Speeding up Graph Transformation through Automatic Concatenation of Rewrite Rules

Jens Müller and Rubino Geiß

Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
{mueller|rubino}@ipd.info.uni-karlsruhe.de

Abstract. The execution of graph transformations is sped up by automatically concatenating rewrite rules. We start by guessing which elements are already matched or created by one rule, and are re-used by the following rule afterwards. Then, we build a rule that combines the modifications of both rules, leaving out unneccesary intermediate steps. We use such combined rules to transform *graph rewrite sequences*, including a fallback for the case that the guess was wrong. Using this method, we achieve a speedup of nearly 50 % in Varró's well-known mutex benchmark.

1 Introduction

Our system, GrGen.NET, is a graph transformation tool optimized for performance and practical applicability. It provides automatic optimization of matching strategies based on an analysis of the host graph. Thus, hand-coding of graph pattern matching is not necessary at all (see section 2). It it based on directed typed multi-graphs and SPO semantics, with extensions such as negative application conditions. Furthermore, it allows the execution of so-called graph rewrite sequences (GRS). This enables the composition of several rules and provides logical and iterative sequence control.

We noticed that GRS often contain sequences of rules that have to be executed repeatedly in an enclosing loop, where one rule re-uses elements already processed by the previous rule.

Our contributions are:

- We build rules that combine the modifications made to the host graph by two separate rules. These rules are only (but not necessarily) applicable when the separate ones would be.
- We transform our graph rewrite sequences using the newly created rules, including a fallback for the case that the heuristically determined mapping of re-used graph elements is not applicable. Thus, we save repeated matching of elements and creation of temporary graph elements, maintaining correctness (see section 3).

We apply our optimization to the well-known mutex benchmark by Varró et al., and achieve a speedup of nearly 50 % (see section 4).

2 GrGen.NET

GrGen, the Graph Rewrite GENerator [1], is a generative programming system for graph rewriting. Fig. 1 gives an overview of the GrGen system components.

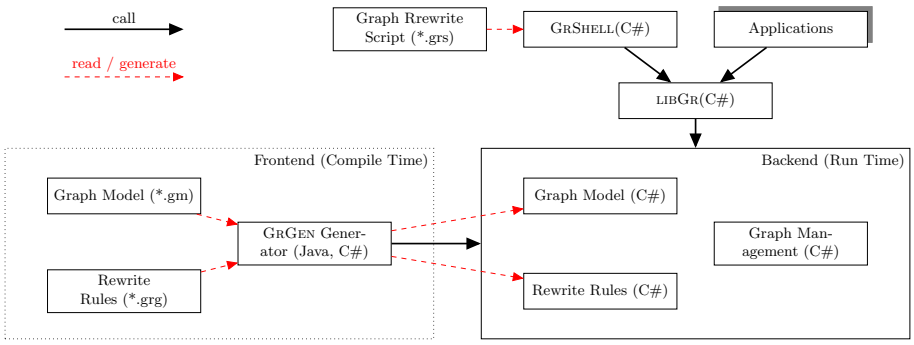

Fig. 1. GrGen system components [2]

GrGen's graph meta-model supports directed multi-graphs with typed nodes and edges. The pattern language supports isomorphic matching (injective mapping), homomorphic matching (possibly non-injective mapping) for selectable sets of nodes or edges, and parameter passing to rules. The rewrite language allows to specify rewrites either as changes to the match or as an replacement of the whole match. In both cases, the semantics is mapped to SPO.

The basis for a graph rewrite system[1] are a rule set specification and zero or more graph model specifications, that are translated into binary software libraries that can be used from applications such as GrShell.

We define the notion of search plans to represent different matching strategies. [3,4,5] The system chooses a good search plan based on a graph analysis and a cost model. When he knows that the graph has changed substantially, the user can request a new searchplan to be generated, but no further manual action or annotation is required.

GrGen also supports rules with negative application conditions (NACs) [6]. NACs are patterns that must not be present in the host graph, otherwise the rule is not applicable. They are independent from the normal pattern graph, except for preset elements that must be mapped to the same host graph element as the corresponding element in the pattern graph.

3 Optimizations on Graph Rewrite Sequences

Table 1 lists some possible graph rewrite expressions (that can be part of graph rewrite sequences) at a glance.

[1] Here, this term means a set of software components

`s ; t`	**Concatenation.** First, `s` is executed, afterwards `t` is executed. The sequence `s ; t` is *successfully* executed iff `s` or `t` is successfully executed.
`s | t`	**XOR.** First, `s` is executed. Only if `s` fails, `t` is executed. The sequence `s | t` is successfully executed iff `s` or `t` is successfully executed.
`s & t`	**Transactional AND.** First, `s` is executed, afterwards `t` is executed. If `s` or `t` fails, the action will be terminated and a rollback to the state before `s & t` is performed.
`s *`	Executes `s` repeatedly as long as its execution does not fail.
`s {n}`	Executes `s` repeatedly as long as its execution does not fail, but anyway `n` times at most.
`Rule`	Only the first pattern match produced by the action *Rule* will be rewritten.
`true`	A constant acting as a successful match.
`false`	A constant acting as a failed match.

Let `s`, `t` be graph rewriting sequences, and `n` $\in \mathbb{N}_0$.

Table 1. Graph rewriting expressions

3.1 The Problem

We notice that there are loops (`s *` and `s {n}`) that might be executed quite often. Thus, the contents of loops are an appropriate target for optimization efforts since the cost for (possibly expensive) optimizations can be amortized by (comparatively small speedups) in each iteration. One commonly occuring pattern in loops are concatenated rules. The well-known STS mutex benchmark [7], e. g., contains the loop (`takeRule ; releaseRule ; giveRule`)$\{n\}$, where n started at 100 000 in our tests.

We noticed that many rules are intended to match elements already processed by the previous rule. In particular, this is true for the loop in the mutex benchmark mentioned above. Therefore, we want to be able to re-use these elements without having to search for them again.

3.2 Mapping Elements

We guess a mapping between replacement graph elements of the first rule and pattern graph elements of the second rule by looking for a maximum common subgraph of both graphs. We do so by using a backtracking algorithm presented by Krissinel and Henrick [8]. This algorithm finds induced common subgraphs with a maximum number of nodes. We apply this algorithm to an (implicit) representation of our pattern and replacement graphs were all elements (both nodes and edges) become nodes and the connections between nodes and edges (when a node and an edge are incident) become edges. Thus we are able to find non-induced subgraphs, too. Additionally, we check that the element in the second rule has the same or a super-type of the type of the element in the first rule, thus respecting inheritance of types in the meta-model.

This approach has some limitations, though. Since it prefers large subgraphs, the intended mapping will not be found if it does not have maximal cardinality, in

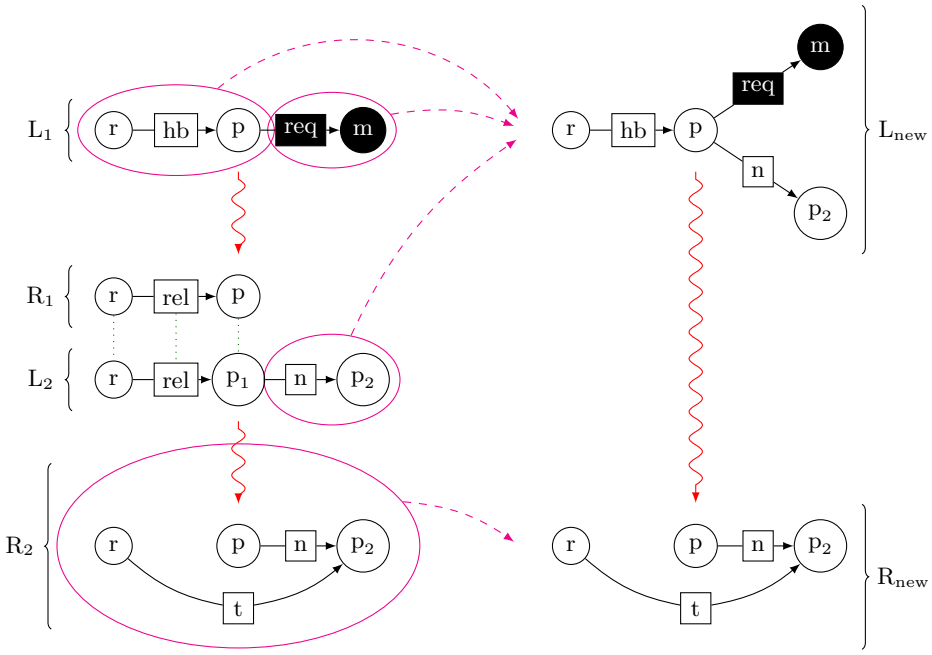

Fig. 2. Combination of `releaseRule` and `giveRule`

particular if it can be extended to a larger one. We encountered such a case when trying to combine two iterations of the mutex benchmark loop: Our technique nearly found the correct mapping, but included one additional wrong mapping, which made the combined rule inapplicable.

3.3 Building combined rules

Our rules follow SPO semantics. Thus they consist of a pattern graph (L), a replacement graph (R), and r, a preservation morphism[2]. We view nodes or edges mapped by r_1, r_2, and the mapping from section 3.2 as identical.

We construct L_{new} and R_{new} as follows: L_{new} consists of L_1 plus the non-mapped elements of L_2. Dangling edges preclude the construction. The construction of R_{new} is analogous: It consists of R_2 plus the non-mapped elements of R_1. Here, dangling edges just don't appear, which conforms to SPO semantics.

We also can just keep NACs from the first rule for the new rule. For NACs from the second rule, matters are a bit more complicated. The graph on which these NACs would have to be checked does not exist in materialized form. Thus, we have to equip the new rule with NACs that are checked before the first rule

[2] In the following, we subscript L, R, and r with 1, 2, and "new" to denote that they belong to the first, second or newly-built rule, respectively.

would be applied, but still are found whenever the NAC in the second rule would have been found.

The following method is applied for each NAC $N_{2,i}$ in the second rule: We try to extend the mapping between R_2 and L_1 to one between R_2 and L_1 plus $N_{2,i}$. Elements of the NAC contained in this mapping are then known to be matchable in the host graph as it would exist after the execution of the first rule. We then build the new NAC $N_{\text{new},i}$: Preset elements are added to $N_{\text{new},i}$ if they exist in L_{new}. Otherwise, the corresponding host graph element is added by the first rule and thus does not exist at the time the NAC is evaluated. For non-preset elements mapped to an element of R_1, we know that a matching host graph element would exist after execution of the first rule. Therefore, they are not added to $N_{\text{new},i}$. Elements of $N_{2,i}$ not mapped to an element of the first rule are added to $N_{\text{new},i}$. This might lead to dangling edges, in that case the construction is not possible. Furthermore, we have to take care of one additional property of NACs: The mapping of the NAC into the host graph is by default injective (unless otherwise specified), but no such property holds for elements of the pattern graph for which no preset element in the NAC exists. This applies to the elements of L_{new} that caused non-preset elements of $N_{2,i}$ not to be added to $N_{\text{new},i}$. We must prevent that non-preset elements in $N_{\text{new},i}$ are mapped to identical host graph elements. In order to achieve this, we add the relevant elements of L_{new} to $N_{\text{new},i}$ as preset elements if such an identical mapping is possible taking the types into account. In addition, we have to prevent that elements of $N_{\text{new},i}$ are mapped to host graph elements that the first rule would delete. We add them as preset elements as well, again taking the types into account.

3.4 Transforming rewrite sequences

We want to replace the sequential execution by the combined rule, but have to include a fallback. Thus, `b; c` becomes `bc|(b; c)`, and `a; b; c` becomes `a(bc)|(a; (bc|b; c))` (where, e. g., `bc` is the combination of `b` and `c`).

4 Results

To measure the effect of our optimization, and to compare our tool with others, we used the mutex benchmark by Varró et al. [7,9]. Two variants of this benchmark are distinguished: **STSmany** does not exploit multiplicity constraints from the graph model (example: a process may have only one outgoing `token` edge), whereas **STSone** does. Such optimizations are not supported in GrGen.NET, but in Fujaba, which then is the fastest tool known to us [10].

We ran our tests on an AMD Athlon XP 3000+ with 1 GiB RAM, using Microsoft Windows XP, the .NET environment (version 2.05.50727.42), and Sun JDK 1.6.0_01. For Fujaba, we used an optimized searchplan found by Kroll [2], for GrGen.NET, we directly called libGr without using GrShell. The shown runtimes apply to the benchmarks main loop, after the ring of processes has been

Table 2. Runtimes for the mutex benchmark [in ms]

Benchmark → / Tool ↓	Mutex (STSmany)						
	10	100	1 000	10 000	100 000	500 000	1 000 000
GrGen.NET	9	9	9	22	124	743	1472
GrGen.NET(auto)	6	5	7	13	69	419	857
GrGen.NET(manual)	7	4	7	10	52	331	665

built. With GrGen.NET, we measured separate execution of the rules, an automatically generated combined rule, and a manually written combined rule. The manually written rule follows the method described above most closely, whereas automatic generation is a bit more conservative at some places, especially when adding additional preset elements to NACs.

Our results are shown in table 2 and figure 3. We see that for a benchmark size of 1 000 000, the automatically combined rule is executed about 40 % faster than the separate rules. The manually written is even executed about 55 % faster, which shows that we can achieve a further significant speedup by implementing the remaining parts of the method described in 3.3.

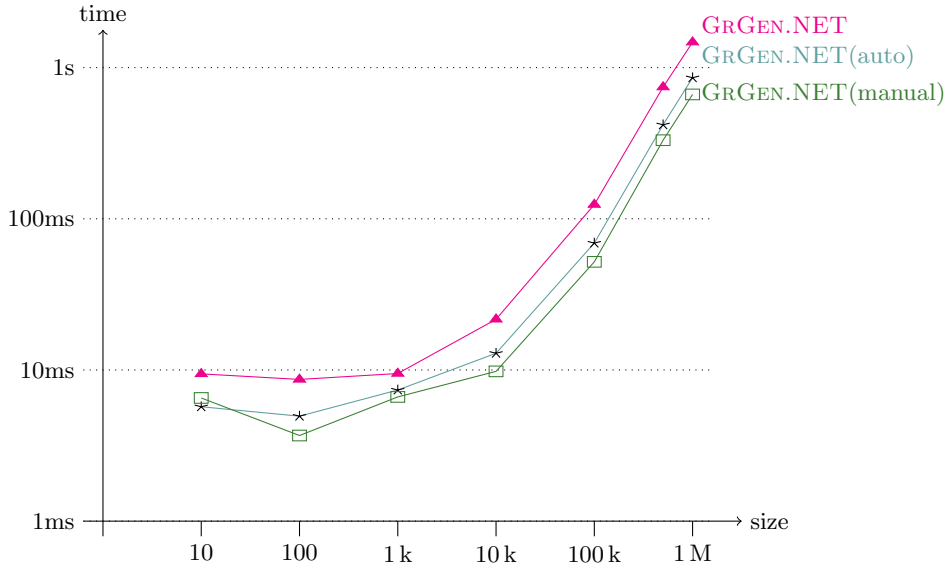

Fig. 3. Runtime for the (`takeRule`;`releaseRule`;`giveRule`){n} loop of the mutex benchmark

Transforming a GRS with the loop from the mutex benchmark takes about 1.3 seconds, primarily due to the overhead from calling the Java-based GrGen

frontend. This computation does not depend on the host graph and thus needs to be done only once.

5 Conclusion

We have shown that rule concatenation can result in a significant performance gain, and thus is a viable approach for optimization of graph rewrite sequences. Further research needs to be done to support as many GrGen features as possible in the rules. By using explicit information from the user instead of guessing a mapping of elements (section 3.2), e. g. by using parameters and return values, we'll be able to preclude most wrong guesses, thus making this optimization more widely applicable.

6 Acknowledgements

We'd like to thank Moritz Kroll who supported us with conducting and evaluating our experiments, Jakob Blomer for some very helpful comments on the nearly final version of this paper, and Christoph Mallon for being a good listener and giving valuable inspirations as we developed our method.

References

1. Blomer, J., Geiß, R.: The GrGen user manual. Technical Report 2007-5 (April 2007) To appear.
2. Kroll, M.: GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen (Mai 2007)
3. Batz, G.V.: An optimization technique for subgraph matching strategies. Technical Report 2006-7, Universität Karlsruhe, IPD Goos (April 2006)
4. Dörr, H.: Efficient Graph Rewriting and Its Implementation. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1995)
5. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A fast SPO-based graph rewriting tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Graph Transformations - ICGT 2006. Lecture Notes in Computer Science, Springer (2006) 383 – 397 Natal, Brasil.
6. Szalkowski, A.M.: Negative Anwendungsbedingungen für das suchprogramm-basierte Backend von GrGen (Oktober 2005)
7. Varró, G., Schurr, A., Varró, D.: Benchmarking for graph transformation. In: VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Washington, DC, USA, IEEE Computer Society (2005) 79–88
8. Krissinel, E.B., Henrick, K.: Common subgraph isomorphism detection by backtracking search. Softw. Pract. Exper. **34**(6) (2004) 591–607
9. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. Technical Report TUB-TR-05-EE17, Budapest University of Technology and Economics (March 2005)
10. Geiß, R., Kroll, M.: On improvements of the Varró benchmark for graph transformations. Technical report (May 2007) To appear.