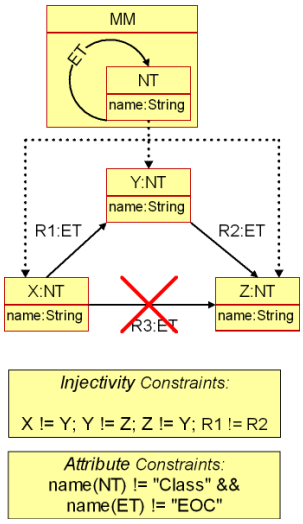
Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformation. A graph transformation rule contains a left-hand side graph LHS, a right-hand side graph RHS, and (one or more) negative application condition graphs NAC connected to LHS. A negative application condition [HHT96] is a graph morphism, which maps the LHS pattern to a *NAC pattern*. 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.

Graph patterns (precondition pattern) consist of the LHS pattern, the NAC pattern, and the mapping between them. They represent conditions (or constraints) that have to be fulfilled by a model in order to execute transformation steps on the model. The most critical step of graph transformation is graph pattern matching, i.e., to find such a matching of the LHS pattern in the model space that is not invalidated by a matching of the negative application condition graph NAC, which prohibits the presence of certain combinations of nodes and edges. So we restrict our current investigations only to graph patterns and graph pattern matching.

Example 2 *An example graph pattern is depicted in Figure 2, where the left side illustrates the graph representation and the right side shows the* VIATRA2 *representation, which is detailed in [BV06]. This pattern is our running example through the paper, as it contains all advance graph pattern elements (e.g., attribute check, nac, etc.), and requires a generic graph representation.*

TransitiveC *pattern is a generic implementation of the transitive closure, which can be used with any metamodel* ***MM*** *passed as a parameter, where* ***NT*** *is matched to the* Node Type *and* ***ET***

to the Edge Type, *which runs between* ***NT*** *type nodes (with a restriction on the name attributes).* ***X*** *represents the inspected element for the transitive closure and* ***Z*** *expresses the closure nodes. The injectivity constraints define that all variables are matched to different elements and the* ***negative*** *application condition expresses that there is no edge* ***R3*** *between the* ***X*** *and* ***Z*** *nodes. For easier readability the explicit "instance of" edges (dotted lines) are only depicted between the* ***NT*** *type element and the* ***X, Y*** *and* ***Z*** *nodes.*




```
pattern transitiveC(X,Z,MM) ={ //meta model
  entity(NT) in MM;
  relation(ET,NT,NT);
  check(name(NT)!="CLASS" && name(ET)!="EOC");
  //instance model nodes
  entity(X);  instanceOf(X,NT);
  entity(Z);  instanceOf(Z,NT);
  entity(Y);  instanceOf(Y,NT);
  X!=Y; Y!=Z; X!=Z;
  //instance model edges
  relation(R1,X,Y); instanceOf(R1,ET);
  relation(R2,Y,Z); instanceOf(R2,ET);
  R1!=R2;
  //nac check
    neg nac1(X,Z,ET)={
    entity(X); entity(Z);
    relation(R3,X,Z); instanceOf(R3,ET);}
}
```

Figure 2: The pattern graph of transitiveC(X,Z,MM)

2.3 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 matching should exist for its NAC pattern.

To drive the pattern matching process, the generation of search plans is a frequently used concept. Informally, a search plan defines the order of traversal (a search sequence) for the nodes of the instance model to check whether the pattern can be matched. The model is traversed according to a specific search plan.

Example 3 *For instance, a matching of the pattern transitiveC of Figure 2 in model Figure 1 with UMLMeta as the input of* ***MM*** *is the following:* ***X*** *= java,* ***Y*** *= lang,* ***Z*** *= jar,* ***R1*** *= sub1,* ***R2*** *= sub2,* ***ET*** *= SUB and* ***NT*** *= Package (****MM*** *= UMLMeta). Where a possible traversal order for the pattern is: (0) MM (it is an input parameter),(1) NT, (2) ET, (3) X, (4) Y, (5) Z, and for the* ***nac1*** *negative application condition is: (1) ET, (2) X, (3) Z.*

3 Unified Search Plan Representation

This section introduces our approach on handling advanced graph patterns. First Subsection 3.1 presents the concept of *search graphs*, which can handle complex constraints on the graph patterns (e.g., containment, generic type parameters etc.). Then Subsection 3.2 introduces the adornment constraints, followed by Subsection 3.3 which proposes our complex constraint cost approximation, and finally, Subsection 3.4 introduces the revised representation of search plans based on the elementary steps of a pattern matching process (search operations).

3.1 Search Graph

A *search graph* is a joint representation of pattern graph elements and operation constraints that drives the pattern matching process. In our interpretation a search graph is a *hypergraph* representing a *constraint net*, where graph nodes reflect variables, and hyperedges express constraints (predicates) between the variables. A search graph is directly derived from the pattern graph as follows:

1. Pattern variable: Each element (node or edge) of the pattern graph is mapped to a *pattern variable*. These elements (depicted by grey ovals, e.g., **X** in Figure 3) represent the arguments of the constraints.

2. Operation Constraint: Each constraint on the pattern graph (containment, connectivity etc.) is mapped to an n-ary (usually binary) operation predicate, (illustrated by rectangles in our figures, e.g., **trg** in Figure 3) that has to be fulfilled during the matching process. The constraints used in our approach are the following (however, the actual set of constraints can be extended easily):

1. **Simple predicates** represent core constraints between two pattern variables.
   - A *source constraint* **src**(Src,E) expresses that node **Src** is the source of edge **E** in the pattern graph.
   - A *target constraint* **trg**(Trg,E) expresses that node **Trg** is the target of edge **E** in the pattern graph.
   - An *'instance of' constraint* **inst**(A,Type) means that **A** is an instance of **Type**, where both elements are represented in the model space.
   - A *containment constraint* **in**(A,Container) expresses that **Container** contains **A**.
2. **Complex predicates** are defined between an arbitrary number of pattern graph elements.
   - The *injectivity constraint* $\mathbf{inj}(A_1,...,A_n)$ means that the A_i must be matched to different graph elements (injective matching).
   - The *negative application condition* $\mathbf{nac}_j(A_1,...,A_n)$ expresses that the check of $\mathbf{nac}_j$ should be initiated with the given input parameters.

- The *attribute check constraint* **attr**$(A_1,...,A_n)$ evaluates a Boolean expression checking, based on the attributes of the pattern nodes, which are accessed by variables $A_1,...,A_n$.

Example 4 *The search graph of Figure 2 is illustrated in Figure 3 (including some parts of the pattern graph itself to improve readability). The search graph contains eight pattern variables;* ***MM, ET*** *and* ***NT*** *represent the metamodel part of the pattern graph, while variables* ***X, Y, Z, R1*** *and* ***R2*** *represent the nodes and edges of the instance model. The operation predicates directly define the constraints of the pattern graph:* ***src, trg*** *define the source and target node of an edge. For example,* ***X*** *is the source of edge* ***R1****,* ***in*** *defines that* ***ET*** *and* ***NT*** *should be contained directly by node* ***MM****,* ***nac1*** *represents the negative application condition with input parameters* ***ET, X*** *and* ***Z****,* ***inj*** *defines the injectivity check between its input variables like between* ***X, Y****, and finally* ***inst*** *represents the direct instance of relations (e.g., between* ***R1*** *and* ***ET****).*

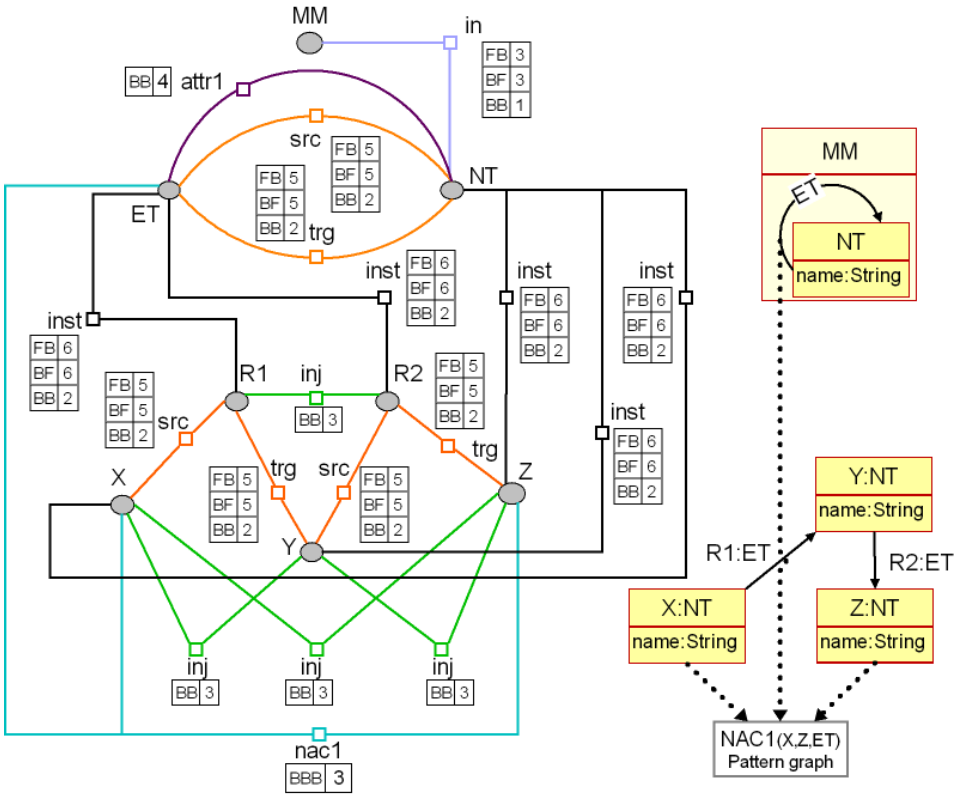


Figure 3: Search graph of GT pattern transitiveC(X,Z,MM)

3.2 Adornment

An operation constraint may represent different concrete search operations depending on the binding of its arguments.

An *adornment* (see in [Ull89]) consists of a string, composed of letters B (bound) and F (free). The meaning of letter B in an adornment is that the variable must be bound to a value in that position. The meaning of the letter F in an adornment is that the variable is not bound in that position.

A *search operation* consists of a constraint and an adornment, they are the atomic units of pattern matching and represents a single step in the matching process. A search operation is either an *extend* type operation which extends the matching by a new element (e.g match the target node along an edge), or a *check* type operation used for checking constraints between pattern elements (e.g., whether an edge runs between two nodes). For example, if adornments FB or BF are attached to a simple constraint, then they represent an extend type search operation, and in case of BB it is a check type operation.

Though 2^n different adornments can be assigned to each n-ary operation constraint in theory, only a subset of these adornments are used, which respect elementary complexity consideration.

Usually for the simple constraints the permitted adornments are FB; BF; BB, while FF represents a far too expensive operation, as we need to cumulate all pairs of elements in the model.

For complex constraints only the B...B adornments are permitted, as all variables must be bound to an element in case of injectivity checking, NAC checking and Boolean term evaluation.

In Figure 3, permitted adornment values are illustrated with small tables near the constraints. The table has two columns, which show the adornment and the cost of the operation (which is discussed in Subsection 3.3), respectively.

3.3 Cost of Search Operations

At this point, a joint representation of search constraints is available. In order to generate efficient search plans cost functions (weights) have to be defined for the operations. Due to space limitations in the paper we are using the more common meta-model based (compile time) weighting and the number values are only used to illustrate the order of magnitude of operation costs, but it is important to mention that the approach is also capable of handling model (runtime) based weighting. The only differences are in case of runtime weighting are (i) the usage of runtime statistics collected from the instance model, (ii) and the more precise weighting of *simple extend* type predicates (e.g., the weight of an *'instance of' constraint* is based on *actual number* of the instance elements in the model), and *nac check* predicates, where the cost can be directly derived from the pattern graph of the *nac*.

Weighting the *simple operation* follows the guidelines of edge multiplicity based cost functions (e.g., if an edge multiplicity is one-to-many, then its cost is higher then if it is one-to-one) with the following restriction: the lowest cost is assigned to the BB adornments (*check* type operation), and there is no difference between the cost of FB and the BF (*extend* type operation). Among the constraints we use the cost ordering based on our earlier transformation experiments and [VSV06]: $in << trg = src << inst$

In case of complex constraints, assigning costs to operations is easier on one hand as they have only one permitted adornment B...B, but on the other hand better cost prediction is possible using a priori knowledge. In case of *inj* and *attr* constraints the number of input parameters provides a good prediction for complexity, while in case of a *nac* constraint the whole *nac* pattern

graph matching cost can be evaluated at compile time. The cost functions are the following:

- For *inj* and *attr* constraint the cost function is linear in the number of parameters.
- For *nac* constraint the cost function is proportional to the number of constraints in the search graph of the *nac* pattern. The idea behind this selection is that a *nac* check may cut the search space significantly when the *nac* pattern is small.

3.4 Search Plans

A *search plan* is a totally ordered list of search operations (one possible traversal of the search graph). As the atomic operations already have cost values attached, we can evaluate the cost of a whole search plan.

The *cost of a search plan* (denoted by $w(P)$) is defined by the formula $w(P) = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j$, where w_j is the weight of the jth operation according to the order defined by the search plan, and n is the number of operation constraints in the original search graph. As described in [VVF05], this formula is an estimation for the size of the search space that has to be traversed during pattern matching, if model-specific search graphs are used and the weight w_j expresses the expected number of iterations performed during the execution of the jth operation. As a consequence, the search plan with the minimum cost $w(P)$ will have the best expected run-time performance. If the weights are fixed and determined at compile-time, this cost function is still an acceptable choice as the search plan that is optimal wrt. $w(P)$ prefers the early execution of low cost operations. For generating the actual search plans, only minor modifications to the techniques described in [VVF05] are needed.

Example 5 *The following example shows two search plans of the running example pattern graph: Figure 4(a) represents the search plan where* ***X,Z*** *and* ***MM*** *parameters are bound, while Figure 4(b) shows the search plan where only* ***X*** *and* ***MM*** *are the fixed input parameters.*

In case of Figure 4(a), the nac_1 *is in the first quarter of the search plan, which means it is an ASAP like positioning, and checks the negative application condition before the extend operation towards the* ***R1,R1*** *and* ***Y*** *variables. While in case of the search plan depicted in Figure 4(b) the* nac_1 *check is in the end as in case of an ALAP like ordering.*

This simple example shows, that the joint search plan representation is capable of handling different positioning for advance constraints, which (like in case of checking a NAC) could result better search plans, compared to hard-wired positioning.

4 Related work

All graph transformation tools use some clever strategies for pattern matching. Since an intensive research has been focused to graph transformation for a couple of decades, several powerful methods have already been developed. First we focus on the three most advanced compiled approaches that use search plan guided and local search based algorithms.

Fujaba [KNNZ00] performs local search starting from the node selected by the system designer and extending the matching step-by-step to neighbouring nodes and edges. Fujaba fixes

constraint	type	comment
$inj^{BB}(Z,X)$	check	Z not equals to X
$in^{FB}(NT,MM)$	extend	NT under MM
$inst^{BB}(X,NT)$	check	X is instance of NT
$inst^{BB}(Z,NT)$	check	Z is instance of NT
$trg^{FB}(ET,NT)$	extend	target of ET is NT
$src^{BB}(ET,NT)$	check	source of ET is NT
$nac_1^{BBB}(X,Z,ET)$	check	nac_1 check
$attr_1^{BB}(ET,NT)$	check	$attr_1$ is evaluated
$trg^{FB}(R2,Z)$	extend	target of R2 is Z
$inst^{BB}(R2,ET)$	check	R2 is instance of ET
$src^{BF}(R2,Y)$	extend	source of R2 is Y
$inst^{BB}(Y,NT)$	check	Y is instance of NT
$inj^{BB}(Y,Z)$	check	Y not equals to Z
$inj^{BB}(X,Y)$	check	X not equals to Y
$trg^{FB}(R1,Y)$	extend	target of R1 is Y
$src^{BB}(R1,X)$	check	source of R1 is X
$inst^{BB}(R1,ET)$	check	R1 is instance of ET
$inj^{BB}(R1,R2)$	check	R1 not equals to R2

(a) X,Z and MM are fixed input parameters

constraint	type	comment
$in^{FB}(NT,MM)$	extend	NT under MM
$inst^{BB}(X,NT)$	check	X is instance of NT
$trg^{FB}(ET,NT)$	extend	target of ET is NT
$src^{BB}(ET,NT)$	check	source of ET is NT
$attr_1^{BB}(ET,NT)$	check	$attr_1$ is evaluated
$src^{FB}(R1,X)$	extend	source of R1 is X
$inst^{BB}(R1,ET)$	check	R1 is instance of ET
$trg^{BF}(R1,Y)$	extend	target of R1 is Y
$inst^{BB}(Y,NT)$	check	Y is instance of NT
$inj^{BB}(X,Y)$	check	X not equals to Y
$src^{FB}(R2,Y)$	extend	source of R2 is Y
$inst^{BB}(R2,ET)$	check	R2 is instance of ET
$inj^{BB}(R1,R2)$	check	R1 not equals to R2
$trg^{BF}(R2,Z)$	extend	target of R2 is Z
$inst^{BB}(Z,NT)$	check	Z is instance of NT
$inj^{BB}(Y,Z)$	check	Y not equals to Z
$inj^{BB}(Z,X)$	check	Z not equals to X
$nac_1^{BBB}(X,Z,ET)$	check	nac_1 check

(b) X and MM are fixed input parameters

Figure 4: Search plans of transitiveC

a single traversal strategy at compile-time for each rule by automatically generating Java code for the pattern matching process. During code generation Fujaba places attribute, injectivity and NAC checks to the earliest allowed location. As a consequence, the corresponding run-time operations are executed immediately after all the necessary variables have been fixed, which means that this engine implements a hard-wired as soon as possible strategy.

PROGRES [Zün96] uses the advanced concept of operation graphs for representing structural constraints on the ordering of basic operations, which are similar to search graphs in the current paper. Costs of search plan operations are defined by using a very sophisticated application domain independent cost model. PROGRES can assign weights to attribute checking operations, which enables their proper scheduling in search plans. On the other hand, injectivity and NAC checks are excluded from the cost model, which results in their hard-wired positioning at runtime. Injectivity constraints are tested as soon as all its arguments are known, while negative application conditions are checked late, i.e., when a complete matching for the pattern has been found. Compared to our approach, PROGRES supports navigation along indexed attributes in addition to attribute checking.

The pattern matching engine of GReAT [AKN+06] only allows injective matchings whose corresponding constraints are checked in an 'as soon as possible' style just like attributes, which are tested immediately whenever a new partial matching has been calculated as a result of an extension of a smaller matching. In GReAT, negative application conditions can only be expressed by zero cardinality edges, which normally restricts the size and complexity of NACs, but reduces expressiveness obviously.

Algorithms that handle pattern matching as a constraint satisfaction problem (CSP) like [LV02]

in AGG [ERT99] do not directly involve the concept of search plans as stated in [VSV06]. However, the underlying constraint solver engine has to define a variable binding order, which can be considered as a search plan derived dynamically at run-time. As a consequence, CSP-based graph transformation engines by their nature support that dynamicity that has been achieved by our approach for local search based algorithms. However, as constraint solver implementations typically use the first-fail principle for determining the variable binding order, this technique still schedules the attribute, injectivity and NAC checking operations to the earliest possible location.

Altogether, we believe that our current contribution is in the increased generality and modularity of the search graphs, and in the more flexible handling of negative application conditions. This way, the current paper is complementary to (and can be integrated with) the recent advances in model-specific search plans (as in GrGen [GBG+06] or [VVF05]).

5 Conclusion

In the current paper, we have presented a general framework for uniformly representing search plan operations. The essence of the approach is to express the operations as cost weighted predicates and assign the weights based on the binding of their input parameters. We used adornments to capture binding constraints on the predicates and introduced a compile-time weighting for a variety of advance pattern graph elements. The implementation based on these techniques will be available in the new VIATRA2 release.

In the future, it will be interesting to apply our solution on recursive graph patterns, where recursive call represents a new predicate in the search graph, which does not have any restrictions on its adornment, making cost assignment complicated.

Acknowledgements: The paper is partially supported by the SENSORIA European IP (IST-3-016004).

Bibliography

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *Software and Systems Modeling* 5(3):261–288, September 2006.

[Ata99] M. J. Atallah (ed.). *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[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.

[GBG+06] R. Geiß, V. Batz, D. Grund, S. Hack, A. M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of the 3rd International Conference on Graph Transformation*. 2006. Accepted paper.

[GSR05] L. Geiger, C. Schneider, C. Reckord. Template- and Modelbased Code Generation for MDA-Tools. In Giese and Zündorf (eds.), *Proc. of the 3rd International Fujaba Days*. Pp. 57–62. Paderborn, Germany, September 2005. ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf.

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

[KNNZ00] T. Klein, U. Nickel, J. Niere, A. Zündorf. From UML to Java And Back Again. Technical report, University of Paderborn, 2000.

[LV02] J. Larrosa, G. Valiente. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical Structures in Computer Science* 12(4):403–422, 2002.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1: Foundations. World Scientific, 1997.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Volume II: The New Technologies. Computer Science Press, 1989.

[VSV06] G. Varró, A. Schürr, D. Varró. Experimental Evaluation of Optimization Techniques in Graph Transformation Tools by Benchmarking. *Software and Systems Modeling*, 2006. Submitted paper.

[VVF05] G. Varró, D. Varró, K. Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai and Taentzer (eds.), *Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05)*. ENTCS 152, pp. 191–205. Tallinn, Estonia, September 2005.

[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.