Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans [1]

Gergely Varró [2] Katalin Friedl [4]

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

Dániel Varró [3]

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

Abstract

The current paper makes two contributions for the graph pattern matching problem of model transformation tools. First, model-sensitive search plan generation is proposed for pattern traversal (as an extension to traditional multiplicity and type considerations of existing tools) by estimating the expected performance of search plans on typical instance models that are available at transformation design time. Then, an adaptive approach for graph pattern matching is presented, where the optimal search plan can be selected from previously generated search plans at run-time based on statistical data collected from the current instance model under transformation.

Key words: graph transformation, adaptive graph pattern matching, search plans.

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

[1] This work was funded by the Péter Bizáky Puky Scholarship and the János Bolyai Scholarship.
[2] Email: gervarro@cs.bme.hu
[3] Email: varro@mit.bme.hu
[4] Email: friedl@cs.bme.hu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

graph-based model structures. Tools based on the visual, rule and pattern-based formal paradigm of *graph transformation (GT)* [13,5] already integrate research results of several decades. Informally, a graph transformation rule performs local manipulation on graph models by finding a matching of the pattern prescribed by its left-hand side (LHS) graph in the model, and changing it according to the right-hand side (RHS) graph.

A recent survey [18] assessing the performance of graph transformation tools following essentially different approaches on various benchmark examples revealed that approaches (such as Fujaba [7], PROGRES [14] or GReAT [1]) *compiling transformation rules into native executable code* (Java, C, C++) are very powerful for model transformation purposes. The performance of the executable code is optimized at compile time by evaluating and optimizing different *search plans* [21] for the traversal of the LHS pattern, which typically *exploits the multiplicity and type restrictions* imposed by the metamodel of the problem domain.

While in many cases, types and multiplicities provide a very powerful heuristics to prune the search space, in practical model transformation problems, one has further domain-specific knowledge on the potential structure of instance models of the domain, which is typically not used in these approaches. Furthermore, in case of intensive changes during the evolution of models, the characteristic structure of a model may change as well, therefore a search plan generated a priori at compile time might not be flexible and powerful enough.

The current paper makes two contributions for the graph pattern matching problem of model transformation tools. First, model-sensitive search plan generation is proposed for pattern traversal (as an extension to traditional multiplicity and type considerations of existing tools) by estimating the expected performance of search plans on typical instance models that are available at transformation design time (Sec. 3). Then, an adaptive approach for graph pattern matching is presented, where the optimal search plan can be selected from previously generated search plans at run-time based on statistical data collected from the current instance model under transformation (Sec. 4).

It is worth emphasizing that the concepts of the first technique is directly applicable to furtherly fine-tune the performance of the above mentioned compiler-based GT approaches, while we believe that the second technique is a step towards efficient incremental model transformation engines where the consistency of several models need to be maintained while the different models are being manipulated.

Overview of the Approach

The proposed workflow of using these techniques is summarized in Fig. 1.

Optim First, typical models of the domain are collected (from transformation designers, end users, etc.) from which the optimizer generates one search plan with the best average performance for each typical model.

Cdgen Still at transformation design time, executable (object-oriented) code is generated as the implementation of each search plan.

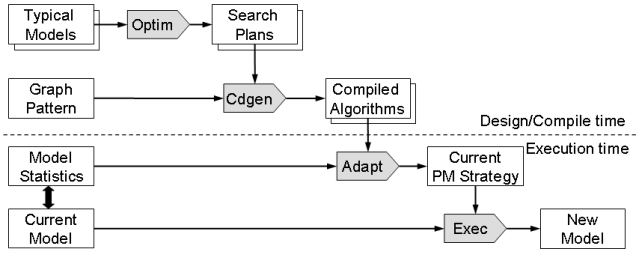


Fig. 1. Overview of the Approach

Adapt At execution time, statistical data is collected on-the-fly from the current model under transformation. Based on this data, a pattern matching strategy (i.e. the implementation of a search plan) is selected which yields the best expected performance cost. It is important to ensure that model statistics causes little memory overhead, and the cost of each pre-compiled search plan can also be estimated rapidly.

Exec Finally, the transformation rule is applied on the instance model using the selected pattern matching strategy.

The current paper focuses on steps *Optim* and *Adapt*, while the detailed discussion of step *Cdgen* (see [2]) is out of scope for the current paper.

2 Model manipulation by graph transformation

We first briefly introduce the main notions of metamodels and models, and then show how these models can be manipulated by using graph transformation.

2.1 Metamodels

In order to present the concepts of models, metamodels and transformations, a standard object relational mapping (see e.g. [16]) will be used throughout this paper as a running example, which generates a relational database schema from a UML class diagram.

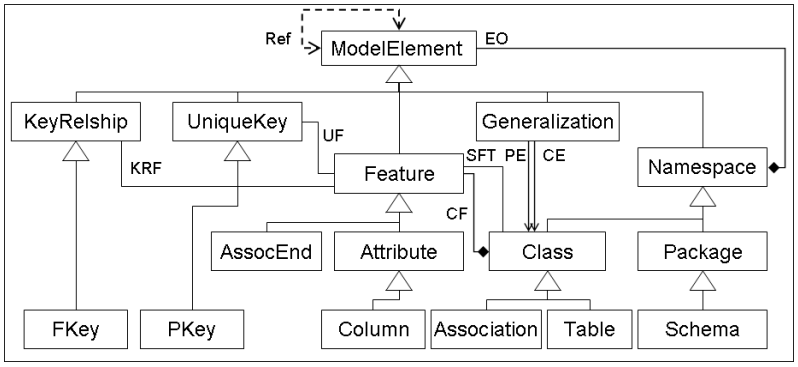


Fig. 2. An extended metamodel for the object relational mapping

The *metamodel* describes the abstract syntax of a modeling language, which can

be formally represented by a type graph. The metamodels of UML class diagrams and relational database schemas (following the CWM standard [12]) are depicted in Fig. 2. Nodes (e.g. Schema, Table) of the type graph are called *classes*. A class may have *attributes* that define some kind of properties of the specific class. (E.g. the name of a table could be an attribute, but it is not depicted in Fig. 2.) *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra attributes. Note that the CWM standard derives database notions like tables, columns, etc. from UML notions by inheritance (see Fig. 2).

Associations like EO, CF, KRF and UF define connections between classes. Both ends of an association may have a *multiplicity* constraint attached to them, which declares the number of objects that, at run-time, may participate in an association. We consider the most typical multiplicity constraints, which are (i) the at most one (denoted by arrows or diamonds), and (ii) the arbitrary (denoted by line ends without arrows and diamonds). Furthermore, we use one-to-one reference edges (denoted by bidirectional dashed lines in instance models) connecting source and target model nodes. Finally, we assume without the loss of generality that both ends of associations are navigable.

2.2 Instance models with statistics

The *instance model* (or, formally, an instance graph) describes concrete systems defined in a modeling language and it is a well-formed instance of the metamodel. Nodes and edges are called *objects* and *links*, respectively. Objects and links are the instances of metamodel level classes and associations, respectively. Attributes in the metamodel appear as *slots* in the instance model. Inheritance in the instance model imposes that instances of the subclass can be used in every situation, where instances of the superclass are required.

Example 2.1 A well-formed instance model of this domain (shown in Fig. 4(a)) has 4 instances of class Class. Generalization nodes g1 and g2 express that c1 is a superclass of c2 and c2 is a superclass of c3, respectively. Class c2 has already been transformed by the object-relational mapping algorithm, which means that table t2 is attached to class c2 via an edge of type Ref. Table t2 has a single column cl2 and a primary key constraint p2 referring to the column cl2.

We assume that the total number of nodes of a certain type (denoted by #type) are maintained during the evolution of each model. Furthermore, we establish a similar counter for edges of a certain type leading between nodes of the corresponding types (denoted by #t_edge(t_src,t_trg)). For instance, in Fig. 4(a), #Class = 5 (note that tables are classes as well according to the CWM metamodel), and #CE(Generalization,Class) = 2. This model statistics will be heavily used both for the search plan optimization and the search plan adaptation steps. Note that the overhead caused by the maintenance of this data is relatively cheap: one option is to use class-level (static) attributes and methods. In fact, in many cases, such details are already provided by the execution environment (e.g. a relational DBMS, if

models are persisted in a database).

2.3 Graph transformation

Graph transformation [13,5] provides a pattern and rule based manipulation of graph-based models. Each rule application transforms a graph by replacing a part of it by another graph.

A *graph transformation rule* $\mathsf{r} = (\mathsf{LHS}, \mathsf{RHS}, \mathsf{NAC})$ contains a left–hand side graph LHS, a right–hand side graph RHS, and negative application condition graph NAC [9]. The LHS and the NAC graphs are together called the precondition PRE.

In the paper, we use the graphical representation initially introduced in [7] where the union of these graphs is presented. Elements to be deleted are marked by the del keyword, elements to be created are labelled by the new, while elements in the NAC graph are denoted by the neg keyword.

The *application* of r to a *host (instance) model* M replaces a matching of the LHS in M by an image of the RHS. This is performed by (i) finding a matching of LHS in M (by graph pattern matching), (ii) checking the negative application conditions NAC (which prohibit the presence of certain objects and links) (iii) removing a part of the model M that can be mapped to LHS but not to RHS yielding the context model, and (iv) gluing the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) obtaining the *derived model* M'. A *graph transformation* is a sequence of rule applications from an initial model M_I.

Example 2.2 A single graph transformation rule (GeneralizationR in Fig. 3(a)) is selected as an example for the paper, which handles the inheritance of classes. The rule expresses that if there is already a database table (with a primary key column) related to both the parent and the child class, then the rule becomes applicable, and a new foreign key constraint is generated to express that the identifier of (an instance of) a child should also be found in the table of the parent. The entire object relational mapping formalized as graph transformation rules can be found in [19].

3 Search plan generation

It is well-known that the most critical step for the performance of graph transformation is the graph pattern matching phase. The pattern matching is determined by only the precondition of a graph transformation rule, so we restrict our current investigations only on this part of GT rules.

For this purpose, the generation of *search plans* is a frequently used and efficient strategy. 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 search space traversed according to a specific search plan is represented as a *search space tree (SST)* which contains all the decisions that can be made at a certain point during pattern matching. The root node of a SST represents a partial matching as provided by fixing the input parameter nodes of rules. Each path of a

SST starting from the root node extends this partial matching by the matching of a fresh (unmatched) node in the pattern.

In the current section, we present a model-specific search plan generation technique in the following way.

(i) First , we introduce the concept of search graphs to obtain an easy to manage representation of GT rule preconditions in Sec. 3.1.

(ii) Based on the statistics of typical models, model-specific search graphs are prepared by adding numerical weights on the edges of search graphs (Sec. 3.2).

(iii) The concept of search plans is defined together with a cost function that helps estimating the performance of search plans and formulating when a search plan is optimal in Sec. 3.3.

(iv) Finally, two algorithms are presented that implement the generation of low cost search plans for model-specific search graphs (Sec.3.4).

3.1 Search graphs

In the first phase of the search plan generation process, a search graph is created for each pattern.

A *search graph* is a directed graph with the following structure. Each node of the pattern is mapped to a node in the search graph. We also add a *starting node* to the graph.

(i) Directed edges connect the starting node to every other search graph nodes. When such an edge is selected in the search graph for a certain search plan, then the graph pattern matching engine executing this search plan needs to iterate over all objects in the model of the corresponding type.

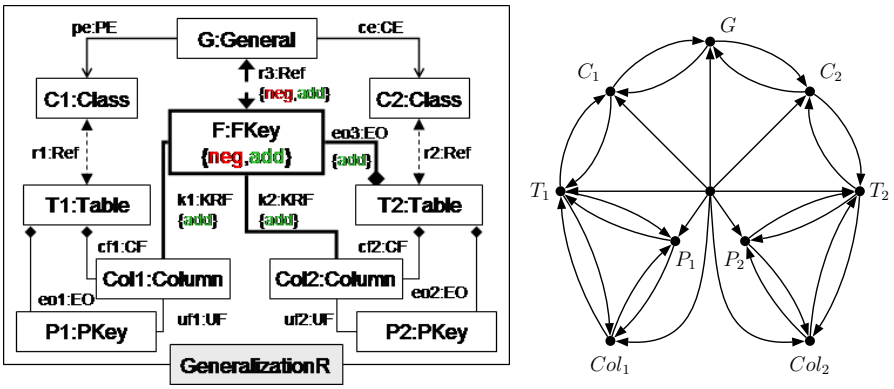
(ii) Each edge of the pattern is mapped to *a pair of edges* in the search graph that connect the corresponding end nodes in both directions expressing bidirectional navigability. [5] A such edge can be selected by the pattern matching engine only when the source pattern node is already matched. In this case, the selection of such a search graph edge means a navigation along the corresponding pattern edge towards the unmatched (target) pattern node.

Search graphs for negative application conditions can be handled similarly. In this case, all the matched nodes (i.e. the ones that are shared with LHS graphs) have to be considered as starting nodes. Negative application conditions are typically checked after a complete matching has been found for the LHS, but simple checks (e.g. like testing whether edges leaving the shared nodes in the NAC has zero cardinality) can be immediately performed as soon as shared nodes are processed during the traversal of LHS.

The graph transformation rule of Ex. 2.2 and its corresponding search graph are depicted in Fig. 3(a) and 3(b), respectively. Note that nodes and edges of the pattern

[5] In case of navigability restrictions, only the navigable direction is generated.

with add (or del) annotation have no corresponding elements in the search graph as they denote nodes and edges to be added (or removed) in the updating phase.



(a) GeneralizationRule

(b) Search graph for GeneralizationRule

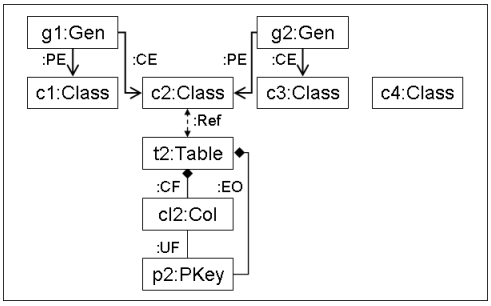
Fig. 3. A sample graph transformation rule and its corresponding search graph

3.2 *Model-specific search graphs*

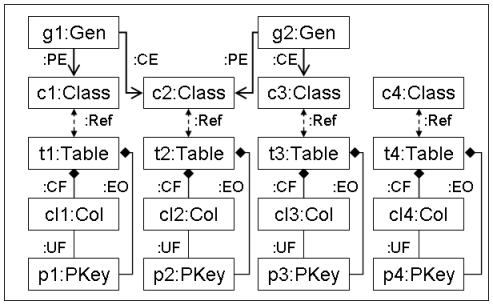
The initial step for search plan generation takes typical models from the problem domain, e.g. typical UML class diagrams and corresponding database schemas in our case. Node and edge statistics of these typical models are also available, so weights can be defined for the edges of the search graph based on the statistical data collected from a model.

A *weighted search graph* is a search graph with numeric weights on its edges. (Weights are depicted as labels of edges in Figs. 4(c) and 4(d).) Informally, the weight of an edge can be considered as an average branching factor of a possible SST at the level, when the pattern matching engine selects the given pattern edge for navigation. Such a choice for edge weights provides an easy to calculate cost function that estimates the size of the search space.

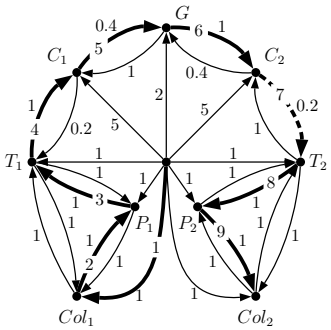
Two models and their corresponding weighted search graphs are depicted in Fig. 4. The weight calculation rule is demonstrated on the edge of Fig. 4(c) (denoted by a dashed line), which corresponds to the traversal of pattern edge r2 of type Ref in the Class-to-Table direction. According to our statistics, Model1 contains 5 Classes (since a Table is a Class in CWM) and 1 reference edge between Classes and Tables, respectively. As a consequence, if the pattern matching engine matches a Class to the pattern node C2 at some time during the execution, then the probability to find a valid Table for pattern node T2 by navigating along a reference (Ref) edge is 0.2 derived by the formula #Ref(Class,Table)/#Class. In case of navigation in the opposite direction, the formula can be expressed as #Ref(Class,Table)/#Table, thus the corresponding weight is 1.



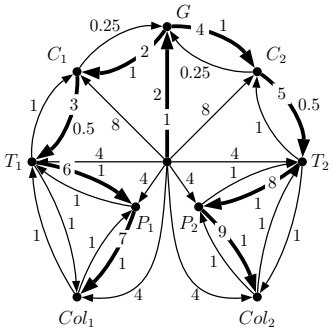
(a) Model1



(b) Model2



(c) Weighted search graph generated for the GeneralizationRule from the statistics of Model1 and a possible search plan



(d) Weighted search graph generated for the GeneralizationRule from the statistics of Model2 and a possible search plan

Fig. 4. Sample instance models and corresponding search plans

3.3 Search trees and plans

At this point, a weighted search graph is available for each typical model selected by the domain engineer. In this section, first, we introduce the concept of search trees and search plans based on weighted search graphs. Then a cost function is defined for search plans to predict its performance.

A *search tree* is a spanning tree of the weighted search graph. As the starting node has no incoming edges, all other nodes should be reachable on a directed path from the starting node. [6] Edges of a search tree are denoted by thick lines in Figs. 4(c) and 4(d).

A *search plan* is one possible traversal of a search tree. 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(c) and 4(d). Two sample search plans (with their corresponding search trees) are shown in Fig. 4(c) and 4(d).

The *cost of a search plan* (denoted by $w(P)$) is the estimated number of traversed nodes in the corresponding search space tree (SST). The number of nodes at the ith depth-level of the SST is the product of branching factors of edges up to the

[6] The search tree concept can be generalized to handle the completion of partially matched patterns. The generalized concept allows several starting nodes. In this case, a search tree is a forest rooted at starting nodes and they should ensure reachability for all other nodes on edges of trees.

level i in the search plan, which is $\prod_{j=1}^{i} w_j$, where w_j is the weight of the jth edge according to the order defined by the search plan. The total number of nodes can be calculated by summing the nodes of the SST on a level-by-level basis, which yields to a formula $w(P) = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j$. By using this cost function for the search plan of Fig. 4(c) on the model of Fig. 4(a) and for the search plan of Fig. 4(d) on the model of Fig. 4(b), we get cost values 5.04 and 8.5, respectively.

As weights denote branching factors, the minimization of a search plan with such a cost function results in a SST that is expected to be optimal in size. Moreover, such a search plan fulfills the first-fail principle criteria as it represents a SST that is narrow at the levels near to its root.

3.4 Algorithms for finding optimal search plans

Two traditional greedy algorithms are adapted to solve the problems of finding (i) a low cost search tree for a given weighted search graph and (ii) a low cost search plan for a given search tree. Note that traditional algorithms use a different cost function (i.e. the sum of weights) for determining the cost of a spanning tree, which means that their solutions are not necessarily optimal in our case.

For *finding a minimum search tree in a weighted search graph*, the Chu-Liu / Edmonds algorithm [3,4] is used. This algorithm searches for a spanning tree in a directed graph that has the smallest cost according to a cost function defined as the sum of weights. This algorithm can be outlined in Alg. 1.

Algorithm 1 *Given a weighted search graph with a starting node.*
Step 1: Discard the edges entering the starting node.
Step 2: For each other node, select the incoming edge with the smallest weight. Let the selected $n-1$ edges be the set S.
Step 3: If there are no cycles formed by the edges of S, then the selected edges constitute a minimum spanning tree of the graph and the algorithm terminates. Otherwise the algorithm continues.
Step 4: For each cycle formed, contract the nodes in the cycle into a pseudo-node k, and modify the weight of each edge entering node j in the cycle from some node i outside the cycle according to the following equation.

$$c(i,k) = c(i,j) - (c(x(j),j) - min_l\{c(x(l),l)\})$$

where $c(x(j),j)$ is the weight of the edge in the cycle which enters j.
Step 5: For each pseudo-node, select the entering edge, which has the smallest modified weight. Replace the edge, which enters the same real node in S by the new selected edge.
Step 6: Go to step 3 with the contracted graph.

In case of *finding a low cost search plan in a given search tree*, a simple greedy algorithm is used, which is sketched in Alg. 2.

We do not state currently that these simple algorithms provide optimal solutions also for our cost model, but best engineering practice suggests that if edges with

Algorithm 2 *Given a search tree with a starting node.*
Step 0: Set the counter to 1 and let S be the set consisting of the starting node.
Step 1: Select the smallest tree edge e that goes out from S.
Step 2: Set the label of e to the value of the counter.
Step 3: Increment the counter by 1 and add the target node of e to S.
Step 4: If the search tree still has a node that is not in S, then go back to Step 1.

weights giving the minimum sum are selected, then the search tree and the search plan consisting of the same edges also have low cost when our special cost function is employed. Simplicity and speed are further arguments in favour of the successful application of such algorithms.

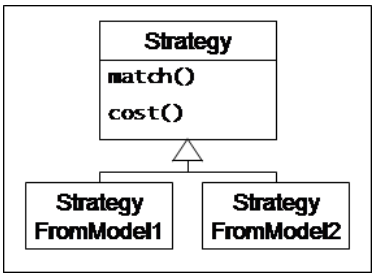
4 Adaptive Pattern Matching

Based on different typical models, several search plans were elaborated in Sec. 3. Then, for each search plan, a separate Java class is generated by a standard compilation phase resulting in an executable Java code. A detailed description of this code generation step can be found in [2].

We now present how this (or related) compiled graph transformation approach can be made adaptive. Our solution for code generation uses the Strategy design pattern [8] (see the figure below), which means that each Java class generated from a search plan extends an abstract Strategy class, which has two basic functionalities.

(i) One method implements the actual pattern matching algorithm, which basically consists of a set of loops embedded into each other as defined in [2].

(ii) The other relevant functionality is the calculation of cost for the given pattern matching strategy based on the statistics of the actual instance model available at run-time. It is worth emphasizing that the complexity of cost calculation is linear in the size a graph pattern.



The latter functionality suggests the way how an adaptive behaviour can be achieved. A `GraphTransformation` class is created, which maintains references to strategies available for a given rule and to the instance model. When a rule is to be applied, this central class invokes the cost calculation method of each strategy of the rule in turn by also passing the actual model. Then the costs of these strategies are compared and the strategy with the smallest cost is executed. Since the cost of a single search plan may vary depending on the current instance model, the relationship between costs of different strategies may change as transformation progresses.

Example 4.1 For illustrating adaptivity, let us consider that Model1 of Fig. 4(a) is evolved into Model2 of Fig. 4(b) as a result of some other rule applications.

Initially, when Model1 is active, costs of search plans are 5.04 and 5.2, respec-

tively, thus, the first plan is selected for execution. Since the GeneralizationRule is not applicable for Model1, the failure of pattern matching is obviously detected by both pattern matching strategies. On the other hand, note that the first strategy recognizes earlier that the pattern cannot be matched.

When the model has been evolved into Model2 by applying some other rules, a new situation appears, since search plan costs are now 19.5 and 8.5, respectively. As a result, the pattern matching engine executes the second strategy for this model.

It is worth pointing out that the target platform of our adaptive technique is not traditional batch model transformations with explicit control structures (i.e. a rule is to be applied as a well-defined step in the transformation flow). Our intention is to exploit this technique for incremental model transformation providing the consistent on-the-fly maintenance of models between multiple domains. Here, a large set of rules should be applied independently at any phase of model evolution where the optimal pattern matching strategy for a rule may vary during this evolution. By following the first-fail principle, our approach facilitates the early detection of pattern matching failure, which is highly demanded in incremental model transformations as the applicability of several rules has to be quickly determined. On the other hand, this technique provides no solution for incremental query evaluation.

5 Related work

All graph transformation based 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.

While many graph transformation approaches (such as [11] in AGG [6], VIATRA [17]) use algorithms based on *constraint satisfaction*, here we focus on the three most advanced compiled approaches with *local searches using search plans*.

Fujaba [10] performs local search starting from the node selected by the system designer and extending the matching step-by-step by neighbouring nodes and edges. Fujaba fixes a single, breadth-first traversal strategy at compile-time (i.e. when the pattern matching code is generated) for each rule. Fujaba uses simple rules of thumb for generating search plans. A sample rule is that navigation along an edge with an at most one multiplicity constraint precedes navigations along edges with arbitrary multiplicity.

PROGRES [21] uses a very sophisticated cost model for defining costs of basic operations (like enumeration of nodes of a type and navigation along edges). These costs are not domain-specific in the sense that they are based on assumptions about a typical problem domain on which the tool is intended to be used. Operation graphs of PROGRES, which are similar to search graphs in the current paper, additionally support the handling of path expressions and attribute conditions. The compiled version of PROGRES generates search plan at compile-time by a greedy algorithm, which is based on the a priori costs of basic operations.

The pattern matching engine of GReAT [20] uses a breadth-first traversal strat-

egy starting from a set of nodes that are initially matched. This initial binding is referred to as pivoted pattern matching in GReAT terms. This tool uses the Strategy design pattern for the purpose of future extensions and not for supporting different pattern matching strategies like in our approach.

Object-oriented database management systems also use efficient algorithms [15] for query optimization, but their strategy significantly differs as queries are formulated as object algebra expressions, which are later transformed to trees of special object manager operations during the query optimization process.

6 Conclusions

In the current paper, we first proposed a model-sensitive approach for generating search plans for compiled graph transformation approaches. The essence of the technique is to use a priori knowledge obtained from typical designer models. A weighted search graph is derived from statistical data taken from these models. Low cost search plans are defined by tailoring well-known greedy algorithms for the cost function of a traversal.

Then we proposed an adaptive technique for switching between different pattern matching strategies by exploiting run-time model statistics using the Strategy design pattern. We expect to use this approach for incremental transformations that consistently maintain models taken from different domains.

In the future, we plan to carry out a thorough experimental evaluation of our approach. Initial experiments show that model-sensitive search plans perform at least as well as existing approaches, but it is too early to make firm statements on performance issues.

References

[1] Agrawal, A., G. Karsai, Z. Kalmar, S. Neema, F. Shi and A. Vizhanyo, *The design of a simple language for graph transformations*, Journal in Software and System Modeling (2005), in review.

[2] Balogh, A., G. Varró, D. Varró and A. Pataricza, *Generation of platform-specific model transformation plugins*, submitted to ECMDA 2005.

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

[4] Edmonds, J., *Optimum branchings*, Journal Research of the National Bureau of Standards (1967), pp. 233–240.

[5] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Handbook on Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools," World Scientific, 1999.

[6] Ermel, C., M. Rudolf and G. Taentzer, "In [5], chapter The AGG-Approach: Language and Tool Environment," World Scientific, 1999 pp. 551–603.

[7] Fischer, T., J. Niere, L. Torunski and A. Zündorf, *Story diagrams: A new graph rewrite language based on the Unified Modeling Language*, in: G. R. G. Engels, editor, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, LNCS **1764** (1998), pp. 296–309.

[8] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995, 1st edition.

[9] Habel, A., R. Heckel and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae **26** (1996), pp. 287–313.

[10] Klein, T., U. Nickel, J. Niere and A. Zündorf, *From UML to Java and back again*, Technical report, University of Paderborn (2000).

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

[12] Poole, J., D. Chang, D. Tolbert and D. Mellor, "Common Warehouse Metamodel," John Wiley & Sons, Inc., 2002.

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

[14] Schürr, A., A. Winter and A. Zündorf, "In [5], chapter PROGRES: Language and Environment," World Scientific, 1999 .

[15] Straube, D. D. and M. T. Özsu, *Query optimization and execution plan generation in object-oriented data management systems*, Knowledge and Data Engineering **7** (1995), pp. 210–227.

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

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

[18] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, in: *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, USA, 2005, pp. 79–88.

[19] Varró, G., A. Schürr and D. Varró, *Benchmarking for graph transformation*, Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (2005), http://www.cs.bme.hu/~gervarro/publication/TUB-TR-05-EE17.pdf.

[20] Vizhanyo, A., A. Agrawal and F. Shi, *Towards generation of efficient transformations*, in: G. Karsai and E. Visser, editors, *Proc. of 3rd Int. Conf. on Generative Programming and Component Engineering (GPCE 2004)*, LNCS **3286** (2004), pp. 298–316.

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