INTRODUCTION TO GRAPH GRAMMARS
WITH APPLICATIONS TO SEMANTIC NETWORKS

HARTMUT EHRLIG
Technische Universität Berlin, Fachbereich Informatik
Franklinstraße 28/29, D-1000 Berlin 10, Germany

ANNEGRET HABEL, HANS-JÖRG KREOWSKI
Universität Bremen, Fachbereich Mathematik und Informatik
Bibliothekstraße, D-2800 Bremen 33, Germany

Abstract—In the first half of this paper, we give an introductory survey on graph grammars that provide rule-based mechanisms for generating, manipulating and analyzing graphs. In the second half, two potential applications of graph-grammar concepts to semantic networks are indicated.

1. INTRODUCTION

Graphs are frequently used in various fields of computer science and artificial intelligence for representing knowledge of complex structure. If they serve more than merely illustrative purposes, one needs concepts and methods to build graphs up systematically, to analyze them, to retrieve information from them, to update the stored information, etc. This is the domain of graph algorithms and graph grammars where the former are mainly concerned with the inspection of graphs for certain properties, while the latter provide means for generating, manipulating and reducing graphs. (See, e.g., [1,2] for a survey on graph algorithms and the proceedings of the international workshops on graph grammars [3–5] for an insight into the spectrum of graph-grammar models and [6,7] for syntactic pattern recognition as the most important area of graph-grammar applications.)

In this paper, we are going to outline some basic features of graph grammars and to indicate their significance for the area of semantic networks. This is an attempt to bring these two flourishing areas together. We think it is worthwhile because the area of semantic networks is one of the most intriguing and challenging applications of graphs while the area of graph grammars provides rule-based methods for handling graphs in a systematic and mathematically precise way.

The paper is organized in two sections. The first section introduces the reader to graph grammars where we follow the so-called algebraic approach initiated by Ehrig, Pfender, Rosen and Schneider [8,9]. After recalling the notions of a graph and a graph morphism, we discuss operations on graphs that allow the removal and the addition of nodes and edges. Finally, the key notions of a rule and its application to a graph are defined. A rule specifies WHAT is to be removed and added (independent of an actual graph in process). The application performs the actual operations after choosing WHERE it should take place. We illustrate all concepts in this section by examples that are related to a very small imperative programming language. In the second section, we discuss potential applications of the concepts of graph grammars to the area of semantic networks with main emphasis on the consistent definition of semantic networks and rule-based type definitions.

The authors are very grateful to Fritz Lehmann and the anonymous referees for various helpful suggestions.

Typeset by AMS-TEX
2. INTRODUCTION TO GRAPH GRAMMARS

One encounters quite a variety of graph-grammar approaches in the literature (the oldest and most influential ones developed by Rosenfeld, Pfaltz and Milgram [10,11], Ehrig, Pfender, Rosen and Schneider [8,9] and Nagl [12]). The major stimulus for introducing and studying graph grammars has come from application areas such as pattern recognition, picture processing, data base systems, semantics of programming languages, compiler construction, process control, specification of data types and non-sequential systems. A few approaches were motivated primarily by purely mathematical aspects.

A typical formal generative grammar for a language (be it a natural language like English or a purely artificial language like a computer programming language) consists of a set of production rules or rewriting rules in which one symbol representing a part of a sentence of the language is removed and a string of symbols is substituted in its place. Often one starts with the simple symbol $S$ (for "sentence") and repeatedly applies the rewriting rules until a complete sentence of the language is derived. An example of such a grammar can be found in Section 2.8. The rewriting rules with left and right-hand sides are presented in "Backus-Naur" form. Such grammars were first studied formally in the 1920's by the Norwegian Axel Thue. Formal grammars for natural languages were developed by Zellig Harris and others and were advanced and widely popularized by Noam Chomsky in the 1950's. Since the 1960's this work has been used to formalize computer programming languages and automata theory (see, e.g., [13,14]).

The foregoing approaches deal only with linear languages, based on strings. Graph grammars provide such rule-based mechanisms for generating, manipulating and analyzing graphs. In this section, we give an introduction to graph grammars stressing the notions of a rule, a derivation and a derived graph.

As in the string case, a graph grammar rule comprises a left-hand side graph $L$, specifying what should be "ripped out," and a right-hand side graph $R$, specifying what should be substituted instead. If such a rule is applied to a host graph $M$, the graph $L$ must be matched with a part of $M$ (which is done in practically all graph grammar approaches by choosing a graph morphism from $L$ to $M$, which may be subject to further application conditions). Some edges of $M$ may be incident to nodes of the occurrence of $L$ in $M$ without being in the occurrence themselves. If the whole occurrence of $L$ is removed, they become dangling edges that must be reconnected in some way after the graph $R$ is added. Various graph grammar approaches follow this line of graph rewriting. They differ from each other in the way the dangling edges (the number of which can become arbitrarily large) are processed and reconnected. We advocate a more protective, strongly local approach. For this purpose, we assume that $L$ and $R$ share a common context graph $K$ and that edges of $M$ outside the occurrence of $L$ in $M$ are incident with nodes of $K$ if they are incident with the occurrence at all. In other words, $K$ contains the "attaching points" for the environment of $L$. Now the application of a rule can be done by removing everything from the occurrence of $L$ except $K$ and by adding $R$ where the corresponding context graphs are glued together. In this way, the numbers of items lost and of items gained in one step is bounded by the sizes of $L$ and $R$. We show in the following that this "protective approach" is easily introduced if one starts with the simple operations that allow one to remove or add single nodes or edges and to compose these operations properly into more complex applications of rules.

We start by recalling the notions of a graph and a graph morphism. Moreover, a reading operation is provided relating graphs and strings.

2.1. Graphs

We restrict our considerations to labelled directed graphs each consisting of a set of labelled nodes and a set of labelled edges where every edge has a source node and a target node.

Let $\text{LAB}$ be an arbitrary but fixed set of suitable labels.

Formally, a (labelled directed) graph (over $\text{LAB}$) is a system $M = (V, E, s, t, l, m)$ where $V$ is a set of vertices or nodes, $E$ is a set of edges, $s, t : E \rightarrow V$ are functions assigning each edge $e \in E$ a source $s(e)$ and a target $t(e)$, and $l : E \rightarrow \text{LAB}$ as well as $m : V \rightarrow \text{LAB}$ are functions labelling every edge and node respectively. The collection of all graphs over $\text{LAB}$ is denoted by
**Introduction to graph grammars**

\( G_{LAB} \). The components \( V, E, s, t, l, \) and \( m \) of a particular graph \( M \) may be referred to as \( V_M, E_M, s_M, t_M, l_M \) and \( m_M \), respectively.

We assume that \( LAB \) contains a default label "*". If a node is labelled by *, it may be interpreted as unlabelled. If all nodes of a graph are labelled by * (this is if all nodes can be considered as unlabelled), the graph is an *edge-labelled graph*. Analogously, a graph is said to be *node-labelled* if all edges are labelled by *. In other words, directed graphs the nodes and edges of which are unlabelled or only partially labelled can be considered as special cases of the notion above.

Figure 2.1 shows some graphs. On the left is the name of each whole graph.

![Graphs](image)

Figure 2.1. Five graphs consisting primarily of unlabelled nodes and labelled edges.

All five graphs happen to belong to the graphical description of a small programming language which will be discussed in Section 2.8 in more detail. The nodes are represented by the dots, small circles and diamonds. All of them are unlabelled, that is, they are labelled by the default label * which is omitted in drawings of graphs. The edges are given by the arrows with their labels next to them. Each arrow head points to the target of the edge.

### 2.2. Relating Strings with Graphs

If one reads the labels of a sequence of items of a graph in a certain order, one gets a *string* in which each member of its "alphabet" is a label. Conversely, there is a straightforward way of interpreting strings as graphs relating sequential data structures nicely with graphs.

More formally, let \( p = e_1 \ldots e_n \) be a sequence of edges of a graph \( M \). Then the corresponding sequence of edge labels \( l_M(e_1) \ldots l_M(e_n) \) yields a single string. This applies particularly if \( p \) is a path from a node \( v \) to a node \( v' \) (i.e., \( s_M(e_i) = v, t_M(e_n) = v' \) and \( t_M(e_i) = s_M(e_{i+1}) \) for \( i = 1, \ldots, n - 1 \)). Reading along all paths from \( v \) to \( v' \) yields a set of strings denoted by \( READ(M, v, v') \). Clearly, reading is also possible for a sequence of nodes and a mixed sequence of nodes and edges.
The following examples of the results of the function \textsc{Read} concern the graphs given in Figure 2.1:

\[ \text{\textsc{Read}}(\text{\textsc{test}}, \text{o}, \text{o}) = \{< \text{variable} > \neq < \text{variable} >\} \]

\[ \text{\textsc{Read}}(\text{\textsc{expression}}, \text{o}, \text{o}) = \{\text{\textsc{succ}} < \text{variable} >, \text{\textsc{pred}} < \text{variable} >, 0\} \]

\[ \text{\textsc{Read}}(\text{\textsc{statement}}, \text{o}, \text{o}) = \{< \text{variable} > : = < \text{\textsc{expression}} >, \text{\textsc{while}} < \text{\textsc{test}} > \text{\textsc{do}} < \text{\textsc{statement}} >, \text{\textsc{begin}} \text{\textsc{end}}\} \cup \{\text{\textsc{begin}} < \text{\textsc{statement}} > ; \text{\textsc{end}} \mid n \geq 0\} \]

Conversely, every string \( w = z_1 \ldots z_n \) with \( z_i \in \text{LAB} \) for \( i = 1, \ldots, n \) gives rise to the edge-labelled string graph

\[ S(w) = \circ z_1 \bullet z_2 \bullet \cdots \bullet z_n \circ \]

consisting of \( n + 1 \) nodes and \( n \) edges that form a path. The \( i \)-th edge is labelled by the \( i \)-th symbol of the given string. In the drawing, each label is placed next to its edge. We mark the beginning of the string graph with \( \circ \) and the end with \( \circ \).

Obviously, we have \( \text{\textsc{Read}}(S(w), \text{o}, \text{o}) = \{w\} \) such that the string graph \( S(w) \) represents the string \( w \) uniquely. As an example, the reader may notice that the graph \( \text{\textsc{test}} \) in Figure 2.1 equals the string graph \( S(< \text{variable} > \neq < \text{variable} >) \).

### 2.3. Subgraphs, Isomorphic Graphs, Graph Morphisms

Quite often, a graph must be matched with another one to find structural similarities. If one considers graphs as abstract structures in the sense that the labelling and the interrelation of edges and nodes matter, but not the particular identity of nodes and edges, the appropriate notion is the graph morphism.

Let \( M \) and \( N \) be two graphs. A graph morphism \( f : M \rightarrow N \) consists of two functions \( f_v : V_M \rightarrow V_N \) and \( f_e : E_M \rightarrow E_N \) subject to the condition that the labels and the interrelations of edges and nodes are preserved, i.e., for all \( e \in E_M \) and \( v \in V_M \) the following properties hold:

1. \( f_v(s_M(e)) = s_N(f_e(e)) \) and \( f_v(t_M(e)) = t_N(f_e(e)) \).
2. \( l_N(f_e(e)) = l_M(e) \) and \( m_N(f_e(v)) = m_M(v) \).

Note that a graph morphism may map the nodes and edges of the graph \( M \) to different individuals of \( N \) as long as it keeps the structural information intact. Different nodes of \( M \) may be identified if they carry the same label, and different edges may be identified if they carry the same label and their sources are equal or mapped to the same node and their targets as well.

If the functions \( f_v \) and \( f_e \) of a graph morphism \( f : M \rightarrow N \) are both bijective, the graphs \( M \) and \( N \) are called isomorphic, denoted by \( M \cong N \). Isomorphic graphs are structurally equal, but corresponding nodes and edges may have different names.

If the two functions are inclusions, i.e., \( V_M \subseteq V_N \) and \( E_M \subseteq E_N \), then the labellings and source and target functions of \( M \) are restrictions of those in \( N \). In other words, \( M \) is a subgraph of \( N \) in this case, denoted by \( M \subseteq N \).

Here are some examples. The string graph \( S(\text{\textsc{begin}} \text{\textsc{end}}) \) can be considered as a subgraph of \( \text{\textsc{statement}} \), but the string graph

\[ \text{\textsc{begin}} \text{\textsc{statement}} \cdots \text{\textsc{end}} \]

cannot. But if we identify the nodes 1 and 3 as well as 2 and 4 and the two edges between them, we do get a graph morphism into the graph \( \text{\textsc{statement}} \). The graph \( \text{\textsc{test}} \) has two distinct subgraphs isomorphic to the string graph \( S(< \text{variable} >) \). In other words, there are two distinct graph morphisms from \( S(< \text{variable} >) \) into \( \text{\textsc{test}} \).

So far, graphs are handled as static objects with a way of retrieving information by reading. But this is inadequate in most applications. If a graph represents an actual state of a system or actual knowledge of some kind, there is a need for updating, adding or removing information. Hence, we are going to outline a more dynamic view of graphs starting with the most elementary operations of removing and adding single items ending up with the notions of a rule and its application to a graph.
2.4. Removal and Addition of a Single Item

It is easy to imagine that a finite graph can be transformed into any other finite graph by removing and adding nodes and edges in a proper succession. The four elementary operations needed are defined on a graph \( M \) as follows:

1. For an edge \( e \in E_M \), \( \text{rem}(M, e) \) removes \( e \) from \( E_M \) without changing anything else. In particular, we get \( \text{rem}(M, e) \subseteq M \).
2. For a node \( v \in V_M \), \( \text{rem}(M, v) \) removes \( v \) from \( V_M \) provided that \( v \) is a disconnected node. In particular, \( \text{rem}(M, v) \subseteq M \).
3. For a node \( v \notin V_M \) and \( x \in \text{LAB} \), \( \text{add}(M, v, x) \) adds \( v \) as a new node labelled by \( x \). The new node is disconnected from the rest of the graph. In particular, \( M \subseteq \text{add}(M, v, x) \).
4. For some \( e \notin E_M \), \( v, v' \in V_M \) and \( y \in \text{LAB} \), \( \text{add}(M, e, v, v', y) \) add \( e \) as a new edge with source \( v \), target \( v' \) and label \( y \). In particular, \( M \subseteq \text{add}(M, e, v, v', y) \).

Note that it is easier to remove an edge than a node because one must check a condition in the latter case. The condition makes sure that, after the removal of a node, no dangling edges are left. Clearly, we can also remove nodes that are not isolated if first their incident edges are removed.

Let us illustrate the operations by applying them to the graph \( \text{STATEMENT} \) (shown in Figure 2.1). Let \( v_0 \) be the node which is incident to the edge \( e_1 \) labelled by \( \text{begin} \) and to the edge \( e_2 \) labelled by \( \text{end} \). If one wants to get rid of the substructure formed by \( v_0, e_1 \) and \( e_2 \), one succeeds by the following removals:

\[
\text{STATEMENT}' = \text{rem}(\text{rem}(\text{rem}(\text{STATEMENT}, e_1), e_2), v_0).
\]

Instead of the removed items, one may like to add an edge which replaces the removed information. This can be achieved by the following addition:

\[
\text{STATEMENT}'' = \text{add}(\text{STATEMENT}', e_3, \text{o}, \text{o}, \text{skip})
\]

where \( e_3 \) is some name for an edge not yet used.

2.5. Removal and Addition of Sets of Items

It is a tedious task to update a graph by removals and additions of single items only. Fortunately, both operations are easily extended to sets of nodes and edges if one provides the necessary information.

If \( E \subseteq E_M \) and \( V \subseteq V_M \), we let \( \text{rem}(M, E, V) \) yield the subgraph of \( M \) which is specified by \( E_M - E \) as set of edges and \( V_M - V \) as set of nodes, where \( A - B \) denotes the difference of the set \( A \) and \( B \) containing all elements of \( A \) that do not belong to \( B \).

The resulting graph \( \text{rem}(M, E, V) \) is obtained by removing all edges in \( E \) and all nodes in \( V \) one by one (provided that \( E \) and \( V \) are finite sets). To meet the condition that only isolated nodes should be removed, an edge \( e \) incident to a node \( v \in V \) must belong to \( E \). This property of the pair \( (E, V) \) is called the contact condition.

If \( V' \) and \( E' \) are sets disjoint from \( V_M \) and \( E_M \), \( \text{add}(M, V', E', s', t', l', m') \) yields a supergraph of \( M \) with the disjoint union \( V_M + V' \) as set of nodes and the disjoint union \( E_M + E' \) as set of edges provided that \( s', t' : E' \rightarrow V_M + V', t' : E' \rightarrow \text{LAB} \) and \( m' : V' \rightarrow \text{LAB} \) are functions specifying the sources, the targets and the labels of new edges and the labels of new nodes respectively.

If, especially, all sources and targets of new edges are nodes in \( V' \), the added information specifies a graph \( M' \). In this case, the resulting graph consists of the two subgraphs \( M \) and \( M' \) disconnected from each other. This disjoint union of \( M \) and \( M' \) is denoted by \( M + M' \).

2.6. Specifying Removal and Addition

The operations \( \text{rem} \) and \( \text{add} \) have at least three drawbacks that lead to further adjustments:

1. The nodes and edges to be removed or added must be chosen as explicit subsets of items in the presence of the graph in process. This establishes a quite liberal form of interaction,
but turns out to be inadequate in many other circumstances. Quite often removals and additions are of the same kind for a variety of host graphs and can be expressed independent of the actual host. For example, one may like to say: Whenever there is a substructure of the form

\[
\begin{align*}
\text{begin} & \quad \text{end}
\end{align*}
\]

remove it, and add an edge labelled by \textit{skip} instead. This applies not only to the graph \textit{STATEMENT}, but to any graph where the substructure occurs.

(2) So far, nodes and edges to be removed or added must be enumerated. There is no other indication how they may be specified systematically. For example, the kind of substructure we refer to in Point 1 intuitively, cannot be used as a parameter of the \textit{rem}-operation officially by our definition.

(3) Arbitrary changes of graphs are possible by removals and additions if they are chosen appropriately. But often some specific updates with particular combinations of removals and additions are intended. An example of this kind has appeared already in Point 1.

The following observations help to deal with the drawbacks in Point 1 and 2. Point 3 will be reconsidered in the next subsection.

Let \( M \) be a graph and \( V \subseteq V_M, E \subseteq E_M \) such that the pair \((E, V)\) satisfies the contact condition. Then there are subgraphs \( K \subseteq L \subseteq M \) such that \( V = V_L - V_K \) and \( E = E_L - E_K \). This means that each pair of a set of nodes and a set of edges can be obtained as the difference of two subgraphs. Consequently, we can specify the pair \((E, V)\) by the graphs \( K \) and \( L \) and denote \( \text{rem}(M, E, V) \) by \( \text{rem}(M, L - K) \). Moreover, \( K \) is a subgraph of the resulting graph because no item of \( K \) gets removed. The smallest possible \( K \) consists of all nodes that are not in \( V \), but are sources and targets of edges in \( E \).

Let \( g : L \rightarrow M \) be a graph morphism (defined above in Section 2.3) and \( K \subseteq L \) as well as \( I \subseteq M \). Then the images \( g(K) \) and \( g(L) \) of \( K \) and \( L \) in \( M \) under \( g \) are subgraphs of \( M \) with \( g(K) \subseteq g(L) \). And if \( g(K) \subseteq I \), then the restriction \( d : K \rightarrow I \) of \( g \) is a graph morphism, where \( d \) is explicitly defined by \( d_v(v) = g_v(v) \) for all \( v \in V_K \) and \( d_E(e) = g_E(e) \) for all \( e \in E_K \). Consequently, \( g \) and \( K \subseteq L \) specify the removal \( \text{rem}(M, g(L) - g(K)) \).

Summarizing, a potential removal is just specified by a graph \( L \) and a subgraph \( K \) of \( L \) independent of the processed graph. If we have gotten an actual graph \( M \) and the removal should be performed, we must match \( L \) with a part of \( M \) choosing a graph morphism \( g : L \rightarrow M \). Then \( I = \text{rem}(M, g(L) - g(K)) \) is defined by removing \( E = E_{g(L)} - E_{g(K)} \) and \( V = V_{g(L)} - V_{g(K)} \) provided that the pair \((E, V)\) satisfied the contact condition. Moreover, we have \( g(K) \subseteq I \) which allows one to define the restriction \( d : K \rightarrow I \) of \( g \) which keeps the context in which the removal has taken place. Note that there may be more than one place in the host graph where the graph \( L \) matches and the removal could be done. But all of them are distinguished by different graph morphisms. In other words, the choice of the graph morphism identifies the place of removal uniquely.

The information needed for adding can be specified in a similar way. Given a graph \( R \) and a subgraph \( K \), the difference provides the sets \( V' = V_R - V_K \) and \( E' = E_R - E_K \) to be added (independent of any processed graph). The restrictions of source and target and label functions of \( R \) to \( V' \) and \( E' \) provide all further information needed for an addition except for the case of an edge in \( E' \) with source or target not in \( V' \). But if we have got an actual graph \( I \) and we want to perform the addition, the missing information is obtained by a graph morphism \( d : K \rightarrow I \) because a source or a target of \( e \in E' \) which is not in \( V' \) must be in \( V_K \) and can be mapped to \( I \). The image turns out to be a suitable source or target of the added edge. Moreover, let \( N \) denote the resulting graph, then the graph morphism \( d \) can be extended to a graph morphism \( h : R \rightarrow N \) by the identity on the added items.

An example may help to clarify the situation. Let

\[
L = S(< \text{variable} >) + S(< \text{expression} >)
\]

be the disjoint union of the two string graphs and \( K \) be the subgraph consisting of the four nodes of \( L \), see Figure 2.2.
This specifies the potential removal of the edges. There is a unique graph morphism from $L$ into the $STATEMENT$ because in this case the labels $<variable>$ and $<expression>$ occur only once in $STATEMENT$. The contact condition is trivially satisfied because no node is going to be removed. If we do the removal according to this specification, we get the graph in Figure 2.3.

The resulting graph morphism $d$ from $K$ to $I$ maps each node of $K$ into the corresponding node of $I$ (where correspondence is established by the source and target relation of the removed edges). Let $R$ be the disjoint union of the graphs $VARIABLE$ and $EXPRESSION$ and $K$ be again the graph consisting of the four nodes that are drawn as circles and diamonds. Then an addition can be done with respect to the graph morphism $d$ above yielding the graph in Figure 2.4.

2.7. Rules and Derivations

Now it is only a small step to combine a removal and an addition into a composite update operation.

A rule $r$ consists of two graphs $L$ and $R$ sharing a common subgraph $K$. $L \supseteq K$ specifies a potential removal and $K \subseteq R$ a potential addition. The shared context $K$ means that addition will take place to the context of the removed parts only.

According to the considerations above, a rule is applied to a graph $M$ in the following way:

1. Choose a graph morphism $g : L \rightarrow M$ matching the left-hand side of the rule to the host graph and check the contact condition.
2. Remove $g(L) - g(K)$ from the host yielding the intermediate graph $I$ with the context morphism $d : K \rightarrow I$.
3. Add $R - K$ to $I$ with respect to $d$ yielding the resulting graph $N$ and the graph morphism $h : R \rightarrow N$. 
The application of a rule to a graph is summarized in the diagram in Figure 2.5, where the upper row, which is the rule, may be specified in advance independent of any particular host graph, whereas the graph morphism \( g \) must be chosen for each application of the rule. The rest is constructed automatically. This diagram is called the *derivation diagram* because it establishes a direct derivation of a graph \( N \) from a graph \( M \). But one should be aware of the fact that usually the significant aspects of graphs are their structures and their labels. In other words, every graph \( N' \) isomorphic to the explicitly constructed graph \( N \) represents the result of an application of the rule \( r \) to the graph \( M \). This leads to the following definition.

Given some graph \( N' \) isomorphic to the constructed graph \( N \), we say that \( M \) directly derives \( N' \) through \( r \) and \( g \), denoted by \( M \Rightarrow^r g N' \) where \( r \) is the applied rule and \( g \) is the chosen graph morphism. We drop the superscripts \( r \) and \( g \) whenever they are not interesting or are clear from the given situation.

Because the application of a rule transforms a graph into a graph, the procedure can be iterated. A sequence of direct derivations of the form

\[
M_0 \Rightarrow M_1 \Rightarrow M_2 \Rightarrow \ldots \Rightarrow M_n
\]

is called a *derivation* from \( M_0 \) to \( M_n \), and \( M_n \) is said to be derived from \( M_0 \). This fact is denoted by \( M_0 \Rightarrow^P M_n \) where all applied rules belong to a given set \( P \) of rules.

Consequently, every graph \( Z \) as start graph or axiom and every set \( P \) of rules specify a *graph language* consisting of all graphs derived from \( Z \) by applying rules from \( P \):

\[
L(P, Z) = \{ M \in GLAB | Z \Rightarrow^P M \}.
\]

Moreover, if we consider only a part \( T \) of the label alphabet \( LAB \) as terminal labels (i.e., those labels permissible in the final output graph) and the symbols in \( LAB - T \) as auxiliary only, we may accept only those derived graphs that are fully labelled by symbols from \( T \):

\[
L(T, P, Z) = L(P, Z) \cap G_T.
\]

The triple \((T, P, Z)\) comprises the notion of a *graph grammar*, and \( L(T, P, Z) \) is its generated language.

Readers familiar with the traditional notion of a Chomsky grammar applying to linear strings of symbols may like to see how it fits into the framework of graph grammars. Let \( G = (T, P, \alpha) \) be a Chomsky grammar where \( T \) is a terminal alphabet, \( P \) is a set of productions each of which consisting of two strings and \( \alpha \) is a start string. Then the graph-grammar version \( S(G) \) of \( G \) is obtained by taking the same \( T \), transforming each production \((u, v)\) into the rule \( S(u) \supseteq \circ \circ \subseteq S(v) \) and taking \( S(\alpha) \) as start graph. If the set of transformed rules is denoted by \( S(P) \), the derivations and languages of the two grammars are nicely related: \( w \in T^* \) is derived from \( \alpha \) by applying productions from \( P \) if and only if \( S(w) \) is derived from \( S(\alpha) \) by applying rules from \( S(P) \). And hence, we have \( L(T, S(P), S(\alpha)) = \{ S(w) | w \in L(G) \} \) if \( L(G) \) denotes the language generated by \( G \) which consists of all terminal strings derivable from \( \alpha \). Note that the \( \circ \) and \( \circ \) symbols are the left and right “attaching points” for any segment of text during the production. Applying a production rule means ripping out the left-hand side of a rule as discovered within the string, and putting the right-hand side string segment in its place.

### 2.8. A Graph-Grammar Example

Kfoury, Moll and Arbib [15] introduce a tiny programming language for defining computable functions. The language provides the non-negative integers as the only data type. The variables
are always global and start with \( X \) followed by a sequence of ciphers. The only arithmetic
expressions are the constant 0 and the successor or predecessor of a variable, the only test is the
inequality of variables. Finally, the statements available are the assignment, the composition, the
while-loop and the empty statement. More formally, the syntax of the language is defined by the
following context-free productions in extended Backus-Naur form:

\[
\begin{align*}
< \text{statement} > & ::= < \text{variable} > ::= < \text{expression} > | \\
& \begin{array}{l}
\text{begin} < \text{statement} > ( ; < \text{statement} > ) \text{ end} \\
\text{while} < \text{test} > \text{ do } < \text{statement} > \text{ begin} \text{ end}
\end{array} \\
< \text{variable} > & ::= X < \text{cipher} > ^ * \\
< \text{cipher} > & ::= 0 | \ldots | 9 \\
< \text{expression} > & ::= 0 | \text{SUCC} < \text{variable} > | \text{PRED} < \text{variable} > \\
< \text{test} > & ::= < \text{variable} > \neq < \text{variable} >
\end{align*}
\]

where all terms in brackets are nonterminal.

If we reformulate the syntax in the style of PASCAL-like syntax diagrams, we get a graph
grammar using the graphs of Section 2.1:

Rule 1: \( S(< \text{statement} >) \supset o o \subseteq \text{STATEMENT} \)
Rule 2: \( S(< \text{variable} >) \supset o o \subseteq \text{VARIABLE} \)
Rule 3: \( S(< \text{expression} >) \supset o o \subseteq \text{EXPRESSION} \)
Rule 4: \( S(< \text{test} >) \supset o o \subseteq \text{TEST} \)
Rule 5: \( S(< \text{cipher} >) \supset o o \subseteq \text{CIPHER} \)

Obviously, each of the five rules is of a simple form with just two “attaching points” in the
context graph and one edge in the left-hand side. Nevertheless, the application of such a rule
shows nearly all effects one can expect in general except for the contact condition which is
automatically satisfied. But this is not a big affair because the checking of the contact condition
can always be done locally in constant time. More complex rules can be found in Section 3. In
Section 2.9(4), we introduce the construction of parallel rules from given rules which yield also
more complex rules from the simple ones above. Actually, the illustrative example in Section 2.6
turns out to be an application of the parallel rule composed of \textit{Rule 2} and \textit{Rule 4}. Another
derivation is given in the following figure:

![Figure 2.6. A sample derivation generating the terminal test graph.](image)

Both syntactic descriptions of the language are nicely related by the reading operation defined
in Section 2.2. A string \( w \) is derivable from some category \( < \text{xyz} > \) (using the set of linear
string production rules in the Backus-Naur form) if and only if there is a graph \( M \) derivable from
S(<xyz>) (using the set of corresponding graph grammar rules) such that w can be retrieved from M by reading, i.e., \( w \in \text{READ}(M, o, o) \).

### 2.9. Some Results

It is beyond the scope of the paper to elaborate the theory of graph grammars. But we are going to mention some of the results to give a flavour of the achievements of the theory.

1. The derivation process handles graphs as abstract objects in the sense that isomorphic transformations do not matter. If \( M \cong M' \) and \( N \cong N' \), then \( M \) derives \( N \) if and only if \( M' \) derives \( N' \).
2. A rule with a right-hand side equal to the context graph specifies a removal operation because nothing has to be added. Dually, a rule with a left-hand side equal to the context graph specifies an addition because nothing has to be removed. So our notion of a rule covers all considerations in the Sections 2.4–6.
3. Consider the derivation diagram in Section 2.7. Our notion of a rule is obviously symmetric such that each rule \( r = (L \supseteq K \subseteq R) \) defines an inverse rule \( r^{-1} = (R \supseteq K \subseteq L) \). This rule turns out to be applicable to the derived graph \( N \) because the graph morphism \( h \) always satisfies the contact condition with respect to the rule \( r^{-1} \). This observation is of some significance if we want to parse our graphs in a bottom-up fashion. Unfortunately, the graph \( M' \) derived from \( N \) through \( r^{-1} \) and \( h \) is not isomorphic to the original host graph \( M \) in general. To fix this drawback, we must require an additional condition. Let \( g \) be the chosen graph morphism satisfying an identification condition which requires that different items of \( L \) that are identified by \( g \) must belong to \( K \). Then the following holds:

\[
M \Leftrightarrow N \text{ if and only if } N \overset{r^{-1}}{\Rightarrow} h M.
\]

This is the reason why we require that contact condition and identification condition are satisfied if a rule is applied. The assumption has another nice consequence. The two subdiagrams of the derivation diagram can be shown to be pushout diagrams in the sense of mathematical category theory which can be helpful in proving properties of derivations like the following in Points 4 and 5. A more detailed discussion of the first three points can be found in Ehrig [16].

4. Using the special addition in Section 2.5, two rules \( r_i = (L_i \supseteq K_i \subseteq R_i) \) for \( i = 1, 2 \) induce a parallel rule \( r_1 + r_2 = (L_1 + L_2 \supseteq K_1 + K_2 \subseteq R_1 + R_2) \) (recalling that \( G + G' \) means that \( G \) and \( G' \) are disconnected graphs on one surface).

A direct derivation \( M \overset{r + r'}{\Rightarrow} X \) can be sequentialized, which means that a graph \( N \) and direct derivations \( M \Rightarrow N \Rightarrow X \) exist provided that the identification condition is satisfied by the given derivation. Hence, the use of parallel rules can speed up the derivation process, but does not increase the generative power. Sequentialization is not possible in general if we drop the identification condition.

5. The converse is true; this means that successive direct derivation steps can be parallelized if they satisfy a certain independence condition. It cannot hold in general because the first step may produce items needed by the second one. For more details concerning the properties and potential applications of parallel derivations, see Kreowski [17].

6. As graph grammars generalize Chomsky grammars, the membership problem and nearly all interesting questions are undecidable for general graph grammars. But various classes of graph grammars are studied in the literature where much better analytical and structural possibilities are made available.

For example, hyperedge replacement grammars (where, in terms of the introduced concepts, one nonterminal node with all incident edges is removed in a derivation step and its direct neighbors, that are all terminal, form the context graph of the applied rule) behave very nicely. Just to mention two of the results, the generated languages can be shown to be certain least fixed points, and it is decidable whether all generated graphs are planar, connected, acyclic, etc. (see, e.g., Habel [18] for more details).
(7) Other graph grammar models like plex grammars, web grammars, tree grammars, node-label-controlled graph grammars, etc., are either special cases of the introduced method or can be related to it in a meaningful way. Moreover, other types of graphs like undirected graphs and hypergraphs can be handled by the same techniques. For more details, the reader is referred to the proceedings of three graph grammar workshops [3-5].

3. APPLICATIONS TO SEMANTIC NETWORKS

In the literature, one encounters quite a variety of different models of semantic networks (see, e.g., [19–23]). Usually semantic networks are described by labelled directed graphs that are used in artificial intelligence and computational linguistics for representing knowledge of a certain domain such as (fractions of) economy, botany, geometry, etc. or—where we take our illustrative examples from—the theory of graph grammars itself. Our consideration is based on the versions of semantic networks developed by Sowa [21–23] and Brachman and Schmolze [19] where we follow mainly the latter approach concerning the technical details. We are going to point out some potential applications of the framework and theory of graph grammars to semantic networks. We assume that the reader is somewhat familiar with the basic ideas of semantic networks, but we hope that there is also a fair chance to catch the meaning from the illustrations and explanations of how graph grammar concepts and methods may be employed for the description of semantic networks.

3.1. Consistent Generation of Semantic Networks

Using graph grammars themselves as our subject domain, our illustrative version of a semantic network comprises concepts such as GRAPH, RULE, GRAMMAR, etc., and conceptual relations or roles such as CONSISTS OF, IS LEFT OF, IS SUBGRAPH OF, IS APPLIED TO. Concepts and roles are typed in such a way that the subtype relations between concepts on the one hand and roles on the other hand are established. Concepts and roles are represented by nodes where concepts are drawn as boxes and roles get rounded forms. Edges between concepts establish the subconcept relation, edges between roles the subrole relation, and other edges connect roles and concepts with each other. For example, the sentences PLANAR GRAPHS AND CONNECTED GRAPHS ARE SPECIAL TYPES OF GRAPHS, A FULL SUBGRAPH IS A SUBGRAPH (a subgraph is called full if each edge of the supergraph between subgraph nodes belongs to the subgraph), K IS SUBGRAPH OF L may be expressed by the nets in Figure 3.1.

![Figure 3.1. Three graphs illustrating the representation of knowledge in semantic networks.](image_url)

For simplicity, we omit other kinds of nodes for attributes, individuals, values etc., and other kinds of edges. In particular, subtype links as well as ordinary relational links are simply represented by edges differentiated only by the forms and types of the nodes they connect. The use of edge labels could help to distinguish the various kinds of edges explicitly. Readers familiar with Sowa’s work may notice that the representation of subconcepts and subroles is the major difference between Sowa’s version of networks and ours. We describe the subconcept relation and the subrole relation by edges between corresponding nodes while Sowa assumes extra order relations in his “type lattice.” Moreover, there may be the danger of confusing the reader by applying graph grammars to semantic networks and using graph grammars as the subject domain at the same time. But both levels of considerations are clearly separated. Whenever we deal with graphs, we attempt to apply the concepts of graph grammars whereas the interpretation of
graphs (based on the labels of the nodes and the actual incidences of the edges) yields knowledge which belongs to the subject domain.

To ensure that semantic networks represent meaningful knowledge, one may require additional *well-formedness conditions* like the following:

1. A subconcept can take over all roles of its superconcept (inheritance).
2. Each role has at least an edge incoming from a concept and another one outgoing to a concept.
3. The subconcept relation is a partial order (or semilattice or lattice).
4. The subrole relation is a partial order.

Conditions of this kind can be found in [19]. A different type of conditions is discussed by Sowa [21] as so-called canonical formation rules. In any case, a set of such conditions defines a particular set of well-formed, intentionally meaningful semantic networks. Such an approach causes the problem of how to guarantee that only well-formed, consistent networks are processed. The good news: graph grammars may help.

The idea is simple enough. Choose a particular (small) well-formed network as start graph, and specify a set of graph grammar rules the application of which preserves well-formedness. Then every derived network is well-formed and consistent (by induction on the length of derivations).

Let us consider an example. We start with the graph shown in Figure 3.2, representing the most general concept being in conceptual relation to itself. The following rules are available to insert a subconcept, to restrict and to differentiate roles where all labels are variable in the sense that they can be chosen appropriately before a rule in Figure 3.3 is applied.

![Figure 3.2. A semantic network with a most general concept used as start graph.](image)

**Rule 1. Insertion of a subconcept**

![Rule 1](image)

**Rule 2. Insertion of a subrole with restricted interrelation**

![Rule 2](image)

**Rule 3. Insertion of a subrole with differentiated interrelation**

![Rule 3](image)

![Figure 3.3. Three rules for the generation of semantic networks in the style of KL-ONE.](image)

The notation of rules follows the conventions introduced in Section 2.7. In each of the three cases, the left-hand side equals the context graph such that nothing will be removed in applications of the rules. The first rule adds a new subconcept which inherits a number of roles from the
superconcept. By the other two rules, one can add subroles where, in the first case, the role name is kept and the argument concept as well as the value concept are restricted to subconcepts while, in the second case, the new role gets a subconcept as new value and the argument concept is kept. The third rule allows one in particular to introduce new roles as subroles of the general role of the start network.

It is not hard to prove that these rules preserve well-formedness as far as the four conditions given above are concerned. The rules allow one to generate fairly large semantic networks. To give some indication of the significance, we would like to mention Tank's unfortunately unpublished work [24]. He shows that a major part of KL-ONE (see [19]) can be covered by a few more rules than the three above.

Summarizing, we suggest (as Tank does in his paper and somewhat implicitly Sowa in [21]) the specification of a set of well-formed semantic networks as the language generated by a graph grammar. This approach may have some advantages:

1. Let the intended networks be defined by a set of well-formedness conditions. Then all generated networks are well-formed if the start graph is and the applications of a rule preserve well-formedness. Both may be simple to prove because the start graph is usually small and a direct derivation works locally endangering the well-formedness of the host network only in the small area matched with the left-hand side of the applied rule.

2. Clearly, one would like to design a set of rules that cover all well-formed networks. To achieve such a set of rules—if possible at all—seems to be a matter of the creative designer. But the theory of graph grammars supports at least the proof of completeness. For example, our rules above enlarge the actual graph in each step. If we would be able to apply a rule from right to left to an arbitrary well-formed network and if also the inverse rules preserve well-formedness, then we would get a smaller well-formed network and could deduce the derivability by induction on the size of the networks.

3. If the rules are monotone, that is, if the graphs in derivations are never shrinking (as in the case above), then the membership problem is decidable, and the rules can be used for the syntactical analysis of arbitrary networks. That is, the semantic network can be automatically parsed. Some nice cases of classes of graph grammars are known that have a fast solution of the membership problem.

4. If the rules are used as a kind of syntax-directed editor to build up well-formed semantic networks, one may be interested in speeding the process up. The theory of graph grammars provides some methods for parallelizing sequential derivations (cf. Section 2.9 (4, 5)).

It is beyond the scope of this paper to work the sketched ideas out fully for semantic networks. We discuss an application of graph grammars to data base systems of similar nature in our paper [25] where we demonstrate the listed advantages explicitly.

3.2. Rule-Based Type Definition and Expansion

If all the available knowledge of a certain domain is put into a single semantic network, one may end up with a large amount of information hard to handle and difficult to understand. An additional structuring principle is needed to be able to simplify networks without losing information. Sowa introduces in [22] the notion of a type definition where a part of a semantic network can be encapsulated as a new type. The other way round, a concept or a role of the new type can be expanded by its defining body wherever it occurs. Networks with nested propositions (see, e.g., [23]) provide a similar structuring principle. Graph grammars may also help in this area.

Let us look at an example first. In a large network, we may find the subnetwork in Figure 3.4, where CONTEXT, LHS and RHS are of type GRAPH (which is automatically satisfied in a network if the role IS-SUBGRAPH-OF is connected to concepts of type GRAPH only) and where the dotted line encloses a part connected with the concept C only. Then we may observe that the enclosed structure defines a rule in the sense of Section 2.7 justifying the simplification of Figure 3.5.
If we choose the node labelled by \( C \) as common subgraph \( K' \) of \( L' \) and \( R' \), we get a graph grammar rule \( r' = (L' \supseteq K' \subseteq R') \) which performs this simplification whenever applied. The reader may notice that the contact condition makes sure that the rule cannot be applied if the concepts and roles in the dotted line are connected with an outside node other than concept \( C \).

In general, a type definition can be defined by every rule \( r = (L \supseteq K \subseteq R) \) where \( K \) contains the context to which all nodes in \( L - K \) are connected, \( L - K \) is the part of the semantic network being the defining body of the new type and \( R \) must be chosen small enough to represent a new type (this may mean that it contains essentially only one new concept or role). A more liberal view may even accept that \( R \) is of a similar composite and complex nature as \( L \). If one inverts a type definition rule, one gets the corresponding type expansion \( r^{-1} = (R \supseteq K \subseteq L) \) which replaces the defined type by its defining body wherever it occurs.

With a set of type definition and expansion rules, a semantic network is no longer just a graph. But every network derivable from a given one by applying given rules should be considered as a different view of the same body of knowledge. The derived view may look simpler or more abstract if definition rules are applied. And it looks more detailed in those parts where expansion rules are applied. So the derivation process can be used as a kind of “zooming in” (as with a zoom lens on a camera) to adapt the description of knowledge in hand to an appropriate level of detail.

Moreover, if one considers a set of proper type expansion rules (meaning that each left-hand side of a rule is smaller than the right-hand side) and starts with a single network, then derivability between two derived graphs establishes a partial order on the generated language with the start graph as infimum. In other words, we obtain a kind of type hierarchy. The theory of graph grammars provides sufficient conditions and some help to prove that the generated language becomes a true mathematical lattice with respect to the partial order (see, e.g., [16, 17]).

Finally, we would like to point out that our small example grammar in Section 2.8 (our graph grammar for Kfoury’s, Moll’s and Arbib’s small programming language) can be interpreted as a type definition grammar. To see the relationship better, the reader may realize that labelled edges can be transformed into nodes (and so into concepts)

After this transformation, the graph grammar description of the programming language looks similar to a semantic network (with type expansion rules).

4. CONCLUSION

The paper presents our very first attempt to bring the areas of graph grammars and semantic networks together because we strongly believe that both can profit from each other. We introduce the reader to the basic elements of the theory of graph grammars and point out some potential and promising applications of graph grammars to the consistent definition of semantic networks and to the definition and expansion of types. In this way, we hope to raise some curiosity about graph grammars by those readers who are mainly interested in the topic of knowledge representation and processing.
We would like to invite everybody attracted to join us in further investigations of the mutual relationship of graph grammars and semantic networks. Among the topics of the future, we expect the following:

(1) In Section 3.1, we listed some points in favour of a graph-grammatical definition of well-formed semantic networks. Clearly, our expectations will not be justified until the ideas are applied to an accepted model of knowledge representation covering the syntactical and semantical aspects completely.

(2) The rule for introducing subconcepts (cf. Figure 3.3) handles inheritance explicitly because the new subconcept inherits the roles of the superconcept. Using graph grammars, there is quite a different approach to inheritance. Since a graph $M$ with nonterminal or auxiliary labels can be interpreted as concept or type representation (cf. Figure 3.2), all terminal graphs derivable from it, i.e., the generated language $L(T, P, M)$ for some suitable set of concept- or type-explaining rules, may be seen as its meaning. If we consider now a graph $N$ derivable from $M$ as a kind of subconcept or subtype, we get obviously $L(T, P, N) \subseteq L(T, P, M)$ such that the meaningful objects of $N$ have all the properties of the meaningful objects of $M$. It seems worthwhile to investigate this kind of implicit inheritance.

(3) The theory of graph grammars provides some constructions that transform derivations into rules and make new rules out of given ones (like the construction of parallel rules in Section 2.9). If the grammar describes knowledge in the sense of Section 3.2, these constructions infer new rules from the given ones (with a known effect). In other words, we have got a kind of learning and it may be interesting to study how powerful it is with respect to the requirements of knowledge processing.

(4) As we have pointed out in Section 2.9(6), there are classes of graph grammars for which one can automatically check whether the generated graphs have certain properties like planarity, connectivity, acyclicity and many others. Are such classes of graph grammars interesting for semantic networks? Clearly, one would like to know such properties of semantic networks. Moreover, we would expect that hypergraph grammars rather than graph grammars are significant for semantic networks because they allow one to deal with $n$-adic relations rather than dyadic relations more easily. From a technical point of view, the difference between hypergraph grammars and graph grammars is small.

REFERENCES