PERFORMANCE OPTIMIZATION OF EXHAUSTIVE RULES IN GRAPH REWRITING SYSTEMS

Tamás Mészáros, Márk Asztalos, Gergely Mezei and Hassan Charaf
Department of Automation and Applied Informatics, Budapest University of Techology and Economics
Goldmann György tér 3, Budapest, Hungary

Keywords: Model transformation, Graph rewriting, Model-driven engineering, Exhaustive matching.

Abstract: Graph rewriting-based model transformation is a well known technique with strong mathematical background to process domain specific models represented as graphs. The performance optimization techniques realized in today's graph transformation engines usually place focus on the optimization of a single execution of the individual rules, and do not consider the optimization possibilities in the repeated execution. In this paper we present a performance optimization technique called deep exhaustive matching for exhaustively executed rules. Deep exhaustive matching continues the matching of the same rule from the next possible position after a successful rewriting phase, thus we can achieve noticeable performance gain.

1 INTRODUCTION

Domain-specific modeling (DSM) is a powerful technique to describe complex systems in a well-understandable yet precise way. The strength of DSM lies in its property, that the language itself is specialized for a concrete application domain, therefore it can efficiently describe systems in the domain.

Such models may be the source of other models, source code or even documentation, therefore, efficient tools are required to perform the conversion between the different formats. As domain-specific models are often represented by directed, attributed graphs, a possible method to perform the conversion is graph transformation (Ehrig et al., 2006). Latest modeling environments also start to incorporate graph transformation engines for model processing. It is essential however, to reduce the execution time of the transformations to save development time.

A graph transformation system in the mathematical sense consists of a set of graph production rules that describe the individual modification steps on the host graph. A production rule consists of two parts: the left-hand side (LHS) pattern that must be found in the host graph, and the right-hand side (RHS) that replaces the found pattern during the execution. Searching for the LHS pattern is usually called the *matching phase* of the rules, while the replacement of the LHS is often referred to as the *rewriting phase*.

The execution time of the rewriting phase can be considered constant (i.e. it does not depend on the size of the host graph). Thus, the bottleneck of a rule application is the matching phase: it determines the speed of the overall transformation. Unfortunately, the problem of matching an isomorphic subgraph is NP complete in the size of the subgraph. However, heuristic methods produce reasonably fast solutions.

Although, there are several different graph transformation tools available (Varró et al., 2006; Geiß et al., 2006; Zündorf, 1996), they follow the same rule application principle: the execution of the production rules is completely separated. After the execution of a production rule, the information regarding the match is thrown away, the matching phase of the same or next production rule starts from scratch.

In this paper, we present a novel approach, that optimizes the execution speed of *exhaustive*ly executed rules. Deep exhaustive matching optimizes the execution of a rule by not restarting the matching from scratch after a successful rewriting, but continuing it from the next potential point. Exhaustive execution means, that the same rule is executed as long as a match can be found. In our approach, we assume deterministic rule application: the rules follow each other according to a strict order while the matches of an individual rule are chosen arbitrarily.

Mészáros T., Asztalos M., Mezei G. and Charaf H. (2010).
PERFORMANCE OPTIMIZATION OF EXHAUSTIVE RULES IN GRAPH REWRITING SYSTEMS.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 292-295
DOI: 10.5220/0003008402920295
Copyright © SciTePress

2 BACKGROUND

In this section, we present the necessary definitions and the execution semantics of graph rewriting rules. The basic definitions of typed-graphs, *E*-graphs, attributed typed graphs are understood according to (Ehrig et al., 2006).

Definition 2.1. ***(Match).*** *The match m of L subgraph in the G host graph is an injective morphism* $m : L \to G$*.*

We transform graphs with attributes, thus instead of graph productions we use typed attributed graph productions. These productions consist of graphs attributed over a term algebra $T_{DSIG}(X)$ with variables X (Ehrig et al., 2006).

Definition 2.2. ***(Graph Production).*** *Given an attributed type graph ATG with a data signature DSIG. A typed attributed typed graph production* $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, X)$ *consists of typed attributed graphs L, K, R with a common DSIG-algebra* $T_{DSIG}(X)$ *and a set of constraints* X*. Also, we have injective* l, r*.*

A production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R, X)$ *is* applicable *to a typed attributed graph G via the match m if there exists a context graph D such that* (1) *is a pushout and m satisfies constraints X, i.e.* $m \models X_s$*.*

$$\begin{tikzcd}
L \arrow[d, "m"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dl, phantom, "(1)"] & R \\
G & D \arrow[l, "f"] &
\end{tikzcd}$$

In our implementation the set of constraints X_s is expressed using the Object Constraint Language (OCL) (OMG, 2006).

In the following we illustrate traditional matcher and rewriting algorithms for single and exhaustive rule execution. Recall that the matching phase can be considered as an isomorphic subgraph searching problem, with the extension, that the nodes and the edges of the graph are typed and attributed, furthermore, constraints (e.g. OCL (OMG, 2006)) can be assigned to both the individual elements and to the entire match as well.

The matching algorithm itself is usually very simple. It iterates through the elements of the pattern graph in an appropriate order, and searches for a matching element satisfying the following conditions for each of them. (i) The type of the matched element should be equal to (or compatible with - in case of inheritance) the type of the pattern element, (ii) it should satisfy the individual constraints of the related pattern element, and (iii) of course, the found element should be connecting to the already matched parts to ensure the edge preserving property of the mapping between the host graph and the pattern graph. Condition (iii) expresses, that if a node and an edge is connecting in the pattern graph, then their images in the host graph should be connecting as well.

The execution of a typical exhaustive rewriting rule is illustrated by Algorithm 1.

Algorithm 1. Exhaustive rewriting rule execution.

```
while $EXECUTE_p(G)$ do
  nop

function $EXECUTE_p(G)$
  for all $e_1 \in T_1(G) : \gamma_{p^1}(e_1)$ do
    for all $e_2 \in T_2(G) : \gamma_{p^2}(e_2)$ do
      ...
      for all $e_n \in T_n(G) : \gamma_{p^n}(e_n)$ do
        if $\gamma_p(< e_1, ..., e_n >)$ then
          $REWRITE_p(G, < e_1, ..., e_n >)$
          return true
      ...
  return false
end function
```

$EXECUTE_p$ implements the p rewriting rule executed over the G host graph. The algorithm consists of nested cycles: one cycle is assigned for each p^i element in the pattern graph. $T_i(G)$ denotes those elements of the G host graph, the type of which is compatible with the p^i pattern element in the p rule. $\gamma_{p^i}(e_i)$ summarizes the conditions (ii) and (iii). I.e. it returns *true*, if e_i satisfies the individual constraint assigned to it in the p rule, and if e_1 .. e_i elements preserve the edges of p. If a matching element has been found for each element of the pattern graph, then the global constraint for the pattern is verified ($\gamma_p(< e_1, ..., e_n >)$), and the rewriting is performed ($REWRITE_p(G, < e1, ..., e_n >)$). In case of exhaustive execution, the rewriting algorithm is applied in a loop as long as a match can be found.

Recall that most graph transformation engines apply an algorithm similar to the presented one. The main differences in the implementations are the data structures they use to represent the graphs, and the order of the pattern elements processed during the matching. The order of the pattern elements is often referred to as the *search plan* of the rule. The search plan highly influences the overall speed of the matching phase.

3 DEEP EXHAUSTIVE EXECUTION

A drawback of Algorithm 3 is that after finding a match for p, and performing the rewriting phase, the $EXECUTE_p$ function immediately returns, and the current state of the $forall$ cycles is dropped. A straightforward idea is to discover, when and which cycles could continue searching provided that the remaining matches are still valid matches in the host graph.

The rewriting phase ($REWRITE_p$) of a rule may modify the host graph in the following six ways, or in their combination:

1. Create a new element, that cannot be matched by either p^i pattern element. (φ_1)
2. Modify the attributes of an existing e_i element without influencing the evaluation of either γ_{p^j} or γ_p. (φ_2)
3. Delete an e_i element from the host graph. (φ_3)
4. Modify the attributes of a matched e_i element, and changing thus γ_{p^i} or γ_p from $true$ to $false$. (φ_4)
5. Create a new element, that can be matched by either p^i. (φ_5)
6. Modify the attributes of a matched e_i element, with the possibility to change either γ_{p^j} or γ_p that was previously $false$ to $true$ (not only for the current match, but all the other matches as well). (φ_6)

Note, that φ_x properties are verified not for a specific $< e_1, ..., e_n >$ match and G host graph, but independently from them. The matcher algorithm is created based on the transformation definition only, thus, no further overhead is necessary at runtime. However, it will not be able to exactly evaluate each property without the host graph and the match. We have created a reference implementation that verifies the φ_x properties. It is capable of handling primitive cases only (with simple attribute modifications, and attribute relations). If the heuristics cannot evaluate φ_x unambiguously, then in case of $\varphi_4, \varphi_5, \varphi_6$ we assume, that $REWRITE_p$ has those properties. This is important not to loose matches or find invalid matches.

φ_1 and φ_2 are irrelevant for the matcher algorithm. Cases φ_3 and φ_4 express, that the current $< e_1, ..., e_n >$ match became invalid, because e_i has been changed (or deleted) in a way, that it cannot be part of a valid match anymore. Thus, because $e_{i+1}...e_n$ are matched after e_i, they are considered to be invalid as well. If there are several such e_i elements, and the one with the smallest i index is e_{min}, then $< e_1, ..., e_{min-1} >$ partial match can still be extended to a valid $< e_1, ..., e_{min-1}, e_min{in}', ..., e_n' >$ match. This idea is called *deep matching* and is illustrated by Algorithm 2.

Algorithm 2. Exhaustive rewriting rule execution with deep matching.

```
while $EXECUTE_p(G)$ do
    nop

function $EXECUTE_p(G)$
    foundOne := false
    for all $e_1 \in T_1(G) : \gamma_{p^1}(e_1)$ do
        ...
        for all $e_{min} \in T_{min}(G) : \gamma_{p^{min}}(e_{min})$ do
            ...
            for all $e_n \in T_n(G) : \gamma_{p^n}(e_n)$ do
                if $\gamma_p(< e_1, ..., e_n >)$ then
                    $REWRITE_p(G, < e_1, ..., e_n >)$
                    foundOne := true
                    goto nextTurn
            ...
            label nextTurn:
        ...
    return foundOne
end function
```

The difference with the original exhaustive algorithm is, that after performing the rewriting phase, instead of leaving the function, the control jumps to the end of the forall cycle body for matching e_{min}. The $< e_1, ..., e_{min-1} >$ partial match is kept, and the matching continues with the following element for p^{min}.

φ_5 and φ_6 express, that the modification of the host graph may produce further matches by creating new elements or modifying attributes of existing elements. If $REWRITE_p$ has the φ_5 or φ_6 properties, then no further modification of Algorithm 2 is needed, as the possible new matches are discovered in the next turn of the top level while cycle.

3.1 Example

Let us consider the hostgraph and the rewriting rule illustrated on Figure 1.

The rule matches a p^1 element of type A connected to a p^2 element of type B (for the sake of simplicity, the edges are not typed and labeled in this case), and deletes the p^2 node and the connecting edge. This rule is executed in an exhaustive manner. Using Algorithm 1, the cycle for p^1 would match e.g. a_1, the cycle for p^2 would match l_1 finally, p^3 would match b_1. Then, after deleting l_1 and b_1, the matching would start from scratch, and p^1 would match a_1

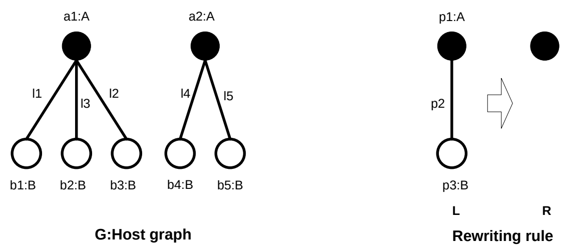

Figure 1: Deep exhaustive execution example.

again. In contrast, Algorithm 2 does not exits the $EXECUTE$ function, instead, it continues the cycle matching the p^2 edge, and finds e.g. l_2 and so on.

4 RELATED WORK

Incremental pattern matching (Varró and Varró, 2004) is an online technique implemented in *VIATRA2* (Varró et al., 2006) to store parts of matches and to reuse them in another rewriting rules within the same transformation. Their approach uses sophisticated data structures and algorithms to store matches and to efficiently update the data structures during the transformation. The drawback of this approach is that it performs much computation at runtime, and the overhead of maintaining these data structures in complex cases may make the transformation even slower compared to the non-optimized version. *VIATRA2* uses a cost model to generate a search plan for the execution of a rule. Cost calculation is based on the creation of a *search graph*, which models the basic operations of the matching as nodes, and the weight of the connecting edges is derived from the number of potential backtracks during matching an element. The search plan is calculated by finding a minimum directed spanning tree in the search graph.

The cost model of GrGen.NET (Geiß et al., 2006) extends the solution of VIATRA2 by considering the cost of performing a primitive step in the matching as well. Furthermore, instead of optimization by searching for a minimum directed spanning tree in the *plan graph*, the solution searches for a minimum multiplicative directed spanning tree, minimizing thus, the product of the costs instead of their sum, thus, they can reach more accurate solutions.

The *PROGRES* (Zündorf, 1996) approach is similar to that of VIATRA2 and GrGen.NET. However, cost calculation does not take into account the statistics of the current host graph or the properties of the metamodel of the host graph, but uses the assumptions based on a typical domain the tool is planned to be used on.

5 CONCLUSIONS

In this paper, we have presented an optimization technique for exhaustively executed rewriting rules. Deep exhaustive execution accelerates the matching phase of a single rule. It is based on the idea, that if the same rule is executed repeatedly, in certain circumstances the matching can be continued from a later point in the algorithm without starting the complete process anew. Although, we cannot present the measurement results due to space limitations, depending on the type of the host graph and the transformation rules we have achieved even more than an order of magnitude decrease of the execution time. Our future research interest includes elaborating more precise heuristics to discover the applicability of the technique: deep exhaustive execution requires more sophisticated solutions to evaluate the still valid parts of the match after the rewriting phase.

REFERENCES

Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Geiß, R., Batz, G. V., Grund, D., Hack, S., and Szalkowski, A. (2006). Grgen: A fast spo-based graph rewriting tool. In *ICGT*, pages 383–397.

OMG (2006). *Object Constraint Language, version 2.0*. http://www.omg.org/technology/documents/formal/ocl.htm.

Varró, G. and Varró, D. (2004). Graph transformation with incremental updates. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 109 of *ENTCS*, pages 71–83. Elsevier.

Varró, G., Varró, D., and Friedl, K. (2006). Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Karsai, G. and Taentzer, G., editors, *GraMot 2005, International Workshop on Graph and Model Transformations*, volume 152 of *ENTCS*, pages 191–205. Elsevier.

Zündorf, A. (1996). Graph pattern matching in progres. In *Selected papers from the 5th International Workshop on Graph Gramars and Their Application to Computer Science*, pages 454–468, London, UK. Springer-Verlag.