Universität-Gesamthochschule Paderborn
Fachbereich Informatik

„Analysis of Design Graph Grammar Properties“

Studienarbeit im Rahmen der Bachelorprüfung

vorgelegt bei:
Prof. Dr. Hans Kleine Büning
AG Wissensbasierte Systeme

Steinmetz, Rita

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Studienarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmitteln benutzt, sowie Zitate kenntlich gemacht habe.

_____   _____
(Ort und Datum)   (Unterschrift)

Analysis of Design Graph Grammar Properties

Rita Steinmetz

December 13, 2001

Contents

1 Introduction

Design graph grammars, which were introduced in [Schulz et al. [2001]] are a special sort of graph grammar. They are developed to manage the synthesis, analysis and optimization of chemical systems as follows:

A simplified model of the chemical system is transformed into a graph, whereas the nodes represent the devices and the edges represent the connections between them. The labels of the edges and nodes provide a way to handle the technical data of the devices or the properties of the flows between the devices respectively.

In contrast to arbitrary string grammars, to which graph grammars in general are considered as equivalent [Schulz et al. [2001], p 12], design graph grammars do not have their advantageous properties.

For example whereas the result of a derivation tree of a context-free grammar is independent of the order in which the rules are applied, this property does not hold for context-free graph grammars [Rozenberg and Engelfried [1997], p 8].

Another important difference is the complexity of the membership problem. For grammars in Chomsky Normal Form this problem can be solved in $O(n^3)$ – thus in polynomial time – by the Cocke-Younger-Kasami-Algorithm which was introduced by [Kasami [1965]] and [Younger [1967]], whereas it is NP-Complete for design graph grammars.

But if they are restricted to design graph grammars which have certain properties (e.g. confluence), the complexity of the membership problem and the subgraph matching problem, which are very important for the task of analysis and synthesis can be reduced drastically.

This document is about these properties, their importance for design graph grammars, the way to implement a computer program which will examine, perhaps – if possible – even determine them.

It is structured in the following way:

- Section 2 explains why the examined properties are important and which advantages result from their existence. Also an informal description of the properties will be given.

- Section 3 provides us formal definitions of design graph grammars and of the properties and provides some Lemmas and Theorems that will help us with the implementation of a tool, which examines a graph grammar with regard to these properties.

- Section 4 shows how – based on the formalisms of Section 3 – the algorithms are developed and which problems they hold. It also demonstrates which properties can be determined absolutely and where the limit of the other ones is.

- Section 5 contains the summary of the results of this document.

- Appendix A is the online-help of the PropertyExaminer – the program which implements the results of this paper – where the program run can be seen.

- Appendix B is a description of the Rule Description File – the format in which the rules must be written down, so that the PropertyExaminer can check these rules.

Section 2, 3 and 4 are divided into several subsections; in each of them one property is examined.

2 Motivation

A design graph grammar has in principal two major tasks: generation of a design in accordance with the given in- and outputs and feasibility analysis of a given system.[1]

Generation means the creation of an apparatus based the given input substances and the desired output with the help of the design graph grammar. The apparatus consists out of labeled nodes, which represent the in- and output substances and a "black box", a nonterminal label, which represents the unknown part of the apparatus. This is the initial graph of our design graph grammar (See Figure 1). Then the system is generated by applying the transformation rules until no nonterminal labels remain in the graph.
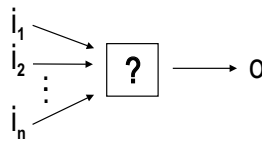

Figure 1: The initial graph of our graph grammar.

In order to analyze the feasibility of a given system, it is checked whether the graphical model of this system can be generated out of the design graph grammar. More precisely, it means that it has to be determined whether the graph which represents this system is an element of the language[2] of the design graph grammar – this question is known as the membership problem.

This two tasks lead us to two lower-level problems, which must be solved:

1. To apply a transformation rule on a graph, the *subgraph matching problem* has to be solved – a matching[3] of the target graph must be found in the host graph.

2. For the analysis of a system the *membership problem* must be solved – as mentioned above. This problem includes the *subgraph matching problem*, because it means mainly to apply the transformation rules backward.

But both of these problems are NP-Complete for arbitrary design graph grammars – to reduce their complexity, further restrictions must be made; some of the properties, which are described below, are required.

As shown in Figure 2, some of these properties are connected: one property or a combination of several properties imply another one. Hence sometimes it is easier to examine some other properties, which are often much easier to determine and which are not of direct use, but they help to determine the more complex property.

---

[1]It is also used for the evaluation and optimization of a system, but these steps are – with regard to the examined properties – not so important.

[2]Informally, the language of a graph grammar is the set of graphs which can be derived from it.

[3]A matching of a graph G in a graph H is a subgraph M of H which is similar to G, so that M could be replaced by G.
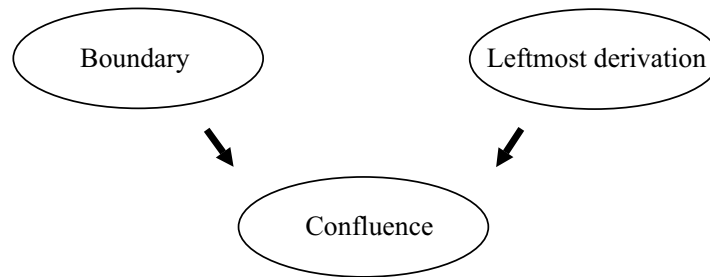
3

Boundary   Leftmost derivation

Confluence

Figure 2: This picture shows the connections between some of the properties.

2.1 Confluence

The strongest of all properties examined in this paper is confluence, which is also known as the finite Church Rosser Property [Schuster [1987], p 26]. If a graph grammar is confluent, the order in which the transformation rules are applied is of no importance – the result is always the same.

Therefore the membership problem can be solved by a dynamic programming algorithm – a modified sort of Cocke-Younger-Kasami-Algorithm – nondeterministic in polynomial time [Brandenburg [1983], p 41] and [Rozenberg and Engelfried [1997], p 82].

2.2 Boundary

Another property, which helps to reduce the complexity of the membership problem is boundary. In a boundary design graph grammar, edges between two nodes with nonterminal labels[4] are forbidden.

Boundary design graph grammars are a subclass of confluent design graph grammars. That can easily be seen because between two nonterminal nodes there is always at least one terminal node which cannot be changed. So the nonterminal nodes are independent from each other and thus the order of rule-application, too. Therefore boundary implies confluence.[5]

The complexity of the membership problem of boundary design graph grammars is even lower than the complexity of this problem for confluent design graph grammar. For boundary design graph grammars the membership problem for connected graphs with fixed maximal degree k is solvable in polynomial time [Welzl [1986], p 599] and [Rozenberg and Welzl [1986], p 160].

This fact can be used to develop an algorithm which decides the membership problem for boundary design graph grammars in general. The graphs are relabeled and nodes with the same label are melted together in such a way that the maximal degree has a fixed upper bound k [Rozenberg and Welzl [1986], p 163]. Then the above paragraph implies that the membership problem can be solved in polynomial time.

---

[4]Nonterminal labels are labels which appear on both sides of a transformation rule, whereas terminal labels appear only on the right side of a transformation rule.

[5]See Figure 2

2.3 Leftmost derivation

Leftmost derivations are an aid to construct confluent design graph grammars. On the nodes of the design graph grammar a linear order is imposed and the transformation rules are always applied on the first node of this order. Of course, the right-hand-sides of a production must be ordered as well, so that the graph which originates from this is ordered, too.

The language derived leftmost from a design graph grammar is always confluent, and if the graph language derived leftmost from a design graph grammar is equal to the language generated by this design graph grammar in any arbitrary way, this design graph grammar is confluent [Rozenberg and Engelfried [1997], pp 38-42].

2.4 Context-free

If a design graph grammar is *not* context-free, there exists at least one role whose target graph is embedded into a context graph. For the complexity that means that one subgraph matching problem must be solved, to find a matching of the context graph within the host graph, and a second subgraph matching problem must be solved to find an isomorphic copy of the target graph within the context graph.

That implies that the complexity of the membership problem is reduced by just looking at context-free design graph grammars.

2.5 Node-based

Node-based design graph grammars are grammars, where the target graph consists of one node only.

This reduces the complexity of the membership problem drastically, because a single node can be found within a connected graph, e.g. by breadth-first search, in $O(V + E)$ time – in linear time – whereas V is the number of nodes and E the number of edges in the host graph [Cormen et al. [1990], pp 463–485].

2.6 Monotonicity

To measure the quality of a generated design G, the distance between this design and the optimal design G^* – developed by an expert – is calculated. Concretely the derivational distance[6] is calculated.

For this calculation the common ancestor needs to be known. But this task also involves the subgraph matching problem, hence the complexity of the calculation is very high. But the search of a common ancestor is much less complex, if only monotonic design graph grammars are used [Schulz et al. [2001], p 59].

---

[6]Whereas the direct distance is the effort of the derivation $G \Rightarrow G^*$, the derivational distance is the effort of the derivation $G \Rightarrow G_A \Rightarrow G^*$, whereas G_A is an ancestor of both G and G^* with respect to the graph grammar.

A monotonic design graph grammar is a design graph grammar in which each graph which can be derived from it can be derived without involving deletion operations[7].

2.7 Shortcut-free

Shortcut-free design graph grammars are a subclass of monotonic design graph grammars, hence shortcut-freedom implies monotonicity.

A design graph grammar is called shortcut-free if the shortest derivation of each graph which can be derived from our design graph grammar can be derived without deletion operations.

---

[7]A deletion operation is a step of a derivation where either the number of nodes or the number of edges is decreased.

3 Theoretical Background

The definitions and lemmas of this section stem mainly from [Schulz et al. [2001]]. The appropriate proofs, which will not be given in this paper, can be looked up there.

To understand the formal definition of a design graph grammar, two fundamental definitions must be given:

Definition 1 *(Isomorphism, Isomorphism with labels)*

Let $G = \{V_G, E_G\}$ and $H = \{V_H, E_H\}$ be two graphs. An isomorphism is a bijective mapping $\varphi : V_G \to V_H$ for which holds:

$$\{a, b\} \in E_G \Leftrightarrow \{\varphi(a), \varphi(b)\} \in E_H, \text{for any } a, b \in V_G.$$

If such a mapping exists, G and H are called isomorphic.

G and H are called isomorphic with labels, if G and H are labeled graphs with labeling functions σ_G and σ_H, and the following additional condition holds:

$$\sigma_G(a) = \sigma_H(\varphi(a)) \text{ for each } a \in V_G, \text{ and } \sigma_G(e) = \sigma_G(\varphi(e)) \text{ for each } e \in E_G,$$

$$\text{where } \varphi(e) = \{\varphi(a), \varphi(b)\} \text{ if } e = \{a, b\}.$$

In the preceding section 2, the subgraph matching problem was mentioned, which is one of the major problems with regard to design graph grammars. This leads us to the basic definition of matching in general.

Definition 2 *(Matching, Context)*

Given are a labeled graph $G = \{V, E, \sigma\}$ and another labeled graph, C. Each subgraph $\{V_C, E_C, \sigma_C\}$ in G, which is isomorphic to C is called a matching of C in G.

Moreover, let T be a subgraph of C, and let $\{V_T, E_T, \sigma_T\}$ denote a matching of T in G. If $V_T \subset V_C$, $V_T \neq \emptyset$, then the graph $\{V_C, E_C, \sigma_C\}$ is called a context of T in G.

In general, two different sorts of design graph grammars are distinguished: context-sensitive design graph grammars on the one hand and context-free design graph grammars on the other hand.

Context-sensitive design graph grammars have a stronger expressive power, but are of course much more complex to handle. Context-freedom is one of the properties which are examined in this paper, and therefore all design graph grammars which are not yet tested with the regard to context-freedom are regarded as context-sensitive design graph grammars, otherwise it is mentioned explicitly.

Definition 3 *(Context-sensitive Design Graph Grammar)*

A context-sensitive design graph grammar is a tuple $\mathcal{G} = (\Sigma, P, s)$ with

- Σ is the label alphabet used for nodes and edges[8],

- P is the finite set of graph transformation rules or productions,

- and s is the initial symbol.

whose productions in the set P are graph transformation rules of the form $(T, C) \rightarrow (R, I)$ with

- $T = (V_T, E_T, \sigma_T)$ is the target place to be replaced,

- C is a super-graph of T, called the context,

- $R = (V_R, E_R, \sigma_R)$ is the possibly empty replacement graph,

- I is the set of embedding instructions for the replacement graph transformation graph R.

  The semantics of a graph transformation rule $(T, C) \rightarrow (R, I)$ is as follows: Firstly, a matching of the context C is searched within the host graph. Secondly, an occurrence of T within the matching of C along with all incident edges is deleted. Thirdly, an isomorphic copy of R is connected to the host graph according to the semantics of the embedding instructions.

  The set of embedding instructions consists of tuples ((h, t, e), (h, r, f)), where

    - *$h \in \Sigma$ is a label of a node $v \in G \backslash T$,*
    - *$t \in \Sigma$ is a label of a node $w \in V_T$,*
    - *$e \in \Sigma$ is the label of the edge {v, w},*
    - *$f \in \Sigma$ is another edge label not necessarily unequal to e, and*
    - *$r \in V_R$ is a node in R.*

  An embedding instruction ((h, t, e), (h, r, f)) is interpreted as follows: If there is an edge with label e connecting a node labeled h with the target node t, then the embedding process will create a new edge with label f connecting the node labeled h with node r.

  The execution of a graph transformation rule p on a host graph G yielding a new graph G′ is called a derivation step and denoted by $G \Rightarrow_p G'$. A sequence of such derivation steps is called derivation. The set of all graphs that can be generated with $\mathcal{G}$ is designated by $L(\mathcal{G})$.

3.1 Confluence

As stated in subsection 2.1, a design graph grammar is confluent if the order of rule application is irrelevant. This leads us to the following formal definition of confluence:

Definition 4 *(Confluence)*

A context-free design graph grammar $\mathcal{G} = (\Sigma), P, s)$ is confluent, if for every pair of rules $T_1 \rightarrow (R_1, I_1)$ and $T_2 \rightarrow (R_2, I_2)$ with R_i contains a matching of T_{3-i} for $i \in 1, 2$, and for any arbitrary host graph H containing matchings of T_1 and T_2 the following equality holds:

---

[8]Labels are used to specify types and as variables for other labels. To avoid confusion, variable labels will be denoted by capital letters, and all other labels with small letters.

$$H[T_1|R_1][T_2|R_2] = H[T_2|R_2][T_1|R_1]$$

Example. The following context-free design graph grammar is confluent:

Let $\mathcal{G} = (\Sigma, P, s)$ be a context free design graph grammar and H = $(\{h_1, h_2, h_3\},$ $\{(h_1, h_2), (h_1, h_3), (h_2, h_3)\}, \{(h_1,a), (h_2,b), (h_3,c), (\{h_1, h_2\}, e), (\{h_1, h_3\}, g), (\{h_2, h_3\}, g)\})$ the host graph as shown in Figure 3.
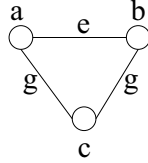

Figure 3: The host graph H.

P = $\{r_1, r_2\}$ as follows:

$r_1 : a \rightarrow (R_1, I_1)$ with $R_1 = (\{v_1, v_2\}, \{(v_1, v_2)\}, \{(v_1, c), (v_2, b), (\{v_1, v_2\}, g)\})$ and $I_1 =$ $\{i_{1,1}:((b,a,e),(b,b,f)), i_{1,2}:((a,a,f),(a,b,e)), i_{1,3}:((c,a,g),(c,b,h))\}$[9](See Figure 4)
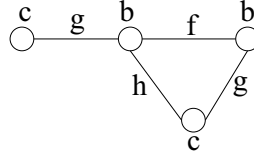

Figure 4: The graph after applying only rule r_1.

$r_2 : b \rightarrow (R_2, I_2)$ with $R_1 = (\{v_3\}, \emptyset, \{(v_3,a)\})$ and $I_2 = \{i_{2,1}:((b,b,f),(b,a,e)), i_{2,2}:((a,b,e),(a,a,f)),$ $i_{2,3}:((c,b,g),(c,a,i))\}$(See Figure 5)
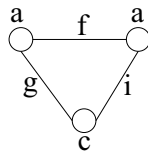

Figure 5: The graph after applying only rule r_2.

Then the order in which these two rules are applied does not matter, the result is always the same: H[a | R_1][b | R_2] = H[b | R_2][a | R_1](See Figure 6).

This definition is very useful to show the advantages of confluent design graph grammar, but it is not very usable to test this property because the result of many rule-applications would have to be examined.

The following definition which was provided by [Rozenberg and Engelfried [1997]] for edNCE grammars and which was modified for design graph grammars by [Schulz et al. [2001]] is much more useful because the transformation rules are examined statically.

---

[9]The embedding instructions are numbered, so that they can be referenced later.
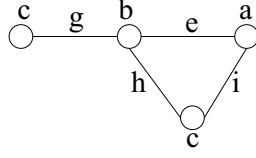
Figure 6: The graph after applying both rules, no matter in which order.

Definition 5 *(Confluence 2) A context-free design graph grammar $\mathcal{G} = (\Sigma, P, s)$ is confluent, if for all graph transformation rules $T_1 \to (R_1, I_1)$ and $T_2 \to (R_2, I_2)$ in P, all nodes $x_1 \in V_{R_1}, x_2 \in V_{R_2}$, and all edge labes $\alpha, \beta \in \Sigma$, the following equivalence holds:*

$$\exists \beta \in \Sigma : ((t_2, t_1, \alpha), (t_2, \sigma(x_1), \beta)) \in I_1 \text{ and } ((\sigma(x_1), t_2, \beta), (\sigma(x_1), \sigma(x_2), \delta)) \in I_2$$

$$\Leftrightarrow$$

$$\exists \gamma \in \Sigma : ((t_1, t_2, \alpha), (t_1, \sigma(x_2), \gamma)) \in I_2 \text{ and } ((\sigma(x_2), t_1, \gamma), (\sigma(x_2), \sigma(x_1), \delta)) \in I_1$$

It is very easy to understand, why this definition is equivalent to the first definition of confluence, if the effects are visualized by drawing them.
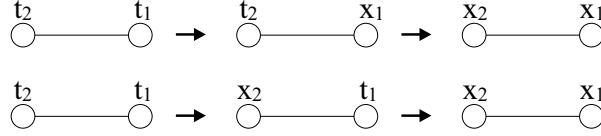

Figure 7: Graphical visualization of definition 5.

By regarding Figure 7 it can be seen that both drawings start with the same partial graph. In the first one a rule with embedding instruction I_1 is applied first and then a rule with embedding instruction I_2, and vice versa in the second one, but then the result is the same. Therefore, this definition just says that if there are in two different embedding instructions two rules which influence each other, there must also be rules in this embedding instructions which achieve the same effects, if the order of these embedding instructions is changed. This meets exactly the requirements of definition 4.

By comparing definition 5 with the given example, one can see that the following holds:

$$\exists f \in \Sigma : i_{1,1} \in I_1 \text{ and } i_{2,1} \in I_2$$

$$\Leftrightarrow$$

$$\exists f \in \Sigma : i_{2,2} \in I_2 \text{ and } i_{1,2} \in I_1$$

which corresponds perfectly to definition 5. For $i_{1,3} \in I_1$ (and $i_{2,3} \in I_2$) there are no embedding instructions in $I_2(I_1)$ which influence them.

3.2 Boundary

Arbitrary graph grammars do not have only the alphabet Σ of labels, but they also distinguish the alphabet Δ of terminal labels. $\Sigma \backslash \Delta$ is then called the alphabet of nonterminal labels.

Design graph grammars normally do not make this distinction, but nonterminal graphs and nonterminal label can be defined as follows:

Definition 6 *(Terminal graphs, terminal labels)*

Let $\mathcal{G} = (\Sigma, P, s)$ be a design graph grammar.

1. *All graphs T, which are the left-hand side of a production $T \rightarrow (R, I)$, are called nonterminal graphs. If this graph consists of a single node only, this node is called nonterminal node.*

2. *All node-labels, which occur in a nonterminal graph or nonterminal node are called nonterminal labels.*

Because each nonterminal node can be viewed as a nonterminal graph, in the following only nonterminal *graphs* are considered in order to cover the not node-based case, too.

With this definition, boundary design graph grammars can be defined:

Definition 7 *(Boundary Design Graph Grammar)*

A design graph grammar $\mathcal{G} = (\Sigma, P, s)$ with directed edges and edge labels is boundary, or a boundary design graph grammar, if for every Production $T \rightarrow (R, I)$,

- *R does not contain edges between nonterminal nodes or nonterminal graphs, and*

- *I does not contain embedding instructions $((\sigma, t, \beta), (\sigma, x, \gamma))$ where σ is nonterminal.*

Example. The following context-free design graph grammar is boundary, because the only terminal nodes in this grammar are v and y with labels a and c, and there do not exist edges between them:

Let $\mathcal{G} = (\Sigma, P, s)$ be a context free design graph grammar and H = ({v, w, x, y}, {(v,w), (v,x), (w,x), (w,y)}, {(v,a), (w,d), (x,b), (y,c), ({v,x}, e), ({v,w}, e), ({w,x}, f), ({w,y}, e)}) the host graph as shown in Figure 8.
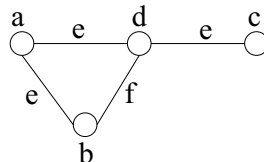

Figure 8: The host graph H.
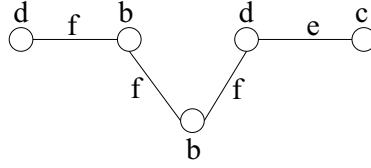
P = $\{r_1, r_2\}$ as follows:

Figure 9: The graph after applying only rule r_1.

$r_1 : a \rightarrow (R_1, I_1)$ with $R_1 = (\{v_1, v_2\}, \{(v_1, v_2)\}, \{(v_1, b), (v_2, d), (\{v_1, v_2\}, f)\})$ and $I_1 = \{(b,a,e), (b,b,f)\}$(See Figure 9)

$r_2 : b \rightarrow (R_2, I_2)$ with $R_1 = (\{y_1\}, \emptyset, \{(y_1, d)\})$ and $I_2 = \{(d,c,e), (d,d,f)\}$(See Figure 10)
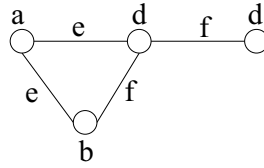

Figure 10: The graph after applying only rule r_2.

Then the order in which these two rules are applied does not matter, the result is always the same: $H[a \mid R_1][b \mid R_2] = H[b \mid R_2][a \mid R_1]$(See Figure 11). Thus, this boundary design graph grammar is confluent.
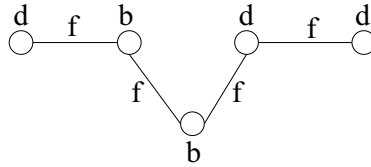
Figure 11: The graph after applying both rules, no matter in which order.

As already mentioned in section 2.2, and supported by the preceding example, boundary design graph grammars are a subclass of confluent design graph grammars:

Lemma 1 *(Expressiveness of Boundary Design Graph Grammars)*

Every boundary design graph grammar is confluent.

By comparing definition 7 to definition 5, it can be seen that the class of confluent design graph grammars can be divided into two disjoint subclasses: The subclass of boundary design graph grammars on the one hand, and the subclass of design graph grammars, where β and γ according to definition 5 exist on the other hand.

As shown in Subsection 3.1, these β and γ can only exist if and only if there is an edge between two nodes with nonterminal labels – thus, this design graph grammar cannot be boundary.

That leads us to a weaker form of confluence, which is faster to determine than confluence. This property we will call "weak confluence" in the following.

3.3 Weak confluence

As mentioned in the previous section (section 3.2), if the design graph grammar is considered isolated, the only place where "nonconfluent embedding instructions" – embedding instructions, in which the conditions of definition 5 are injured – can occur, are edges between two nonterminal graphs or nodes. That means that one point that has to be checked for weak confluence are all instructions, where the host node label and the target node label might be the node labels of this edge.

Another point which has to be regarded carefully is the embedding instructions themselves. If we do not restrict them, a nonterminal graph or even two connected nonterminal graphs could be built. So, embedding instructions where such a graph could be created – i.e. embedding instructions, where either the host node label or the replacement node label are nonterminal labels – have to be checked, too. In this case, the embedding instructions where the *host node label* is a nonterminal label are checked.[10]

These two requirements lead to the new definition of weak confluence.

Definition 8 *(Weak confluence)*

Let $\mathcal{G} = (\Sigma, P, s)$ be a context-free design graph grammar. Let E' be the set of edges between two nonterminal graphs or nodes and let Γ be the set of nonterminal labels. Let then $P' \subset P$ be the set of productions $T' \to (R', I')$ where for all $((a, b, c), (a, d, e)) \in I' : a \in \Gamma$ or $E = \{a, b\} \in E'$.

$\mathcal{G}$ is called weak confluent, if for all graph transformation rules $T'_1 \to (R'_1, I'_1)$ and $T'_2 \to (R'_2, I'_2)$ in P', all nodes $x_1 \in V_{R'_1}, x_2 \in V_{R'_2}$, and all edge labes $\alpha, \beta \in \Sigma$, the following equivalence holds:

$$\exists \beta \in \Sigma : ((t_2, t_1, \alpha), (t_2, \sigma(x_1), \beta)) \in I'_1 \text{ and } ((\sigma(x_1), t_2, \beta), (\sigma(x_1), \sigma(x_2), \delta)) \in I'_2$$

$$\Leftrightarrow$$

$$\exists \gamma \in \Sigma : ((t_1, t_2, \alpha), (t_1, \sigma(x_2), \gamma)) \in I'_2 \text{ and } ((\sigma(x_2), t_1, \gamma), (\sigma(x_2), \sigma(x_1), \delta)) \in I'_1$$

It would be very useful if this new property would show the same advantageous qualities as confluent design graph grammars. It would be very desirable if the order of rule applications would have no influence on the resulting graphs for all weak confluent design graph grammars, just as for all confluent design graph grammars.

Up to now, it is assured that inside the design graph grammar the embedding instructions have no influence on the application of each other. But nothing can be known about the host graph, on which these embedding instructions are applied, in advance. It has to be guaranteed that inside the host graph itself, no two embedding instructions can influence each other, i.e. the hostgraph has to be sort of boundary.

The property of boundary does not exist for graphs, because there is – and can be – no distinction between nonterminal and terminal labels. But boundary in this case just means

---

[10]The replacement node labels do not need to be checked, either. Either the host node labels or the replacement node labels are sufficient.

that in the graph there must be no connection between two nodes with nonterminal labels, whereas the nonterminal labels are specified by the design graph grammar.

This property, which will be defined here for any graph, will be called "boundary according to a design graph grammar" in the following.

Definition 9 *(Boundary according to a design graph grammar)*

Let $G = (V_G, E_G, \sigma)$ be a labeled graph and let $\mathcal{G} = (\Sigma, P, s)$ be a design graph grammar. Let $\Gamma \subset \Sigma$ be the set of nonterminal labels.

Then G is called boundary according to $\mathcal{G} \Leftrightarrow$ there does not exist an edge $E = \{v_G, w_G\} \in E_G$ with $v_G, w_G \in V_G$ and $\sigma(v_G) \in \Gamma$ and $\sigma(w_G) \in \Gamma$.

Now it can be seen easily that weak confluent design graph grammars in connection with hostgraphs which are boundary according to this design graph grammar show qualities equal to these of confluent design graph grammars, i.e. that they have – of course if and only if they are exclusively applied on host graphs which are boundary according to the design graph grammar – the same advantages as described in section 2.1 (The complexity of the membership problem is reduced drastically.).

Lemma 2 *(Expressiveness of weak confluence)*

Let $\mathcal{G} = (\Sigma, P, s)$ be a weak confluent design graph grammar and let $H = (V_G, E_G, \sigma)$, the initial graph, be a graph which is boundary according to $\mathcal{G}$. Then for $\mathcal{G}$ started with the host-graph H the following holds:

For every pair of rules $T_1 \rightarrow (R_1, I_1)$ and $T_2 \rightarrow (R_2, I_2)$ with R_i contains a matching of T_{3-i} for $i \in 1, 2$, and for any arbitrary host graph H', with $H' \Rightarrow_{\mathcal{G}}^* H$ containing matchings of T_1 and T_2 the following equalitiy holds:

$$H'[T_1|R_1][T_2|R_2] = H'[T_2|R_2][T_1|R_1]$$

Proof sketch.

This lemma says, that for a design graph grammar, which is started with a host graph, which is boundary according to this design graph grammar, the order of rule application does not matter, that means that any two productions do not influence each other. Two productions can only influence each other, if there exists an edge between their target graphs. The target graphs are part of the left hand side of a production and thus, according to Definition 6, nonterminal graphs. The host graph is claimed to be boundary according to the design graph grammar, which is applied on it. That means, there do not exist edges between nonterminal graphs, thus in the host graph itself no two productions, which are applied on it, can influence each other.

The only way, two productions, which influence each, other can appear, is that edges between nonterminal graphs are generated by applying the design graph grammar on the host graph. In this case two different cases have to be considered:

1. *There exist two connected nonterminal graphs inside the embedded replacement graph.*

2. *Two connected nonterminal graphs are generated by connecting the replacement graph to the host graph.*

In the first case the edge, which connects the two nonterminal graphs, would be an element of the set E', defined according to Definition 8, and in the second case, the host node label of the embedding instruction, through which the graphs are connected, would have to be a nonterminal label, that means, it is an element of the set Γ, defined according to Definition 8. But because the design graph grammar is weak confluent, it is guaranteed by Definition 8 that for all productions, which contain elements of these two sets, the order in which they are applied does not matter.

This means, that the design graph grammar started with a host graph, which is boundary according to this design graph grammar, shows the same advantageous property as confluent design graph grammars: the order in which the rules are applied is of no importance.

3.4   Leftmost derivation

As mentioned in section 1, arbitrary String grammars can easily be transformed into a equivalent String grammar in Chomsky Normal Form, which have the same positive qualities as confluent design graph grammars – especially the Membership problem can be solved very fast.[11] One of the characteristics of String grammars, which makes this problem to be solved so fast, is that they have an automatically implied order – from the left to the right.

In [Schulz et al. [2001]] and [Rozenberg and Engelfried [1997]] it was shown that if an order is applied on the right-hand side of rule-applications, the languages generated in that way show the same properties as languages generated by confluent design graph grammars.

But, first, it has to be defined what the order of a graph is.

Definition 10 *(Ordered Graph)*

A graph $G = (V_G, E_G, \sigma_G)$ is an ordered graph, if there is a linear order $(v_1, ..., v_n)$ with $v_i \in V_G$ for $1 \leq i \leq n$ and $|V_G| = n$.

The next point which needs to be known is the general term of a derivation.

Definition 11 *(Derivation)*

A derivation is a sequence of graphs $\pi = (G_1, ..., G_n)$ for which the simple derivation $G_i \Rightarrow_{v,p} G_{i+1}, i \in \{1, ..., n-1\}$, has been achieved by applying graph transformation rule p on node v. $\pi_\mathcal{G}$ denotes a derivation based on graph transformation rules of a design graph grammar $\mathcal{G} = (\Sigma, P, s)$, and $\pi_\mathcal{G}(G)$ denotes a derivation $(s, ..., G)$.

If there is an ordered Graph $G = (V_G, E_G, \sigma_G)$ with the linear order $(v_1, ..., v_n)$ and there exists a production $v_i \rightarrow (R_i, I_i)$ for $1 \leq i \leq n$ and R_i has the linear order $(r_1, ..., r_m)$, then after the derivation step the generated graph is ordered, too. Its linear order is: $(v_1, ..., v_{i-1}, r_1, ..., r_m, v_{i+1}, ..., v_n)$.

---

[11]In time $O(n^3)$ by the Cocke-Younger-Kasami-Algorithm.

Definition 12 *(Leftmost Derivation)*

Let $\mathcal{G}$ be a design graph grammar. For an ordered graph H, a derivation step $H \Rightarrow_{v,p} H'$ of $\mathcal{G}$ is a leftmost derivation step, if v is the first nonterminal node on the order of H . A derivation is leftmost, if all steps are leftmost.

The graph language leftmost derived by $\mathcal{G}$ is denoted by $L_{lm}(\mathcal{G})$.

As mentioned above, languages which are leftmost derived out of a design graph grammar have the same qualities as confluent design graph grammars. In fact, the class of leftmost generated languages is equal to the class of confluent languages (see [Rozenberg and Engelfried [1997]], pp. 39-41).

Lemma 3 *(Expressiveness of Leftmost Generated Languages)*

For every confluent design graph grammar $\mathcal{G}$ it holds that $L_{lm}(\mathcal{G}) = L(\mathcal{G})$.

Leftmost derivations are very useful to generate confluent design graph grammars, but they provide no means in order to determine confluence, so they will not be treated in the next section (Section 4).

3.5 Context-free

A context-free design graph grammar is defined as follows:

Definition 13 *(Context-Free Design Graph Grammar)*

A context-free design graph grammar is a tuple $\mathcal{G} = (\Sigma, P, s)$ defined as in the context-sensitive case.

The productions of the set P are graph transformation rules of the form $T \rightarrow (R, I)$ with

- *$T = (V_T, E_T, \sigma_T)$ is the target graph to be replaced,*

- *$R = (V_R, E_R, \sigma_R)$ is the possibly empty replacement graph,*

- *I is the set of embedding instructions for the replacement graph R.*

  The semantics of a context-free graph transformation rule $T \rightarrow (R, I)$ is as follows: Firstly, a matching of the target graph T is searched within the host graph G. Secondly, this occurrence of T along with all incident edges is deleted. Thirdly, an isomorphic copy of R is connected to the host graph according to the semantics of the embedding instructions.

  The set I of embedding instructions consists of tuples ((h, t, e), (h, r, f)), where

  - *$h \in \Sigma$ is a label of a node $v \in G \backslash T$,*
  - *$t \in \Sigma$ is a label of a node $w \in V_T$,*
  - *$e \in \Sigma$ is the label of the edge {v, w},*
  - *$f \in \Sigma$ is another edge label not necessarily unequal to e, and*

16

– $r \in V_R$ is a node in R.

An embedding rule ((h, t, e), (h, r, f)) is interpreted as in the context-sensitive case.

By comparing this definition to definition 3, it is easy to see that the difference between context-free design graph grammars and context-sensitive design graph grammars is the lack of a context graph in context-free design graph grammars – as the term already says.

Examples of context-free design graph grammars are the example in section 3.1 and the example in section 3.2.

In this document a design graph grammar will also be called context-free, if for all rules r where a context-graph C_r exists this context-graph C_r is an isomorphic copy of the target-graph T_r of the same rule, because the effect will be the same – only one matching per rule has to be found and not two (one matching of the context graph within the host graph and one matching of the target graph within the context graph) as in the context-sensitive case.

3.6 Node-based

Node-based design graph grammars are design graph grammars where only nodes and not entire graphs are replaced by the replacement graph:

Definition 14 *(Node-based design graph grammar)*

Let $\mathcal{G} = (\Sigma, P, s)$ be a design graph grammar.

$\mathcal{G}$ is called node-based, if for every rule $T_i \to (R_i, I_i)$ holds: T_i just consists out of one node only.

Otherwise this design graph grammar is called graph-based.

3.7 Monotonicity

A design graph grammar is called monotonic, if every graph can be derived monotonically, that means that within each derivation step $G \Rightarrow G'$ the size of the derived graph G' must not be smaller than the size of the graph G. Put in other words, it is monotonic if no nodes and no edges are deleted.

The formal definition of a deletion or deletion operation is the following:

Definition 15 *(Deletion Operation)*

A deletion operation is a graph transformation step $G \Rightarrow G'$ such that

- $|V_T| > |V_R|$ *or*
- $|E_T| > |E_R|$

whereby (V_T, E_T, σ_T) and (V_R, E_R, σ_R) represent the target and replacement graphs, respectively.

There exist two different definitions of monotonicity:

Definition 16 *(Monotonicity)*

Let G, G′ be graphs and $\mathcal{G}$ a design graph grammar. A derivation $\pi = (G,...,G')$ is called monotonic, if and only if ρ_π does not involve deletion operations.

$\mathcal{G}$ is monotonic, if and only for every $G \in L(\mathcal{G})$ there exists a monotonic derivation $\pi_\mathcal{G}(G)$.

The second definition is a little stronger than the first one. It claims not only that there must not be a deletion operation within the derivation of a graph, but it also demands that for each graph transformation rule the target graph is a subgraph of the replacement graph.

This means that not only the size of the graph has to increase steadily, but that in fact the graph itself remains intact, and only additional nodes can be inserted, there is no possibility to replace an existing node.

Definition 17 *(Monotonicity 2)*

Let a design graph grammar $\mathcal{G} = (\Sigma, P, s)$ be given. $\mathcal{G}$ is monotonic, if the following holds for every graph transformation rule $r = (T, C) \rightarrow (R, I)$ of P: R encompasses a matching of T.

In this document and in the program, which is developed on the results of the document, only the first definition of monotonicity will be used.

3.8   Shortcut-free

Shortcut-free design graph grammars are very similar to monotonic design graph grammars according to the first definition (definition 16). But in this case not only the derivation of a graph which can be derived by the design graph grammar has to be monotonic, but above all the shortest derivation, i.e. the derivation with the smallest number of derivation steps, has to be monotonic.

Definition 18 *(Shortcut-Free)*

Let G, G′ be graphs and $\mathcal{G}$ a design graph grammar.

$\mathcal{G}$ is called shortcut-free, if for every $G \in L(\mathcal{G})$ the shortest derivation is a monotonic derivation.

4 Implementation

This section describes how the examined properties – except leftmost derivation – can be determined. This is realized in a tool called "PropertyExaminer", which is also part of this thesis.

The general program run and some screen-shots can be viewed in the appendix (section A.1).

This tool does not examine whole design graph grammars, but only their rule sets, independent from the initial symbol s, which can be changed without changing the results on the properties.

The set of rules has to be written down in a special format – as a "Rule Description File (RDF)". The syntax of such files is described in the appendix (section B).

4.1 Confluence

The test for confluence checks mainly the conditions of definition 5.

Before it starts this, it checks whether this design graph grammar is context-free, because confluence is only defined for context-free design graph grammar(See definitions 4 and 5). If it is not context-free, it returns false.

After that, it is checked whether this design graph grammar is boundary, because according to lemma 1 each boundary design graph grammar is confluent, and the test for boundary is faster than the test for confluence.

Then all pairs $((t_2, t_1, \alpha), (t_2, \sigma(x_1), \beta))$ and $((\sigma(x_1), t_2, \beta), (\sigma(x_1), \sigma(x_2), \delta))$ are searched. Here it has to be ensured that the rules of these embedding instructions are different.

Afterwards all $((t_1, t_2, \alpha), (t_1, \sigma(x_2), \gamma))$ are searched, which have to be out of the same rule as $((\sigma(x_1), t_2, \beta), (\sigma(x_1), \sigma(x_2), \delta))$. This may be several embedding instructions, because δ is not specified, yet. If for one pair $((t_1, t_2, \alpha), (t_1, \sigma(x_2), \gamma))$ cannot be found, the condition of definition 5 is injured – the design graph grammar cannot be confluent – it is returned false.

Finally $((\sigma(x_2), t_1, \gamma), (\sigma(x_2), \sigma(x_1), \delta))$ is searched. All its labels and its rules are specified by the three embedding instructions found above which match together according to definition 5. If it is not found, the design graph grammar is not confluent, false is returned.

After having checked whether there exist the according third and fourth embedding instructions for all pairs, line 44 is reached, it is returned true, and the design graph grammar is confluent.

The following pseudo-code algorithm performs the above described steps:

boolean TESTCONFLUENCE(DESIGNGRAPHGRAMMAR $\mathcal{G}$)

```
(1)   {
(2)      if(not TESTCONTEXTFREE(G)) then return false; fi
(3)      if(TESTBOUNDARY(G)) then return true; fi
(4)      INSTRUCTION [] instructions;
(5)      for(r ∈ G.rules) do
(6)         for(i ∈ r.instruction) do
(7)            instructions := instructions ∪ i;
(8)         od
(9)      od
(10)     INSTRUCTION [] [] twoMatchingInstructions;
(11)     for(i1 ∈ instructions) do
(12)        for(i2 ∈ instructions) do
(13)           if(not i1.rule=i2.rule and i1.replacementlabel=i2.hostlabel and
(14)              i1.hostlabel=i2.targetlabel and i1.newedgelabel=i2.edgelabe) then
(15)              twoMatchingInstructions := twoMatchingInstructions ∪ (i1 ∪ i2);
(16)           fi
(17)        od
(18)     od
(19)     INSTRUCTION [] [] threeMatchingInstructions;
(20)     for((i1 ∪ i2) ∈ twoMatchingInstructions) do
(21)        boolean found = false;
(22)        for(i3 ∈ instructions) do
(23)           if(i3.rule=i2.rule and i3.hostlabel=i1.targetlabel and
(24)              i3.targetlabel=i1.hostlabel and i3.edgelabel=i1.edgelabel and
(25)              i3.replacelabel=i2.replacelabel) then
(26)              found := true;
(27)              threeMatchingInstructions := threeMatchingInstructions ∪ (i1 ∪ i2 ∪ i3);
(28)           fi
(29)        od
(30)        if(not found) then return false; fi
(31)     od
(32)     for((i1 ∪ i2 ∪ i3) ∈ threeMatchingInstructions) do
(33)        boolean found = false;
(34)        for(i4 ∈ instructions) do
(35)           if(i4.rule=i1.rule and i4.hostlabel=i2.replacelabel and
(36)              i4.targetlabel=i1.targetlabel and i4.replacelabel=i1.replacelabel and
(37)              i4.edgelabel=i3.newedgelabel andi4.newedgelabel=i2.newedgelabel) then
(38)              found := true;
(39)              break;
(40)           fi
(41)        od
(42)        if(not found) then return false; fi
(43)     od
(44)     return true;
(45)  }
```

4.2 Weak Confluence

The test for weak confluence is mainly the same as for confluence, but the lines 4 to 9 of **boolean** TESTCONFLUENCE(DESIGNGRAPHGRAMMAR $\mathcal{G}$) have to replaced by the following lines:

```
(1)     LABEL [] nonterminalLabels :=
(2)           IDENTIFYNONTERMINALLABELS(IDENTIFYNONTERMINALGRAPHS(𝒢));
(3)     INSTRUCTION [] instructions;
(4)     EDGE [] boundaryEdges := GETBOUNDARYEDGES(𝒢);
(5)     for(r ∈ 𝒢).rules) do
(6)       for(i ∈ r.instruction) do
(7)         if(i.hostlabel ∈ nonterminalLabels) then
(8)           instructions := instructions ∪ i;
(9)           continue;
(10)        fi
(11)        if(EDGE(i.hostlabel, i.targetlabel) ∈ boundaryEdges or
(12)          EDGE(i.targetlabel, i.hostlabel) ∈ boundaryEdges) then
(13)          instructions := instructions ∪ i;
(14)        fi
(15)      od
(16)    od
```

The method EDGE [] GETBOUNDARYEDGES(DESIGNGRAPHGRAMMAR $\mathcal{G}$) is a slightly modified algorithm **boolean** TESTBOUNDARY(DESIGNGRAPHGRAMMAR $\mathcal{G}$) which does not return a boolean value but the set of all edges which connect two nonterminal graphs inside a replacement graph.

The lines 7 to 11 of the method **boolean** TESTBOUNDARY(DESIGNGRAPHGRAMMAR $\mathcal{G}$) have to be deleted and the line 22 has to be replaced by **if**(GETCONNECTINGEDGES(g, f, R)), whereas (EDGE [] GETCONNECTINGEDGES(GRAPH g, GRAPH f, GRAPH R)) is a method which searches all edges which connect nodes of g and f within R and stores them in an array. The algorithm EDGE [] GETBOUNDARYEDGES(DESIGNGRAPHGRAMMAR $\mathcal{G}$) returns an array of all found connecting edges or null, if none is found.

By replacing the lines 4 to 9 of the algorithm **boolean** TESTCONFLUENCE (DESIGN-GRAPHGRAMMAR $\mathcal{G}$) with these lines, the number of instructions is decreased drastically – of course the effect is dependent on the design graph grammar – and so the runtime of the algorithm is decreased, too. (The runtime of both algorithms is at about $O(n^2)$, where n is the number of examined embedding instructions. That means if for confluence 10 embedding instructions would have to be examined, and for weak confluence just 2, the test for weak confluence would be 25 times faster than the test for confluence.)

This algorithm takes – according to definition 8 – only the instructions, in which the host node label is a nonterminal label, and those which connect two nonterminal graphs of nonterminal labels. The rest of the algorithm tests the conditions of this definition – exactly the same condition as for confluence.

4.3 Boundary

In order to check whether a graph grammar is a boundary design graph grammar, first the nonterminal graphs have to be identified, because it has to be examined whether there exist edges between two nonterminal graphs inside a replacement graph.

The nonterminal graphs here are all target graphs, therefore the following algorithm stores all target graphs into an array and returns it:

GRAPH [] IDENTIFYNONTERMINALGRAPHS(DESIGNGRAPHGRAMMAR $\mathcal{G}$)

```
(1)    {
(2)        GRAPH [] nonterminals;
(3)        for(r ∈ G.rules) do
(4)            nonterminals := nonterminals ∪ r.targetgraph;
(5)        od
(6)        return nonterminals;
(7)    }
```

The nonterminal labels have to be identified, too, because it has to be checked, whether there is a nonterminal host node label inside one embedding instruction.

LABEL [] IDENTIFYNONTERMINALLABELS(GRAPH[] graphs)

```
(1)    {
(2)        LABEL [] nonterminals;
(3)        for(G ∈ graphs) do
(4)            for(v ∈ V_G) do
(5)                nonterminals := nonterminals ∪ v;
(6)            od
(7)        od
(8)        return nonterminals;
(9)    }
```

The test for boundary checks the two conditions of definition 7.

First it checks whether one of the host-labels is nonterminal within the embedding instructions. If yes, the design graph grammar cannot be boundary – the algorithm returns false.

Then it tests for each replacement graph, how many nonterminal graphs it contains. If the number is less or equal one, nothing more has to be checked for this replacement graph. Otherwise, it has to be checked whether at least two of these nonterminal graphs are connected within this replacement graph. If they are, false is returned.

If the line 29 is reached, the design graph grammar has to be boundary, because no injuries of the conditions have been found – true is returned.

The following algorithm in pseudo-code implements this, whereas **boolean** ISSUB-GRAPH(GRAPH R, GRAPH g) checks whether g is a subgraph of R and the method **boolean** ARECONNECTED(GRAPH g, GRAPH f, GRAPH R) checks whether there is an edge between

nodes of g and f inside the graph R.

```
   boolean TESTBOUNDARY(DESIGNGRAPHGRAMMAR 𝒢)
(1)    {
(2)        GRAPH [] nonterminalGraphs :=
(3)                IDENTIFYNONTERMINALGRAPHS(𝒢);
(4)        LABEL [] nonterminalLabels :=
(5)                IDENTIFYNONTERMINALLABELS(nonterminalGraphs);
(6)        for(r ∈ 𝒢.rules) do
(7)          INSTRUCTION [] instr := r.instructions;
(8)          LABEL hostlab := instr.hostlabel;
(9)          if (hostlab ∈ nonterminalLabels) then
(10)             return false;
(11)         fi
(12)         GRAPH R := r.replacementgraph;
(13)         GRAPH [] foundGraphs;
(14)         for(g ∈ nonterminalGraphs) do
(15)           if(ISSUBGRAPH(R,g)) then
(16)              foundGraphs := foundGraphs ∪ g;
(17)           fi
(18)         od
(19)         if(|foundgraphs| > 1) then
(20)           for(g ∈ foundGraphs) do
(21)             for(f ∈ foundGraphs) do
(22)               if(ARECONNECTED(g, f, R))
(23)                  return false;
(24)               fi
(25)             od
(26)           od
(27)         fi
(28)       od
(29)       return true;
(30)   }
```

4.4 Context-free

A design graph grammar is context-free, if there does not exist a context graph or if the context graph is an isomorphic copy of the target graph.

The following pseudo-code algorithm tests for each rule whether there is a context graph – that means whether the number of nodes of the context graph is greater than 0 – and if it has found one, it tests whether this context graph is an isomorphic copy on the target graph in the same rule. If this is not the case, it returns false and true otherwise.

The method **boolean** AREISOMORPHIC(GRAPH g1, GRAPH g2) is an algorithm which tests whether two graphs are isomorphic copies of each other.

```
boolean TESTCONTEXTFREE(DESIGNGRAPHGRAMMAR 𝒢)
 (1)   {
 (2)      for(r ∈ 𝒢.rules) do
 (3)         GRAPH T := r.targetgraph;
 (4)         GRAPH C := r.contextgraph;
 (5)         if(|V_C| > 0) then
 (6)            if(not AREISOMORPHIC(T, C) then
 (7)               return false;
 (8)            fi
 (9)         fi
(10)      od
(11)      return true;
(12)   }
```

4.5   Node-based

A design graph grammar is called node-based, if the target graph of each rule consists of a single node only. Therefore the grammar has to be searched for target graphs in which the number of nodes is greater than one. If one is found, the design graph grammar cannot be node-based, otherwise it is.

The following pseudo-code algorithm implements this test:

```
boolean TESTNODEBASED(DESIGNGRAPHGRAMMAR 𝒢)
 (1)   {
 (2)      for(r ∈ 𝒢.rules) do
 (3)         GRAPH T := r.targetgraph;
 (4)         if(|V_T| > 1) then
 (5)            return false;
 (6)         fi
 (7)      od
 (8)      return true;
 (9)   }
```

4.6   Monotonicity/Shortcut-free

Monotonicity and shortcut-free are the two properties which cannot be decided completely.

As mentioned in the beginning of section 4, the rule sets are examined isolatedly, i.e. without any knowledge of the initial graph, with which this design graph grammar is started.

Definition 16 says that a design graph grammar $\mathcal{G}$ is monotonic, if each graph $G \in \mathcal{L}(\mathcal{G})$ can be derived monotonically. But $\mathcal{L}(\mathcal{G})$ cannot be known without any knowledge of the host graph. But that also means that monotonicity cannot be determined completely.

But definition 16 also says that a derivation is monotonic, if it does not involve deletion

operation. That means that if there do not exist deletion operations within the design graph grammar, this graph grammar has to be monotonic, because each derivation is monotonic.

This leads to the following algorithm in pseudo-code, which yields the following integer values:

1. 1(true – the design graph grammar is monotonic) or

2. ∞(maybe – the design graph grammar might not be monotonic) or

3. 0(false – the design graph grammar is definitely not monotonic). But this case cannot be decided, therefore this case does not occur within the algorithm.

```
        int TESTMONOTONICITY(DESIGNGRAPHGRAMMAR 𝒢)
(1)     {
(2)         for(r ∈ 𝒢.rules) do
(3)             GRAPH T := r.targetgraph;
(4)             GRAPH R := r.replacementgraph;
(5)             if(|E_T| > |E_R|) then
(6)                 return ∞;
(7)             fi
(8)             if(|V_T| > |V_R|) then
(9)                 return ∞;
(10)            fi
(11)        od
(12)        return 1;
(13)    }
```

The test for shortcut-free is exactly the same as for monotonicity because it cannot be determined completely due to the above stated reasons. But if each derivation is monotonic, because there do not exist deletion operations within the design graph grammar, of course the shortest derivation has to be monotonic, too, therefore the design graph grammar is shortcut-free.

5 Summary

This approach of an analysis of design graph grammar properties only examines statically the rule set of design graph grammars.

Because of this, two properties – shortcut-free and monotonicity – cannot not be determined completely. Only in a few cases it can be said that these properties are not true, but it cannot be determined that they are true. But these two properties – especially monotonicity – are important to be known, because, if they are true, the effort of the measure of the quality of a design is reduced.

In order to also be able to completely determine these two properties, it would be necessary to examine the whole design graph grammar. But the runtime of this test would be very high, because the language of this design graph grammar would have to be determined. That means, each possible derivation would have to be calculated, and afterwards, it would have to be checked, whether each graph g $\in \mathcal{L}(\mathcal{G})$ can be derived without the use of a deletion operation.

But all in all it can be said that this approach was a successful approach – all other properties can be determined completely and within a decent runtime. Only the runtime of the important property confluence may be longer – particularly if the design graph grammar is very large.

But in many cases, if the user has special knowledge on the initial graph on which this design graph grammar is applied, he can choose weak confluence instead of confluence if he is sure that the host graph is boundary according to the examined design graph grammar. In most cases this test has a much smaller runtime as the test for confluence – as shown in section 4.2.

A PropertyExaminer-Help

A.1 Getting Started

First choose File → Load in the menu, then select the Rule Description File in the file-chooser dialog-box, which opens.

If this description-file is syntactically correct, the file is loaded, and a new dialog opens. Select the properties which shall be examined.
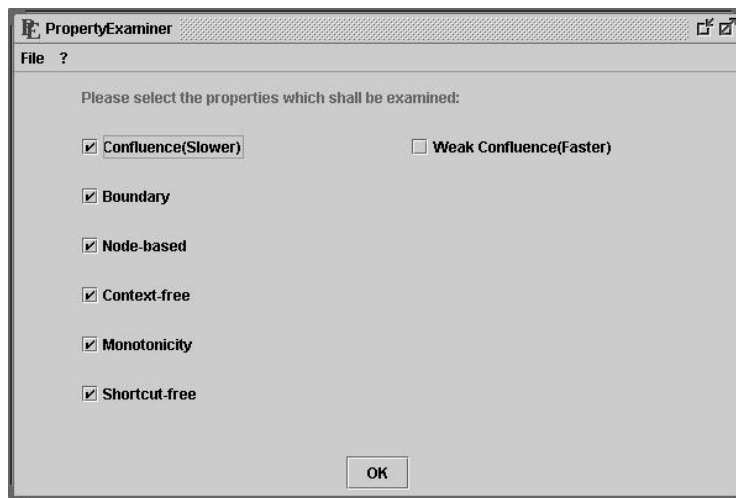

Figure 12: Mask, in which the examined properties can be selected.

After a short calculation a new dialog appears, which presents the results. Behind each property you can find a table entry, which says either "Yes"(the property is met), "No"(the property is not met) or "Maybe"(it cannot be decided whether this property is met or not).

Figure 13: Mask showing the results.

If you click on the button "Details...", more information is displayed.

If the property "Weak Confluence" is determined to be "Maybe", a button "More" appears. If you click on this button, a further calculation is started, and the result for confluence changes either to "Yes" or "No"(This calculation may take some time).

A.2   The Menu

A.2.1   File

A.2.1.1 Load

Loads a file with extension ".rdf" into this PropertyExaminer.

If you click on this menu entry, a file-chooser dialog-box opens, and you can select the design graph grammar which shall be examined.

If this description file is syntactically correct, the file is loaded, and a new dialog opens.

Otherwise, if this file does not correspond to the RDF-Description, an error message is displayed, and the file is not loaded into the PropertyExaminer. For a detailed error message choose File → Validator.

A.2.1.2 Edit

Opens the actually loaded file into an editor.

There, you have the possibility to either just view the content of this file, or even to change the content.

After changing, you can save the changes by choosing the menu entry File → Save.

A syntax analysis is performed. If the changes have created a valid RDF-File, it is saved and loaded into the PropertyExaminer. Otherwise an error message with the first wrong line is displayed, and the changes are not saved.

A.2.1.3 Validate

Examines the syntax of a RDF-File.

If you click on this menu entry, a file-chooser dialog-box opens, and you can select the design graph grammar to be examined. An editor with the selected file opens.

If this description file is syntactically correct, an information message opens saying that it is correct. You can either just close the editor or change this file.

Otherwise an error message with the first wrong line is displayed, and you can change this file and save the changes. The syntax is checked once again.

A.2.1.4 Visualize

Shows a visualization of the design graph grammar.

In the combo-box you can choose the rule which shall be displayed.

Four tabs are shown, on which the context graph, the target graph, the replacement graph and the embedding instructions are visualized.

The graph always moves its nodes, until the edges have a length of about 70 pixels so that the graph is not too scattered or too crammed. If this shall be stopped, just select "Freeze" besides the combo-box.

All nodes of the graph can be moved by drag & drop to the desired place in order to improve the clarity of this visualization.

A.2.1.5 Exit

Closes this PropertyExaminer.

A.2.2   Help

A.2.2.6 Help

Opens the PropertyExaminer-Help-Console, in which you can browse as in any HTML-Browser.

With the buttons "<" and ">" you can navigate through the pages.

The button "Home" opens the Table of Contents.

A.2.2.7 Theory

Opens a PostScript-Console with the contents of this thesis.

With the slider you can zoom the page bigger or smaller.

With the buttons "<" and ">" you can go one page back or forward.

Into the textfield, you can write a page number and jump directly to this page.

A.2.2.8 About

Opens a dialog-window with information on this version of the PropertyExaminer.

A.3 Confluence vs. Weak Confluence

If you want to test the design graph grammar with the regard to confluence, you have two possibilities to choose: Confluence or Weak Confluence.

Weak Confluence is the faster test, and just tests whether the design graph grammar is confluent, if the host graph is boundary according to this design graph grammar. If this test says "Yes"("No"), the design graph grammar is (not) confluent no matter which host graph. If this test says "Maybe", this design graph grammar is confluent, if the host graph is boundary according to this design graph grammar. Then a button "More" appears, which will start the normal confluence test.

Confluence is - especially with a great number of instructions - the slower test. So if Weak Confluence is sufficient, you should choose this one.

B   RDF-Description

RDFFILE	::= VER RULES
VER	::= "V 1.0"
RULES	::= RULE RULES | RULE
RULE	::= "rule" RULENAME CONTEXT TARGET REPLACEMENT EMBINSTRUCTIONS
CONTEXT	::= "c" GRAPH
TARGET	::= "t" GRAPH
REPLACEMENT	::= "r" GRAPH
GRAPH	::= "nodes" NUMBEROFNODES (NODE)^{NumberOfNodes} "edges" NUMBEROFEDGES (EDGE)^{NumberOfEdges}
NODE	::= NUMBER ";" NODELABEL ["*"]
EDGE	::= "e" NUMBER ";" NODELABEL ";" NODELABEL ";" EDGELABEL ";" ("G" | "U")
EMBINSTRUCTIONS	::= "I" "entries" NUMBEROFINSTR (INSTR)^{NumberOfInstr}
INSTR	::= NODELABEL ";" NODELABEL ";" EDGELABEL ";" NODELABEL ";" NODELABEL ";" EDGELABEL
NUMBEROFNODES	::= NUMBER
NUMBEROFEDGES	::= NUMBER
NUMBEROFINSTR	::= NUMBER
NUMBER	::=("1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0")*
NODELABEL	::= LABEL
EDGELABEL	::= LABEL
LABEL	::=("A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"| "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"| "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0")*

Comments are started with '%' and reach up to the end of the line.

References

F.-J. Brandenburg. On the complexity of the membership problem of graph grammars. *Proceedings of the WG '83*, pages 40–49, 1983.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

T. Kasami. An efficient recognition and syntax algorithm for context-free languages. *Scientific Report AFCRL-65-758*, 1965.

G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*. World Scientific, 1997.

G. Rozenberg and J. Engelfried. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

G. Rozenberg and E. Welzl. Boundary NLC graph grammars – basic definitions, normal forms, and complexity. *Information and Control*, 69:136–167, 1986.

A. Schulz, B. Stein, and A. Kurzok. On the automated design of technical systems. University of Paderborn – to appear, 2001.

R. Schuster. *Graphgrammatiken und Grapheinbettungen: Algorithmen und Komplexität*. PhD thesis, Fakultät für Mathematik und Informatik der Universität Passau, 1987.

E. Welzl. Boundary NLC and partition controlled graph grammars. *Lecture Notes in Computer Science*, 291:593–609, 1986.

D. Younger. Recognition and parsing of context free languages in time n^3. *Information and Control*, 10:189–208, 1967.