GrGen: A fast SPO-based graph rewriting tool

Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski

Universität Karlsruhe (TH), 76131 Karlsruhe, Germany,
rubino@ipd.info.uni-karlsruhe.de,
WWW home page: http://www.info.uni-karlsruhe.de/~rubino

Abstract. Graph rewriting is a powerful technique that requires graph pattern matching, which is an NP-complete problem. We present GrGen, a generative programming system for graph rewriting, which applies heuristic optimizations. According to Varró's benchmark it is at least one order of magnitude faster than any other tool known to us.
Our graph rewriting tool implements the well-founded single-pushout approach. We define the notion of search plans to represent different matching strategies and equip these search plans with a cost model, taking the present host graph into account. The task of selecting a good search plan is then viewed as an optimization problem.
For the ease of use, GrGen features an expressive specification language and generates program code with a convenient interface.

1 Introduction

Over the last 30 years graph rewriting theory has become mature. The constant rise of applications requires tools that are all *theoretically sound*, *fast* and *easy to use*. Currently available tools meet these requirements only partially, with varying emphases. Our tool GrGen, which is presented in this paper, fulfills these requirements [1].

1.1 Graph Rewriting

The concept of *graph rewriting*, as implemented by GrGen, follows the *single-pushout* (SPO) approach (see section 2.4). A SPO graph rewrite rule

$$p : L \xrightarrow{r} R$$

consists of a *pattern graph* L, a *replacement graph* R and a partial graph homomorphism r between L and R (GrGen allows more information to be added). An application of p to a *host graph* H is called a *direct derivation*. It requires a partial graph homomorphism m from L to H called a *match* (GrGen demands *total* matches). The direct derivation leads to a *result graph* H', see figure 1. For each node or edge x in L there exists a corresponding node or edge in H, namely $m(x)$. Note that m does not need to be injective. The *preservation morphism* r

Preliminary version to be submitted to ICGT 2006

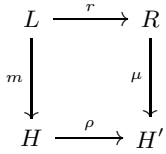

Fig. 1. The principle setting of SPO-based graph rewriting

determines what happens to $m(x)$: It maps all items from L to R, which are to remain in H during the application of the rule. The images under m of all items in L which have no image under r are to be deleted. The others are retained. Items in R which have no pre-image under r are added to H'. Note that in general ρ is neither surjective nor total. It is partial, because nodes from H may be deleted to get H'. Homomorphism ρ can be non-surjective, because new nodes may be introduced in H'—these new nodes are not in the image of ρ but in the image of μ.

1.2 Our Contributions

For pattern graphs of potentially unbounded size, subgraph matching is an NP-complete problem (see Garey and Johnson, problem GT48 [2]). Hence, the question of performance is essential for the practical relevance of graph rewriting. The multi-purpose graph rewrite generator GrGen allows high-speed graph rewriting. The main features and concepts of GrGen are:

1. *An expressive graph concept.*
   GrGen uses an extension of labeled directed multigraphs, namely *attributed typed directed multigraphs*. The type system features multiple inheritance on node and edge types (see section 2.1).
2. *Separation of meta model and rewrite rules*
   A *meta model* defines the allowed node and edge types as well as the attributes associated with each type. To restrict the set of *well-formed* graphs, the user can give so called *connection assertions*. Meta model and rewrite rules can be specified separately. This enables the developer to utilize different rule sets together with the same meta model description (see section 3.1).
3. *A notion of rewriting close to theory.*
   GrGen implements an extension of the SPO approach to graph rewriting. The differences consist in the use of the extended graph concept, some restrictions regarding the allowed matches and the ability of graph rewrite rules to request the re-labelling (i.e. retyping) of nodes (see section 2.4).
4. *Additional matching conditions and attribute computations.*
   The set of valid matches can be restricted beyond graph patterns by the assignment of *attribute conditions*, *type constraints* and *negative application conditions* (NACs) to every rule. Additionally, *attribute computations* can be associated with each rule (see section 3).

Preliminary version to be submitted to ICGT 2006

5. *Optimization of the matching process.*
   Subgraph matching is an NP-complete problem. To deal with this challenge in practice, the system is able to optimize the matching process at run time using knowledge about the current host graph (see section 2.3).
6. *Convenient user interface.*
   GrGen features an expressive and concise specification for meta models, rewrite rules, and rule application strategies (see section 3). The generated code can be invoked through an interface, which is easy to use.

We compare GrGen with the most prominent tools, namely PROGRES [3], AGG [4], Fujaba [5], and an approach presented by Varró [6]. Regarding a benchmark also introduced by Varró [7], our graph rewrite engine outperforms all of these tools by at least one order of magnitude (see section 5). While being the fastest graph rewriting system we know, we will show that GrGen is even one of the most expressive ones (see section 4).

2 Fundamental Problems and their Solutions

Thinking of graph rewriting raises three major questions:

1. What is a graph?
2. How is an occurrence of a pattern graph found?
3. What does rewriting mean in detail?

For tool development, the first question should be answered from an engineering point of view. We have to decide wisely, because our choices may have significant influence on the benefit a user gets from a graph rewrite tool. Two main aspects are concerned: which graph concept to use (see section 2.1) and which abilities to specify a meta model we give to the user (see section 3).

The second question regards finding a match. Computationally this is a quite complex task, because subgraph matching is known to be NP-complete [2]. We propose a heuristically optimizing approach to subgraph matching. Moreover, the optimization is done dynamically at runtime depending on the present host graph (see section 2.2 and 2.3). In our experience, all this leads to good performance (see section 5), in several cases even linear runtime may be achieved. However, the worst case is still exponential.

In the literature the third question is treated thoroughly [8,9,3]. Despite this fact it is not a computationally complex problem at all. Any known rewriting method is linear in the number of nodes and edges of the applied rule. The approaches differ substantial in understandability and readability of specifications as well as their expressiveness. Also, their degree of theoretical foundation is quite different. So defining the meaning of rewriting is an important ingredient for a successful graph rewriting tool (see section 2.4).

2.1 Graphs

GrGen features attributed typed directed multigraphs. These are directed graphs with typed nodes and edges, where between two nodes more than one edge of

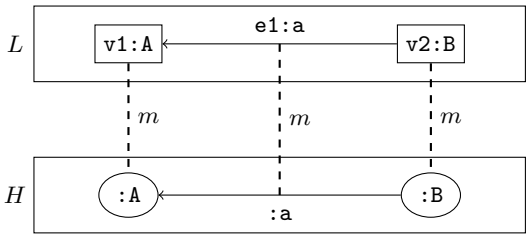

Fig. 2. Named pattern graph L and host graph H together with a match m

the same type and direction is permitted. According to its type, each node or edge has a defined set of attributes associated with it. Moreover, the type system features multiple inheritance on node and edge types. A meta model defines the allowed node and edge types as well as the attributes associated with each type. Furthermore it allows to restrict the set of well-formed graphs by so called *connection assertions*. For an example specification see section 3.1.

Throughout this paper graphs are depicted as follows: Nodes are either displayed by rectangles or ellipses. Rectangles are used in pattern graphs, ellipses are used in host graphs. The directed edges are displayed by arrows. Figure 2 shows a pattern graph L and a host graph H. The types of the nodes and edges are represented by node and edge labels with a preceding colon. In case a node or edge is given a name, it is written before the colon.

2.2 Finding a Match

We describe a match as a *graph homomorphism* between the pattern graph L and the host graph H. Such a graph homomorphism is a pair of maps $m = (m_V, m_E)$, where m_V assigns the nodes of L to nodes of H and m_E the edges of L to edges of H. In figure 2 the nodes and edges mapped to each other are connected by dashed lines.

The tightest upper bound for the runtime of subgraph matching known to us is $O(|L||H|^{|L|})$, where $|\cdot|$ denotes the sum of the numbers of nodes and edges of a graph. If we consider only fixed size patterns, subgraph matching can be regarded to as polynomial (possibly with a high polynomial degree). This seems to be good news, because we do not have to deal with an exponential runtime. But a runtime of, e.g. $O(|H|^{10})$, is still not feasible even for small constant factors, especially for host graphs H with hundreds or thousands of nodes and edges. Assuming that many application domains provide sparse graphs and a rich type system, we expect that our optimizing approach to subgraph matching leads to acceptable runtimes.

To enable the optimization of the matching process, we perform the subgraph matching according to a so called *search plan*. A search plan is a sequence of primitive search operations. Each such operation represents the matching of a single node or edge of the pattern graph to an appropriate node or edge of the

host graph. The whole search plan describes the stepwise construction of all (or one) possible matches between L and H. We call a partly constructed match a *candidate*. The runtimes caused by different search plans depend on the present host graph and can vary significantly. Therefore the key idea for finding a match fast is to create a preferably good search plan taking the structure of the present host graph into account. The necessary information is taken from an analysis of the host graph performed at runtime. GrGen also provides default search plans. They are statically created according to optional user hints.

Consider a search plan $P = \langle s_0, \ldots, s_q \rangle$, i.e., a sequence of primitive search operations s_i. We allow two kinds of search operations: At first there are *lookup* operations. They are denoted by $s_i = \mathsf{lkp}(x_i)$, where x_i is a node or edge of the pattern graph. At second there are *extension* operations $s_i = \mathsf{ext}(v_i, e_i)$, where v_i is a pattern node and e_i is a pattern edge. A lookup operation $\mathsf{lkp}(x_i)$ represents the expansion of a candidate by any node or edge of the host graph, which is suitable for the given x_i. If x_i is a pattern node, an appropriate host graph node must have the same type as x_i or a subtype thereof (we call this an *admissible type*). If x_i is a pattern edge, the incident nodes must also have admissible types (GrGen supports no lookup operations for edges, yet). An extension operation $\mathsf{ext}(v_i, e_i)$ represents the expansion of a candidate by an edge e_i coming from an already matched node v_i along the edge e_i. Of course an appropriate host graph edge and the node at its other end must also have admissible types.

The matching of a node can happen explicitly by the execution of a node lookup $\mathsf{lkp}(v)$ or implicitly by the matching of an edge incident to that node. An edge e can also be matched in two different ways (both explicitly): by an edge lookup $\mathsf{lkp}(e)$ or by an extension $\mathsf{ext}(v, e)$. E.g. consider two possible search plans for the pattern graph L shown in figure 3.

$$P_0 = \langle \mathsf{lkp}(\mathtt{v1}), \mathsf{ext}(\mathtt{v1}, \mathtt{e1}), \mathsf{lkp}(\mathtt{v3}), \mathsf{ext}(\mathtt{v2}, \mathtt{e2}) \rangle$$
$$P_1 = \langle \mathsf{lkp}(\mathtt{e1}), \mathsf{lkp}(\mathtt{v3}), \mathsf{ext}(\mathtt{v3}, \mathtt{e2}) \rangle$$

On the execution of a primitive search operation more than one appropriate node or edge may be found. In this case a candidate is replaced by several new candidates, one for every possible node or edge. However, it is not necessary to materialize all candidates at the same time. If a candidate can be expanded by more than one host graph element, we process only one of these. The other alternatives are treated by backtracking.

2.3 Generating Good Search Plans

The runtime of different search plans can vary significantly for a given host graph. For the generation of preferably good search plans, we use an approach originally presented by Batz [10]. It is an extension of a technique invented by Dörr [11].

The execution of an operation s_i can cause the splitting of a candidate into several new candidates. If this is the case for a significant ratio of the operations of a search plan, this leads to an exponential growth of the set of candidates.

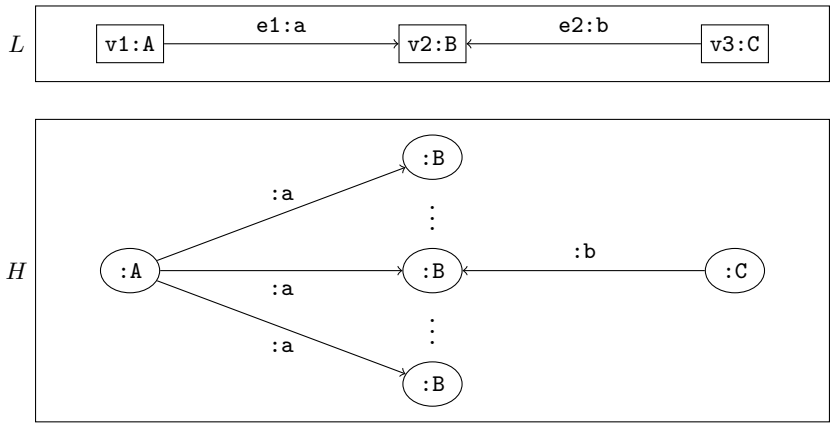

Fig. 3. The operation ext(v1, e1) causes an intense needless splitting of candidates

If splitting operations could be avoided by a search plan, less runtime would be needed. If the execution of a search plan causes no splitting at all, linear runtime for *sparse* host graphs H is achieved, that is $O(|L|)$.

Consider e.g. the pattern graph L and the host graph H shown in figure 3. In H a single node of type A is connected to a number of nodes of type B (let's say 20), each by an edge of type a. Now let us assume that the search plan

$$P_2 = \langle \mathsf{lkp}(\mathtt{v1}), \mathsf{ext}(\mathtt{v1}, \mathtt{e1}), \mathsf{ext}(\mathtt{v2}, \mathtt{e2}) \rangle$$

is executed. The first operation lkp(v1) leads to the creation of one new candidate. Now the node of type A is incident to 20 outgoing edges of type a, each leading to a node of type B, so in the worst case the candidate splits into 20 new ones. In contrast the execution of the search plan

$$P_3 = \langle \mathsf{lkp}(\mathtt{v3}), \mathsf{ext}(\mathtt{v3}, \mathtt{e2}), \mathsf{ext}(\mathtt{v2}, \mathtt{e1}) \rangle$$

requires no splitting at all. In case of the extension operation for edge e1, the crucial point is that P_3 follows e1 in the opposite direction as P_2 does. That is where Dörr's approach applies to: The direction an edge is followed can determine whether a candidate splits or not. In contrast to extension operations, for lookup operations the splitting depends on the number of present elements having an admissible type.

However, for extension operations, splitting cannot always be avoided. In the following we refer to equally typed edges of equal direction which connect equally typed nodes as *isomorphic*. If there are isomorphic edges present on *both* nodes incident to an edge, splitting occurs inevitably. In such a situation it only remains to choose the direction with *less* splitting. Moreover, we are looking for search plans which cause a low *overall* amount of splitting. Therefore, we extend Dörr's technique by a cost model to direct the optimization of search plans.

For this purpose we assign a cost to every operation which might possibly occur in a search plan: An operation $\mathsf{ext}(v, e)$ gets assigned the average number of splittings for a candidate. A $\mathsf{lkp}(x)$ gets assigned the number of present elements of admissible type. Having done this, we compute the costs of a possible search plan $P = \langle s_0, \ldots, s_q \rangle$ by the formula

$$C_P := c_0 + c_0c_1 + c_0c_1c_2 + \cdots + c_0c_1c_2 \cdots c_q$$

where c_i is the cost of the operation s_i.

Essentially the formula estimates the number of host graph elements matched while executing P. If operation s_0 is executed, up to c_0 host graph elements will be matched. This also means that up to c_0 new candidates will be created. Now if operation s_1 is performed, for all these candidates on average c_1 further elements will be matched. Overall this results in an average amount of up to c_0c_1 matched elements and newly created candidates. Continuing this, one gets the above formula. But if a candidate fails to complete, of course no further candidates will be created from it. So, except for constant factors, the above formula yields an *overestimation* of the expected number of nodes or edges processed while executing P.[1]

We do not know an efficient algorithm yielding a search plan P with minimal costs C_P. So, we use the following heuristic method: In the first step, we minimize the most significant term occurring in the above formula, namely $c_0c_1c_2 \cdots c_q$. This is done by choosing a possibly cheap selection from the set of all possible search operations for L. In the second step, we compute an order for the selected operations, such that the cheap operations appear preferably early and the expensive operations as late as possible. This exploits the fact, that a splitting has more impact on C_P, the earlier the according operation occurs in P. The costs of the possible operations are derived from an analysis of H, which can be performed in time $O(|H|)$. A detailed description of this heuristics is given in a technical report [12].

2.4 Meaning of Rewriting

There are many approaches (either ad hoc or mathematically founded) to define the meaning of "rewriting". It is impossible to tell which is the best for all purposes, especially because this is a matter of taste to some extent. E.g. compiler construction (our application domain) yields some requirements: Firstly, we need to find patterns without specifying the full context of nodes. Secondly, we want to delete nodes and edges without matching all adjacent nodes, dangling edges should be deleted. Thirdly, if we iterate the application of a set of graph rewrite rules then this should be a Turing-complete formalism. Finally, the expressiveness should be adequate to formulate retyping of nodes, attribute recalculation and rich structural extra conditions on a match.

[1] This holds under two assumptions: Firstly, $c_0, \ldots, c_q$ (understood as a collection of random variables) is stochastically independent. Secondly, the occurring host graphs are sparse and have uniformly distributed edges. The latter yields the working hypothesis that every node of H has roughly $O(1)$ incident edges.

This rules out a lot of the approaches but leaves us still with some choice. We have chosen the well-known SPO approach. For conciseness we omit a description of the SPO approach itself (for an introduction see [8]). Except for partial matching GrGen implements the SPO approach to the full extent, but provides additional features not covered by SPO. These are: Attribute and type conditions, NACs, node type changes and attribute recalculation. Attribute and type conditions imposed on a match restrict the set of admissible matches. Retyping and attribute evaluations are performed after the SPO rewrite is done. A formalization of such extensions based on category theory for the DPO approach is presented by Ehrig et al. [13].

3 The Tool

In this section, we present the most important features of GrGen along with its input language which enables the user to define a meta model for graphs, a set of graph rewrite rules as well as a sequence of rule applications.

The structure of the generated graph rewriters (we call them *graph engines*) yielded by GrGen arises from the separation of four concerns: defining the type of graph elements, storing the graph data, finding the match, and performing the rewrite. This gives us the freedom to easily change certain aspects of the implementation. The GrGen(SP) graph engine uses our search plan approach to subgraph matching sketched in section 2.2 and 2.3 (for a technical description see Batz and Szalkowski [10,12,14]). GrGen(PSQL) is a graph engine variant that uses a Postgres database for storing and matching graphs [15,16].

3.1 Meta Model

The key features of GrGen's meta model are exemplarily shown in listing 1.1.

Types. Nodes and edges (classes) can have types. The syntax is similar to common programming languages (keywords `node class` and `edge class`).

Attributes. Nodes and edges can possess attributes. The set of attributes assigned to a node or edge is determined by its type. The attributes itself are typed, too.

Inheritance. Types (classes) can be composed by multiple inheritance. This eases the way of specifying patterns and improves the expressiveness of graphs. `Node` and `Edge` are the built-in root types of node and edge types, respectively. Moreover, inheritance eases the specification of attributes, because subtypes inherit the attributes of their super types.

Connection Assertions. To specify that certain edge types can only connect specific nodes, we included connection assertions (keyword `connect`). Using these, the system is optionally able to check whether a host graph is well-formed or not. For example, line 12 of listing 1.1 specifies, that nodes of type `NodeTypeA` can have arbitrary outgoing edges of type `EdgeTypeA`. Furthermore these edges must connect to a node of type `NodeTypeB`, whereas one to five such edges may be incoming at a single `NodeTypeB` node.

Listing 1.1. A meta model

```
node class NodeTypeA {
  a1 : int;
  a2 : int;
}
node class NodeTypeB extends NodeTypeA {
   a3 :int;
}
node class NodeTypeC extends NodeTypeA, NodeTypeB;

edge class EdgeTypeA
     connect NodeTypeA  [0:1] -> NodeTypeA  [0:1],
             NodeTypeA  [*]   -> NodeTypeB  [1:5];

edge class EdgeTypeB extends EdgeTypeA
     connect NodeTypeB  [4:*] -> NodeTypeA  [1]
{
  a1 : string;
}
```

3.2 Graph Rewrite Rules

We present an example graph rewrite rule with name `SomeRule` (see listing 1.2). For convenience, we denote the preservation morphism r implicitly by using named nodes and edges: Identical names in pattern and replacement graph indicate that this nodes or edges are mapped to each other by r.

The part introduced by the keyword `pattern` denotes the pattern graph consisting of a node named `n3` of type `NodeTypeB` as well as two nodes named `n1` and `n2` of type `NodeTypeA`. Anonymous edges can be denoted by an arrow (`-->`). Additionally, we can specify an edge name and type through inserting them into the arrow (`-EdgeName:EdgeType->`). The semantics of the example rule is sketched in the following.

Isomorphic/Homomorphic Matching. The tilde operator (`~`) between the nodes `n1` and `n2` specifies that these nodes may be matched homomorphically. In contrast to the default isomorphic matching of morphism m the nodes `n1` and `n2` *may* be mapped to the same node in the host graph.

Negative Application Conditions (NACs). With negative application conditions (keyword `negative`) we can specify graph patterns which forbid the application of a rule if one of them is present in the host graph.

Attribute Conditions. The attribute conditions (keyword `if`) in the pattern part allows for further restriction of the applicability of a rule.

Type Constraints. By writing `n4 : Node \ NodeTypeB` we declare a node that is a subtype of `Node` but not of `NodeTypeB`.

Replace Part. Because node instances `n1` and `n3` (declared in the pattern part) are used in the replace part (denoting the replacement graph), these nodes

Listing 1.2. A rewrite rule specification

```
rule SomeRule {
  pattern {
    node (n1 ~ n2) : NodeTypeA;
    n1 --> n2;
    n3 : NodeTypeB;
    negative {
      n3 -e1:EdgeTypeA-> n1;
      if { n3.a1 == 42 * n2.a1; }
    }
    negative {
      node n4 : Node \ NodeTypeB;
      n3 -e1:EdgeTypeB-> n4;
    }
  }
  replace {
    n5 : NodeTypeC<n1>;
    n3 -e1:EdgeTypeB-> n5;
  }
  eval {
    n5.a3 = n3.a1 * n1.a2;
  }
}
```

are kept. The anonymous edge instance between `n1` and `n2` only occurs in the pattern and therefore gets deleted. The edge `e1` is only declared in the replace part, thus it has to be created. Note that edge `e1` from the replace part and the negative parts are all different, because of their scopes.

Retyping. Node `n5` is a retyped node stemming from node `n1`. This enables us to keep all edges and all attributes stemming from common super types of a node while changing its type.

Eval Part. If a rule is applied, then the attributes of matched and inserted nodes and edges may be recalculated.

3.3 Rule Application

To control the application of rules, we define the set $\mathcal{R}$ of *regular graph rewrite sequences (RGS)*, where $\mathcal{P}$ is a set of rewrite rules:

$$\mathtt{p} \in \mathcal{P} \Rightarrow \mathtt{p} \in \mathcal{R} \qquad \mathtt{p} \in \mathcal{P} \Rightarrow [\mathtt{p}] \in \mathcal{R}$$
$$\mathtt{R}_1, \mathtt{R}_2 \in \mathcal{R} \Rightarrow \mathtt{R}_1\mathtt{R}_2 \in \mathcal{R} \qquad \mathtt{R} \in \mathcal{R} \Rightarrow (\mathtt{R}) \in \mathcal{R}$$
$$\mathtt{R} \in \mathcal{R} \Rightarrow \mathtt{R}* \in \mathcal{R} \qquad \mathtt{R} \in \mathcal{R}, n \in \mathbb{N} \Rightarrow \mathtt{R}\{n\} \in \mathcal{R}$$

The syntax of RGSs is largely borrowed from regular expressions, but its semantics are only related. The main difference is: Determined and undetermined

iteration expressions `R`$\{n\}$ and `R`$*$ cause an execution of `R` until no rule contained in `R` can be applied (or the iteration count exceeds n, respectively). This includes that the subsequence $\mathtt{R}_2$ of $\mathtt{R}_1\mathtt{R}_2$ is executed even if $\mathtt{R}_1$ is not applicable. [`p`] denotes the simultaneous application of all matches of rule p.

E.g. we can express Varró's STS mutex benchmark of size 1000 by the following RGS:

```
newRule{998} mountRule requestRule{1000}
  (takeRule releaseRule giveRule){1000}
```

4 Related Work

Over three decades, graph rewrite theory has evolved well. Amongst others, there are two major schools: Firstly, the algebraic rewriting school, which considers graphs as algebraic objects and defines rewriting via mappings. Algebraic rewriting itself has a rich variety of approaches: There is the single-pushout approach (SPO, see section 1.1 and 2.4), the double-pushout approach (DPO) and the pullback approach. These approaches are all based on category theory and differ mostly in the fashion of defining the rewrite rules and the behaviour when deleting nodes. Regarding the latter, SPO is more powerful then DPO. Secondly, there is the programmed approach. It defines rules and rewrites in a more operational style. Its semantics is more complex and hard to define, which on the other hand eases the integration of special application driven needs to the tool. For example, consider the formal definition of a part of PROGRES [17].

In table 1 the most prominent graph rewriting tools are compared. For this purpose we consider five key properties, which gives a coarse-grained insight in the theory and implementation of each tool.

Semantics. How is the rewriting described theoretically and how powerful is a single rewriting step? SPO refers to single-pushout approach (see section 1.1 and 2.4). If the tool uses negative application conditions to enhance its expressiveness then we write NAC. By programmed we mean that semantics is rather defined through an operational sequence than a theory.

Storage. The storage property describes how the graph is stored and whether it is persistent: In-memory storage is not persistent, always. RDBMS and GRAS are both database backed graph storages where the first stands for of the shelf relational database system, the latter is a special graph database implementation.

Matching. The tools vary significantly in the handling of the matching problem. Some transform the matching problem into another well understood and tool supported domain, like constraint satisfaction (CSP) or relational algebra (SQL). Others perform a local search (LS) on the graph structure to find the matchings. This search process can be driven by chance or be planned ahead.

Mode. Does the tool generate code in a conventional programming language and compiles it to perform the matching? Or are the graph rewrite rules just interpreted by the tool, hence no code is generated.

Table 1. Features of graph rewriting tools

Tool	Semantics	Storage	Matching	Mode	Language
PROGRES	programmed	GRAS	planned LS	int.&comp.	C/C
AGG	SPO&NAC	memory	CSP	interpreted	Java
Fujaba	programmed	memory	LS	compiled	Java/Java
VarróDB	SPO&NAC	RDBMS	SQL	interpreted	Java
GrGen(PSQL)	SPO&NAC	RDBMS	SQL	compiled	Java/C
GrGen(SP)	SPO&NAC	memory	planned LS	compiled	Java/C

Language. This refers to the language that is used for implementing the rewrite generator and if applicable the matcher to be compiled.

One of the first graph rewrite tools is PROGRES and it is still amongst the most expressive one [3]. As described by Zündorf [18], its matching algorithm is based on planned local search. A more contemporary tool is AGG, which also has the desirable property to rely closely on the theoretical foundations of the SPO approach [4]. The matching of AGG is done by reducing the problem to a constraint satisfaction problem [19]. To call Fujaba a graph rewrite tool is a kind of an understatement [5]. Fujaba is a tool for software visualization and two-way transformation based on UML. Some of its functionality relies on graph transformations. These parts can be utilized to perform general graph rewriting. The graph rewriting rules are programmed story diagrams in the sense of extended UML use case diagrams. Varró describes a technique for performing graph rewriting based on relational algebra [6]. Up to now, his tool is not accessible, but we have some example runs available [20].

The Optimix system proposed by Uwe Assmann has a limited expressiveness [21]. It would be impossible to perform the benchmarks of our choice without significant simplifications. Therefore it is not included in our closer examination. But nevertheless Optimix is interesting; because of its limitations it is possible to get some strong theoretical results, such as confluence and guaranteed termination. In general, this is not possible for the other tools mentioned above.

Dörr developed an idea for matching certain graphs in linear time [11]. His technique fails for graphs which contain edges that cause inevitably splitting of candidates. To our knowledge no actual tool was built depending on his ideas. By defining a cost model, we extended his findings to all graphs, but had to sacrifice the linear runtime guarantee (see section 2.3).

5 Performance

The benchmark uses various sizes of graphs and patterns as well as long and short transformation sequences. The example used as a benchmark by Varró was originally proposed to serve as distributed mutual exclusion algorithm. Varró has changed the algorithm slightly for benchmarking.

Our own measurements (for AGG and GrGen) were carried out on an AMD Athlon XP 3000+ with 1GB main memory. Measurements by Varró (for

Table 2. Runtime for several of the Varró benchmarks (in milliseconds)

Benchmark →	STS			ALAP			ALAP simult.			LTS
Tool ↓	10	100	1000	10	100	1000	10	100	1000	1000,1
PROGRES	12	946	459,000	21	1,267	610,600	8	471	2,361	942,100
AGG	330	8,300	6,881,000	270	8,027	13,654,000	–	–	–	$> 10^7$
Fujaba	40	305	4,927	32	203	2,821	20	69	344	3,875
VarróDB	4,697	19,825	593,500	893	14,088	596,800	153	537	3,130	593,200
GrGen(PSQL)	30	760	27,715	24	1,180	406,000	–	–	–	96,486
GrGen(SP)	< 1	8	79	< 1	5	64	< 1	< 1	5	99

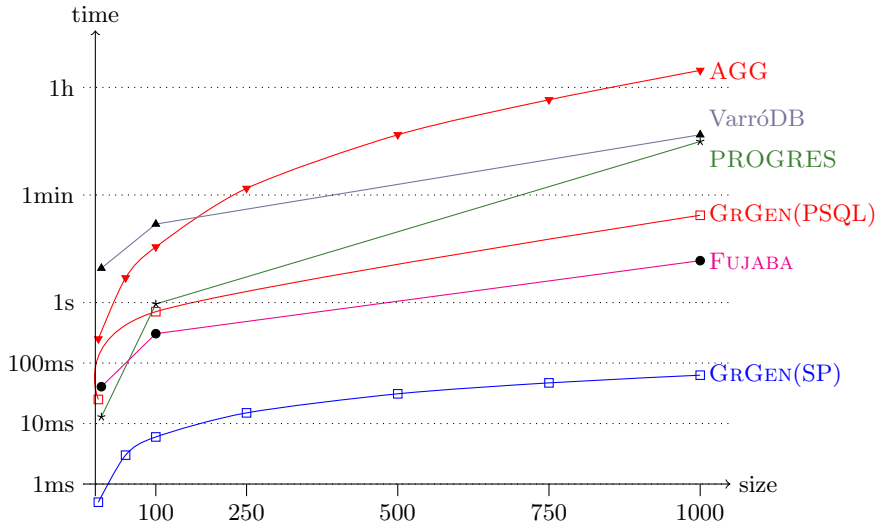

Fig. 4. Runtime of STS mutex benchmark (multiplicity optimizations off, parameter passing off, simultaneous execution off; for parameter details see [7])

PROGRES, Fujaba and VarróDB) were performed on a Intel Pentium 4 at 1.5 GHz with 768 MB main memory [20]. To reuse his results we multiplied Varró's figures by 0.68 which is the speed difference of both processors according to the SPEC organization [22].

Figure 4 shows the runtime of two GrGen instances compared with the most prominent tools, namely AGG [4], Fujaba [5], PROGRES [3] and an approach presented by Varró [6], which we call VarróDB. GrGen(SP) uses our most advanced graph engine, whereas GrGen(PSQL) is based on a Postgres database for storing and matching graphs (see section 3). Further benchmark results, shown in table 2, support the overall impression. The other benchmarks proposed by Varró show analogous results and are omitted, here (see [1]).

The memory usage of GrGen(SP) for the largest mutex benchmark was below 1.6 MByte. In any benchmark we conducted GrGen(SP) outperformed the next fastest tool at least by a factor of 40. Regarding the STS mutex benchmark

GrGen(SP) achieves even linear runtime in terms of benchmark size, i.e., the average runtime for a single rewrite rule is constant regardless the host graph size. The spread between GrGen(SP) and the slowest tool is more than 6 orders of magnitude.

6 Conclusion

Graph rewriting has complex theoretical and practical aspects. We meet the computational challenge of *finding a match* with a heuristically optimizing approach based on search plans. The definition of the *rewrite semantics* closely follows the well-established SPO approach and provides some extensions.

We still have to answer the most important question: Can the user actually put the power of the theory to work? Therefore, let us consider what users might expect from GrGen. The user wants to: define elements of a domain as graph elements, get expressive and concise rewrite specifications, get the results fast without excessive memory consumption, and easily integrate the graph rewriting into his applications.

GrGen meets all those needs: In the meta model attributes and types can be defined both for nodes and edges. It is possible to check graphs against given connection assertions, but graphs not conforming to these assertions can also be processed. The specification language is expressive and concise. The type hierarchy defined by the meta model helps to express graph rewrite rules easily. GrGen supports different rule application strategies: interactive application, regular graph rewrite sequences (RGS), and a low level selection by user supplied program code. An interactive environment for stepwise execution of graph rewrite rules and graph inspections is also provided. The performance of a rule application, especially of the potentially expensive pattern matching, is at least one order of magnitude faster than of any other tested system. The memory consumption of our search plan based graph engine is low, too. 10 million graph elements can be handled in 1 GB main memory. In other words: On average about 100 bytes were consumed per node or edge (without attributes assigned) including all administration overhead. The integration effort of the dynamically linked graph engines produced by GrGen is small.

Thus, tool supported graph rewriting can be done both, fast and easy to use, based on the well established theoretical foundations of SPO built into the declarative graph rewrite language of GrGen.

Acknowledgements. Thanks to all co-workers and students that helped during the design and implementation of GrGen as well as the writing of this paper. Especially we want to thank Michael Beck, Dr. Markus Noga, Dr. Andreas Ludwig and Tom Gelhausen. We thank Gergely Varró for both his most influential work on the benchmarking of graph rewrite tools as well as allowing us to reuse his measurements. Finally, all this would not happened without the productive atmosphere and the generous support that Prof. Goos provides at his chair.

References

1. Geiß, R.: GrGen. http://www.info.uni-karlsruhe.de/software.php/id=7 (2006)
2. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
3. Schürr, A.: The Progres Approach: Language and Environment. In: [23]. Volume 2. (1999) 487–550
4. Ermel, C., Rudolf, M., Taentzer, G.: The AGG Approach: Language and Environment. In: [23]. Volume 2. (1999) 551–603
5. Fujaba Developer Team: Fujaba-Homepage. http://www.fujaba.de/ (2005)
6. Varró, G., Friedl, K., Varró, D.: Graph Transformations in Relational Databases. In: Proc. GraBaTs 2004: Intl. Workshop on Graph Based Tools, Elsevier (2004)
7. Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. Technical report, Department of Computer Science and Information Theory, Budapest University of Technology and Economics (2005)
8. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation - Part II: Single Pushout A. and Comparison with Double Pushout A. In: [23]. Volume 1. (1999) 247–312
9. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation - Part I: Basic concepts and double pushout approach. In: [23]. Volume 1. (1999) 163–245
10. Batz, G.V.: Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Master's thesis, Universität Karlsruhe (2005)
11. Dörr, H.: Efficient Graph Rewriting and its Implementation. Volume 922 of LNCS. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1995)
12. Batz, G.V.: An Optimization Technique for Subgraph Matching Strategies. Technical Report 2006-7, Universität Karlsruhe, Fakultät für Informatik (2006)
13. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer (2006)
14. Szalkowski, A.M.: Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen (2005) Studienarbeit, Universität Karlsruhe.
15. Hack, S.: Graphersetzung für Optimierungen in der Codeerzeugung. Master's thesis, Universität Karlsruhe (2003)
16. Grund, D.: Negative Anwendungsbedingungen für den Graphersetzer GrGen (Studienarbeit) (2004) Studienarbeit, Universität Karlsruhe.
17. Schürr, A.: Logic based programmed structure rewriting systems. Fundamenta Informaticae, Special Issues on Graph Transformations **26**(3/4) (1996)
18. Zündorf, A.: Graph Pattern Matching in PROGRES. In: Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science. Volume 1073 of LNCS., Springer (1996) 454–468
19. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: TAGT'98: Selected papers from the 6th Intl. Workshop on Theory and Application of Graph Transformations. Volume 1764., LNCS (1998) 238–251
20. Varró, G.: Graph transformation benchmarks page. http://www.cs.bme.hu/~gervarro/benchmark/2.0/ (2005)
21. Assmann, U.: Graph rewrite systems for program optimization. ACM Trans. Program. Lang. Syst. **22**(4) (2000) 583–637
22. Standard Performance Evaluation Corporation: All SPEC CPU2000 results published by SPEC page. http://www.spec.org/cpu2000/results/cpu2000.html (2005)
23. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific (1999)

Preliminary version to be submitted to ICGT 2006