# An Abstract Machine for the Execution of Graph Grammars

**Heiko Dörr**
**Institut für Informatik**
**Freie Universität Berlin**
**Takustraße 9**
**D-14195 Berlin**
**doerr@inf.fu-berlin.de**

**Report B-94-07**
**13/4/94**

An abstract machine for graph rewriting is the central part of the middle layer of the implementation of a grammar based graph rewriting system. It specifies the interface between a compiler for graph grammars and a system performing actual graph transformations. By the introduction of a middle layer, the analysis of the given graph grammar can be used to optimize its execution. The costs of expensive analysis are thus shifted from run to compile time. Each implementation of the abstract machine can optimize the utilization of available hardware.
We give the specification of the state and the instruction set of the abstract machine. For an example grammar we show how compile time analysis can reduce execution time, and we present code generation rules to implement a grammar on the abstract machine.
In comparison to abstract machines, well-known from the implementation of functional languages, our machine can execute rewriting specified by graph grammars which is far more general than graph reduction.
The abstract machine for graph rewriting is part of a project which addresses the efficient implementation of the execution of graph grammars.

## 1 Introduction

Graph grammars are often used as formal, operational models. In this application area, there occur two components: a fixed grammar which describes the operational, overall behaviour of a modelled system, and a sequence of grammar rules to be applied iteratively on the initial graph. The sequence represents a specific run of the system. Göttler gives example applications ([Gött88]). In other applications graph grammars are used as specifications ([Lew88], [EnSch89], [Schü90]).

In general, graph grammar rewriting is computationally intensive. For each rewrite step, the subgraph isomorphism problem, which is combinatorial exploding as the number of vertices grows, has to be solved ([Gou88]). Usually certain analyses, tests, and replacements are performed to execute rewriting a graph given an applicable rule. But each rule together with the definition of the rewrite step can also be viewed as a fixed algorithm to be applied on a graph. In this interpretation, a middle layer of abstraction is introduced. Each rewrite rule is viewed as a sequence of instructions which are going to be executed when the rule is applied to a graph. The abstract machine for graph rewriting is the major component at this level of abstraction. It is capable of executing the algorithms representing the rewriting effect of a rule.

As a model of a system, the grammar is fixed. Thus, an analysis of the graph grammar can be performed by a compilation step which precedes actual rewriting. Computational cost is shifted from the rewriting phase to a previous compilation. Compilation of a graph grammar translates each rule to an instruction sequence for the graph rewriting abstract machine. This sequence encodes the algorithmic content of the rules of a grammar.

The work presented here is part of a project which develops an efficient implementation of a graph rewrite system. Besides the analysis which is given in the paper, major work in the project is devoted to graph rewriting in linear time. The basic components of the project are given in [Dö92].

## 2 Related Work

As a consequence of the computational complexity of a rewriting step, just a few implementations of graph rewrite systems have been realised. Implementations are presented in [Schü91], [Zün92] or [Him89]. The first two papers describe work done at the RWTH Aachen. It covers mainly the software development environment IPSEN. The specification language Progres within this environment is a powerful tool based on graph grammars. Its atomic actions are graph rewrite rules. They are compound by an imperative control language extended by some statements for non deterministic choice. The development environment statically checks the type consistency of the graph specification. But no analysis to speed up the execution of a graph rewrite step is performed. Furthermore no specification of a middle layer of the rewriting system is given.

The implementation presented by [Him89] is set on top of a graph editor. The interpreter can execute context-free graph rewrite rules but only with support by the user. Thus its functionality is far less then the Progres system.

The work presented here adopts an approach used in the implementation of functional languages ([Joh84], [PeJo87]). Abstract machines were implemented explicitly to test a proposed execution model. After some times of experience, the implementors were able to build compilers producing machine code based on the execution model given by the abstract machine. We take the same approach. But note that the formalisms to implement are definitely different although both are dealing with graphs. The structure of the graphs to deal with is quite different. Graphs representing functional expressions are far more regular than the graphs defined by a graph grammar. Thus, the specific implementation techniques developed in the functional language area are not applicable to grammar based graph rewriting.

In the following section we list the basic notation used in the sequel. Then, we present the specification of the abstract machine for graph rewriting. An example compilation unfolds the general idea of compiling and analysing a graph grammar to a rewriting machine. We will use a graph grammar with rule sets, unique vertex labels, and set-labelled vertices.

## 3 Preliminary Definitions

We first introduce notions for directed graphs with set labelled vertices.
Let $\Sigma_V, \Sigma_E$ be finite alphabets the sets of *vertex* resp. *edge labels*. The finite set $V$ is the *set of vertices*. $E \subseteq V \times \Sigma_E \times V$ is the *set of edges*. Let $l: V \to \Sigma_V$ be a total function, the *vertex labelling function*. Then $g = (V, E, l)$ is an *ordinary labelled graph* over $\Sigma_V, \Sigma_E$, or just graph. In case $l: V \to \wp(\Sigma_V) \backslash \varnothing$ is a total function, it is a *set-labelling function* and graph $g = (V, E, l)$ is a graph with set labelled vertices (over $\Sigma_V, \Sigma_E$), short *slv-graph*.[1] The graph $g_\varnothing = (\varnothing, \varnothing, \perp)$ is the *empty graph*, where $\perp$ is the undefined function.
The relation between set-labelled and ordinary graphs is set up by the concept of an instance of a set-labelled graph. An ordinary labelled graph $g_1 = (V_1, E_1, l_1)$ is an *instance* of a slv-graph $g_2 = (V_2, E_2, l_2)$ iff $V_1 = V_2$, $E_1 = E_2$, and the label $l_1(v)$ of all vertices $v \in V_1$ is element of $l_2(v)$. The set of all instances of $g_2$ is $inst(g_2)$.
Let $g, g_1, g_2$ be (slv-)graphs. Let $G = \{g_1, \ldots, g_n\}$ be a set of (slv-)graphs. The *partial subgraph* relation is written $g_1 \subseteq_p g_2$. *Complete subgraphs* are denoted $g_1 \subseteq_c g_2$. Two graphs $g_1$ and $g_2$ are *consistently labelled* iff the labelling functions $l_1, l_2$ are identical for all elements of the intersection $v \in V_1 \cap V_2$. $G$ is *consistently labelled* iff all pairs $g_i$ and $g_j$ are consistently labelled with $i, j \in \{1 \ldots n\}$.
Let $W \subseteq V$. The (slv-)graph $G(W, g) = (W, E \cap (W \times \Sigma_E \times W), l|_W)$ is the *induced graph* of $W$ in $g$. The set of edges of $g$ *incident* to $v \in V$ is $inc_g(v)$. The *graphmorphism* based on a vertex map $h$ is $\hat{h}$.

## 4 The Graph Rewriting Abstract Machine

The Graph Rewriting Abstract Machine (GRAM) is the main concept to be presented in this paper. It serves as a specification of a middle layer of a graph rewriting system. Thus a grammar should be compiled to code executable on the GRAM. Explicit compilation is a supposition for the analysis of a given grammar without degradation of the execution time which can even be reduced by an adequate analysis. Moreover, the definition of the machine is a guideline of its implementation either in a higher language or as an assembly version. The machine is introduced by describing its state and the semantics of the instructions.

### 4.1 The State of the Machine

The state of the abstract machine for graph rewriting consists mainly of a *labelled graph* $g$ which is successively rewritten. In a rewrite step, specific vertices or edges have to be determined because they are part of an application condition, or because they serve as a handle for insertion. There should be a set of unique labels $U$ given by a grammar. The function $u: U \to V$ accesses uniquely labelled vertices via their label. It is the inverse function of the labelling function restricted to $U$. With this function initial vertices are accessed for graph navigation.
The instruction sequence compiled out of a rewrite rule drives the rewriting according to this rule. One rewrite step consists of four phases:

- search of a partial isomorphic subgraph and determination of the rule to be applied,
- insertion of new graph components,
- construction of the embedding, and
- deletion of vertices and edges.

1. The components of a structure are indexed or designated according to the index or designator of the structure, i.e. $g' = (V', E', l')$, or $g_i = (V_i, E_i, l_i)$. $V_r$ is the vertex set of a (slv-)graph $g_r$.

The compiled rules and an instruction sequence which builds the initial graph constitute the code to be executed by the GRAM. The function $Code: Adr \rightarrow inst$ maps an address onto an instruction where *Adr* is the address space and *inst* the instruction set. The function *Code* is no element of the state since it is fixed throughout a run of the machine. Thus it might be represented elsewhere. The initialization sequence starts at address 0. The code is accessed using a program counter $pc \in Adr$.

The *rule sequence* $RS \in Adr^*$ drives the actual execution of the machine. Note that this sequence is not the code to execute a particular rewrite step. It consists of entry points to the corresponding rewrite rule.

Two stacks support the application of a rewrite rule. First, there is a *stack of sets of current bindings* $B \in \text{stack}(\{b | b: V_{rule} \rightarrow V\})$ used in the search phase. Generally, a binding is a partial map from the vertices of a rule set $V_{rule}$ to the graph stored in the machine. In the search phase, each binding represents an isomorphism between partial subgraphs of the left-hand side of a rule and the graph. In each search step, a set of new bindings is constructed out of the bindings defined in the previous step by mapping a further vertex or edge to the graph. The graph is walked breadth-first following the search algorithm produced by the compilation of the rewrite rule. If an extension of a binding is possible, extended bindings can be constructed and are pushed on top of the stack. If not, the search backtracks to another alternative. The search is successful when a binding is a total map from a left-hand side into the graph. The corresponding rule can be applied to the graph, and the binding determines the subgraph on which the rule is going to be applied.

The *embedding stack* $M \in \text{stack}(\wp(V) \times \wp(E))$ holds information of the embedding phase. In the embedding phase, the *embedding vertices and edges* determined by a cut-description are pushed on the stack. After the evaluation of all descriptions they serve for the construction of new embedding edges.

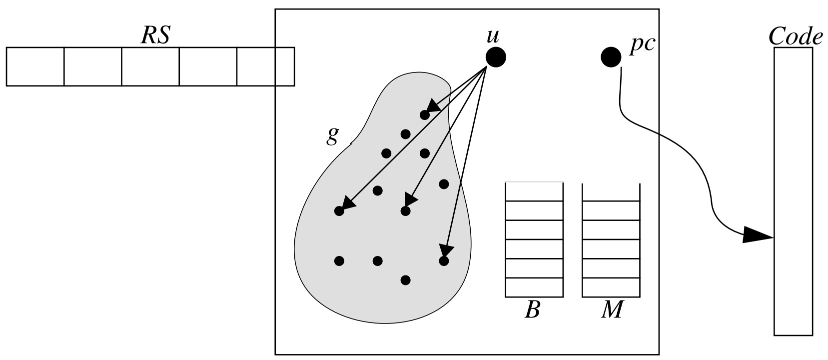


Figure 1 the graph rewriting abstract machine (GRAM)

Formally, the *state of the machine* is the tuple $[g, u, pc, RS, B, M]$. The *initial state* of the machine is $[(\emptyset, \emptyset, \perp, \perp), \perp, 0, RS, \{\perp\}, \varepsilon]$ where $\varepsilon$ is the empty stack, and *RS* is a specific rule-parameter sequence with first element (init,$\varepsilon$). Execution starts at address 0 and interprets the initialization code. It builds up the initial graph and finishes with an end-rewriting-step instruction.

## 4.2 Semantics of the Instructions

In the following, selected instructions of the GRAM are defined to exemplify the operation of the machine. They are grouped according to the phase in which they are executed. Obviously all instructions address vertices of the graph indirectly via a binding. The notations for operands of instructions are: $v$ vertex, $s$ source, $t$ target of an edge, $vl$ vertex, $el$ edge label, and $ret$ return address.

### 4.2.1 Search Instructions

The search for a partial subgraph isomorphic to a left-hand side of a rule starts with the search for a vertex with an unique label $vl$. It is then bound to $v$, the corresponding vertex of the left-hand side.

$Code(pc)$ = find-unique-vertex $vl\ v\ ret$:
$$[(V, E, l), u, pc, RS, \{b\} : B, \varepsilon] \Rightarrow [(V, E, l), u, ret, RS, B, \varepsilon], \text{ if } u(vl) = \bot$$
$$\Rightarrow [(V, E, l), u, pc + 1, RS, \{b'\} ; B, \varepsilon], \text{ if } u(vl) = v' \in V$$
$$\text{where } b'(v) = v', b'|_{dom(b)} = b.$$

In case this vertex exists, the binding $b'$ is pushed on the binding stack. When "find-unique-vertex" is the first instruction of the execution of a rewrite step, $b$ is undefined. A new binding $b'$ for $v$ is defined only when a corresponding vertex exists in the graph. If it does not exist, execution proceeds at the return address *ret*.
The successive matching of subgraphs assumes at least one initial binding found by "find-unique-vertex". For each search step concerning a new vertex, a new frame is put on top of the binding stack. This frame will hold extension of current bindings which are given as the top element of the stack $B$.

$$Code(pc) = \text{start-find-neighbour}: [g, u, pc, RS, B, \varepsilon] \Rightarrow [g, u, pc + 1, RS, \emptyset : B, \varepsilon].$$

After this preliminary step several alternative images of one distinct left-hand side vertex are determined. They are stored as an extension to one binding constructed in the previous step. The instruction "find-neighbour-forward" binds vertices of the graph to vertex $t$. It is a part of the left-hand side or a rule which is joined by an $el$-labelled edge with the image of $s$. This new binding is given in the definition of $B'''$. The set $T_i$ contains all vertices of the graph labelled with $vl$. Furthermore, they have to be target of an edge labelled with $el$ incident to the image of the source $s$ under a binding $b_i$ found in the previous search phase. Only those target vertices are included which are not bound by $b_i$ yet. These target vertices $t_{i,j}$ are bound to $t$. They extend $b_i$ to the new binding $b_{i,j}$ which is added to the set of current bindings $B'''$.

$Code(pc)$ = find-neighbour-forward $s\ el\ vl\ t$:
$$[(V, E, l), u, pc, RS, B'' : \{b_1, \ldots, b_n\} : B, \varepsilon]$$
$$\Rightarrow [(V, E, l), u, pc + 1, RS, (B''' \cup B'') : \{b_1, \ldots, b_n\} : B, \varepsilon]$$
$$\text{where } T_i = \pi_3(E \cap (b_i(s) \times el \times [l^{-1}(vl) \backslash rg(b_i)])), \{t_{i,1}, \ldots, t_{i,m_i}\} = T_i, i = 1 \ldots n$$
$$\text{and } B''' = \{b_{i,j} | \forall i = 1 \ldots n, \forall j = 1 \ldots m_i : b_{i,j}|_{dom(b_i)} = b_i, b_{i,j}(t) = t_{i,j}\}.$$

After testing all occurrences of a vertex in the graph, the search step is finished. The instruction

$Code(pc)$ = end-find-neighbour $ret$:
$$[g, u, pc, RS, B' : B, \varepsilon] \Rightarrow [g, u, pc + 1, RS, B' : B, \varepsilon], \text{ if } B' \neq \emptyset$$
$$\Rightarrow [g, u, ret, RS, B, \varepsilon], \text{ if } B' = \emptyset,$$

branches to another search alternative if no new binding could be found. Otherwise the search continues with the new bindings or, if a whole left-hand side matches, the execution enters the rewriting phase initiated with "chose-binding".
In case a search path is not successful, the constructed bindings have to be removed from the stack to enable backtracking to former partial matches.

$$Code(pc) = \text{pop-binding}: [g, u, pc, RS, B' : B, \varepsilon] \Rightarrow [g, u, pc + 1, RS, B, \varepsilon].$$

An application condition of a rule can be met by several subgraphs. Therefore search may result in a number of bindings. One of them, hence one isomorphic partial subgraph is selected for application.

$Code(pc)$ = choose-binding:
$$[g, u, pc, RS, B' : B, \varepsilon] \Rightarrow [g, u, pc + 1, RS, \{b\}, \varepsilon], \text{ with } b \in B'.$$

This instruction finishes the search phase of a rewrite step and enters actual rewriting.

### 4.2.2 Embedding

The evaluation of an embedding rule begins with pushing a new frame on the embedding stack. This frame holds the edges and vertices determined during the evaluation of a cut-description.

$Code(pc)$ = prepare-embedding-search:
$$[g, u, pc, RS, \{b\}, M] \Rightarrow [g, u, pc + 1, RS, \{b\}, (\emptyset, \emptyset) : M].$$

Afterwards, embedding edges and vertices are successively determined. An inward embedding is defined by the label $vl$ given to the source of an edge with label $el$ going to the target vertex $t$. It is part of the left-hand side and therefore addressable by the binding. In case there are edges incident to the image of $t$ with the given characteristics, these edges and their sources are added to the frame. The restriction $\backslash rg(b)$ ensures that only vertices of the surrounding graph are taken into consideration.

$$Code\,(pc) \;=\; \text{find-embedding-in}\;\; vl \;\; el \;\; t:$$
$$[\,(V, E, l)\,, u, pc, RS,\ \{b\}\,,\ (MV, ME) : M]$$
$$\Rightarrow\ [\,(V, E, l)\,, u, pc+1, RS,\ \{b\}\,,\ (\pi_1(EE) \cup MV, EE \cup ME) : M]$$
$$\text{where}\quad EE \;=\; E \cap\ (l^{-1}(vl) \backslash rg(b) \times \{el\} \times \{b(t)\}\,)\ .$$

After the evaluation of all cut-descriptions of a rewrite rule, the contribution of the embedding description is mapped to the graph. First, the definition of the rule-set rewrite step requires the deletion of those edges determined by a cut-description.

$$Code\,(pc) \;=\; \text{delete-embedding-edges}:$$
$$[\,(V, E, l)\,, u, pc, RS,\ \{b\}\,,\ (MV, ME) : M]$$
$$\Rightarrow\ [\,(V, E \backslash ME, l)\,, u, pc+1, RS,\ \{b\}\,,\ (MV, ME) : M]\ .$$

Next, new edges connecting right-hand side vertices to the surrounding graph are introduced according to given paste-descriptions. Here too, the two possible directions of edges lead to two similar insert instructions. One of them is:

$$Code\,(pc) \;=\; \text{add-embedding-edges-in}\;\; el \;\; t:$$
$$[\,(V, E, l, val)\,, u, pc, RS,\ \{b\}\,,\ (MV, ME) : M]$$
$$\Rightarrow\ [\,(V, E', l, val)\,, u, pc+1, RS,\ \{b\}\,,\ (MV, ME) : M]$$
$$\text{where } E' \;=\; E \cup\ \{\,(v, el, b(t)) \mid v \in MV\}\ .$$

After updating the graph according to the current embedding rule, the next rule is processed. Therefore the results of the current one are popped from the embedding stack.

$$Code\,(pc) \;=\; \text{end-embedding-rule}:$$
$$[g, u, pc, RS,\ \{b\}\,,\ (MV, ME) : M] \;\Rightarrow\; [g, u, pc+1, RS,\ \{b\}\,, M]\ .$$

### 4.2.3 Control Instructions

At the end of a rewrite step, the head of the rule sequence is dropped. Execution proceeds at $n_2$, the entry point of the new rule set to be applied to the graph. The binding and the embedding evaluation stacks are initialized.

$$Code\,(pc) \;=\; \text{end-rewrite-step}:$$
$$[g, u, pc, n_1 : n_2 : RS, B, M] \;\Rightarrow\; [g, u, n_2, n_2 : RS,\ \{\bot\}\,, \varepsilon]\ .$$

### 4.2.4 Insertion and Deletion of Vertices and Edges,

Instructions for insertion and deletion of vertices and edges are not listed here. They manipulate the central data structure in an intuitive way.

## 5 Example Compilation

In this section we introduce a graph grammar type suitable for the execution on the GRAM. It gives an example for execution speed-up by compile time analysis. Its main characteristic is the usage of a set of rules instead of a singular rule. By this means pattern matching, well-known from functional languages, is introduced into graph grammars. The concept of rule sets is closely related to the or-statement in the specification language Progres ([Sch91]).

From the application point of view, the introduction of rule sets enhances the usability of graph grammars for the representation of state transition system. Commonly, the effect of a state transition is dependent on the input state. This situation is modelled by a rule set where for each input state there is a specific rule representing the

effect of a state transition. Depending on the actual system state represented by a graph, the according rewrite rule will be selected for application by the rewrite mechanisms defined for rule sets.
First, the grammar and the corresponding notion of rewriting is introduced. Then, the construction of a search tree is developed which optimizes the execution of a rule set rewrite step. The resulting search tree is the input for code generation. Finally, the code generation rules are presented.

## 5.1 Graph Grammars with Rule Sets

Graph grammars consist of mainly the same components as string grammars. Only the embedding of a right-hand side of a rule is not as straightforward as in the string case. The embedding description is part of a rewrite rule. It defines a set of edges connecting vertices of the rest graph with those of the inserted graph. A description consists of a number of embedding rules. Each rule is a pair of cut- and paste-descriptions. The embedding addresses direct neighbour vertices by their label which cannot be changed during embedding. The set of *cut-descriptions* being defined over a vertex set $V$ is $C(V) = V \times \wp^{+}(\Sigma_E) \times \wp^{+}(\Sigma_V) \times \{\mathrm{in}, \mathrm{out}\}$. The set of *paste-descriptions* related to $V$ is $P(V) = (V \times \wp^{+}(\Sigma_E) \times \{\mathrm{in}, \mathrm{out}\}) \cup \{\mathrm{del}\}$. Two cut-descriptions $c_i = (v_i, EL_i, VL_i, d_i)$ $i = 1, 2$ are not overlapping iff either $v_1 \neq v_2$, $d_1 \neq d_2$, $vl_1 \cap vl_2 = \varnothing$ or $el_1 \cap el_2 = \varnothing$.
The function *fits* interprets the cut-description in the context of a specific rewrite step. Thus a labelling function $l$ and a map $h$ of the applied rule to the rewritten graph are given. The boolean function $fits_{h,l}$ determines the *fit of a cut-description* $c = (v, EL, VL, d) \in C(V)$ to an edge $e = (s, el, t) \in V' \times \Sigma_E \times V'$ depending on the direction of the edge. If $d = \mathrm{out}$ then $fits_{h,l}(c, e) = el \in EL \wedge h^{-1}(s) = v \wedge l(t) \in VL$. In case $d = \mathrm{in}$ source and target vertex change their roles: $fits_{h,l}(c, e) = el \in EL \wedge h^{-1}(t) = v \wedge l(s) \in VL$.

Based on the embedding description, a single graph rewriting rule can be defined. It employs set-labelled graphs to identify rules which differ just in some vertex labels but have the same graph structure. To be able to interpret this modification correctly, there are restrictions to the labelling of the right-hand side of a rule mentioned in the first condition of the definition. In case there occurs a set labelled vertex on the right-hand side, it must be part of the left-hand side, too. Furthermore cut-descriptions have to be consistent concerning the deletion of embedding edges. Thus, they must not overlap. The unique labels will serve as an anchor for rewriting a graph.

**Definition 1** graph rewrite rule

Let $g_l, g_r$ be slv-graphs, $M \subseteq C(V_l) \times P(V_r)$. A *graph rewrite rule* with set labelled vertices, or slv-rule, is the tuple $r = (g_l, g_r, M)$ iff $\forall v \in V_r : |l_r(v)| > 1 \Rightarrow v \in V_l \wedge l_l(v) = l_r(v)$ and for the embedding descriptions hold $\forall (c_1, \{\mathrm{del}\}), (c_2, p) \in M : p \neq \{\mathrm{del}\} \Rightarrow c_1 \cap c_2 = \varnothing$. The *unique labels* of $r$ are:

$$U(r) = \{u \in \Sigma_V | \exists v \in V_l \cap V_r : l_l(v) = \{u\} = l_r(v) \wedge u \notin \bigcup_{v' \in V_l \setminus \{v\}} l_l(v') \cup \bigcup_{v' \in V_r \setminus \{v\}} l_r(v')\} .$$

The set of *common vertices* is $V_c = V_l \cap V_r$. The set of vertices which change their label during rewriting is $V_{c*} = \{v \in V_c | l_l(v) \neq l_r(v)\}$. The set of *common edges* is $E_c = E_l \cap E_r$ and the according complementary sets are $E_{\bar{l}} = E_l \backslash E_c, E_{\bar{r}} = E_r \backslash E_c$.

A rule is applicable when there exists an instance of a left-hand side of a rule in the applied set, which is a partial subgraph of the rewritten graph. The rewrite step is defined with the usual semantics but the terminology is chosen such that actual changes are minimized. The context graph $(V_c, E_c, l|_{V_c})$ needs not to be removed without respective of the actual change intended by the rewrite rule.

**Definition 2** applicability, rewrite step

Let $r = (g_l, g_r, M)$ be a slv-rule, $g$ be a graph. $r$ is *applicable* to $g$ iff there exists a graph isomorphism $\hat{h}_l$ such that $\hat{h}_l(g_l) \subseteq_p g$. Let $r$ be applicable to the graph $g$.

The graph $g' = (V', E', l', val')$ is the result of rewriting $g$ with $r$ iff its vertex set is $V' = (V \backslash h(V_{\tilde{l}})) \cup V^*$, the edges are $E' = (E \backslash (inc_g(h(V_{\tilde{l}})) \cup h(E_{\tilde{l}}) \cup Emb_{del})) \cup (h(E_{\tilde{r}}) \cup Emb_{ins})$, and the labelling function $l': V' \to \Sigma_V$ is derived from the right-hand side with $l' = l_r \bullet h^{-1}$ for all vertices $v \in V^* \cup V_{c*}$ and equals $l$ for all other vertices $v \in V' \backslash (V^* \cup V_{c*})$.

The following subdefinitions are used: To construct the graphmorphism $h$ chose a vertex set $V^*$ disjoint from $V$ and $|V^*| = |V_{\tilde{r}}|$. Based on $V^*$, $h_l$ can be extended to an injective map $h: V_l \cup V_r \to V \cup V^*$ such that $h|_{V_l} = h_l$ and $h: V_r \backslash V_l \to V^*$. The edge sets $Emb_{del}$ and $Emb_{ins}$ are defined by the evaluation of the embedding descriptions based on the edges embedding the image of the left-hand side in the rest graph, $E_{emb} = inc_g(V_{\tilde{g}}) \cap inc_g(V \backslash V_{\tilde{g}})$. Embedding edges are deleted whenever there is an embedding description with a fitting cut-description: $Emb_{del} = \{e \in E_{emb} | \exists (c, p) \in M: fits_{h,l}(c, e)\}$. New embedding edges will be inserted according to a corresponding paste-description: $Emb_{ins} = \bigcup_{m \in M} eval_{h,l}(m, Emb_d)$.

A rule set is a collection of rewrite rules which are consistently labelled. They have at least one common unique label.

**Definition 3** rule set

Let $r_i = (g_{l_i}, g_{r_i}, M_i)$ be slv-rules for $i = 1 \ldots n$. The set $R = \{r_i | i = 1 \ldots n\}$ is a *slv-rule set* with unique labels $U(R)$ iff $\{g_{l_i} | i = 1 \ldots n\}$ is consistently labelled, and $U(R) = \bigcap_{i=1}^{n} U(r_i)$ is not empty.

A rule set is applicable to a graph if there exists at least one applicable rule in the set. The application of this rule is the result of the set rewrite step.

**Definition 4** applicability of a rule set, set rewrite step

Let $g$ be a graph. Let $R = \{(l_i, r_i, M_i) | i = 1 \ldots n\}$ be a slv-rule set. $R$ is *applicable* to $g$ iff $\exists \tilde{g} \subseteq_p g \exists g_l \in \bigcup_{i=1}^{n} inst(l_i): g_l \cong \tilde{g}$. If $R$ is applicable on $g$, a graph $g'$ is the result of a *slv-set rewrite step*, $g \Rightarrow^R g'$, iff $\exists r \in R: g \to^r g'$.

Finally the GRAM graph grammar is defined. There are two main differences to common grammar types. First, the usage of rule sets as defined before and second the explicit enumeration of unique labels.

**Definition 5** GRAM-graph grammar

Let $\emptyset \neq U \subseteq \Sigma_V$, $g = (V, E, l)$ be a graph over $\Sigma_V, \Sigma_E$ with $U \subseteq l(V)$ and for all $v \in V$ with $l(v) \in U$ does not exist $v' \in V$ with $v \neq v'$ and $l(v) = l(v')$. Let $S = \{R_i | i = 1 \ldots n\}$ be a set of slv-rule sets with $U \cap U(R_i) \neq \emptyset, \forall i = 1 \ldots n$. A *GRAM-graph grammar* over $\Sigma_V, \Sigma_E$ with unique labels $U$, initial graph $g$, and a set of rule sets $S$ is the tuple $gg = (\Sigma_V, \Sigma_E, U, g, S)$.

## 5.2 The Construction of a Search Tree

An optimization for the implementation of the rewriting step of a GRAM graph grammar will be presented in terms of partial order theory. The base of the optimization lies in the usage of rule sets. Given such set one has to decide which element of the set can be applied. We now can test the applicability of each rule sequentially to find an applicable element of the set. In the optimized implementation we test the applicability of the rules with less effort. Obviously, it is not necessary to test the existence of subgraphs common to several rules repeatedly. Therefore, we analyse the existence of common subgraphs of the left-hand sides of the rules in the set. We sketch the analysis which is to be performed in the compiler. The analysis will output a search tree which controls the execution of the applicability test.

The applicability test is performed by traversing the graph to be rewritten according to the left-hand side of a rule. Thus, the subgraphs covered by the traversal are all partial, connected subgraphs of the underlying graph. For a (slv-)graph $g$ we define the set of all connected partial subgraphs of $g$ $S(g) = \{g' | g' \subseteq_p g \wedge g' \text{ connected}\}$. Since we deal with rule sets we define for any set $G = \{g_1, \ldots, g_n\}$ of consistently labelled set of (slv-)graphs $S(G) = \bigcup_{i=1}^{n} S(g_i)$.

We define an order relation on the set of all partial connected subgraphs. Let $G$ be a finite, consistently labelled set of (slv-)graphs, and $g_1, g_2 \in S(G)$. The graph $g_1$ is smaller than $g_2$ with respect to G, $g_1 \leq_G g_2$ iff in case $V_1 = V_2$: $E_1 \subseteq E_2$ or in case $V_1 \subset V_2$ there exists a graph $g \in G$ such that $G(V_1, g_2) = G(V_1, g)$.

Based on these definitions, we give the construction of a tree which represents the common subgraphs in the left-hand sides of a rule set. This tree is the basic structure of the input of the code generation phase in the compiler. The produced code performs the applicability test for rule sets with fewer search steps.

**Construction 6** $tree(g_{\varnothing}, G)$, tree over $S(G)$

Let $G$ be a finite, consistently labelled set of (slv-)graphs. A tree over $S(G)$ with root $g_{\varnothing}$ and leaves $g_i \in G$ is constructed by $tree(g_{\varnothing}, G)$ using the following definitions:

$tree(g_r, \varnothing) = \varnothing$,

$tree(g_r, G_0) = ch \cup \bigcup_{i=1}^{n} tree(u_i, \overline{G_i})$

where graph $\tilde{g}$ is a maximal lower bound of $G_0$, and $ch$ is a maximal chain between $g_r$ and $\tilde{g}$. For all upper neighbours $u_i$ of $\tilde{g}$ we define the sets $\overline{G_i} = \{g \in G_{i-1} | u_i \leq_G g\}$ and $G_i = G_{i-1} \backslash \overline{G_i}$ for $i = 1 \ldots n$.

We apply the construction in the analysis of a rule set $R = \{(g_{l_i}, g_{r_i}, M_i) | i = 1 \ldots n\}$ as follows. Let $U$ be the set of unique vertex labels of the grammar under concern. When we calculate $tree(g_0, \{g_{l_i} | i = 1 \ldots n\})$ for a graph $g_0 = (\{v\}, \varnothing, l)$ with $l(v) \in U \cap U(R)$ then we have an optimized search algorithm for the left-hand sides of the rule set. We just have to traverse the tree. When we reach a leaf, we have successfully constructed a map from a left-hand side of a rule to some vertices in the graph. This map will serve as the base for the application of the rule.

## 5.3 The Code Generation Rules

For the code generation for a given graph grammar with rule sets, we assume that the left-hand sides of a rule set are analysed and transformed to a search tree according to Construction 6. Its nodes represent individual search steps. The leaves are augmented with information about the corresponding rewrite rule.

The first set of code generation rules splits the rule sets and the initial graph to be dealt separately. The following variables are used: g graph, v vertex, s source, t target, vl vertex label, el edge label, tr tree, cr crown, the sequence of subtrees rooted at a node, r rewrite rule, n search tree node, and er embedding rule. The variables iv, ie, cv, dv, and de denote inserted, changed, or deleted vertices resp. edges. x* is a sequence of x. Terminal symbols are "in" and "out". The labels *n* and *lab.n* are in Dewey notation over natural numbers.

```
                GG [ g r* ] U   =>       INIT [ g ] U
                                         RULESET [ r* ] 1
    RULESET [ v vl cr r* ] n    =>   n:  find-unique-vertex vl v n.0
                                         TREE [ cr ] n.1
                                   n.0:  not-applicable
                                         RULESET [ r* ] (n+1)
               RULESET [ ] n    =>   n:  skip
```

The compilation of the initial graph is supplied with the grammar set of unique vertex labels *U*. The search tree of a rule set is rooted by a vertex with unique label. Thus, a search instruction is output and compilation of the tree proceeds with compiling the crown made up of a list of subtrees.

```
                    TREE [ r ] lab   =>     chose-binding
                                            RULE [ r ]
               TREE [ n cr ] lab.l   =>     TREE [ n ] lab.(l+1)
                                            TREE [ cr ] lab.l.1
                                            pop-binding
              TREE [ tr tr* ] lab.l  =>     TREE [ tr ] lab.l
                          lab.(l+1):        TREE [ tr* ] lab.(l+1)
               TREE [ s el t ] lab   =>     exists-edge s el t lab
   TREE [ t el vl_1 ... vl_n s in ] lab  => start-find-neighbour
                                            FOR i=1..n
                                                find-neighbour-backward t el vl_i s
                                            end-find-neighbour lab
  TREE [ s el vl_1 ... vl_n t out ] lab  => ...
```

Each tree is traversed and its nodes are transformed to search instructions. When a leaf is reached, the code for the rule is generated. The search for set labelled vertices is split into individual search instructions.

```
 RULE [ iv* ie* cv* dv* de* er* ]   =>   INS [ iv* ]; INS [ ie* ]; CHG [ cv* ];
                                         EMB [ er* ]; DEL [ dv* ];
                                         DEL [ de* ]; end-rewrite-step;
                INS [ v vl  lv* ]   =>   add-vertex v vl; INS [ lv* ];
                INS [ s el t  e* ]  =>   add-edge s el t; INS [e* ];
                CHG [ v vl lv* ]    =>   new-label v vl; CHG [ lv* ];
                    DEL [ v v* ]    =>   delete-vertex v; DEL [ v* ];
                DEL [ s el t e* ]   =>   delete-edge s el t; DEL [ e* ];
      DEL [ ], CHG [  ], INS [ ]    =>   ()
```

The code generation function **RULE** outputs the instructions for the actual rewriting. The rewriting effect of a rule is given by lists of inserted resp. deleted vertices and edges, vertices with changing label, and embedding rules.

Further code generation functions for the initial graph and embedding rules are skipped for sake of brevity.

## 6 Conclusion

We gave the specification of an abstract graph rewriting machine. It defines the middle layer of a rewriting system for graph grammars. The instruction set covers all relevant elements of a graph rewriting step. There-

fore, the graph rewriting machine serves as an interface between different grammar types on one side and different implementations on the other. It is a suggestion for the common coin of various graph grammar rewriting projects.
We have shown that the separation of compilation and execution can be exploited for the analysis of graph grammars. Especially speeding-up the rewrite step is the major goal for a compile-time analysis.
To give an example for possible optimization, we presented a graph grammar with rule sets. We sketched the optimization method which provides a matching of maximal common subgraphs. Hence several elements of a rule set are applied at one time. As a consequence the time needed for the application of a rule set is reduced. The main compilation rules for the analysed grammar were listed.

The prototype of the rewriting machine is implemented. It is the central component of a project concerning efficient grammar graph rewriting. Further analysis steps will be included in the compiler to improve the efficiency of the generated code. An application area is the analysis of parallel systems.

## References


[Dö92] Dörr, Heiko: 'Monitoring with Graph-Grammars as formal operational Models', *Report B-19-92*, Fachbereich Mathematik, Freie Universität Berlin, 1992.

[EnSch89] Engels, Gregor; Schäfer, Wilhelm: '*Programmentwicklungsumgebungen. Konzepte u. Realisierung*'; Teubner, Stuttgart, 1989

[Gött88] Göttler, Herbert: 'Graphgrammatiken in der Softwaretechnik', *Informatik-Fachberichte 178*, Springer, Berlin, 1988.

[Gou88] Gould, Ronald: '*Graph Theory*'; The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.

[Him89] Himsolt, Michael: 'Graphed: An interactive Graph Editor', in *STACS 89*, *LNCS 349* Spinger Verlag, Berlin, 1989.

[Joh84] Johnsson, Thomas: 'Efficient Compilation of Lazy Evaluation', in *ACM SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices, **19**, (6) 58-69, (1984)

[Lew88] Lewerentz, Claus: 'Interaktives Entwerfen großer Programmsysteme', *Informatik-Fachberichte 194,* Springer, Berlin, 1988.

[PeJo87] Peyton Jones, Simon: '*The implementation of functional programming languages*', Prentice Hall, Hemel Hempstead, 1987.

[Schü90] Schürr, Andreas: 'Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language', in W. Nagl (ed.) *Fifthteenth International Workshop WG '89, Graph-Theoretic Concepts in Computer Science*, Castle Rolduc, The Netherlands, June 1989, *LNCS 411*, Springer, Berlin, 1990, pp. 151-166.

[Schü91] Schürr, Andreas: '*Operationales Spezifizieren mit programmierten Graphersetzungssystemen*', Deutscher Universitäts-Verlag, Wiesbaden, 1991.

[Zün92] Zündorf, Albert: 'Implementation of the imperative/rule based language PROGRES', *Aachener Informatik-Berichte Nr. 92-38*, RWTH Fachgruppe Informatik, Aachen, 1992.