Journal of Visual Languages and Computing (1997) 8, 27–55

Defining and Parsing Visual Languages with Layered Graph Grammars

J. REKERS* AND A. SCHÜRR[†]

**IBM Nederland N.V., Watsonweg 2, 1423 ND Vithoorn, The Netherlands, jan_rekers@nl.ibm.can.,*
[†]Lehrstuhl für Informatik III, RWTH Aachen, Ahornstr. 55, D-52074 Aachen, Germany, andy@i3.informatik.rwth-aachen.de

Received 27 January 1995; revised 3 January 1996; accepted 20 January 1996

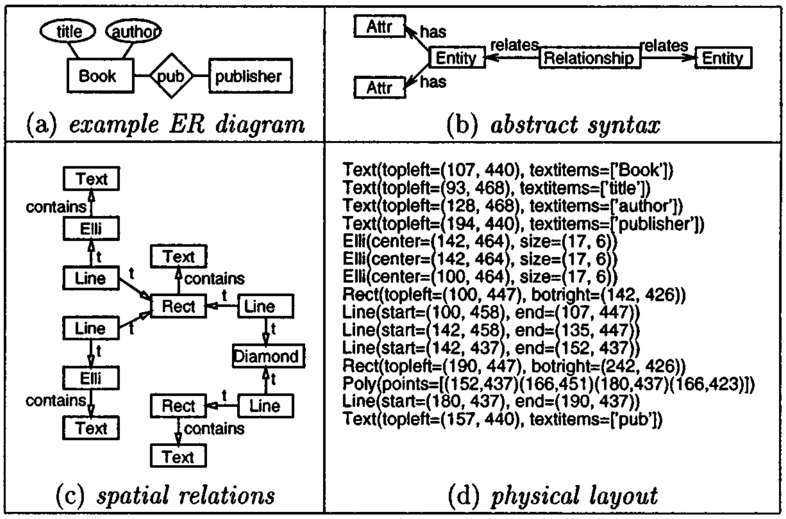
Sentences of Visual Languages (VLs) may often be regarded as assemblies of pictorial objects with spatial relationships like 'above' or 'contains' between them, i.e. their representations are a kind of directed graphs. Such a spatial relationship graph is often complemented by a more abstract graph, which provides information about the syntax (and the semantics) of the visual sentence in a more succinct form. As both representations are graphs, graph grammars are a natural means for defining the concrete and the abstract syntax of VLs. They can be used to generate syntax directed VL editors, which support 'free editing' and parsing of their underlying graph structures. Unfortunately, all efficiently working graph grammar parsing algorithms deal with restricted classes of context-free graph grammars only, while more general classes of graph grammars are necessary for defining many VLs. This motivated us to develop the notion of layered context-sensitive graph grammars, together with a bottom-up/top-down parsing algorithm.

1. Introduction

WHEN READING the visual language literature, or any book on software engineering, one cannot help but notice that a large variety of visual languages exists, of which only a few are equipped with a proper formal syntax definition. In this paper we will show how graph grammars can be used as syntax definition formalisms for graphical languages, and we will develop a graphical parsing algorithm based on these grammars. We start with the context of our work by discussing the internal representations necessary to support editing and execution of visual programs, by showing how graph grammars fit in, and by arguing why graph parsing would be useful for users of visual languages.

1.1. The Internal Representations of a Diagram

For an user of a visual language, the two most important aspects of a visual program are its physical layout (what the user sees and manipulates), and its meaning (what the user expresses with it). Any implementation of the language has to maintain the correspondence between these two aspects. We discuss this in more detail and will use Entity–Relationship (ER) diagrams as an example language to illustrate the proposed data structures.



```
Text(topleft=(107, 440), textitems=['Book'])
Text(topleft=(93, 468), textitems=['title'])
Text(topleft=(128, 468), textitems=['author'])
Text(topleft=(194, 440), textitems=['publisher'])
Elli(center=(142, 464), size=(17, 6))
Elli(center=(142, 464), size=(17, 6))
Elli(center=(100, 464), size=(17, 6))
Rect(topleft=(100, 447), botright=(142, 426))
Line(start=(100, 458), end=(107, 447))
Line(start=(142, 458), end=(135, 447))
Line(start=(142, 437), end=(152, 437))
Rect(topleft=(190, 447), botright=(242, 426))
Poly(points=[(152,437)(166,451)(180,437)(166,423)])
Line(start=(180, 437), end=(190, 437))
Text(topleft=(157, 440), textitems=['pub'])
```

Figure 1. The internal representation of a diagram

- The *abstract syntax graph* (ASG) representation of a diagram reflects the internal (logical) structure of the diagram according to its visual language. Nodes and edges in this graph representation correspond to language constructs, but do not determine how these constructs look. For instance, the graph of Figure 1(b) is an appropriate ASG for the ER diagram of Figure 1(a). This representation contains just everything necessary to *interpret* the diagram.
- However, the ER diagram has to be represented on a screen to the user, and in order to do so, a physical representation of the diagram must exist. The *physical layout* of a visual program is built up from some predefined graphical objects (such as *Line, Rect, Elli,* and *Text*) with properties (such as *location, size, color,* and *pen kind*). This level defines what the user sees and manipulates. For example, the physical layout of the ER diagram of Figure 1(a) is shown in Figure 1(d). This representation contains just everything necessary to *display* the diagram.
- In order to relate these two, quite different, representations of a diagram some intermediate data structure is necessary that represents the *pictorial structure* of the diagram. The *spatial relations graph* (SRG) abstracts from the physical layout: it contains all graphical objects, but instead of containing all individual properties, it represents the higher-level spatial relations which hold *between* its objects (*touch, contains, left of, labels*). Furthermore, it may use higher-level objects, such as *Diamond* instead of *Poly.* An example of an SRG is given in Figure 1(c). Note how much more detail this graph contains compared to the corresponding ASG.

The SRG and the ASG representations of a visual sentence are both graphs, but only very specific kinds of graphs represent actual visual sentences. A *graph grammar* can be used to define which graphs are correct, and specifies how such graphs may be constructed. For example, Section 1.2 presents such a graph grammar for connected ER abstract syntax graphs. Together with a grammar for the allowed spatial relations graphs it forms a complete syntax definition for the visual language of ER diagrams.

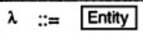
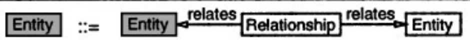
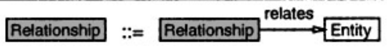
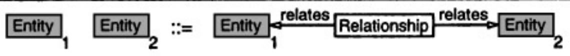
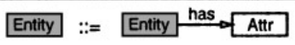
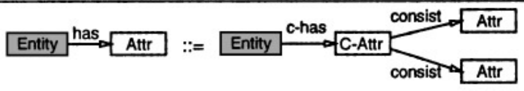
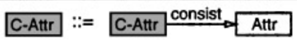
1	 The axiom production
2	 Introduction of an entity such that the diagram remains connected
3	 In order to allow for n-ary relationships
4	 Entity 1 Entity 2 ::= Entity 1 relates Relationship relates Entity 2 Introduction of a relationship between two already existing entities
5	 Entities may have attributes
6	 Entity has Attr ::= Entity c-has C-Attr consist Attr consist Attr Attributes may be composed of other attributes
7	 In order to allow for n-ary composition

Figure 2. The grammar for correct ER abstract syntax graphs

1.2. A Graph Grammar for ER Diagrams

Figure 2 presents the graph grammar of ER abstract syntax graphs in our formalism. This grammar takes care to accept connected ER diagrams only, it allows for *n*-ary relationships, and it allows for (composite) attributes on entities. In this formalism, every production is of the form '$L ::= R$', L and R being graphs which may have a common set of (grey) vertices and edges. This common part is the so-called *context* and states how the application of the production is to be embedded in the surrounding graph.

For example, the production which introduces composite attributes in ER diagrams is shown in the upper part of Figure 3. The bottom of Figure 3 shows an application of the production in the *generation* ($L \rightarrow R$) direction. This production may be applied if the host graph has a subgraph which matches the entire left-hand side '$\boxed{\text{Entity}}\xrightarrow{\text{has}}\boxed{\text{Attr}}$'. The application then deletes the portion '$\xrightarrow{\text{has}}\boxed{\text{Attr}}$' from the host graph, and adds the portion '$\xrightarrow{\text{c-has}}\boxed{\text{C-Attr}}\rightrightarrows\ldots$' to it. Next, the edge labeled 'c-has' is connected to the vertex in the host graph that matched the context element '$\boxed{\text{Entity}}$'. The reverse application of a production ($R \rightarrow L$), needed for *parsing* purposes, is defined analogously.

1.3. What is the Use of Parsing Visual Sentences?

Given a syntax definition of a visual language, one can easily imagine an editor which supports the creation of diagrams according to the syntax. However, without a parser, such an editor has to insist that every intermediate diagram is syntactically correct. We

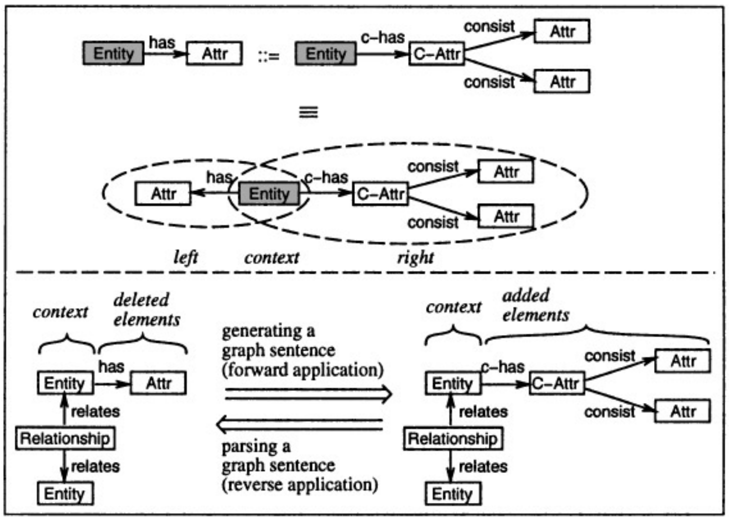


Figure 3. The application of a production

are convinced that this will lead to awkward user interfaces which will only be used if there are no other options available. This would be alleviated if the editor allows the user to enter a special *free-editing mode* in which he is free to insert, delete, change, or move any graphical object without any interference by the syntax directed nature of the editor. At the moment when the user indicates that he is done, the resulting diagram needs to be parsed in order to discover the new structure. This clearly indicates the need for graphical parsing, even in the context of syntax-directed graphical editors.

1.4. The Class of Graph Grammars Supported

This paper describes a graph grammar formalism and an associated graph parsing algorithm. Both have been developed with the application of graphical syntax definition and graphical parsing in mind, but will be useful in any context which requires the definition of graph languages and parsing of graphs. *Promising application fields*, where graph grammars were already used in the past, are modeling the development of plants, document image analysis, pattern recognition for 3D objects, music scores, and the like (cf. [2, 6, 8–10] for further details about graph grammar applications).

It has turned out that a restriction to context-free graph grammar productions (where every left-hand side consists of a single non-terminal node) makes it awkward to define the syntax of a large portion of visual languages and would be too restrictive. We therefore allow for *context-sensitive graph grammars* (in which both left- and right-hand side of a production are graphs), even though the performance penalty might be serious. However, as the parsing problem is intractable for general context-sensitive graph grammars, we restrict ourselves to *layered graph grammars* in which the left-hand side of every production must be lexicographically smaller than its right-hand side, and which thus avoids cyclic derivations.

2. Analysis of Visual Syntax Definition Formalisms

When inventing a new syntax definition and parsing approach for graphical languages, the most important thing is to come up with a reasonable solution for the so-called *embedding problem*. In Section 2.1 we compare our solution to the embedding problem with solutions which are part of other visual language definition formalisms. Section 2.2 then discusses the pros and cons of related approaches in more detail. It explains why we believed it to be necessary to develop yet another new formalism—layered graph grammars—together with a suitable parsing algorithm.

2.1. Embedding Problems of Graphical Languages

In the case of linear textual languages it is clear how to replace a nonterminal in a sentence by a corresponding sequence of (non-)terminals. But in the case of graphical languages, with many possible relationships between language elements, we need a far more complicated mechanism for (re-)establishing relationships between the surroundings of a replaced nonterminal and its replacing (non-)terminals.

In the previous section, we have extended the left- and right-hand sides of productions with *context elements* in order to be able to create edges between new vertices and preserved vertices in the rewritten host graph. This is just one popular solution of the embedding problem. The others are:

- *Implicit embedding:* Formalisms such as picture layout grammars [12, 13] or constraint multiset grammars [5, 4] do not distinguish between vertex and relationship objects. All necessary relationships between objects are implicitly defined as constraints over their attribute values. Therefore, attribute assignments within productions have the implicit side effect of creating new relationships to unknown context elements.
- *Embedding rules:* A quite different solution is an essential part of various forms of graph grammars like those presented by Kaul [15] and Rozenberg and Welzl [20]. These formalisms have separate embedding rules which allow for the redirection of arbitrary sets of relationships from a replaced nonterminal to its replacing (non-terminals.

All three embedding approaches have their specific advantages and disadvantages. The main drawbacks of the *implicit embedding* approach are: users are not always aware of the consequences of attribute assignments, and parsers have to spend a lot of time to extract implicitly defined knowledge about relationships from attributes and constraints. Furthermore, approaches based on this paradigm are usually not able to define productions which rewrite a vertex object and change (relabel) its relationships at the same time, such as production 6 of Figure 2 which relabels a *has* edge.

The *context* approach is, in our opinion, the most readable one, but the unrestricted use of context elements requires a quite complex parsing algorithm as we will see later. Furthermore, it is difficult in this setting to rewrite nonterminals which may participate in a statically unknown number of relationships.

In the latter case, the *embedding rule* approach is the most convenient one. But embedding rules are difficult to understand and all known parsing algorithms for productions with embedding rules are either hopelessly inefficient or impose too hard restrictions on left- and right-hand sides of productions. Furthermore, embedding rules are only able to redirect or relabel already existing relationships. Therefore, they do not allow for the definition of a production, such as production 4 of the ER grammar, which establishes new relations between previously unconnected vertex objects.

2.2. Related Work

Summarizing the explanations above, related parsing approaches should be studied and compared by answering the following questions (cf. Table 1):

- Is the *left-hand side* of a production restricted to a *single nonterminal*, which will be replaced by its right-hand side (context-free production)?
- Are there any *restrictions* for the *right-hand side* of a production?
- Does the formalism allow for references to additional *context elements*, which have to be present but remain unmodified during the application of a production?
- Does the proposed type of grammar have more or less complex *embedding rules*, which establish connections between new elements (created by a production) and the surrounding structure?
- Are there *additional restrictions* for the set of productions or the form of graphs, which do not fall in the above-mentioned categories?
- Is the *time and space complexity* of the proposed algorithm linear, polynomial, or even exponential with respect to the size of an input graph?

The *precedence graph grammar* parser of Kaul is an attempt to generalize the idea of operator precedence-based parsing. It has a linear time and space complexity. The parsing process is a kind of handle rewriting, where graph handles (subgraphs of the input graph) are identified by analysing vertex and edge labels of their direct context. Unfortunately, this approach works only for a very restricted class of graph languages.

The next three entries in the table contain references to Earley-style parsing approaches. The first one by Bunke and Haller [3] uses *plex grammars*, which are a kind of context-free graph grammars with rather restricted forms of embedding rules. Any nonterminal has only a fixed number of connection points to its context. The second one by Wittenburg [25] uses dotted rules to organize the parsing process for *relational grammars*, but without presenting any heuristics how to select 'good' dotted rules. Furthermore, it is restricted to the case of relational structures, where relationships of the same type define partial orders.

Finally, the approach of Ferrucci *et al.* [11] with so-called *1NS-RG grammars* is a translation of the graph grammar approach of Rozenberg and Welzl [20] into the terminology of relational grammars. In this approach, right-hand sides of productions may not contain nonterminals as neighbors, thereby guaranteeing local confluence of graph rewriting (parsing) steps. Furthermore, polynomial complexity is guaranteed as long as generated graphs are connected and an upper boundary for the number of adjacent edges at a single vertex is known in advance.

All approaches presented up to now are not adequate for generating abstract syntax graphs for connected ER diagrams. Their embedding rules are not able to connect previously unconnected *Entity* vertices by means of new *Relationships* vertices as we do in production 4 of Figure 2, and even the remaining two approaches of Marriott and Golin would find it difficult to define our language of connected ER diagrams. Their

Table 1. A comparison between various graph parsing algorithm

	Left-hand side	Right-hand side	Context	Embedding rules	Additional restrictions	Complexity
Kaul [15]	Nonterminal	Graph	No	Yes	Implicitly def. vertex order	Linear
Bunke and Haller [3]	Nonterminal	Plex structure	No	Fixed no. of connections	No	Exponential
Wittenburg [25]	Nonterminal	Relational structure	No	Yes	Explicitly def. vertex order	Exponential
Ferrucci *et al.* [11]	Nonterminal	Relational structure; no nonterminal neighbors	No	Yes	Bounded degree	Exponential
Rozenberg and Welzl [20]	Nonterminal	Graph; no nonterminal neighbors	No	Yes	Bounded degree	Exponential
Marriott [5, 4]	Nonterminal	Multiset	Yes	Implicit	Deterministic	Polynomial
Golin [12, 13]	Nonterminal	Maximal two (non-)terminals	1 terminal	Implicit	Finite set of attribute values	Polynomial
Rekers and Schürr [18]	Graph	Graph	Graph	No	Layering	Exponential

parsing algorithms generalize the bottom-up algorithms of Tomita or Cocke-Younger-Kasami for context-free textual grammars.

Marriott's *constraint multiset grammar* formalism [5, 4] offers the concept of context elements, but the parsing algorithm presented by Chok and Marriott [5] does not check whether the resulting derivation is consistent. This means that a syntax specification has to be made *deterministic* by additional 'not exists' constraints which prevent any possible overlap between the right-hand sides of productions.

The *picture layout grammar* approach of Golin [12, 13] allows for terminal context elements, but has the main focus on productions with one nonterminal on the left-and side and at most two (non-)terminals on the right-hand side, with predefined spatial relationships between them. A definition of a grammar which generates our language of connected ER diagrams should be feasible, but would be quite unreadable.

To summarize, all the presented parsing approaches have some difficulties with the definition of connected ER diagrams and they are certainly unable to deal with the running example of the following section, well-structured process flow diagrams, in a proper way. There is a strong need for a new syntax definition and parsing approach, where both *left- and right-hand sides* of productions are *arbitrary graphs* which share a common context graph. Such a formalism together with its parsing algorithm will be presented in the sequel.

3. Introduction to Graph Grammars

The subject of graph grammars was inaugurated 25 years ago with two papers on so-called 'web grammars' [17] and 'Chomsky systems for partial orders' [22]. Nowadays, a surprisingly large variety of graph grammar formalisms exists (cf. graph grammar proceeding volumes [6, 8–10]). Almost all of them belong to one of the following two families:

- The *algebraic family* of graph grammars, which adheres to the explicit embedding approach by means of context elements and uses category theory as its foundation [7].
- The *algorithmic family* of graph grammars, which uses powerful embedding rules instead of context elements and has set theory as its underlying formalism [16].

Approaches which support *parsing of graphs* now belong mainly to the algorithmic branch of context-free graph grammars. But we have already seen that graph grammars that support embedding by means of context elements and which allow for almost arbitrary left- and right-hand sides would be a very convenient tool for the *definition of visual languages*. It is, therefore, the goal of our research and this paper to define a class of graph grammars which has the main properties of algebraic graph grammar approaches—although we will not use category theory within definitions—but is still (efficiently) parsable.

The rest of this section is organized as follows. Section 3.1 introduces a more elaborate example which will be used in the sequel. Section 3.2 then provides the reader with a formal definition of graphs, graph grammar productions, and mappings between graphs. These mappings will be used to define the relationship between the left-hand side of a production and the rewritten subgraph in a given host graph, as well as between the right-hand side of a production and the generated subgraph in the resulting host graph. Next, we will introduce a rather general form of graph grammars and their generated languages in Section 3.3. Finally, Section 3.4 defines the class of *layered graph grammars* our parsing algorithm is able to deal with.

3.1. The Running Example

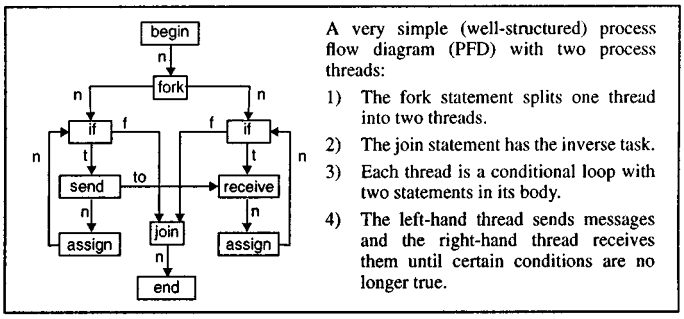
ER diagrams as they were introduced in Section 1 are well-known and therefore adequate for introductory discussions. Unfortunately, they have a rather straightforward syntax definition. They are not well-suited for a more detailed explanation of layered graph grammars and graph grammar parsing throughout the rest of this paper. As a consequence, we have to exchange the running example and to use a new visual language of *process flow diagrams* (PFD). It is a hybrid of well-formed control flow diagrams and Message Sequence Chart diagrams [14]. It inherits from both types of diagrams the property of having linear sequences of statements (actions). Furthermore, its control structures (if and while) are borrowed from well-structured control flow diagrams. Last but not least, it allows for the definition of multiple control flow threads which exchange asynchronous messages in the same way as Message Sequence Charts do. Figure 4 shows an example of a PFD graph which contains all the above-mentioned elements.

Figure 5 contains the graph grammar for these graph instances. Production 1 of this grammar replaces its axiom, a PFD, by two terminal vertices and one nonterminal vertex, which are connected by means of two *n(ext)* edges. Production 2 deletes a single *Stat* vertex and creates a new *assign* vertex, which inherits an incoming and an outgoing control flow edge from the deleted nonterminal vertex. Two grey context vertices are used for this purpose. They have to be present when the production is applied, but remain unmodified. The left context vertex is one of *begin*, *fork*, *if* or *Stat* and either the source of a *n(ext)* or a *t(rue)* or a *f(alse)* edge. The right context vertex has either the label *end* or *assign* or … and is the target of a *n(ext)* edge. Separately defined *label wildcards* are used to construct a single production for all the above-mentioned combinations of possible vertex and edge labels.

The other productions have a similar outline: they extend *Stat* lists, create new process threads with a *fork* operation at the beginning and a *join* operation at the end, establish communication channels between them, and produce conditional loops as well as branches. Note that such a small grammar already contains reasonable examples of productions which do not delete any nonterminals (production 3 and 4b), replace more than one nonterminal at the same time (production 5), and relabel embedding edges between created vertices and preserved vertices [production 7 replaces a *n(ext)* edge to the *T* ? vertex by a *f(alse)* edge to the same vertex].

3.2. Graphs, Graph Morphisms, and Productions

It is now time to provide the reader with a formal definition of graphs, productions, and matches of (left-hand sides of) productions in graphs. We will see that the definition of matches, the so-called *redexes*, is not as straightfoward as it might seem at a first glance. Therefore, we will use our new running example to explain the intricacies of finding redexes and the resulting formal definition of a redex.



A very simple (well-structured) process flow diagram (PFD) with two process threads:

1) The fork statement splits one thread into two threads.
2) The join statement has the inverse task.
3) Each thread is a conditional loop with two statements in its body.
4) The left-hand thread sends messages and the right-hand thread receives them until certain conditions are no longer true.

Figure 4. An abstract syntax graph for a process flow diagram

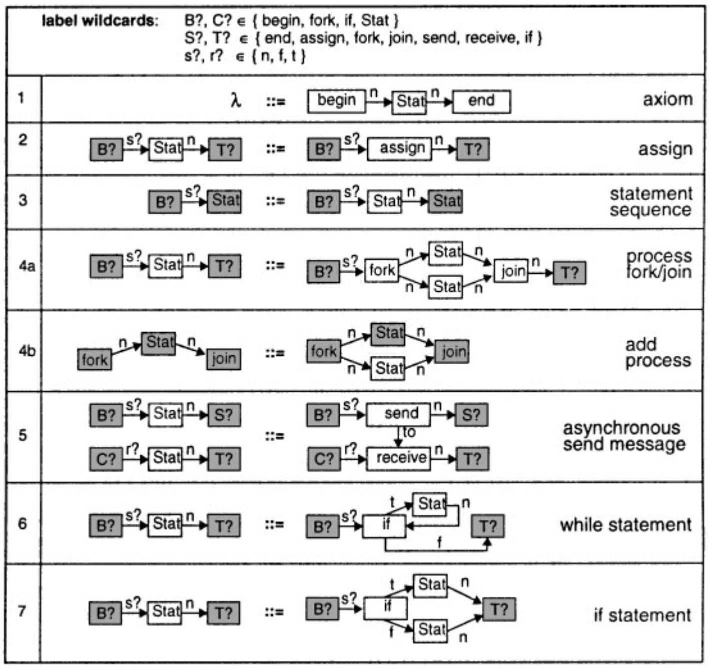

label wildcards: B?, C? ∈ { begin, fork, if, Stat }
S?, T? ∈ { end, assign, fork, join, send, receive, if }
s?, r? ∈ { n, f, t }
1 λ ::= begin Stat end axiom
2 ::= assign
3 ::= statement sequence
4a ::= process fork/join
4b ::= add process
5 ::= asynchronous send message
6 ::= while statement
7 ::= if statement

Figure 5. A graph grammar for process flow diagrams

Definition 3.1. $G := (V, E, l_V, l_E, s, t)$ is a *graph* over two given label sets L_V, L_E with:

- $V(G) := V$ and $E(G) := E$ are finite sets of vertices and edges,
- $l_V(G): V \to L_V$ and $l_E(G): E \to L_E$ are their labeling functions,
- $s(G): E \to V$ and $t(G): E \to V$ assign each edge its source and target,
- $l(G) := l_V(V) \cup l_E(E)$ will be used as an abbreviation for the set of all vertex and edge labels in a graph G.

Furthermore, we omit the suffix 'V' or 'E' of labeling functions whenever it is clear from context, and we will use $x \in G$ as an abbreviation for $x \in V(G) \vee x \in E(G)$.

Note that the presented graph data model is too simple to be useful in practice. Necessary extensions concern the introduction of *node and edge attributes*, as well as the definition of *label wildcards*. They are omitted here in order to keep formal definitions and algorithms as simple as possible. But these extensions were already used within the preceding examples. The ER abstract syntax graphs of Section 1 would need attributes to store the names of various constructs, and the PFD grammar of Section 3.1 already uses label wildcards. For further details concerning label wildcards (hierarchies) the reader is referred to another paper by Rekers and Schürr [18]. For the purpose of this paper we will simply assume that a single production with wildcards is replaced by an appropriate set of productions without wildcards. The introduction of attributes is the subject of ongoing implementation activities.

The following definition formalizes the notion of productions with context elements. These context elements are modelled as common subgraphs of the left-hand and the right-hand sides of productions.

Definition 3.2. A (graph grammar) *production* $p := (L, R)$ is a tuple of graphs over the same alphabets of vertex and edge labels L_V and L_E. Its left-hand side $lhs(p) := L$ and its right-hand side $rhs(p) := R$ may have a common (context) subgraph K if the following restrictions are fulfilled:

- $\forall e \in E(K) \Rightarrow s(e) \in V(K) \wedge t(e) \in V(K)$ with $E(K) := E(L) \cap E(R)$ and $V(K) := V(L) \cap V(R)$, i.e. sources and targets of common edges are common vertices of L and R, too.
- $\forall x \in L \cap R \Rightarrow l(L)(x) = l(R)(x)$, i.e. common elements of L and R do not differ with respect to their labels in L and R.

In the sequel, we will often use the abbreviations $Xlhs(p)$ and $Xrhs(p)$ for all graph elements of $lhs(p)$ and $rhs(p)$, respectively, which are not elements of their common subgraph $common(p) = K$.

For the definition of the application of a graph grammar production p to a given graph G, a precise definition of the *match of the left-hand side* of p in a given host graph G is necessary. Such a match, in the sequel termed *redex*, is a special case of a morphism (mapping) between two graphs over the same alphabets of vertex and edge labels L_V and L_E.

Definition 3.3. A pair of functions $h := (h_V, h_E)$ is a *graph morphism* $h: G \rightarrow G'$ from graph G to graph G' with $G := (V, E, l_V, l_E, s, t)$ and $G' := (V', E', l'_V, l'_E, s', t')$ iff:

- $h_V : V \rightarrow V'$ and $h_E : E \rightarrow E'$ are total mappings,
- $\forall v \in V : l'_V(h_V(v)) = l_V(v) \wedge \forall e \in E : l'_E(h_E(e)) = l_E(e)$,
- $\forall e \in E : s'(h_E(e)) = h_V(s(e)) \wedge \forall e \in E : t'(h_E(e)) = h_V(t(e))$.

In the sequel, we will often use $h(x)$ instead of $h_V(x)$ or $h_E(x)$, if the omitted subscript is clear from context.

Beside these definitions of graphs, productions, and graph morphisms, the usual definition of the image of a graph under a graph morphism as being a *subgraph* of that graph, as well as the operations $\cap$, $\cup$, and $\setminus$ for *intersection*, *union*, and *difference* of two graphs with a common subgraph will be used from now on to define the *application of*

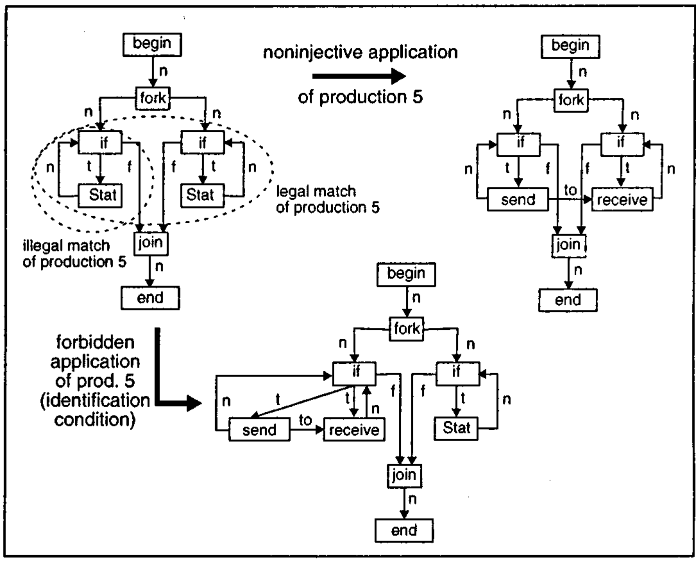


Figure 6. Noninjective matches and the identification condition

a graph grammar production. The most difficult point with such a definition is to decide which matches of a production's left-hand side are allowed and which are to be disallowed.

Consider, for instance, the production 5 of Figure 5 which creates a communication channel between two processes. It should be able to rewrite the top-left graph into the top-right graph of Figure 6, although some of its context vertices have to share their matches in the host graph. Therefore, *noninjective* graph morphisms are useful in practice. But on the other hand, it should not be possible to rewrite the top-left graph into the bottom graph of Figure 6. Therefore, we have to prohibit situations, where two *Xlhs* elements of a production match the same host graph element. This is the purpose of the so-called *identification condition* within Definition 3.4.

Another problem arises in the treatment of edges at vertices which have to be deleted. Consider, for instance, production 2 of the PFD grammar of Figure 5. It would be possible to apply the production to the left-hand graph of Figure 7. The question is now, what shall we do with the edge which has the deleted *Stat* node as source? It is not matched by the production's left-hand side and, therefore, not explicitly removed. One solution would be to remove this edge too. But this leads in almost all cases to unwanted results (cf. right-hand graph of Figure 7). Therefore, the so-called *dangling edge condition* of Definition 3.4 has been introduced. It prevents the application of a production under these circumstances.

Please note that the identification and the dangling edge condition together guarantee that the application of a graph grammar production is *reversible* [7]. This is a very important property of graph grammars which simplifies the development of a parsing algorithm considerably. It is sufficient to exchange the roles of left- and right-hand sides, i.e. to remove all host graph elements matched by *Xrhs* elements and to add for any *Xlhs*

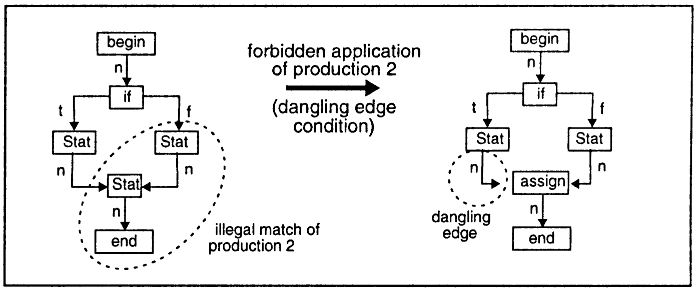

begin
n
if
t
f
Stat
Stat
n
n
Stat
n
end
forbidden application of production 2
(dangling edge condition)
illegal match of production 2
begin
n
if
t
f
Stat
Stat
n
assign
n
n
dangling edge
end

Figure 7. Deleting vertices and the dangling edge condition

element one copy to the host graph. The dangling edge condition ensures that we do not have to add edges which are not copies of *Xlhs* elements, and the identification condition ensures that we have indeed to add a separate copy for any *Xlhs* element.

Definition 3.4. A morphism $h := (h_V, h_E : L \to G)$ identifies a *redex* of L in G with respect to another graph R iff:

- Dangling edge condition:

$$\forall v \in V(L)\backslash V(R),\ e \in E(G):$$
$$(s(e) = h_V(v) \vee t(e) = h_V(v)) \Rightarrow \exists e' \in E(L)\backslash E(R): h_E(e') = e$$

- Identification condition:

$$\forall x \in L\backslash R,\ x' \in L: \quad h(x) = h(x') \to x = x'$$

Furthermore, a morphism is called a *potential redex* if the identification condition is fulfilled, but maybe not the dangling edge condition.

3.3. Graph Grammars and Their Languages

During parsing we have to search for redexes of right-hand sides of productions in a given input graph. Checking the identification condition is possible without taking other production applications into account. But the dangling edge condition needs knowledge about the existence of incident edges, i.e. we have to know which edges are already recognized (deleted) by inverse applications of other production instances. This leads to the distinction between redexes and potential redexes in Definition 3.4.

Our parsing algorithm, which we present in the next section, will be divided into two phases. The first phase has not enough knowledge for checking dangling edge conditions. It is only able to find a collection of potential (inverse) production applications, henceforth called *potential production instances.* The second phase is afterwards able to eliminate—among other things—those potential production instances violating the dangling edge condition, and it creates a subset of *production instances* which together generate the given input graph (if existent).

Definition 3.5. A *production instance* of a production $p := (L, R)$ is a tuple $pi := (p, h, h')$ such that $h : L \to G$ and $h' : R \to G'$ define the *application* of p to a graph G with result G', where:

- h is a redex of L in G with respect to R,
- h' is a redex of R in G' with respect to L,
- $h|_K = h'|_K$, with $K = common(p)$, and
- $G \backslash (h(L \backslash R)) = G' \backslash (h'(R \backslash L))$.

The application of a production p to a graph G with result G' will be denoted as $G \overset{p}{\Longrightarrow} G'$. A *potential production instance* is a production instance (p, h, h') for which h and h' are *potential* redexes.

Based on the definition of productions and production instances (applications), graph grammars and their languages are defined as follows:

Definition 3.6. A *graph grammar gg* is a tuple $(A, \mathscr{P})$, with A a nonempty initial graph (the axiom), and $\mathscr{P}$ a set of graph grammar productions. To simplify forthcoming definitions, the initial graph A will be treated as a special case of a production with an empty left-hand side λ. The set of all potential production instances of gg is abbreviated with $\mathscr{P}\mathscr{I}(gg)$.

Definition 3.7. Let $gg := (A, \mathscr{P})$ be a graph grammar. Its *language* $\mathscr{L}(gg)$ is defined as follows with G and G' being graphs:

$$G \in \mathscr{L}(gg) : \Leftrightarrow A \Rightarrow^* G$$

with

$$G \Rightarrow G' : \Leftrightarrow \exists p \in \mathscr{P} : G \overset{p}{\Rightarrow} G'$$

and $\Longrightarrow^*$ being the transitive, reflexive closure of $\Longrightarrow$.

3.4. Layered Graph Grammars and Their Languages

The above definitions of a graph grammar and its language are unusual with respect to vertex and edge labels. Until now, we have made no distinction between *terminal* and *nonterminal* labels, and, therefore, also no distinction between intermediate derivation results, i.e. sentential graph forms, and final results, i.e. elements of the generated language. The reason for this omission is that we need a more *fine-grained decomposition* of our label alphabets into a number of so-called *layers*, instead of the usual decomposition into two layers: a set of terminals and a set of nonterminals. Graph grammars with arbitrary graphs on left- and right-hand sides of productions are able to generate type 0 languages. But it is well-known that the membership problem is undecidable for type 0 languages in the general case [21]. Therefore, we have to impose additional restrictions onto graph grammars in order to be able to develop a graph parsing algorithm. This will be done by defining a kind of lexicographical order on graphs based on decomposition of label alphabets.

Definition 3.8. The decomposition $L_V \oplus L_E = L_0 \oplus \cdots \oplus L_n$ of the vertex and edge label alphabet into n subsets is a *layered label set* ($\oplus$ is the disjoint union of sets). We will use a function layer in the sequel which returns for any element of a given graph G the index of the layer to which its label belongs to, i.e.

$$\forall x \in G: \quad \mathit{layer}\ (x) = i : \Leftrightarrow l(x) \in L_i.$$

Definition 3.9. Given a decomposition $L_0 \oplus \cdots \oplus L_n$ of our label alphabet L_V and L_E, the language of a graph grammar *gg* may be decomposed into a number of *sublanguages* $\mathscr{L}_0(gg), \ldots, \mathscr{L}_n(gg)$, such that $\mathscr{L}_i(gg) := \{G \in \mathscr{L}(gg) \mid l(G) \subseteq \bigcup_{j \leq i} L_j\}$.

Using label layers, we are able to define a rather general class of *layered* graph grammars. For these graph grammars we will present an algorithm which solves the membership problem and returns for any input graph G either the answer 'no' or 'yes' together with one derivation, or all possible derivations of G.

Definition 3.10. A graph grammar $gg := (A, \mathscr{P})$ is called a *layered* graph grammar with respect to a global *layer* assignment $L_0, \ldots, L_n$ to its labels, if $\forall p := (L, R) \in \mathscr{P}$:

- R is a connected graph.
- The left-hand side L is non-empty.
- The right-hand side R without the common elements with L is non-empty.
- $L < R$ with respect to the following order for graphs:

$$G < G' : \Leftrightarrow \exists i : |G|_i < |G'|_i \wedge \forall j < i : |G|_j = |G'|_j$$

with $|G|_k$ defined as $|\{x \in G \mid \text{layer}(x) = k\}|$, i.e. the number of elements in G which have a label of layer L_k.

These additional restrictions guarantee a number of desirable properties which we will need later on for the development of our parsing algorithm:

- The connectedness of right-hand sides allows us to use linear search plans for pattern matching purposes which can be processed step by step just by traversing edges (see Definition 4.1).
- The nonemptiness of the left-hand side guarantees that each application of a production (see Definition 3.5) 'uses' graph elements that have been created by another application or that belong to the initial graph. This implies that the 'derivation history' of a graph is always a *connected* acyclic graph.
- The nonemptiness of $R \backslash L$ implies that we do not have to guess how often such a production has been applied in order to generate a certain graph.
- The layering condition above defines an ordering relation between vertex and edge labels which guarantees the termination of the parsing algorithm.

Note that it is not necessarily the task of a language designer to assign labels to layers. Such a *decomposition of label alphabets* can usually be computed automatically by applying the following default rule to any production p: $\mathit{layer}(x) \geq \mathit{layer}(y)$ for any $x \in Xlhs(p)$, and any $y \in Xrhs(p)$. This determines the assignment of labels to layers completely under the additional assumption that $\mathit{layer}(x) \geq \mathit{layer}(y) \Rightarrow \mathit{layer}(x) > \mathit{layer}(y)$ whenever possible. Applying these rules to the PFD graph grammar would result in a layer 1 with all edge labels and the node label *Stat*. All other node labels belong to layer 0.

The following theorem is now a direct consequence of the introduction of label layers in Definition 3.10.

Theorem 3.11. *The element problem for a layered graph grammar gg is decidable. A naive parsing algorithm, which applies productions with exchanged left- and right-hand sides as long as possible and backtracks when necessary, terminates always and produces the correct answer.*

Proof (*Sketch*). The identification and dangling edge condition guarantee that production applications are reversible by simply exchanging their left- and right-hand sides. The defined ordering of Definition 3.10 guarantees that any sequence of reverse production applications, which starts with a finite graph, has a finite length. Furthermore, graphs of finite size possess a finite number of potential redexes for a finite set of productions. Finally, we can compare all intermediate and final results of computed reverse derivation sequences with the grammar's axiom graph, i.e. the element problem for layered graph grammars is decidable (for further details cf. the paper by Rekers and Schürr [18]). □

4. The Parsing Algorithm

We have seen that the membership problem for layered graph grammars may be solved using a naive exhaustive search algorithm. The main problem with an exhaustive search is that it may recompute already found matches (subderivations) for a part A of the input again and again when it explores different parsing alternatives for an unrelated part B of the input. One way to improve this behavior is to replace depth-first exhaustive search by a kind of *breadth-first search algorithm* such that possible subderivations are constructed and extended in parallel instead of recomputing them multiple times. Filters have to be used to discard useless subderivations as soon as possible.

Here we will sketch the main ideas of such a parsing algorithm. Any details concerning filtering functions and correctness proofs may be found in the paper by Rekers and Schürr [18]. The parsing algorithm has to solve the following two tasks:

1. Finding matches of right-hand sides of productions and completing them to production instances (reverse production applications). This is an expensive process which works at graph element level.
2. Combining computed production instances with derivations. In the case of ambiguities, it might however happen that more than one derivation exists, or it might happen that a constructed production instance is not useful at all.

During the development of our parsing algorithm it became evident that dealing with these two tasks at the same time results in very complex algorithms. These algorithms would even perform a lot of work which turns out to be useless afterwards. Therefore, we decided to realize a two-phase parsing algorithm which is divided into a bottom-up phase and a top-down phase:

- The *bottom-up phase* searches the graph for matches of productions' right-hand sides. On the recognition of such a right-hand side, a production instance *pi* is created, and the noncontext elements of its left-hand side are added to the graph, but nothing is deleted from it. The bottom-up phase thus generates a *completion* $\bar{G}$ of the input graph G. The additions to the graph might in turn lead to the recognition of other right-hand sides. The result of the bottom-up phase is the collection *PPI* of all production instances discovered plus the completed graph $\bar{G}$ (cf. Section 4.1).

- The production instances created have *dependency* relations among each other, such as $above(pi_1, pi_2)$, which means that production instance pi_1 should occur before pi_2 in a derivation, or $excludes(pi_1, pi_2)$, which states that pi_1 and pi_2 may not occur in the same derivation. These relations can be computed during the bottom-up phase.
- The *top-down phase* composes a subset of *PPI* which creates the given graph. Such a derivation is developed in a pseudo-parallel fashion with a preference for depth-first development (cf. Section 4.3).

Our approach is to concentrate all work which deals with graph *elements* in the bottom-up phase. This phase is not bothered with backtracking, ambiguities, and alternative derivations; it just generates as many matches as possible. The top-down phase does not have to consider individual graph elements, but only deals with *dependencies* between entirely matched *production instances*, and combines constructed production instances into viable derivation sequences.

4.1. The Bottom-Up Phase

4.1.1. Search Plans and Dotted Rules

One of the most severe problems of any graph rewriting system or graph parsing algorithm is keeping track of all potential redexes of a given set of productions, and to incrementally construct them while the graph is modified. We apply a method which constructs a *linear search plan* for the right-hand side of every production. Such a search plan predetermines the order in which the redex must be constructed.

Definition 4.1. The right-hand side of a production $p := (L, R)$ can be linearized into a *search plan*, which is a sequence $[md_0, md_1, \ldots, md_n]$ of *pattern matching directives*. The first item of the sequence, md_0, has the form

- $\langle head(y : l)\rangle$: find a vertex with label l and call it y,

and each of the remaining items md_i, for $1 \leq i \leq n$, has one of the following forms:

- $\langle z : x \xrightarrow{k} (y : l)\rangle$: Start at an already known vertex x of R, follow an edge with label k to a target vertex with label l, and call the edge z and its target vertex y,
- $\langle z : x \;\xrightarrow{k} (y : l)\rangle$: Start at an already known vertex x of R, follow an edge with label k in inverse direction to a source vertex with label l, and call the edge z and its source vertex y, or
- $\langle z : x \;\xrightarrow{k} y\rangle$: Check the existence of an edge with label k between two already known vertices x and y of R, and call it z.

Furthermore, $left(md)$ returns the variable name x of a matching directive md; it is undefined for the head of a search plan.

The set of search plans for a production p is in general quite large, and it is rather difficult to find a 'best' search plan within this set. The quality of the choice depends to some extent on the expected number of vertices and edges with a certain label in the considered language of graphs. We will for now assume that a function $SP(p)$ selects at least a 'good' search plan. We refer to the paper by Rekers and Schürr [18] for a first attempt to define such a function based on estimated costs of search plans.

In constructing a match, the bottom-up phase moves a *dot* through the search plan: vertices and edges to the left of the dot are already matched, the ones to the right have still to be matched. It might happen that a searched-for edge is not present in the host graph. In that case the dotted rule is suspended, and will be awakened when a promising edge appears.

Definition 4.2. A tuple $dr := (p, M, i, h, s)$ is a *dotted rule* with respect to a given graph $\bar{G}$, which is an already constructed completion of an input graph G, if

- $p := (L, R)$ is a production of a layered graph grammar,
- $M := SP(p)$ is a sequence $[md_0, \ldots, md_n]$ of matching directives,
- i, with $1 \le i \le n$, is the position of the 'dot' in the dotted rule. The matching directives $md_0, \ldots, md_{i-1}$ are already fulfilled, the matching directives $md_i, \ldots, md_n$ still have to be fulfilled in the selected order of the search plan,
- $h : R \to \bar{G}$ is a partially recognized redex of R in $\bar{G}$ with respect to L, binding graph elements of R to already discovered graph elements in $\bar{G}$ as the result of processing matching directives $md_0, \ldots, md_{i-1}$, and
- s represents the state of a dotted rule, which can be active or suspended. If active, md_i still has to be checked against $\bar{G}$. If suspended, md_i can only be fulfilled when an appropriate edge is added to $\bar{G}$.

The parsing algorithm stores these dotted rule instances as attachments to vertices in $\bar{G}$. A dotted rule $(p, [md_0, \ldots, md_n], i, h, s)$ will be attached to the vertex $h(left(md_i))$ of $\bar{G}$, which is the already known vertex x of the next pattern matching directive md_i.

For example, a reasonable search plan for production 6 of the PFD grammar of Figure 5 would be

$$\begin{aligned}
M_6 = [\ & \langle head(V_1, \{if\})\rangle, \\
& \langle E_1 : V_1 \overset{\{t\}}{\rightarrow} V_2 : \{Stat\}\rangle, \\
& \langle E_2 : V_2 \overset{\{n\}}{\rightarrow} V_1\rangle, \\
& \langle E_3 : V_1 \overset{\{f\}}{\rightarrow} V_3 : \{end, assign, fork, join, send, receive, if\}\rangle, \\
& \langle E_4 : V_1 \overset{\{n,t,f\}}{\longleftarrow} V_4 : \{begin, fork, if, Stat\}\rangle\ ]
\end{aligned}$$

Now, we consider the graph of Figure 8 with the following dotted rule attached to *Stat* vertex 108:

$$(p_6, M_6, 2, \{(V_1, 105), (E_1, 107), (V_2, 108)\}, active)$$

At the moment this dotted rule is considered for proceeding the parser has to check whether the matching directive just after the dot $\langle E_2 : V_2 \overset{\{n\}}{\rightarrow} V_1\rangle$ can be fulfilled: it has to check whether the vertex matched by V_2 (vertex 108) has an outgoing edge labeled n to the vertex matched by V_1 (vertex 105). This is the case, so that the partial redex is extended with $(E_2, 109)$ and the following dotted rule is attached to vertex 105, as the next matching directive starts at V_1:

$$(p_6, M_6, 3, \{(V_1, 105), (E_1, 107), (V_2, 108), (E_2, 109)\}, active)$$

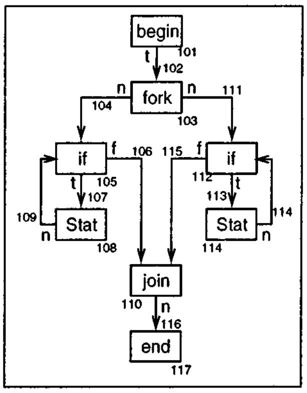


Figure 8. A sample graph for dotted rule processing

However, the parser cannot simply discard the older version of the dotted rule, as another outgoing edge labeled *n* might still appear in the future. We have to leave a suspended version of the dotted rule behind at vertex 108 in order to be able to process these forthcoming edges. It is possible to minimize the number of suspended dotted rules by taking cardinality constraints of edges into account (any *Stat* node is the source of one and only one *n* edge) and by using available layering information (see the remarks at the end of Section 4.1.2).

4.1.2. The Algorithms of the Bottom-Up Phase

The main idea of the bottom-up part of the parsing algorithm is as follows: it starts by attaching initial dotted rules to all matching vertices in the host graph. Next it repeatedly chooses an active dotted rule to advance. If a dot reaches the end of a search plan, the associated production has been recognized completely. That generates a production instance, and the host graph is extended with the elements in $L \setminus R$; new vertices may give rise to initial dotted rules; new edges may activate suspended dotted rules. This is repeated until there are no remaining active dotted rules.

Algorithm 1 (*Bottom-up Loop*). The bottom-up phase of our parsing algorithm extends matchings of right-hand sides of productions step-by-step by 'pumping' dotted rules through the graph. The main loop of the bottom-up parser starts with a call to create-initial-dotted-rules (Algorithm 1.1). Next, it repeatedly checks whether there are dotted rules which can extend their matches, and if so, calls routine proceed (Algorithm 1.2) with a discovered possible extension of an already known match. If this results in a completely recognized production, proceed extends the graph, calls create-initial-dotted-rules for all vertices created, and calls reactivate-dotted-rules (Algorithm 1.3) for all edges created.

```
function BottomUP-Loop(in $G$: graph): set of $PPI$ =
  $\bar{G} := G$; $PPI := \emptyset$
  for every vertex $v \in \bar{G}$ do
    create-initial-dotted-rules($\bar{G}$, $v$)
  od
  while $\exists v \in \bar{G}$ with $dr = (p, M, i, h, active)$ attached to $v$ do
    $M = [\ \dots, md_i, \dots\ ]$
    if $md_i$ is of the form $\langle z : x \xrightarrow{k} (y : l)$ then
      for every edge $e : v \xrightarrow{k'} v' \in \bar{G}$ do
        if $k = k' \wedge l = l(v')$ then
          $proceed(\bar{G}, (p, M, i, h \cup \{z \to e, y \to v'\}$, active))
        fi
      od
    else if $md_i$ is of the form $\langle z : x \xleftarrow{k} (y : l)\rangle$ then
      for every edge $e : v \xleftarrow{k'} v' \in G$ do
        if $k = k' \wedge l = l(v')$ then
          $proceed(\bar{G}, (p, M, i, h \cup \{z \to e, y \to v'\}$, active))
        fi
      od
    else if $md_i$ is of the form $\langle z : x \xrightarrow{k} y\rangle$ then
      for every edge $e : v \xrightarrow{k'} v' \in G$ do
        if $h(y) = v' \wedge k = k'$ then
          $proceed(\bar{G}, (p, M, i, h \cup \{z \to e\}$, active))
        fi
      od
    fi
  change the state of $dr$ from active to suspended
od
return $PPI$
```

Algorithm 1.1 (*Create initial dotted rules*). If a new vertex v is added to the graph, then an initial dotted rule is created for all productions which have a search plan with a matching head.

```
proc create-initial-dotted-rules(inout $\bar{G}$ : graph, in $v$ : vertex) =
  for every production $p : (L, R) \in gg$ with
                          search plan $M := [\langle \text{head}(y : l)\rangle, \dots\ ] = SP(p)$ do
    $h :=$ completely undefined (partial) morphism
    if $l = l(v)$ then
      attach $(p, M, 1, h \cup \{y \to v\}$, active) to $v$ in $\bar{G}$
    fi
  od
```

Algorithm 1.2 (*Proceed with a dotted rule*). Matching directive md_i has been fulfilled. If md_i is not the last one of the matching directives, a new dotted rule is attached to the vertex from which matching directive md_{i+1} has to proceed. Otherwise, its production p has been recognized completely, in which case the left-hand side must be added to the graph and can be processed further on.

```
proc proceed(inout $\bar{G}$ : graph, in $(p, M, i, h', s)$) : dotted rule =
  $M = [md_0, \dots, md_n]$
  if $\exists var \to x, var' \to x \in h : var \neq var' \wedge var \in Xrhs(p)$ then
    return                                    violates identification condition
  fi
```

```
if $i < n$ then
  attach $(p, M, i+1, h', s)$ to $h'(left(md_{i+1}))$
else
  construct a morphism $h$ from $h'$ respecting the conditions of Definition 3.5
  if not inconsistent( $(p, h, h')$ ) then
    $PPI := PPI \cup \{(p, h, h')\}$
    $\bar{G} := \bar{G} \cup h(Xlhs(p))$
    for every vertex $v \in h(V(Xrhs(p)))$ do
      create-initial-dotted-rules($\bar{G}$, $v$)
    od
    for every edge $e \in h(E(Xrhs(p)))$ do
      reactivate-dotted-rules($\bar{G}$, $e$)
    od
  fi
fi
```

Function inconsistent(p, h, h') checks whether the to-be created production instance relies on production instances that exclude each other. In that case it can be discarded right-away. See [18] for a more in-depth explanation.

Algorithm 1.3 (*Reactivate suspended dotted rules*). If we add a new edge e from v to v' to the graph, then it might be the case that there are suspended dotted rules attached to v or v' which can proceed their pattern matching process with this edge. These suspended rules need to be re-activated. However, care should be taken that only new edges are considered and not already traversed edges.

```
proc reactivate-dotted-rules(inout $\bar{G}$: graph, in $e$: edge) =
  $v := s(e)$
  $v' := t(e)$
  for every $dr := (p, M, i, h, suspended)$ attached to $v$ do
    $M = [\ \ldots, md_i, \ldots\ ]$
    if $md_i$ is of the form $\langle z : x \xrightarrow{k} (y : l)\rangle \wedge k = l(e) \wedge l = l(v')$ then
      $proceed(\bar{G}, (p, M, i, h \cup \{z \rightarrow e, y \rightarrow v'\}, active))$
    else if $md_i$ is of the form $\langle z : x \xrightarrow{k} y\rangle \wedge v' = h(y) \wedge k = l(e)$ then
      $proceed(\bar{G}, (p, M, i, h \cup \{z \rightarrow e\}, active))$
    fi
  od
  for every $dr := (p, M, i, h, suspended)$ attached to $v'$ do
    $M = [\ \ldots, md_i, \ldots\ ]$
    if $md_i$ is of the form $\langle z : x \xleftarrow{k} (y : l)\rangle \wedge k = l(e) \wedge l = l(v')$ then
      $proceed(\bar{G}, (p, M, i, h \cup \{z \rightarrow e, y \rightarrow v\}, active))$
    fi
  od
```

In the main loop of the bottom-up phase the next dotted rule to be processed is selected at random. However, the layering can also be used to process active dotted rules in such a way that productions which generate graph elements of lower layers are given priority. This means that the layers of the elements that are added to the graph will be increasing. This implies again that dotted rules which are waiting for an element of a lower layer can safely be discarded. In practice, this measure avoids almost all suspended dotted rules; see the work of Rekers and Schürr [18] for a more in-depth discussion of the layering condition and its consequences.

4.1.3. *Example of the Bottom-Up Phase*

We will parse the process-flow diagram of Figure 9(a) according to the PFD grammar of Figure 5. In this example we will not go into the details of moving dots through dotted rules, but will only explain which production instances are generated, and how they extend the input graph G step by step to $\bar{G}$.

In the graph of Figure 9(a), the right-hand side of production 2 (assign statement) matches the *assign* vertex 105 and its context. The application of this production adds *Stat* vertex 114 and its edges to the graph (see Figure 9(b)), and creates production instance pi_1. The RHS of pi_1 matches the graph elements $\{103, 104, 105, 108, 110\}$ and its LHS matches $\{103, 113, 114, 115, 110\}$. This leads to the assignment of $h(Xlhs)$, $h(common)$ and $h(Xrhs)$ as indicated in the first row of the table of Figure 10. Production 2 can be recognized for a second time in the lower *assign* vertex 107, which leads to pi_2 of Figure 10, and extends $\bar{G}$ to the graph depicted in Figure 9(b).

In the extended graph, the right-hand side of production 4a (the fork / join statement) can be recognized, which leads to production instance pi_3 and the graph of Figure 9(c). However, in this graph the right-hand side of two instances of production 4b (add process to fork / join) can be recognized, too. This leads to pi_4 and pi_5 of Figure 10. It is up to the top-down phase to recognize that these possible production instances do not fit into any derivation. Finally, the right-hand side of the axiom production 1 finds a match in the graph of Figure 9(c), which creates production instance pi_6. That completes the work of the bottom-up phase, and the resulting production instances of Figure 10 will be shipped to the top-down phase for further processing.

4.2. Dependencies Between Production Instances

A production instance represents the application of a production to some version of the graph, and it indicates the graph elements matched by both sides of the production. By operating on graph elements, production instances depend on each other. In order to be able to reason about these dependencies, we have introduced the dependency relations *above*, *consequence*, *excludes*, and *excludes**.

Definition 4.3. A production instance $pi_2 = (p_2, h_2, h_2')$ is a *consequence* of another production instance $pi_1 = (p_1, h_1, h_1')$, or $pi_2 \in \text{consequence}(pi_1)$, if the execution of pi_1 must be followed by the execution of pi_2, i.e. $pi_2 \neq pi_1$ and:

- $h_1'(Xrhs(p_1)) \cap h_2(Xlhs(p_2)) \neq \emptyset \vee$
  (pi_1 creates a graph element which is deleted by pi_2)
- $h_1(common(p_1)) \cap h_2(Xlhs(p_2)) \neq \emptyset$
  (pi_1 needs a context element which is deleted by pi_2).

The transitive, reflexive closure of consequence is *consequence**.

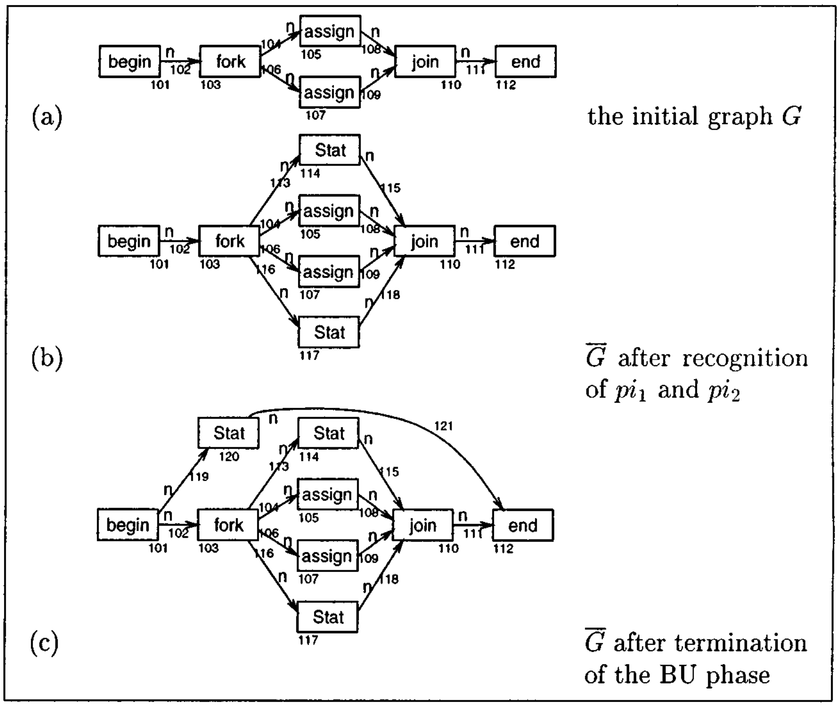

(a)
the initial graph G
(b)
$\overline{G}$ after recognition of pi_1 and pi_2
(c)
$\overline{G}$ after termination of the BU phase
begin
fork
assign
join
end
Stat

Figure 9. Some intermediate graphs $\bar{G}$ of the BU phase

	prod	$h(Xlhs)$	$h(common)$	$h(Xrhs)$
pi_1	2	113 114 115	103 110	104 105 108
pi_2	2	116 117 118	103 110	106 107 109
pi_3	4a	119 120 121	101 112	103 113 114 115 110 116 117 118
pi_4	4b	$\emptyset$	103 113 114 115 110	116 117 118
pi_5	4b	$\emptyset$	103 116 117 118 110	113 114 115
pi_6	1	$\emptyset$	$\emptyset$	101 119 120 121 112

Figure 10. The production instances created by the example of Figure 9

Definition 4.4. A production instance $pi_1 = (p_1, h_1, h_1')$ is *above* another production instance $pi_2 = (p_2, h_2, h_2')$, if pi_1 must be executed before pi_2, i.e. $pi_1 \neq pi_2$ and:

- $pi_2 \in \text{consequence}(pi_1) \vee$
- $h_1'(Xrhs(p_1)) \cap h_2(\text{common}(p_2)) \neq \emptyset \vee$
  (pi_1 creates an element which pi_2 needs as context element).
- $\exists e \in h_1(E(Xlhs(p_1))), \exists v \in h_2(V(Xlhs(p_2))) : s(e) = v \vee t(e) = v.$
  (pi_2 deletes a vertex with an incident edge which is removed by pi_1. In that case pi_1 needs to be applied first in order to avoid dangling edges.)

To summarize, $pi_2 \in consequence(pi_1)$ means that the application of pi_1 must be followed by the application of pi_2. This is a consequence of the fact that the bottom-up phase of our parsing algorithm will guarantee that Xlhs-elements of production instances are never Xlhs-elements of other production instances. Therefore, any intermediate graph element, which is not part of the finally generated graph, must be deleted by applying a uniquely defined production instance. The dependency pi_1 *above* pi_2 is weaker in that it states that if both pi_1 and pi_2 are applied, then pi_1 must be applied earlier than pi_2.

Definition 4.5. A production instance $pi_1 = (p_1, h_1, h'_1)$ *excludes* another production instance $pi_2 = (p_2, h_2, h'_2)$ (and vice versa) if both production instances depend on each other or if they add the same elements to a graph (cover the same elements), i.e. $pi_1 \neq pi_2$ and:

$$pi_1 \text{ excludes } pi_2 : \Leftrightarrow (pi_1 \text{ above } pi_2 \wedge pi_2 \text{ above } pi_1) \\ \vee\ h'_1(Xrhs(p_1)) \cap h'_2(Xrhs(p_2)) \neq \emptyset.$$

The definition of excludes can be generalized to that of *excludes**:

$$pi_1 \text{ excludes* } pi_2 : \Leftrightarrow \exists pi'_1 \in \text{consequence*}(pi_1), \\ pi'_2 \in \text{consequence*}(pi_2) : pi'_1 \text{ excludes } pi'_2.$$

The intuition behind this definition is the following: if pi_1 *excludes* pi_2, then the choice to use pi_1 inhibits the use of pi_2. However, selecting pi_1 might not be a choice, but a necessary *consequence* of an earlier selected production instance pi (if pi creates an intermediate ‘nonterminal’ graph element which must be removed by applying pi_1 afterwards). This leads to the definition of *exclude**, which makes the ‘real’ choice points explicit in the top-down algorithm of the next section.

Our dependency relations between production instances are far more complex to compute and use than the *cover set* approach as used in PLG [12] and CMG [4] parsing. In this approach every symbol *covers* part of the input. The *Xlhs* symbol of a production covers the union of the covers of the *Xrhs* symbols, two *Xrhs* symbols may not have an overlap in their covers, and the start symbol should cover the entire input. However, this quite straightforward approach can only be used for context-free productions, and breaks if a production may have an empty *Xlhs* (where does the union of cover sets go?), or if it may have several *Xlhs* symbols (if these come together again their cover sets do not conflict). For such grammars a history mechanism between production instances is inevitable.

4.2.1. Example of Dependency Relations

If we take the possible production instances of Figure 10 and compute which dependency relations hold between them according to the above definitions, we obtain the relations as depicted in Figure 11. As an example of such a computation, $pi_1 = (p_1, h_1, h'_1)$ is a consequence of $pi_3 = (p_3, h_3, h'_3)$, or $pi_1 \in consequence(pi_3)$, as

$$h'_3(Xrhs(p_3)) \cap h_1(Xlhs(p_1)) \\ = \{103, 113, 114, 115, 110, 116, 117, 118\} \cap \{113, 114, 115\} \\ = \{113, 114, 115\} \neq \emptyset.$$

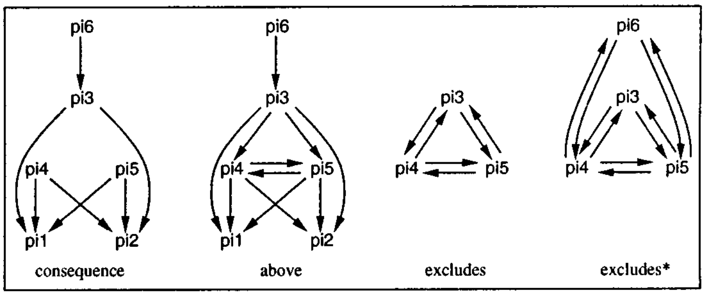


Figure 11. Dependency relations between PPIs of Figure 10

4.3. The Top-Down Phase

The top-down phase of the parsing algorithm receives the entire collection of possible production instances *PPI* from the bottom-up phase, and extracts a subset which would create the given input graph. In the case that several such subsets exists, it selects the first one encountered. The top-down phase maintains several *partial derivations* in parallel. It returns finally a sequence of production instances which generates the given input graph (if existent).

Definition 4.6. A tuple (G_c, API_c, EPI_c) is a *partial derivation* for G in the context of all possible production instances *PPI*. G_c is the graph as built till now by the applied production instances in API_c. The history of the derivation and the production instances in API_c might exclude certain production instances, which are kept in EPI_c. The sets API_c and EPI_c are both subsets of the original collection of potential production instances *PPI*. We will also refer to PPI_c as an abbreviation for $PPI \setminus (API_c \cup EPI_c)$, the production instances which can still be applied.

The main idea behind the top-down phase is as follows: it starts with a production instance for the axiom production and extends this set without violating the *above* restriction. Whenever a production instance is encountered which *excludes* other production instances, this marks a choice point in the algorithm. This means that the derivation on hand splits into two derivations, one for each possibility. These derivations are developed in a pseudo-parallel fashion with a preference for depth-first development.

Algorithm 2 (*Top-down loop*). The top-down algorithm keeps its collection of active partial derivations in a stack, as this facilitates the pursuit of derivations in a depth first manner. A production instance may be applied in a derivation if its lhs is present in G_c, the application of it does not introduce dangling edges, and if it is not yet excluded by already applied production instances. We use the dependency relations between production instances to determine the candidate production instances which fulfill all of these requirements.

If a to be applied production instance *pi* has an *excludes** relation with any of the not yet applied production instances, the application of *pi* indicates a choice point in the algorithm. Therefore, we push two derivations on the stack of derivations: first one in which *pi* is simply excluded, next one in which *pi* is applied. This allows us to continue with the alternative derivation(s) if the choice turns out to be wrong.

The algorithm uses *Cleanup* (not described here, see [18]) to get rid of production instances which are useless as early as possible. Furthermore, it uses *Apply* (Algorithm 2.1) to compute the effects of a selected production instance on the current derivation.

```
function TopDown-Loop(in $G$: graph, in $PPI$: set of $PPI$): set of $PPI$ =
  $D := emptystack$
  for every $pi := ((\emptyset, A), h, h') \in PPI$ do                 the axiom production
    $D := push(Apply(pi, (\emptyset, \emptyset, \emptyset)), D)$
  od
  while $\neg empty(D)$ do
    $d := (G_c, API_c, EPI_c) = top(D); D := pop(D)$
    $d := cleanup(d)$
    $Candidates := \{ pi \in PPI_c |$
        $\exists pi' \in API_c$: $pi'$ above $pi \wedge$
        $\forall pi'' \in PPI$: $pi''$ above $pi \rightarrow (pi'' \in API_c \vee pi'' \in EPI_c)\}$
    if $Candidates = \emptyset \wedge G_c = G$ then
      return $API_c$                                                 successful derivation
    else if $Candidates = \emptyset$ then
      do nothing                                                     dead-end derivation
    else if $\exists pi \in Candidates: \neg pi' \in PPI_c: pi$ excludes* $pi'$ then
      $D := push(Apply(pi, d), D)$                                   simple step
    else
      select some production instance $pi$ from Candidates
      $D := push((G_c, API_c, EPI_c \cup \{pi\}), D)$                choice point
      $D := push(Apply(pi, d), D)$
    fi
  od
return $\emptyset$                                                   no successful derivation found
```

Algorithm 2.1 (*Apply*). Returns a derivation which is the incoming derivation *d* extended with an application of production instance *pi*.

```
function $apply$(in $pi = (p, h, h')$: $PPI$,
                 in $d = (G_c, API_c, EPI_c)$ ): derivation =
  $G_n := (G_c \backslash h(Xlhs(p))) \cup h'(Xrhs(p))$
  $API_n := API_c \cup \{ pi \}$
  $EPI_n := EPI_c \cup \{ pi' \in PPI | pi$ excludes* $pi' \}$
  return $(G_n, API_n, EPI_n)$
```

4.3.1. Example of the Top-Down Phase

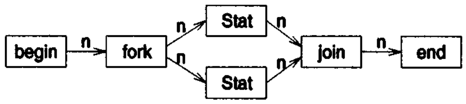
Given the simplicity of the running example, the working of the top-down phase on the possible production instances of Figure 9 is also pretty straightforward:

1. The algorithm starts with an initial derivation tuple created for the production instance of the axiom production, which is pi_6. According to the dependency relations shown in Figure 11, pi_6 has *excludes** relations with pi_4 and pi_5, so the initial derivation becomes

   $(G_1, \{ pi_6 \}, \{ pi_4, pi_5 \})$, with $G_1 = \boxed{\textbf{begin}} \xrightarrow{n} \boxed{\textbf{Stat}} \xrightarrow{n} \boxed{\textbf{end}}$

2. For this derivation, the single candidate production is pi_3, which is not involved in an *excludes** relation with any of the still applicable production instances, so it can be applied directly:

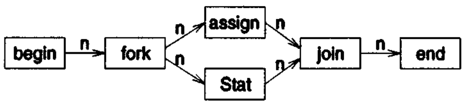
$$(G_2, \{pi_6, pi_3\}, \{pi_4, pi_5\}),$$

$G_2 =$ 

3. The candidate productions are now pi_1 and pi_2. Neither has an *excludes** relation, so we can freely pick one and apply it. This produces

$$(G_3, \{pi_6, pi_3, pi_1\}, \{pi_4, pi_5\}),$$

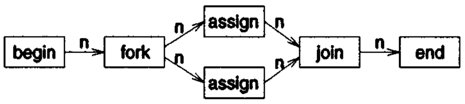
with

$G_3 =$ 

4. Application of pi_2 on this derivation leads to

$$(G_4, \{pi_6, pi_3, pi_1, pi_2\}, \{pi_4, pi_5\}),$$

with

$G_4 =$ 

5. Now the collection of *Candidate* production instances is empty and $G_4 \equiv G$, so we have found a successful sequence of production instances in $\{pi_6, pi_3, pi_1, pi_2\}$, which is returned.

5. Conclusions

Graphs and graph grammars are well-suited means for representing visual language sentences and for defining their syntax. They may be used to develop (generate) syntax-directed editors for visual languages. These editors should offer additional commands for ‘free editing’, which requires the development of efficient parsing algorithms. Graph grammer parsing algorithms available currently impose rather severe restrictions on the class of grammars they are able to deal with. This was our motivation for developing a rather general class of *layered context-sensitive graph grammars*, which can conveniently be used to define the syntax of visual languages, and to design an associated *graph parsing algorithm*.

This paper sketches the developed parsing algorithm and mentions all kinds of additional improvements which could be used to reduce its search space. The associated technical report [18] describes the algorithm in more detail, and proves its *termination and correctness* for any given layered graph grammar and input graph. The layering condition is decidable, so a parser generator can reject any grammar for which termination of the generated parser cannot be guaranteed.

Our approach differs from other visual language approaches in the large class of grammars it accepts. This facilitates considerably the definition of complex syntactical requirements. Furthermore, our algorithm uses quite a complex *history mechanism*—a graph of production instances—which, among other things, enables us to deal with context elements without any restriction. The two phases of our parsing algorithm make it possible to handle multiple derivations correctly and efficiently. Still, all the above-mentioned advantages have their price in a worst-case exponential time and space complexity of the resulting algorithm. Whether the heuristics suggested for reducing the algorithm's search space are sufficient to guarantee a better behavior for real visual languages has to be proved in practice.

A weakness of the parsing algorithm presented here is its inability to identify equivalent subderivations which are the result of local ambiguities. We have an indication on how to solve this problem, but that requires further theoretical work. Furthermore, we know that our solution of the 'embedding problem' via *context elements* has its drawbacks if a single nonterminal node may have an arbitrary number of incident edges. In this case *embedding rules*, which are able to redirect and recolor edge bundles of arbitrary size, would be more appropriate. We are planning, therefore, to add embedding rules to our graph grammar formalism. Other extensions considered are concerned with the introduction of attributes and negative application conditions.

We are currently implementing the parsing algorithm as a stand-alone software package which may be used in different environments. By testing it on syntax definitions of various real-world visual languages we hope to obtain a better insight into the applicability of the developed graph grammar formalism. Furthermore, this will allow us to analyse the efficiency of the parser on actual visual sentences. The implementation will become part of a syntax-directed editor toolkit for visual languages [1, 19] under development, and the graph grammar programming environment PROGRES [23, 24].

Acknowledgements

We would like to thank the referees of the JVLC for their extensive and helpful comments on the first version of this paper.

References

1. M. Andries, G. Engels & J. Rekers, How to represent a visual program? AVITVL, in press.
2. D. Blostein, H. Fahmy & A. Grbavec (1996) Issues in the practical use of graph rewriting. In: *Proceedings of the 5th international workshop on Graph Grammars and their application to computer science—GraGra94*, Williamsburg, Virginia. These proceedings will appear in the LNCS series; an extended version of the paper is available as Technical Report 95-373 from Queen's University, Ontario.
3. H. Bunke & B. Haller (1992) Syntactic analysis of context-free plex languages for pattern recognition. In: *Structured Document Image Analysis* (Baird, Bunke & Yamamoto, eds). pp. 500–519. Also appeared in Springer LNCS 411, Berlin pp. 136–150.
4. S. Chok & K. Marriott (1995) Automatic construction of user interfaces from constraint multiset grammars. In: *Proceedings 11th IEEE symposium on Visual Languages—VL'95*, pp. 242–249.

5. S. Chok & K. Marriott (1995) Parsing visual languages. In: *18th Australasian Computer Science Conference*, Glenolg, South Australia.
6. V. Claus, H. Ehrig & G. Rozenberg (eds) (1979) *International Workshop on Graph Grammars and their Application to Computer Science and Biology*. Springer, LNCS 73, Berlin.
7. H. Ehrig (1979) Introduction to the algebraic theory of graph grammars (a survey). In: *International Workshop on Graph Grammars and their Application to Computer Science and Biology*, pp. 1–69. Springer, LNCS 73, Berlin.
8. H. Ehrig, H. Kreowski & G. Rozenberg (eds) (1991) *4th International Workshop on Graph Grammars and Their Application to Computer Science*. Springer, LNCS 532, Berlin.
9. H. Ehrig, M. Nagl & G. Rozenberg (eds) (1983) *2nd Int. Workshop on Graph Grammars and Their Application to Computer Science*. Springer, LNCS 153, Berlin.
10. H. Ehrig, M. Nagl & G. Rozenberg (eds) (1987) *3rd Int. Workshop on Graph Grammars and Their Application to Computer Science*. Springer, LNCS 291, Berlin.
11. F. Ferrucci, G. Tortora, M. Tucci & G. Vitellio (1994) A predictive parser for visual languages specified by relation grammars. In: *Proceedings 10th IEEE Symposium on Visual Languages—VL'94*, pp. 245–252.
12. E. Golin (1991) A method for the specification and parsing of visual languages. Ph.D. Thesis, Brown University.
13. E. Golin (1991) Parsing visual languages with picture layout grammars. *Journal of Visual Languages and Computing*, 2, 371–394.
14. ITU-T, Geneva (1993) *Recommendation Z.120: Message Sequence Chart* (*MSC*). See also: *http://www.win.tue.nl/win/cs/fm/sjouke/msc.html*.
15. M. Kaul (1983) Parsing of graphs in linear time. In: *4th International Workshop on Graph Grammars and Their Application to Computer Science*, pp. 206–218. Springer, LNCS 153, Berlin.
16. M. Nagl (1979) *Graph-Grammatiken*. Vieweg Verlag, Braunschweig (in German).
17. J. Pfaltz & A. Rosenfeld (1969) Web grammars. In: *International Joint Conference on Artificial Intelligence*, pp. 609–619.
18. J. Rekers & A. Schürr (1995) A parsing algorithm for context-sensitive graph grammars (long version). Technical Report 95-05, Leiden University, the Netherlands. Available via ftp from *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1995/tr95-95.ps.gz*.
19. J. Rekers & A. Schürr (1996) A graph based framework for the implementation of visual environments. In: *Proceedings 12th IEEE Symposium on Visual Languages—VL'96*, pp. 148–155.
20. G. Rozenberg & E. Welzl (1986) Boundary NLC graph grammars—Basic definitions, normal forms, and complexity. *Information and Control*, 69, 136–167.
21. A. Salomaa (1973) *Formal Languages*, ACM Monograph Series. Academic Press, New York.
22. H. J. Schneider (1970) Chomsky-Systeme für partielle Ordnungen. Technical Report 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen.
23. A. Schürr, A. Winter & A. Zündorf (1995) Graph grammar engineering with PROGRES. In: *Proceedings 5th European Software Engineering Conference* (*ESEC '95*). Springer, LNCS 989, Berlin, pp. 219–234.
24. A. Schürr, A. Winter & A. Zündorf (1995) Visual programming with graph rewriting systems. In: *Proceedings 11th IEEE symposium on Visual Languages—VL'95* pp. 326–333.
25. K. Wittenburg (1992) Earley-style parsing for relational grammars. In: *Proceedings 8th IEEE Workshop on Visual Languages—VL'92*, pp. 192–199.