Monitoring with Graph-Grammars as formal operational Models

Heiko Dörr
Institut für Informatik
Freie Universität Berlin
Nestorstr. 8-9
D-1000 Berlin 31
doerr@inf.fu-berlin.de

21/9/92

This paper presents a monitoring method which reduces the costs of event-tracing. To support the monitoring of a system-run, a model is used to manage the system's information which has been collected since the beginning of the run. By this means the collection of redundant event-trace information is avoided. The background and usage of model-based monitoring is discussed, too.
Graph-grammars with grouped rules are proposed and defined as an operational formalism to model the behaviour of a system. They are especially suitable for the description of distributed systems whose functionality can be characterized by structural changes. In graph-grammar models, relevant functional parts become directly addressable. Therefore, system analysis becomes easier.
The formal model of a parallel graph reduction machine is introduced in order to illustrate the monitoring method.

1 Introduction

The main reason for the introduction of parallel computers is the expected speed-up compared to sequential ones. Therefore, **parallelization of applications is necessary**. But parallelization introduces problems of distributed and parallel programming into design and implementation. Some of these problems are handled using parameters to adapt a parallel application to a specific hardware platform. The load distribution strategy for example is one component which has to be fine-tuned by the choice of parameter values.

When the run-time behaviour of the application is unknown, **optimal adaptations cannot be found** at all. The properties of interest are not only quantitative measures like execution times, numbers of messages or processes, and idle times but also qualitative measures like routes of messages or dynamic dependencies between processes. Intense measurements are necessary in order to analyse an application with respect to properties like the ones mentioned above. Thus, monitoring is the first step to a solution of optimization problems.

Software monitoring techniques have, on the one hand, the advantage of being available without going through a long process of hardware development. But on the other hand, software monitoring always intrudes and alters the behaviour of the monitored system. Therefore, when software monitoring is chosen, the disturbance of the monitored system should be minimized.

There are two main kinds of software-monitoring: **event-sampling and event-tracing**. In the first approach the execution is stopped at definite intervals of time and information on the system-state is collected. In event-tracing, code is introduced in the monitored system which sends messages to a record file during the instrumented system's run. The list of messages, called the event-trace, mirrors the run of the system. To get information on the system, the event trace has to be analysed.

Commonly, **successive experiments** are carried out to study different and mostly independent aspects of the run-time behaviour. An example for this approach are the measurements presented by Hammond and Peyton Jones ([HamPey90]). In order to monitor the run-time behaviour of the GRIP multiprocessor, it is instrumented such that only data on processor activity and parallelizing tasks are collected. In the case other aspects are important, other experiments have to be designed and the system is instrumented again.

The analysis strategy using successive experiments has **three drawbacks**.

- Each instrumentation **alters the behaviour of the system in a specific way**. So, the results of the different experiments based on various instrumentations cannot be correlated because different systems were measured in the experimentation runs.
- In each experiment the measured system has to be **instrumented with respect to the requested data**. Thus, each experiment costs the effort of an individual instrumentation.
- Each **instrumentation presupposes a thorough knowledge** of all details of the measured system. When the system designers are gone nobody will be able to do any further instrumentation.

The straightforward way to avoid these drawbacks is to collect great quantities of data which, later on, can be analysed under various concerns. Even if the goal of an experiment is not known at monitoring time, the experiment may be run using the "old" data because they give a consistent view of the system behaviour. Thus, **monitoring results are reused**. Furthermore, a correlation between various experimental results can be stated since they are based on the same instrumented system.

To collect this fine-grained data, event-tracing is adequate. But obviously, there is an immense intrusion when a general trace facility is used naively. This paper proposes **model-based monitoring to minimize this intrusion** by reduction of the amount of collected data. The approach is refined in this paper by the usage of a formal operational model for the support of model-based monitoring.

2 Model-based Monitoring

There are **three sources of intrusion** to the system by event-trace monitoring. First, collecting state information and building an event message slows down the execution of the system. Second, each event message which has to be sent puts a supplementary load on the communication service which, third, is also influenced by the average length of event messages. The consequence of intrusion is usually a performance degradation caused by additional processor's load and the communication overhead. This leads to a change in the control flow of parallel systems.

2.1 Background

Model-based monitoring is a common technique to minimize the overhead of event-tracing. At first sight, in ordinary event-trace monitoring **each message contains information on the system-state** to distinguish individual events. This information has to be collected and sent repeatedly for each event although it may be already included implicitly in former event messages. This source of intrusion is avoidable, since redundant information is collected. Model-based monitoring exploits the causal relation between events and detects redundant information by storing the information already known according to a model of the system.

To deepen this observation and to improve notational precision, the close **relation of event-trace monitoring and the standard state/transition description** of system behaviour is sketched. An abstraction of the static system's properties is defined by selecting distinct system components. A state is defined by the instantiation of this abstraction which gives explicit values to the components. This state might occur during a system-run which is described as a sequence of states. The dynamic behaviour of the system is represented by transition rules defined on the basis of the system abstraction. An individual transition happening during a system-run from a state to its successor state must obey a transition rule. The full description of a transition contains at least the pre-transition state and the changes caused by the transition.

To sum up:

- The **abstract behaviour of a system** is described in terms of abstract system views and transition rules.
- The **concrete behaviour of a running system** is formulated in terms of states and transitions.

The notions used in event-tracing can be defined in a similar way. Instead of looking to states, a system-run can be described as the sequence of transitions performed. In event-tracing this sequence of transitions is recorded in the **event-trace**. Its elements are called **event-instances** instead of transitions. An event comprising a set of event-instances is related to a transition rule, thus describing the similarity of its instances.

Event-trace monitoring by recording all transitions is too expensive in terms of runtime and space. For practical use, the close relation between event instances and states is exploited. As said before, given an initial state, the **sequence of states can be reproduced from an event-trace**. This interpretation makes the trace analysis easier since the sequence of states becomes the subject of system-run analysis. System components can be accessed directly.

A **reduction of intrusion** is caused by the fact, that no whole event-instances have to be included in the event-trace, but just the events comprising several instances. Because an actual state representation is available to the analysis tool, the event instance can be deduced from the signalled event. By this means, the event-trace needs not to be a protocol of changes but consists of events which mark specific changes.

2.2 Usage

The first step in model-based monitoring is the **construction of an abstract model** of the monitored system. This abstraction should comprise all relevant functional units and data structures because it serves as a guideline for the definition of a specific data structure representing the system-state. The instrumentation of the system has to be done based on the model. For the analysis of the event-trace produced by a measurement run, the defined data structure is used to keep the system's state. It is accessed and updated by the event-analyser which may operate on- or off-line.

Haban and Wybranietz developed a monitoring system which uses an **informal model** ([HaWy90]). They defined a set of events which describe the relevant transition rules of their distributed system. Based on this set, they build a database which keeps track of the system-state. It has entries for the system components as well as for each process running. The event-trace analyser runs on-line accessing and updating the database. Since Haban and Wybranietz did not use an operational formal model, they had to program their monitoring system without tool support. The use of their method needs a lot of initial effort because for each monitoring application the model, the database, and the event-trace analyser have to be designed and implemented from scratch.

To save some effort the use of formal operational models is proposed here. The use of **formal models has three advantages** for system's monitoring.

- First the design of a model can be supported by appropriate editors.
- Second the event-trace can be analysed by a set of tools which operate independent on the specific system. Thus event-trace analysers become reusable.
- Third, the use of a formalism is a necessary precondition for reasoning about a model.

The **advantage of operational models** in contrast to static ones lies in the ease of analysis tools' development. A simple example of operational models are finite automata which model state transitions. Because the dynamic behaviour of monitored systems is expressed explicitly in the operational model, it can be directly accessed by the analyser. Moreover, operational models can be used as executable images of a monitored system. Each step of the monitored system, which is signalled by an event message, can be interpreted by performing an operation on a system's model. When a formal operational model is used, the updating procedure can be automated.

The use of graph-grammar models for monitoring support will illustrate this general idea of event-trace monitoring based on formal, operational models. But first the basic formalism has to be introduced.

3 Graph-Grammars with Grouped Rules

Nagl mentions three **motivations for the usage of graph-grammars**, namely specification, description of systems, and decision support for grammar-definable properties ([Nag86a]). Depending on the intended use an algebraic, a set theoretic, or a parallel rewriting type of graph-grammars is more suitable. Graph-grammars of the set theoretic type are a well-known formalism for the operational specification of software. They have been used in projects like IPSEN ([EnSch89], [Lew88], [Eng86], [Nag86b]), those around the PAGG definition ([Gött88]), and others described in [FeSch90], [Schü90]. Thus, this approach is adopted in the following.

3.1 Ordinary Graph-Grammars

Graph-grammars use directed, vertex- and edge-labelled graphs $g = (\mathcal{V}, \mathcal{E}, l) \in \mathcal{G}$ where $\mathcal{G}$ denotes the set of graphs[1]. The labelling function $l: \mathcal{V} \rightarrow \Sigma_{\mathcal{V}}$ assigns a label to each vertex, element of the vertex set $\mathcal{V}$. $\mathcal{E} \subseteq \mathcal{V} \times \Sigma_{\mathcal{E}} \times \mathcal{V}$ is the set of directed edges labelled with elements of $\Sigma_{\mathcal{E}}$. A graph-grammar $gg = (\Sigma_{\mathcal{V}}, \Sigma_{\mathcal{E}}, g_0, \mathcal{R})$ contains two non-empty label sets $\Sigma_{\mathcal{V}}, \Sigma_{\mathcal{E}}$, an initial graph g_0, and a set of graph rewriting rules $\mathcal{R}$. A **graph rewriting rule** or production $r = (g_l, g_r, \mathcal{M}) \in \mathcal{R}$ has two graphs g_l and g_r as left- and right-hand side, and a set of embedding rules $\mathcal{M} \subseteq \mathcal{C} \times \mathcal{P}$. Edges which are used to embed a graph have to fit to a "cut-edge-description"

$$c \in \mathcal{V} \times \Sigma_{\mathcal{E}} \times \Sigma_{\mathcal{V}} \times \{in, out\} = \mathcal{C}.$$

They are replaced eventually by edges described by a "paste-edge-description" $p \in (\mathcal{V} \times \Sigma_{\mathcal{E}} \times \{in, out\}) \cup \{del\} = \mathcal{P}$. These descriptions are defined such that their interpretation addresses edges during a rewrite step which is defined later.

After the definition of the basic structures, the operational aspects of graph-grammars are approached. The concept of **applicability of a production** $r = (g_l, g_r, \mathcal{M})$ to a graph g is defined by the existence of a partial subgraph g' of g isomorphic to g_l.

Two graphs $g_1 = (\mathcal{V}_1, \mathcal{E}_1, l_1)$ and $g_2 = (\mathcal{V}_2, \mathcal{E}_2, l_2)$ are **isomorphic** iff there exists a bijective mapping on vertices $h: \mathcal{V}_1 \rightarrow \mathcal{V}_2$ such that

$$(\forall v_1 \in \mathcal{V}_1, l_1(v_1) = l_2(h(v_1))) \wedge$$
$$(\forall v_1, v_2 \in \mathcal{V}_1, \forall a \in \Sigma_{\mathcal{E}}, (v_1, a, v_2) \in \mathcal{E}_1 \Leftrightarrow (h(v_1), a, h(v_2)) \in \mathcal{E}_2).$$

The corresponding graph isomorphism is denoted by $\hat{h}$.

1. The mathematical symbols in the sequel are introduced in [Epp90]. Additionally, given the functions f and g, the restriction of a function f on a set $\mathcal{S}$ is denoted by $f|_{\mathcal{S}}$ and function composition by $f \bullet g$. Italic chancery symbols denote sets. Variable symbols are chosen to remind to their natural language notions. Variables of the same type occurring in a formula are distinguished by numeric or alphabetic indices. The last kind of indication is chosen to be explanatory, e.g. g_l denotes the **g**raph being the **l**eft-hand-side of a rule.

The graph $g_s = (\mathcal{V}_s, \mathcal{E}_s, l_s)$ is a **partial subgraph** of $g = (\mathcal{V}, \mathcal{E}, l)$, written $g_s \subseteq g$, iff

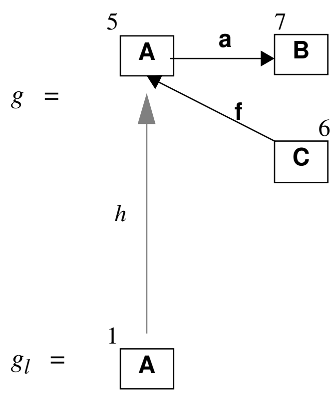
$$(\mathcal{V}_s \subseteq \mathcal{V}) \wedge (l_s|_{\mathcal{V}_s} = l|_{\mathcal{V}_s}) \wedge (\mathcal{E}_s \subseteq \mathcal{E} \cap (\mathcal{V}_s \times \Sigma_{\mathcal{E}} \times \mathcal{V}_s)).$$

Note that the definition of the partial subgraph does not require the equality of the sets of edges. This leads to a relaxation of the common notion of applicability employing proper subgraphs in the sense, that the edges of g_l define a minimal condition on an isomorphic subgraph.

The boolean function

$$fits_{h,l}(c, (v_1, a, v_2)) = (c = (h^{-1}(v_1), a, l(v_2), out)) \vee (c = (h^{-1}(v_2), a, l(v_1), in))$$

determines whether the cut-edge-description c describes the edge (v_1, a, v_2) under a given bijective vertex mapping h and vertex labelling function l. By the definition of $fits_{h,l}$, the cut-edge-description $c = (v, l_1, l_2, out)$ describes an edge with label l_1 starting at the vertex which is the image of v under h and going to a vertex with label l_2.



$\mathcal{M} = \{m_1, m_2\}$

$m_1 = ((1, \mathbf{a}, \mathbf{B}, out), \{del\})$

$m_2 = ((1, \mathbf{f}, \mathbf{D}, in), \{del\})$

$(1, \mathbf{a}, \mathbf{B}, out)$ fits to $(5, \mathbf{a}, 7)$

$(1, \mathbf{f}, \mathbf{D}, in)$ does not fit to $(5, \mathbf{a}, 7)$ and $(6, \mathbf{f}, 5)$

Figure 1 The Fit of cut-edge-descriptions

Figure 1 gives an example. It shows the left-hand side graph and the two cut-edge-descriptions of a rule $(g_l, g_r, \mathcal{M})$, and a graph g on which the rule will be applied under the isomorphism h. In the example, the description $(1, \mathbf{f}, \mathbf{D}, in)$ mentioned in m_2 does not fit to $(6, \mathbf{f}, 5)$ because $l_g(6) \neq \mathbf{D}$.

A **derivation step** transforms a graph $g = (\mathcal{V}_g, \mathcal{E}_g, l_g)$ into a graph $g' = (\mathcal{V}_{g'}, \mathcal{E}_{g'}, l_{g'})$ 2). It uses an applicable production $r = (g_l, g_r, \mathcal{M})$. Written $g\text{-}r\text{->}g'$, the derivation step is defined as follows: Choose a graph isomorphism $\hat{h}_l$ such that

2. Objects of an intermediate step in a derivation are denoted by $\tilde{x}$, parts of the resulting graph not occurring in the initial one by x'.

$\hat{h}_l(g_l) = \tilde{g}_l \subseteq g$. This isomorphism exists since r is applicable to g. Cutting $\tilde{g}_l$, the isomorphic image of the left-hand side out of g leaves the **residual graph** $\tilde{g} = (\mathcal{V}_{\tilde{g}}, \mathcal{E}_{\tilde{g}}, l_{\tilde{g}})$ where the components of the graph are defined as:

$$\mathcal{V}_{\tilde{g}} = \mathcal{V}_g - \mathcal{V}_{\tilde{g}_l}$$

$$\mathcal{E}_{\tilde{g}} = \mathcal{E}_g - (\mathcal{E}_{\tilde{g}_l} \cup \mathcal{E}_i \cup \mathcal{E}_d)$$

$$l_{\tilde{g}} = l_g \big|_{\mathcal{V}_{\tilde{g}}}.$$

The set of **inner edges** is $\mathcal{E}_i = \{ (v_1, a, v_2) \in \mathcal{E}_g - \mathcal{E}_{\tilde{g}_l} \big| v_1, v_2 \in \mathcal{V}_{\tilde{g}_l} \}$ whereas the set of **dangling edges** is defined as

$$\mathcal{E}_d = \{ (v_1, a, v_2) \in \mathcal{E}_g \big| (v_1 \in \mathcal{V}_{\tilde{g}_l} \wedge v_2 \in \mathcal{V}_{\tilde{g}}) \vee (v_2 \in \mathcal{V}_{\tilde{g}_l} \wedge v_1 \in \mathcal{V}_{\tilde{g}}) \}.$$

The dangling edges connect the isomorphic image of the left-hand side $\tilde{g}_l$ and the residual graph $\tilde{g}$. Before inserting g_r in $\tilde{g}$, the vertices of g_r are renamed to prevent accidental identification of vertices. Therefore a vertex set $\mathcal{V}_{\tilde{g}_r}$ is chosen such that there is an bijective mapping $h_r: \mathcal{V}_{g_r} \to \mathcal{V}_{\tilde{g}_r}$ and $\mathcal{V}_{g_r} \cap \mathcal{V}_{\tilde{g}_r} = \varnothing$. According to this right-hand side isomorphism, the **embedding edges**

$$\mathcal{E}_m = C_{in} \cup C_{out} \cup I_{in} \cup I_{out}$$

are determined by $\mathcal{E}_d$, $\hat{h}_l$, l_g, and the embedding rules listed in $\mathcal{M}$.

C_{in} and C_{out} contain edges which are described by a paste-edge-description under the condition that there is a dangling edge $(v_1, a, v_2) \in \mathcal{E}_d$ which fits to the corresponding cut-edge-description. These edges may be changed to incoming or outgoing ones depending on the direction defined by the paste-edge-description. C_{in} and C_{out} are defined as follows:

$$C_{out} = \begin{array}{l} \{ (h_r(v), b, v') \big| \exists (v_1, a, v_2) \in \mathcal{E}_d, \exists (c, p) \in \mathcal{M}, \exists v \in \mathcal{V}_{g_r} \\ fits_{h_l, l_g}(c, (v_1, a, v_2)) \wedge (p = (v, b, out)) \wedge (v' \in \{v_1, v_2\} \cap \mathcal{V}_{\tilde{g}}) \} \end{array}$$

$$C_{in} = \begin{array}{l} \{ (v', b, h_r(v)) \big| \exists (v_1, a, v_2) \in \mathcal{E}_d, \exists (c, p) \in \mathcal{M}, \exists v \in \mathcal{V}_{g_r} \\ fits_{h_l, l_g}(c, (v_1, a, v_2)) \wedge (p = (v, b, in)) \wedge (v' \in \{v_1, v_2\} \cap \mathcal{V}_{\tilde{g}}) \}. \end{array}$$

Note that due to the definition of $\mathcal{E}_d$, $\{v_1, v_2\} \cap \mathcal{V}_{\tilde{g}}$ contains either v_1 or v_2.

A dangling edge which does not fit to any cut-edge-description of the applied rule may not be deleted. It can be transformed if the origin of the adjacent vertex under h_l occurs also in the right-hand-side with the same label. These edges are members of I_{out} and I_{in} formally defined by

$$I_{out} = \begin{array}{l} \{ (v_1, a, h_r \bullet h_l^{-1}(v_2)) \,\big|\, \exists e \in \mathcal{E}_d, \forall (c, p) \in \mathcal{M},\ e = (v_1, a, v_2) \\ \neg fits_{h_l, l_g}(c, e) \wedge h_l^{-1}(v_1) \in \mathcal{V}_{g_r} \wedge (l_{g_r} \bullet h_l^{-1}(v_1) = l_{g_l} \bullet h_l^{-1}(v_1)) \ \} \end{array}$$

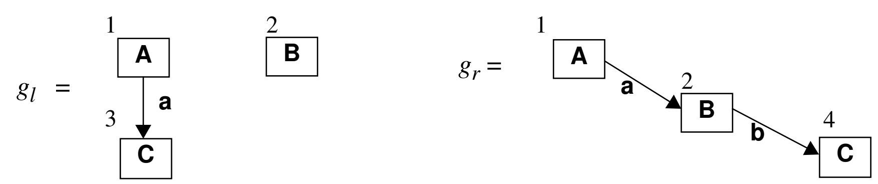
$$I_{in} = \begin{array}{l} \{ (h_r \bullet h_l^{-1}(v_1), a, v_2) \,\big|\, \exists e \in \mathcal{E}_d, \forall (c, p) \in \mathcal{M},\ e = (v_1, a, v_2) \\ \neg fits_{h_l, l_g}(c, e) \wedge h_l^{-1}(v_2) \in \mathcal{V}_{g_r} \wedge (l_{g_r} \bullet h_l^{-1}(v_1) = l_{g_l} \bullet h_l^{-1}(v_1)) \ \}. \end{array}$$

Inner edges are added to $\hat{h}_r(g_r)$ if the sources of the adjacent vertices occurring on the left-hand side are identically passed to the right-hand side. They are contained in the set of unchanged edges

$$\mathcal{E}_u = \{ (h_r(v_1), a, h_r(v_2)) \,\big|\, \exists (v_1, a, v_2) \in \mathcal{E}_i, \forall j \in \{1, 2\},$$
$$h_l^{-1}(v_j) \in \mathcal{V}_{g_r} \wedge (l_{g_r} \bullet h_r^{-1}(v_j) = l_{g_l} \bullet h_l^{-1}(v_j)) \ \}.$$

Based on these definitions the **derived graph** $g' = (\mathcal{V}_{g'}, \mathcal{E}_{g'}, l_{g'})$ is defined by its components

$$\mathcal{V}_{g'} = \mathcal{V}_{\tilde{g}} \cup \mathcal{V}_{\tilde{g}_r}$$
$$\mathcal{E}_{g'} = \mathcal{E}_{\tilde{g}} \cup \mathcal{E}_u \cup \mathcal{E}_m$$
$$l_{g'}\big|_{\mathcal{V}_{\tilde{g}}} = l_g \qquad l_{g'}\big|_{\mathcal{V}_{\tilde{g}_r}} = l_{\tilde{g}_r}.$$



$$\mathcal{M} = \{m_1, m_2, m_3, m_4\} \qquad m_i = (c_i, p_i)$$
$$m_1 = ((1, \mathbf{f}, \mathbf{E}, out), \{del\})$$
$$m_2 = ((3, \mathbf{a}, \mathbf{B}, in), \{del\})$$
$$m_3 = ((2, \mathbf{e}, \mathbf{D}, in), (1, \mathbf{e}, in))$$
$$m_4 = ((2, \mathbf{e}, \mathbf{D}, in), (4, \mathbf{e}, out))$$

Figure 2 Rewrite Rule $r = (g_l, g_r, \mathcal{M})$

3.2 Example of a Derivation Step in ordinary Graph-Grammars

This example has been constructed such that several kinds of edge replacement during a derivation step can be shown. Thus, it is not a natural example. The applied graph rewrite rule $r = (g_l, g_r, \mathcal{M})$ is given in Figure 2. The according derivation is shown in Figure 3.

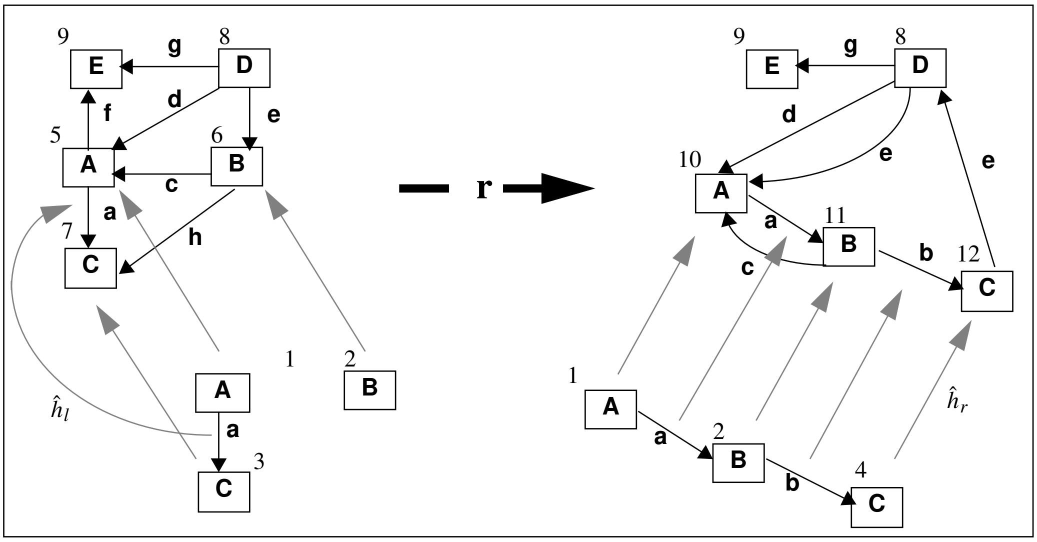


Figure 3 Derivation Step g-r-> g'

To apply r to g the bijective mapping $h_l = \{(1, 5), (2, 6), (3, 7)\}$ must be used. This leads to the following sets of inner and dangling edges:

$$\mathcal{E}_i = \{(6, \mathbf{c}, 5), (6, \mathbf{b}, 7)\}$$

$$\mathcal{E}_d = \{d_1, d_2, d_3\} \; with$$

$$d_1 = (5, \mathbf{f}, 9) \qquad d_2 = (8, \mathbf{d}, 5) \qquad d_3 = (8, \mathbf{e}, 6).$$

The insertion of g_r into the residual graph is done under application of the graph isomorphism based on $h_r = \{(1, 10), (2, 11), (4, 12)\}$.

The cut-edge-description c_1 fits to d_1. Because $p_1 = \{del\}$, the edge d_1 is not used for embedding. Edge d_3 is described by c_3 and c_4. It is duplicated since the two embedding rules m_3 and m_4 contain paste-descriptions. Thus, the edges $(8, \mathbf{e}, 10) \in C_{in}$ and $(12, \mathbf{e}, 8) \in C_{out}$ are generated. No edge of $\mathcal{E}_d$ is described by c_2, therefore m_2 is ignored.

An example for an identical embedding is d_2. No cut-edge-description fits to d_2, but the vertex 1 labelled **A** exists on both sides of the rule. Thus, the edge $(8, \mathbf{d}, 5)$ is transformed to $(8, \mathbf{d}, 10) \in I_{in}$. The set of outgoing identical edges I_{out} is empty.

The inner edge $(6, \mathbf{c}, 5)$ is transferred from g to g' since the vertices 1 and 2 occur on both sides of the rule with the same labels. The second inner edge $(6, \mathbf{b}, 7)$ is deleted because the vertex 3 which is mapped to vertex 7 does not exist in g_r. The deletions takes place although vertex 4 of g_r has the same label.

3.3 Graph-Grammars with Grouped Rules

Up to this stage, the operational model given by a graph-grammar is a fine grained description of the system. Each rewrite rule represents a small number of transitions since it has to fit exactly to that part of the state which is subject to change. The rest of the graph representation of the system will be stable. Thus, the grain size of a rewrite rule, as defined now, lies between that of a transition and a transition rule. The first covers the whole state, whereas the second is mostly applicable to a set of states. The **descriptive level of transition rules** can be introduced into graph-grammars by the notions of **grouped rules and set derivation**. By this means the number of different events is reduced. So, this reduces the amount of data which are requested for the description of a system-run.

Given a graph-grammar $gg = (\Sigma_{\mathcal{V}}, \Sigma_{\mathcal{E}}, g_0, \mathcal{R})$, the construction of a graph-grammar with grouped rules $g^3r = (\Sigma_{\mathcal{V}}, \Sigma_{\mathcal{E}}, g_0, \mathcal{S})$ is done by classifying the rules of $\mathcal{R}$. This is done by the definition of a set of rule groups $\mathcal{S} = \{s_i | i \in \{1...n\}\} \subseteq \wp(\mathcal{R})$. The construction of grouped rules can be done by common set operations as well as by the use of parameterised rewriting rules. For example the labelling functions of left- and right hand side of rules can be parameterised such that a set of rules is applicable to a number of states.

According to this refinement, a **set derivation step** $=s=>$ is defined in terms of the derivation $-r->$ of ordinary graph-grammars. A set of rules $s \in \mathcal{S}$ is applicable on a graph g iff $\exists r \in s$ and r is applicable to g. The graph g' is set derivable from g, written $g=s=>g'$, iff $\exists r \in s$ with $g-r->g'$.

For the practical usability of graph-grammars for modelling, the amount of information input in a derivation step is essential. The definitions of the derivation step in ordinary graph-grammars found in the literature vary in the choice of the isomorphic subgraph as the handle of the derivation mechanism. Nagl assumes that an arbitrary isomorphic subgraph can be chosen in the first step of the derivation ([Nag79]). Thus, the **application of a graph rewrite rule has a non-deterministic result**. Kaul on the other hand claims that the specification of a graph isomorphism is a necessary input for a derivation step ([Kau86]). This guarantees a deterministic derivation step. The difference between both definitions is relevant for practical applications. For the non-deterministic mechanism, a graph and a applicable rewrite rule is a sufficient input. Thus, Nagl's definition is adopted here. In comparison with that, a deterministic derivation needs also a graph isomorphism as input. Thus, to start a derivation explicit knowledge of the graph is necessary to define this isomorphism. This means that each vertex has to be explicitly addressable instead of using the pattern given by the labelling and connectivity of the left-hand side graph.

Under the restriction that unique labels are used in rewrite rules and all derivable graphs, a derivation by the non-deterministic mechanism can have a deterministic result. In particular the unique labels can be used to determine the left-hand-side isomorphism in the derivation. When there exists exactly one left-hand-side isomorphism all derivable graphs belonging to the same equivalence class. They are not identical since there is no restriction on the right-hand-side isomorphism. Because the vertices themselves are interesting only for the execution of a derivation and as a carrier of labels, the rewrite rule is, in this case, said to be **uniquely applicable** on the affected graph.

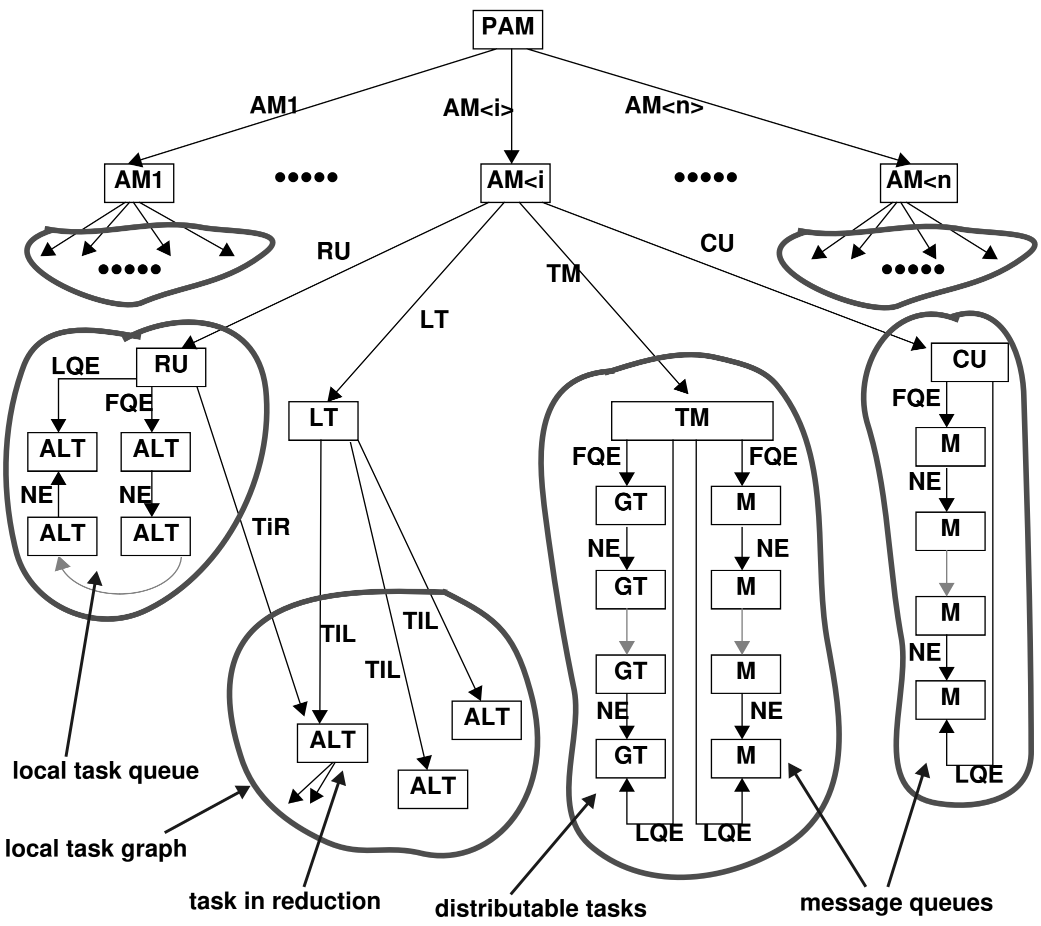
When there are multiple rules of a set applicable to a graph, the result of a set derivation step is not deterministic. The correct modelling of deterministic behaviour depends mainly on the structure of the rule sets: a set containing rules with isomorphic left-hand sides are the best way to model nondeterminism. The strategy used to insure unique applicability in ordinary graph-grammars can be used also to model **deterministic behaviour of set derivation**.

4 The Use of G^3R Models for Monitoring - an Example

To **illustrate the usage of the monitoring method**, it is applied to a parallel graph reduction machine called Parallel Abstract Machine (PAM, [LKI89], [Kuc89], [Loo90]). This machine was designed and implemented as a software prototype at the Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen. A machine performing graph reduction evaluates a functional expression according to a set of given function definitions. In pure functional languages multiple occurrences of subexpressions need to be evaluated just once. Therefore the subtrees representing the common subexpressions can be identified and the abstract syntax tree of the functional expression is transformed to a graph. During the evaluation the machine holds the expression together with bookkeeping information in its memory. Since a graph reduction machine uses graphs as the main data structure, it is a very suitable application for a graph-grammar model. Several other projects have designed and implemented parallel graph reduction machines for example ALICE ([DCFHR87]), or GRIP ([HamPey90],[PCSH87]). So there is a broad application area of monitoring with graph-grammars.

4.1 Modelling the State

The main ideas of the graph representation of the "**P**arallel **A**bstract **M**achine" are shown in Figure 4. The labels are acronyms denoting specific functional units or objects. The machine consists of replicated sequential **A**bstract **M**achines numbered from 1 to n (**AM1**, **AM<i>**, **AM<n>**). The implementation approach for graph reduction chosen in the PAM uses tasks to hold the complete information for the evaluation of subexpressions. The dependencies of tasks form a directed acyclic graph, the task graph. Each individual abstract machine in the PAM holds an individual part of the task graph, the "**L**ocal **T**asks", on which the local "**R**eduction **U**nit" operates. This unit reduces a task to a normal form which is the value of the expression. It accesses the task graph having one "**T**ask **i**n **R**eduction". Whenever tasks are activated in a reduction



PAM	Parallel Abstract Machine	**CU**	Communication Unit	**FQE**	First Queue Element
AM<j>	Abstract Machine number j	**TIL**	Task Is Local	**LQE**	Last Queue Element
RU	Reduction Unit	**TiR**	Task in Reduction	**NE**	Next Element
LT	Local Task graph	**ALT**	Active Local Task	**M**	Message
TM	Task Manager	**GT**	Global Task		

Figure 4 Graph Model of the Parallel Abstract Machine

step, they are added either to the local or global task queue. Each queue is represented by two edges pointing to the "**F**irst **Q**ueue **E**lement" and the "**L**ast **Q**ueue **E**lement" and the connection of list elements by "**N**ext **E**lement"-edges. The local task queue is managed by the reduction unit itself. The global one contains those tasks which might be evaluated on remote reduction machines. It is therefore organized by a "**T**ask **M**anager" which coordinates the distribution of tasks. The task manager holds a list of "**M**essages" which contain state information of remote reduction units or task representations sent by a remote task manager. The cooperation of the sequential reduction machines is realised by message passing. It is supported by separate "**C**ommunication **U**nits".

Figure 5 shows the components of a task. A task is a data structure which holds the information necessary for the evaluation of a function application. This information includes state of the task, a stack, arguments, and pointers to its parent task and other tasks dependent on its value. A task is represented by a vertex labelled with its actual state. The actual states of all three tasks occurring in Figure 5 are "**A**ctive **L**ocal **T**ask". The stack of the task shown in the middle contains a "**V**alue" on which "**S**tack **T**op" points. "**B**ottom" is a "**S**tack **E**lement" of the task and is referred through "**S**tack **N**ext". The task was created by the "**P**arent **T**ask" which needs its value. Other tasks dependent on the result can be organized in a dependency queue ("**L**ast **D**ependent"). The task being reduced has two arguments referred by edges labelled "**A1**" and "**A2**".

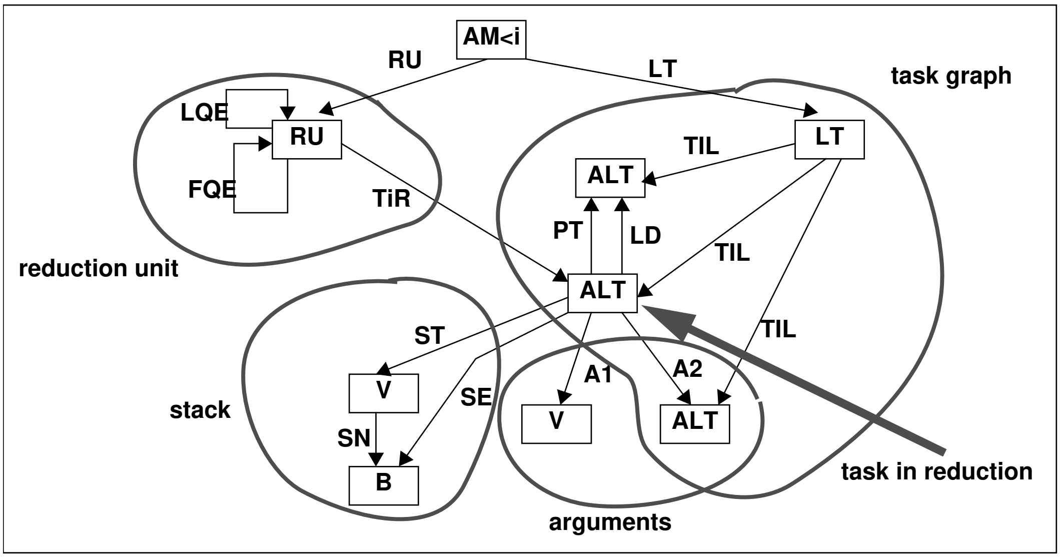


Figure 5 Graph Model of a Task

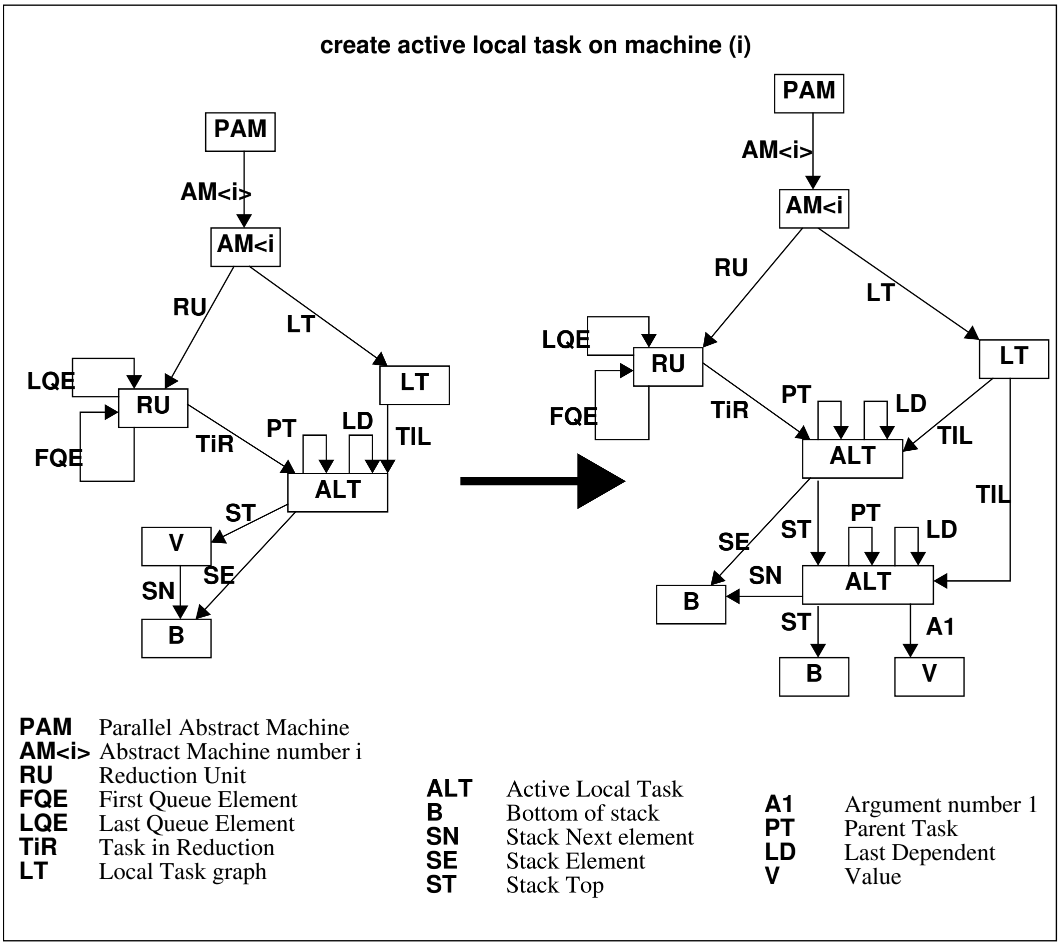


Figure 6 Graph Rewrite Rule describing the Creation of a Task on Reduction Machine i

4.2 Modelling dynamic behaviour

In an adequate graph-grammar model of system behaviour, each **graph is an abstraction of a system-state**. The graph represents that part of the state which is constant between two event-instances. A necessary precondition for the adequate representation is the completeness and consistency of the model. Thus, all possible event-instances have to be represented as rewrite rules and these rules have to cover all effects of the events in the abstract model.

The left- and right-hand-side of a rewrite rule representing the creation of a task with one argument is given in Figure 6. The embedding description is empty in the example, since no explicit changes are intended. The identical embedding, which is the default then, is sufficient. In the example, the value on top of the stack is passed as the first argument to the created task which replaces the values on the stack. In another rule, this task might be added to the local task queue held by the reduction unit. The rule shown is parameterised by the machine number <i>. Thus, it expresses the creation of a task on an arbitrary machine.

4.3 Specifying Monitoring Content by Attributes

After building the operational model of the machine to be examined, the goal of the analysis under consideration has to be specified. Usually this goal can be formulated as a **set of requested values** which makes up the result of the analysis. These values might serve as a total description of the behaviour of the machine or just show some specific properties.

To introduce these data in the graphs representing a system's run, a number of attribute instances are assigned to vertices. Not all vertices need to have the same attributes. The average length of a queue, for example, makes sense as an attribute for the reduction unit, but does not contain information on an element of the stack. Since the labels of a vertex are chosen such that they express a specific functionality or object type, they are used to determine the type of the attribute instances. An **attribute assignment function** defines the set of attribute instances belonging to each vertex dependent on its label. By this means the informational content of the monitoring is specified.

AM<i>	$\mapsto$	*ratio (#local executed tasks) to (#remote executed tasks)*
RU	$\mapsto$	*busy time, idle time, #executed tasks, average queue length, load*
LT	$\mapsto$	*total #tasks, actual #tasks, actual #waiting tasks*
TM	$\mapsto$	*message throughput, #accepted tasks*
ALT	$\mapsto$	*initial time, completion time, wait time, active time, function name*

Figure 7 Attribute Assignment Function

For example, the functional units of one instance of the graph reduction machine can have the attribute instances shown in Figure 7. An interesting detail in the behaviour of the abstract machine is the ratio of the number of locally executed tasks to the number of remotely executed ones. It shows the migration of tasks during an execution. Obvious performance figures of the reduction unit are included in the list of attribute instances. Attributes containing information distilled from the task graph are assigned to the vertex labelled "**LT**" found at the root of the local task graph. With respect to execution performance, interesting information on an active local task are the various times describing its life-cycle. The function name is another attribute instance which supports debugging.

4.4 Monitoring by Updating Attribute Instances

The assignment of values to attribute instances is done according to **attribute evaluation rules** which are attached to each graph rewrite rule. Possible evaluation rules assigned to the rewrite rule shown in Figure 6 are incrementation of the numbers of total and actual tasks of the local task queue, the setting of the initial time, and the function name of the created task. The evaluation rules define the propagation and actualisation of attribute values.

The **execution of the evaluation rules** is performed during the application of the rewrite rule. To formalise the procedure of execution, the definition of the set derivation step has to be extended according to the standard techniques for attribute grammars ([Alb91]). The only non-standard feature is the input of initial values. In the standard formalism values are assigned as explicit constants mentioned in the rules. An initial value “0” for the length of a queue at the beginning of the derivation can serve as an example. To feed run-time data such as timestamps into the graph representation of the state, the parameters of the rewrite rule have to be extended. They take not only structural data like machine numbers determining actual labels used in the rule, but also values to be assigned to attribute instances or used in the calculation of these values.

The extension of rewrite rules by sets of attribute evaluation rules is the precondition of the analysis of the event-trace. Since various decorations are possible, the analysis can be performed under various respects. Based on the same operational model, time analysis as well as debugging support can be defined by different, **application-specific evaluation rules**.

The analysis of the event-trace which has been produced during the run of the monitored system is triggered by the correspondence between events and rewrite rules which has guided the definition of the rules. Each event message is interpreted by the application of the corresponding group of rewrite rules on the graph representing the actual state. Due to the attribution this actual state not only comprises structural information, but also partially analytical results. To be able to initialise attributes, initial values have to be extracted out of the event message and passed to the rewrite rule. After interpreting the event-trace totally, the **monitored data are present as attribute values** and can serve for further interpretation.

In case the perspective of analysis changes, the **event-trace can be reused**. All functional and structural behaviour of the application is already captured by the non-attributed graph-grammar. By defining new attributes and evaluation rules, the same graph-grammar can serve as the base for further analysis. In case no further data to initialise attributes are necessary, the analysis can be performed using already assembled traces.

5 Conclusion and Further Work

This paper presented a monitoring method which **reduces the cost of event-tracing**. The use of a formal and operational model enables the collection of a big amount of data with minimal intrusion on the system. To support the monitoring of a system-run, this explicit model is used to manage the system's information which has been collected since the beginning of the run. By this means the collection of redundant event-trace information is avoided and intrusion is minimized.

To model the abstract state of a system, graph-grammars with grouped rules have been proposed. They are suitable for the description of distributed systems whose functionality can be characterized by structural changes. The expressive power, as well as the adaptability to special application purposes, recommend graph-grammars as formal tools for monitoring. In graph-grammar models, relevant functional parts become directly addressable. Therefore, system analysis becomes easier.

The figures of the last section are part of the graph-grammar description of a graph reduction machine. This model has been completed already. The model of the underlying communication part can be handled in the same formalism. So, all interesting properties of the system will be accessible in one overall formal operational model. The measuring goal of monitoring can be supported by adding attributes and attribute evaluation rules to the modelling formalism. This allows the formal definition and automatic evaluation of measures which describe the system properties in deeper detail.

The basic component of the analysis tool has been implemented as a prototype. With this component it is possible to replay the execution of the monitored system by the interpretation of the event trace. It can be viewed as a interpreter for an arbitrary graph-grammar with grouped rules. Since it is a prototype, it uses only simple algorithms and is quite slow therefore. On top of this tool, measuring instruments will be defined. They have to execute the attribute evaluation rules attached to the rewrite rules of the grammar.

After the completion of the system description, the parallel graph reduction machine PAM will be instrumented. Then, the properties of the monitoring method can be evaluated.

The formal description of the system by means of a graph-grammar can be directly transformed into an animated visualization of the system's progress. Therefore, the evaluation of the event trace might be supported through animation.

References

[Alb91] Alblas, Henk: 'Introduction to Attribute Grammars', in Alblas, H.; Melichar, Borivoj (eds.) *Attribute Grammars, Applications and Systems*, International Summer School, Prague, Czechoslovakia, June 4-13, 1991, *LNCS 545*, Springer, Berlin, 1991, pp. 1-15.

[DCFHR87] Darlington, J.; Cripps, M.; Field, A.J.; Harrison, P.G.; Reeve, M.J.: '*The design and implementation of Alice; a parallel graph reduction machine*', IEEE Computer Society Press, 1987.

[ENRR86] H. Ehrig; Nagl, M.; Rozenberg, G.; Rosenfeld, A.(ed.): *Graph-Grammars and Their Application to Computer Science*, *3rd Int. Workshop on Graph-Grammars*, Warrenton VA, USA, December 2-6, 1986, *LNCS 291*, Springer, Berlin, 1987.

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

[Epp90] Epp, Susanna: '*Discrete mathematics with applications*'; Wadsworth, Belmont CA, 1990.

[FeSch90] Feldmann, Detlef; Alois Schütte: 'Spezifikation einfacher, kontextsensitiver Syntaxkonstrukte von Occam2 durch eine Graphgrammatik', *Fachberichte Informatik 1/90*, Universität Koblenz-Landau, Koblenz, 1990.

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

[HaWy90] Haban, Dieter; Wybranietz, Dieter: 'A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems', *IEEE Trans. on Software Engineering*, **16**, (2) 197-211 (1990).

[HamPey90] Hammond, Kevin; Simon Peyton Jones: 'Some Early Experiments on the GRIP Parallel Reducer', unpublished manuscript, October 24., 1990.

[PCSH87] Peyton Jones, Simon; Clack, C.; Salkild, J.; Hardie, M.: 'Grip - a high-performance architecture for parallel graph reduction', in Kahn, Gilles (ed.) *Functional Programming Languages and Computer Architecture*, Portland OR, USA, *LNCS 274*, Springer, Berlin, 1987.

[Kau86] Kaul, Manfred: 'Practical Applications of Precedence Graph Grammars', [ENRR86], pp. 326-342.

[Kuc89] Kuchen, Herbert: 'Parallele Implementierung einer funktionalen Programmiersprache auf einem OCCAM-Transputersystem unter besonderer Berücksichtigung applikativer Datenstrukturen', *Dissertation,* RWTH Aachen, 1989.

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

[Loo90] Loogen, Rita: 'Parallele Implementierung funktionaler Programmiersprachen', *Informatik-Fachberichte 232*, Springer, Berlin, 1990.

[LKI89] Loogen, Rita; Kuchen, Herbert; Indermark, K.; Damm, W.: 'Distributed Implementation of Programmed Graph Reduction' in Eddy Odijke et al. (ed.) *Conference on Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, June 12-16, 1989*, Vol. 1, LNCS 365* Springer, Berlin, 1898, pp. 136-157.

[Nag79] Nagl, Manfred: *'Graph-Grammatiken, Theorie, Implementierung, Anwendungen'*; Vieweg, Braun-schweig, 1979.

[Nag86a] Nagl, Manfred: 'Set Theoretic Approaches to Graph Grammars', [ENRR86], pp.41-54.

[Nag86b] Nagl, Manfred: 'A Software Development Environment based on Graph Technology', [ENRR86], pp.458-478.

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