A Graph Grammar Approach to Graphical Parsing

J. Rekers
Department of Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
email: rekers@wi.leidenuniv.nl

A. Schürr
Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D-52074 Aachen, Germany
email: andy@i3.informatik.rwth-aachen.de

Abstract

We present a new graph grammar based approach for defining the syntax of visual languages and for generating visual language parsers. Its main advantage — in comparison to other visual language parsing approaches — is its ability to handle context-sensitive productions which may replace more than one non-terminal at the same time and which may contain very complex context requirements. Its implementation will be part of a forthcoming parsing toolkit for visual languages and the already existing graph grammar programming environment PROGRES.

1 Introduction[†]

In reading the proceedings of the past VL workshops 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*. That is regrettable, as such definitions have a number of advantages to offer:

- without a proper syntax definition, new users can only guess the syntax of a graphical language by generalizing from the provided examples,
- a definition could serve as unambiguous specification for syntax directed editors over the language,
- a graphical parser could be generated out of a proper definition, and
- a syntax definition is a necessary precondition for a definition of the semantics of the language.

We think that it would be very beneficiary to the theory of visual languages if a single syntax definition formalism could be agreed on. Such a formalism should be highly expressive, unambiguous, and specifications should be easy to read and develop.

In this paper we will show how *graph grammars* can be used as syntax definition formalism for graphical languages, and we will develop a graphical parsing algorithm based on these grammars. We will however first argue why graphical parsing would be useful for users of visual languages, and we will show that graphical parsing, if used to its full power, is less trivial than it might seem.

A *graphical parser* translates a diagram from its raw picture format into its underlying syntactic structure. If a graphical parser is incorporated in a syntax directed graphical editor, it may offer a far greater freedom to the user, as the editor then only needs to require that the final diagram is syntactically correct, instead of requiring that every intermediate diagram is syntactically correct. Furthermore, graphical parsing makes it possible to process diagrams which have been drawn with ordinary graphical editors.

We explain our view on graphical parsing and graphical syntax definition in Sec. 2, followed by a discussion of related work in Sec. 3. Sec. 4 forms the core of the paper in which we present our graphical parsing algorithm. In Sec. 5 we summarize the paper and discuss some remaining problems.

2 Multi-stage graphical analysis

We propose a multi-stage graphical analysis (parsing) process as depicted in Fig. 1. Usual graphical editing, such as with Idraw or FrameMaker, produces a collection of pictorial elements, which are fed to phase 1 of the analysis process, **graphical scanning**. It interprets the positions, sizes, and shapes of pictorial elements and produces a **spatial relations graph**, in which objects are connected by relations like *contains*, *points at*, and *labels*.

Phase 2, **low level parsing**, maps pictorial elements and their spatial relationships onto more abstract objects and relationships between them. Its output is an **abstract relations graph**. For **F**inite **S**tate **A**utomata this phase would recognize a circle that encloses a string as a *State*, and an arrow with a close-by string as a *Transition* (cf. productions p_5, p_6, and p_7 of [13] or the FSA grammar of Marriott [10]). Fig. 2 shows a FSA diagram and the transformation in its corresponding abstract relations graph. However, low level parsing would also happily accept an unconnected FSA, an

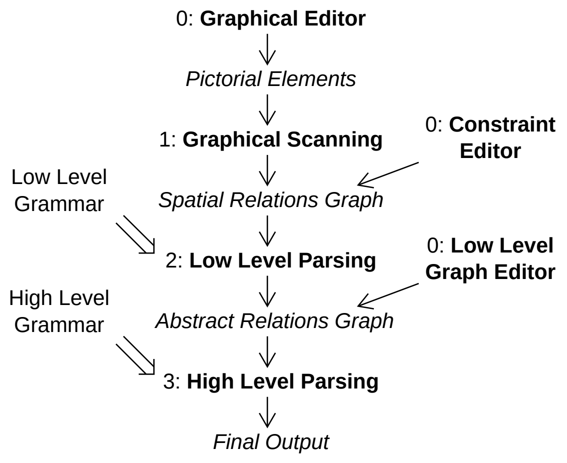
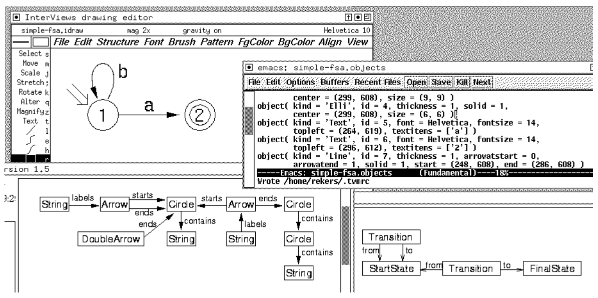


Figure 1: Our multi-stage graphical analysis approach

[†]This is technical report 95-15 of Leiden University and is available via ftp from ftp-server *ftp.wi.leidenuniv.nl*, file */pub/CS/Technical-Reports/1995/tr95-15.ps.gz*. This paper will also appear in the proceedings of the *1995 IEEE Symposium on Visual Languages* (*VL'95*), Darmstadt, Germany, September 5-9, 1995.


InterViews drawing editor
simple-fsa.idraw
mag 2x
gravity on
Helvetica 10
File Edit Structure Font Brush Pattern FgColor BgColor Align View
emacs: simple-fsa.objects
File Edit Options Buffers Recent Files Open Save Kill Next
b
a
1
2
String
labels
Arrow
starts
ends
Circle
contains
DoubleArrow
String
Circle
Circle
String
Transition
from
to
StartState
Transition
FinalState

```
        center = (299, 608), size = (9, 9) )
object( kind = 'Elli', id = 4, thickness = 1, solid = 1,
        center = (299, 608), size = (6, 6) )
object( kind = 'Text', id = 5, font = Helvetica, fontsize = 14,
        topleft = (264, 619), textitems = ['a'] )
object( kind = 'Text', id = 6, font = Helvetica, fontsize = 14,
        topleft = (296, 612), textitems = ['2'] )
object( kind = 'Line', id = 7, thickness = 1, arrowatstart = 0,
        arrowatend = 1, solid = 1, start = (248, 608), end = (286, 608) )
-----Emacs: simple-fsa.objects      (Fundamental)----18%-----------------
Wrote /home/rekers/.twmrc
```

Figure 2: The transformation in a number of steps of an FSA diagram into its abstract relations graph

automaton with unreachable states, one without a start state, or one with several start states.

In order to catch those kinds of errors, additional more complex grammar rules are needed. Phase 3, **high level parsing**, is defined by a grammar which states how sentences should be generated, starting from an *axiom* (cf. productions p_1, p_2 and p_3 of [13]). For the FSA case, the axiom would be an *Automaton*, which may be rewritten into a *StartState*; given a *State*, one may add an outgoing *Transition* and a new *State*; given two *States*, one may connect them by a *Transition*. These rules guarantee connected automata, for which all states are reachable from the start state.

It is also possible to perform only some steps of the proposed multi-stage graphical analysis process. For instance, the constraint based graph editor EDGE [9] may be used to create a spatial relations graphs, and phase 1 would not be necessary. Or a syntax directed editor may directly create an abstract relations graph, thereby skipping phases 1 and 2.

2.1 Process flow diagrams as example

The running example of this paper will be the recognition of well-structured process flow diagrams. An example of the abstract relations graph of a process flow diagram is depicted in Fig. 3. Do note that this graph already suggests work of phases 1 and 2 by the shape of the vertices in this graph, but this is only to make the example less abstract.

Fig. 4 contains the (high-level) grammar for this language. Production 1 of this grammar replaces the axiom *PFD* with 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 **dashed 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*} and therefore source of a *n(ext)*, *t(rue)* or *f(alse)* edge. The right context vertex is one of {*end*, *assign*,...} and therefore always the target of a *n(ext)* edge. Separately defined **label wildcards** may be used to construct a single production as an abbreviation 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, establish communication channels between them, and produce conditional loops as well as branches. Note that already such a small grammar

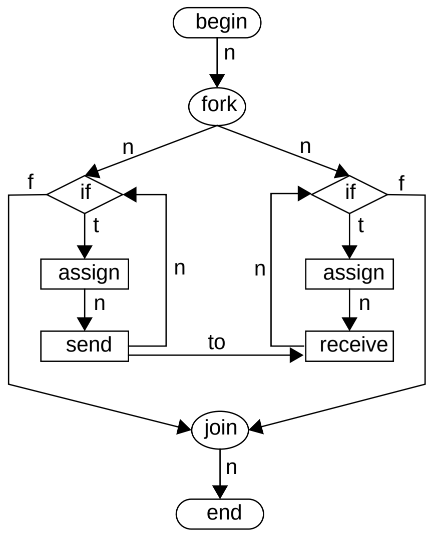


Figure 3: Sample process flow diagram

axiom: PFD
label wildcards:
B?, C? ∈ { begin, fork, if }
S?, T? ∈ { end, assign, fork, join, send, receive, if }
s?, r? ∈ { n, f, t }

productions:

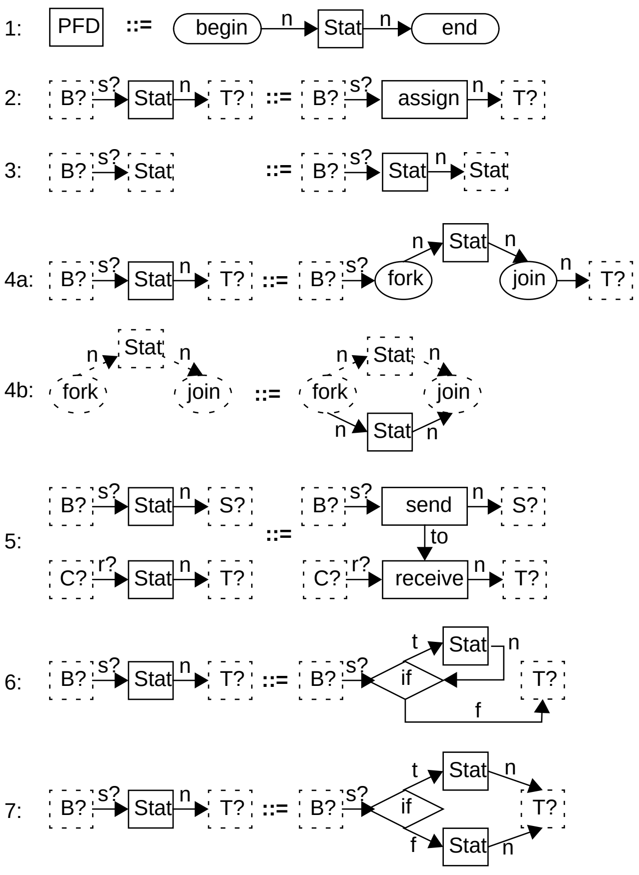


Figure 4: Grammar for process flow diagrams

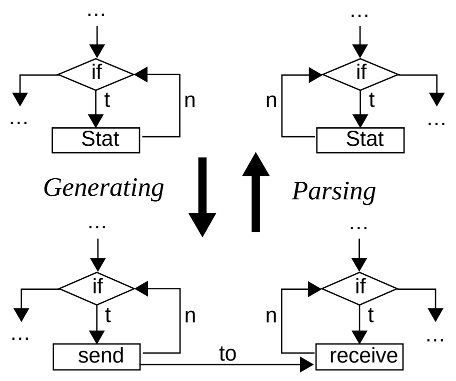


Figure 5: A context-sensitive rewriting step

contains reasonable examples of productions which do not delete any nonterminals (production 3 and 4b) or which replace more than one nonterminal at the same time (production 5).

3 Analysis of Related Approaches

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 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 old context elements of a replaced nonterminal and its replacing (non-)terminals (see productions of Fig. 4 and Fig. 5).

3.1 Embedding Problems of Graphical Languages

There are at least three solutions for the embedding problem:

1. **Implicit Embedding**: formalisms like picture layout grammars [5] or constraint multiset grammars [10] do not distinguish between vertex and relationship objects. As a consequence, all needed relationships between objects are implicitly defined as constraints over their attribute values. Therefore, attribute assignments within productions have the implicit side effect to create new relationships to unknown context elements.
2. **Extended Context**: all approaches which support the concept of relationships between objects directly, need a special mechanism to embed new (non-)terminal objects in their proper context. A straight-forward solution for this problem is to extend left- and right-hand sides of productions with context elements (as we do in Fig. 4). These context elements will not be modified by production applications but may be used as sources or targets for new relationships.
3. **Embedding Rules**: The last and most powerful solution is an essential part of various forms of graph grammars like [7,14]. These formalisms have separate embedding rules which allow the redirection of arbitrary sets of relationships from a replaced nonterminal to its replacing (non-)terminals.

All three approaches have their specific advantages and disadvantages. The main drawbacks of the first 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.

The second approach is in our opinion the most readable one, but the unrestricted use of *context elements* requires very complex parsing algorithms. 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 very hard restrictions on left- and right-hand sides of productions (see below).

3.2 Properties of Graphical Parsing Algorithms

Summarizing the explanations above, related parsing approaches should be studied and compared by answering the following questions:

- 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 sides** of productions?
- Does the formalism allow 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?

Table 1 provides an overview of our related work studies with respect to these questions.

3.3 Analysis of Graphical Parsing Algorithms

The **precedence graph grammar** parser of Kaul is an attempt to generalize the idea of operator precedence based parsing and 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 [2]. The first one by Bunke and Haller [1] 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 [17] 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 Ferucci et al. [4] with so-called **1NS-RG** grammars is a translation of the graph grammar approach of Rozenberg/Welzl [1] into the terminology of relational languages. In this approach right-hand sides of productions may not contain nonterminals as neighbours, 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.

All presented approaches up to now are not adequate for generating process flow graphs. Their embedding rules are not able to rewrite previously unconnected *Stat* vertices to pairs of connected *Send* and *Receive* vertices (as we do in the rewriting step presented in Fig. 5). And even the remaining two approaches of Marriot and Golin would have their difficulties with process flow diagrams. Their parsing algorithms generalize the bottom-up algorithms of Tomita [16] or **Cocke-Younger-Kasami** [18, 6] for context-free textual grammars. Marriot’s **constraint multiset grammars** [10] offer the concept of context elements and would thereby be able to define the graph rewriting step of Fig. 5. But the accompanying parsing algorithm is not yet able do deal with these context elements.

There remains the **picture layout grammars** [5] of Golin. His parsing algorithm allows terminal context elements, but has a main focus on productions with one nonterminal on the left-hand side and at most two nonterminals on the right-hand side with predefined spatial relationships between them. A definition of process flow graphs which obeys these restrictions should be possible but would be quite unreadable. Furthermore, the realized parsing algorithm has the following restrictions:

- Two nonterminals of the same class, which are used for the derivation of a single graphical language sentence, must have different attribute values (otherwise the parser makes the wrong decision to identify them).
- Their must be an upper boundary for the number of possibly used different nonterminal attribute values during parsing (otherwise, the parser tries to create an infinite number of unidentifiable nonterminals).
- Overlapping matches of right-hand sides due to ambiguous derivations may not occur (they cause the parser’s immediate termination with a fatal error).

To summarize, all presented parsing approaches are *inadequate for defining* the language of 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 algorithms will be presented within the next section.

4 The graph parsing algorithm

Our parsing algorithm for graphical languages is in fact a graph parsing algorithm. In this paper we will only sketch its main characteristics; for a complete presentation of all details including a full proof of correctness and termination, the reader is referred to [13].

Definition 1: A **production** is of the form (L, R), with L and R both graphs. L and R may have an intersection.❑

A production $p := (L, R)$ can be read in two directions:

- In *generation*, the host graph G is searched for a match $h(L)$ of the left-hand side L. If found, $h(L)$ is replaced by a copy $h'(R)$ of the right-hand side, resulting in a graph G'.

	Left-hand Side	Right-hand Side	Context Elements	Embedding Rules	Additional Parsing Restrictions	Space/Time Complexity
Kaul [7,8]	one nonterminal	directed nonempty graph	no	yes	implicitly def. vertex ordering by edges	linear
Bunke/Haller [1]	one nonterminal	nonempty plex structure	no	fixed set of connection points	no	exponential
Wittenburg [17]	one nonterminal	nonempty relational structure	no	yes	explicitly def. vertex ordering by relations	exponential
Ferrucci et al. [4]	one nonterminal	rel. structure without nonterminal neighbours	no	yes	(bounded degree, connected structure)	exponential (polynomial)
Rozenberg/ Welzl [14]	one nonterminal	dir. graph without non-terminal neighbours	no	yes	(bounded degree, connected graph)	exponential (polynomial)
Marriot [10]	one nonterminal	arbitrary nonempty multiset	(yes)	implicit	acyclic grammar, no context elements	exponential
Golin [5]	one nonterminal	one or two (non-)terminals	one terminal	implicit	finite set of possible attribute values	polynomial
Rekers/Schürr [13]	directed graph	directed connected, nonempty graph	directed graph	not yet	global layering condition for grammar	exponential

Table 1: Visual language syntax definition and parsing approaches

- In *parsing*, the host graph *G'* is searched for a match *h'*(*R*) of the right-hand side *R*. If found, *h'*(*R*) is replaced by a copy *h*(*L*) of the left-hand side *L*, resulting in a graph *G*.

Definition 2: The application of a production *p* is a **production instance** (p, h, h'), where graph morphisms $h\colon L \to G$ and $h'\colon R \to G'$ relate vertices and edges of *L* and *R* to vertices and edges of *G* and *G'*. The **application** of *p* to *G* with result *G'* is denoted as

$$G \overset{\text{pi}}{\Rightarrow} G' \quad . \qquad ❑$$

Do note that the part in *G* that corresponds to $h(L \cap R)$ and the part in *G'* that corresponds to $h'(L \cap R)$ are equal and thus preserved by the application of *pi* in either direction. This part is the *context* which has to be present, but which itself is not affected by the application of *pi*.

Definition 3: A **derivation** from the axiom graph *A* to the graph parsed *G* is a sequence of production instances $pi_1, \ldots, pi_n$ with the following property:

$$A \overset{\text{pi}_1}{\Rightarrow} G_1 \overset{\text{pi}_2}{\Rightarrow} \ldots \overset{\text{pi}_n}{\Rightarrow} G_n \equiv G \qquad ❑$$

One can distinguish two tasks which have to be performed by a graph parsing algorithm:

1. The *searching* in the host graph *for matches* of right-hand sides of productions. This is an expensive process which works at graph element level.
2. Each completed match results in a production instance. These have to be *combined into a derivation*. In the case of ambiguities, it might however happen that more than one derivation exists, or it might happen that a recognized 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 as follows:

- The **bottom-up phase** searches the graph for matches of the right-hand side of productions. On the recognition of such a right-hand side, a production instance *pi* is created, and the elements in *h'(L \ R)* are added to the graph, but nothing is deleted from it. The additions might in turn lead to the recognition of other right-hand sides. The result of the bottom-up phase is the collection *PI* of all production instances discovered.
  The production instances created have **dependency relations** among each other, such as *above(pi, pi')*, which means that *pi* should occur before *pi'* in a derivation, or *exclude(pi, pi')*, which states that *pi* and *pi'* may not occur in the same derivation. These relations can be computed during the bottom-up phase.
- The dependency relations are used to direct the second phase of the parsing process, the **top-down phase**. It starts with an empty graph and applies production instances of *PI* in such a way that the *above* and *exclude* relations are respected. By knowing all possible production instances and their dependency relations in advance, the top-down phase is able to postpone exploration of alternative derivation branches as long as possible. When necessary, these alternative derivations are developed in a pseudo-parallel fashion, with a preference for depth-first development.

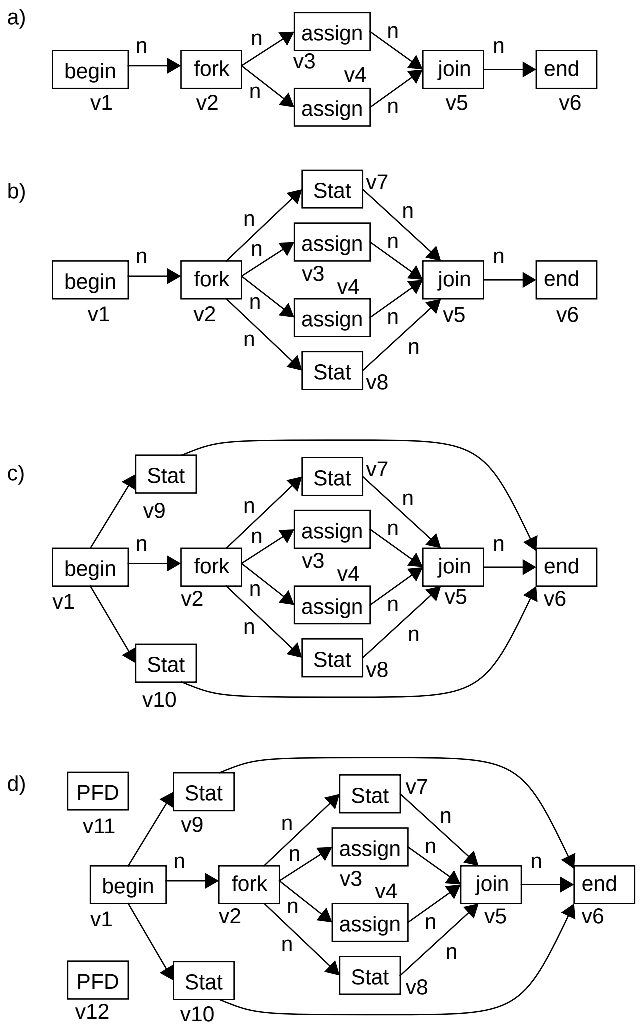


Figure 6: Some intermediate graphs of the BU phase

4.1 The bottom-up phase

We will parse the process-flow diagram of Fig. 6a according to the grammar of Fig. 4. In the end this will lead to two possible derivations, as it is ambiguous which of the two assign statements should match production *4a*.

The bottom-up phase searches the graph for matches of productions. The right-hand side of production *2* matches twice in this diagram: this results in production instances *pi1* and *pi2* of Table 2, and extends the graph to the one of Fig. 6b. The match described by production instance *pi2* is depicted more clearly in Fig. 7.

In the extended graph the right-hand side of production *4a* can be recognized with *Stat* vertex *v7*, but it can also be recognized with *Stat* vertex *v8*. Similarly, production *4b* can be recognized in two ways. This means that production instances *pi3*, *pi4*, *pi5*, and *pi6* of Table 2 are created, and the graph is extended to the one of Fig. 6c. In this graph the right-hand side of the production *1* can be recognized, which

pi	prod.	Xlhs	Common	Xrhs
pi1	2	v7	v2 v5	v3
pi2	2	v8	v2 v5	v4
pi3	4a	v9	v1 v6	v2 v7 v5
pi4	4a	v10	v1 v6	v2 v8 v5
pi5	4b		v2 v8 v5	v7
pi6	4b		v2 v7 v5	v8
pi7	1	v11		v1 v9 v6
pi8	1	v12		v1 v10 v6

Table 2: The production instances created

means that two production instances are created, and two *PFD* vertices (*v11* an *v12*) are added to the graph. That completes the work of the bottom-up phase, and the production instances of Table 2 will be shipped to the top-down phase.

4.2 Search plans and dotted rules

In the description above, we simply stated which matches were possible, but not how these matches are computed. In order to implement searching for matches efficiently, we associate a linear **search plan** to each production's right-hand side, which predetermines the order in which the match must be constructed. Each search plan starts with the matching of a single vertex, and extends its match at every step by following an edge from the already matched part of the host graph into the still unknown part. For example, a reasonable search plan for production 6 of Fig. 4 would be:

N1: vertex({if})

E1: edge(N1, ⟶, {t}, N2: vertex({Stat}))

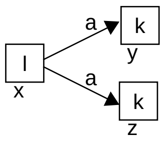
E2: edge(N2, ⟶, {n}, N1)

E3: edge(N1, ⟶, {f}, N3: vertex({end, assign, fork,
join, send, receive, if}))

E4: edge(N1, ⟵, {n, t, f}, N4: vertex({begin, fork, if}))

In order to be able to construct these search plans, right-hand sides of productions have to be connected. In constructing a match, the bottom-up phase moves a **dot** through the search plan: vertices and edges left of the dot are already matched, the ones right of it still have to be matched. Such a **dotted rule** is attached to the vertex in the host graph where the next to be matched edge starts or ends. 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.

Say, we have the following subgraph in the host graph and a



dotted rule is attached to *x* with its dot in front of the matching directive:

E4: edge(N1, ⟶, {a}, N2: vertex({k}))

In this case both edges match, and *y* and *z* both obtain an incremented version of the dotted rule. We also have to leave the original dotted rule (in suspended state) attached to vertex *x*, as another edge with label *a* may still appear.

The bottom-up algorithm starts by attaching initial dotted rules to 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 $L \setminus \mathrm{R}$: vertices may give rise to initial dotted rules, edges may activate suspended dotted rules. This is repeated until there are no remaining active dotted rules.

Our bottom-up phase performs an exhaustive generation of all possible production instances, during which it only extends the graph on every production instance found. Although the algorithm takes care not to perform double work (initial dotted rules are only created for newly created vertices, dotted rules only proceed over newly created edges), this may still lead to non-termination, as the grammar may be cyclic.

We, therefore, have to introduce the following **layering restriction** on the grammar: every edge and vertex label is assigned a layer number, such that terminal labels belong to layer 0, and non-terminal labels belong to a layer greater than 0. This layer assignment has to be such that every production respects the following order $L < R$:

$$L < R \iff \exists_i \, |L|_i < |R|_i \wedge \forall_{j < i} \, |L|_j = |R|_j$$

with $|G|_k$ the number of elements in G which have a label of layer k. This layering restriction guarantees the termination of the bottom-up phase by disallowing cyclic grammars.

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 [13] for details.

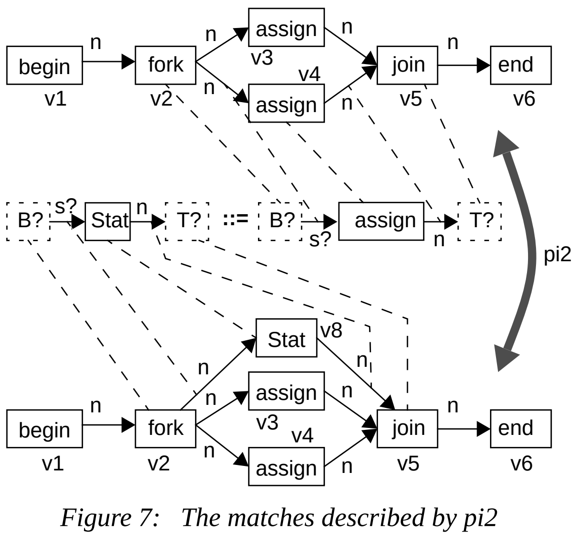


Figure 7: The matches described by pi2

4.3 The dependency relations

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 reason about these dependencies, we first introduce the abbreviations *Xlhs*, *Common*, and *Xrhs* for the different sets of elements matched by a production instance, and next introduce the dependency relations *above, consequence, excludes,* and *excludes**.

Definition 4: We define the following abbreviations for a production instance $pi := (p, h, h')$ with $p := (L, R)$:

- $Xlhs(pi) = h(L \setminus R)$, the exclusive left-hand side, which are (in *generation*) the set of deleted graph elements;
- $common(pi) = h(L \cap R)$, the set of matched but preserved graph elements;
- $Xrhs(pi) = h'(R \setminus L)$, the exclusive right-hand side, which are (in *generation*) the set of created graph elements. ❑

Definition 5: Given these partitions, we can define three cases when a production instance *pi* should be **above** a production instance *pi'* in any derivation in which both occur:

1. $Xrhs(pi) \cap Xlhs(pi')) \neq \emptyset$: *pi* creates an element which is deleted again by *pi'*, or
2. $common(pi) \cap Xlhs(pi')) \neq \emptyset$: *pi* needs a context element which is deleted by *pi'*, or
3. $Xrhs(pi) \cap common(pi')) \neq \emptyset$: *pi* deletes an element which *pi'* needs as context element.

In case 1 or 2, *pi'* consumes an element which is produced or needed by *pi*. Therefore, *pi' must* belong to any derivation which contains *pi, i.e.* pi' is said to be a **consequence** of *pi* (and **cons*** is its transitive closure). ❑

Definition 6: It may also be the case that two production instances **exclude** each other (directly or indirectly) and may never be part of the same derivation:

- $(Xrhs(pi) \cap Xrhs(pi')) \neq \emptyset$: *pi* and *pi'* create the same elements, or
- $above(pi, pi') \wedge above(pi', pi)$: *pi* and *pi'* have a mutual above relation.

The transitive closure **excludes*** is defined as follows:
$\exists\ \overline{pi}, \overline{pi}'$: $\overline{pi}$ *cons* pi* $\wedge$ $\overline{pi}'$ *cons* pi'* $\wedge$ $\overline{pi}$ *excludes* $\overline{pi}'$. ❑

Based on these definitions, the production instances of Table 2 lead to the relations as depicted in Fig. 8. For example, $above(pi8, pi4)$ holds since:

$Xrhs(pi8) \cap Xlhs(pi4) = \{v1, v10, v6\} \cap \{v10\} \neq \emptyset$.

4.4 The top-down phase

The top-down phase composes a subset of the production instances which forms a *derivation* for the graph parsed. Given the production instances of Table 2 and the relations of Fig. 8, correct derivations according to Def. 3 would be: $\{pi7, pi3, pi6, pi1, pi2\}$ and $\{pi8, pi4, pi5, pi1, pi2\}$.

The top-down phase develops such a derivation by keeping a stack of *partial derivations,* of which only the topmost one is active. The lower ones are starting points for alternative derivations, and will only become active if the ones higher on stack fail.

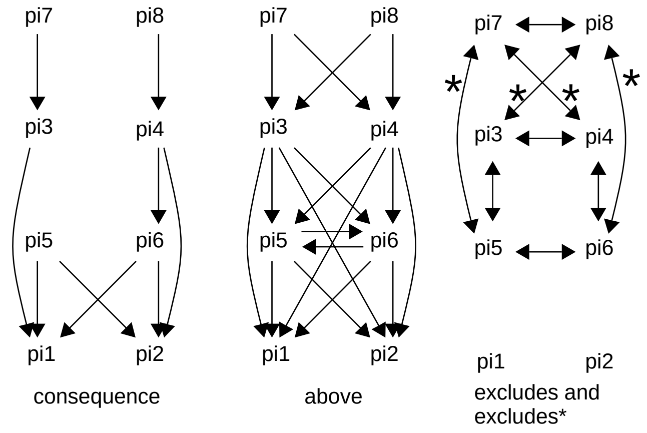


Figure 8: The dependency relations induced by Table 2

Definition 7: A tuple (G_c, API_c, EPI_c) is a **partial derivation** for *G* in the context of all possible production instances *PI*. G_c is the graph built by the applied production instances in API_c; the production instances in EPI_c are the already excluded ones. ❑

The top-down phase starts with a derivation $(A, \emptyset, \emptyset)$ of length 0. Next, it repeats the following steps until a complete derivation has been found, or there are no partial derivations left. It creates at each step a set of *candidate* production instances *CPI*, which are those *pi* in $PI \setminus (API_C \cup EPI_C)$ such that all production instances *above* it are either in API_C or in EPI_C. It depends on the contents of *CPI* what happens next:

- $CPI = \emptyset$: API_C is a successful derivation if $G_C = G$; otherwise, the current partial derivation is dropped and the top-down phase continues with the next one on stack
- Is there a $pi \in CPI$ without an *excludes** relation to another still applicable production instance: apply *pi* by changing the active partial derivation into
  $(G_C \setminus Xlhs(pi) \cup Xrhs(pi), API_C \cup \{p\}, EPI_C)$.
- Else: any $pi \in CPI$ is selected and a partial derivation of the form $(G_C, API_C, EPI_C \cup \{pi\})$ is pushed for the case that *pi* turns out to be a wrong choice. Next, *pi* is applied by changing the current partial derivation into:
  $(G_C \setminus Xlhs(pi) \cup Xrhs(pi),$
  $API_C \cup \{pi\},$
  $EPI_C \cup \{pi' \in PI\text{: } pi \text{ excludes* } pi'\})$. ❑

This process is fully based on production instances and their dependency relations. It generates a derivation if possible, and, in the case of more than one possible derivations, it generates the first one encountered.

4.5 The output of the analysis

It depends on the application domain what the *final result* of the analysis process must be and how it should be specified:

1. It might be the case that we are only interested whether a derivation exists or not. A *yes/no answer* then suffices.
2. Our parsing algorithm represents a derivation as a sequence of *production instance* applications. This pro-

vides all information, but might be too detailed and too abstract to interpret.

3. During the above rewriting process non-terminal elements are created, which are consumed again to create terminal elements. It would also be possible to perform the rewriting by *only creating elements*. In that case, the final result would be a graph which contains the axiom graph *A*, the final graph *G*, and all intermediate graphs as subgraphs. Certain terminals and non-terminals could be filtered out of this graph again.
4. Another solution would be the YACC way: attach an *arbitrary action* to every production; these actions are then executed in the order of the derivation found. This is a quite low-level solution which can however create any desired data structure.
5. A higher level approach is based on *coupling two graph grammars* such that parsing a graph with respect to the first grammar is bound to the generation of another graph with respect to the second grammar [15].

Throughout this paper we have only considered method 2; any further processing of derivation results is subject for future research.

5 Summary and discussion

Graph grammars are a very powerful mechanism for defining the syntax of graphical languages with well-known theoretical properties [3]. The main drawback of graph grammars until now was the lack of efficiently working parsing algorithms or, more general, the lack of almost any tool support. Within this paper we presented the overall ideas of a new *graph grammar parsing algorithm*. A complete formal definition of our new approach together with a proof of correctness of the presented parsing algorithm may be found in [13]. Its implementation will be part of a forthcoming parsing toolkit for visual languages [12] and the already existing graph grammar programming environment PROGRES [11].

A flaw in our parsing algorithm is the inability to identify “equivalent” non-terminals, which are the result of local ambiguities. This deficiency is demonstrated in the example of Fig. 6, in which two equivalent *Stat* nodes (*v9* and *v10*) appear. They are the result of a locally ambiguous interpretation of the *fork-join* statement. In order to avoid reduplication of parsing efforts, it would be nice if these two *Stat* nodes could be identified instead of resulting in two completely different derivations.

Parsing algorithms based on *cover checks*, such as Marriott’s [10], are able to identify non-terminals which are the result of local ambiguities: each element knows which input elements it covers; at the moment two elements with the same label and the same cover set occur, the two are identified. A drawback of cover set based parsing algorithms is that they require that the left-hand side of every production is a single non-terminal. They are even not able to deal with non-terminal context elements, since cover sets do not record the history of derivations. This is the reason why Marriott's parsing algorithm works only for grammars without context elements and why Golin’s parsing algorithm [5] disallows non-terminal context elements.

However, context elements and more complex left-hand sides are indispensable for producing readable syntax definitions of visual languages. Therefore, our parsing algorithm uses complex *history relations* instead of simple cover sets and is, thereby, able to handle productions properly, which

- delete more than one vertex at the same time,
- delete nothing at all, i.e. extend a graph only, and
- make use of context elements.

Future research is necessary to find a way to identify non-terminals with equivalent histories during parsing.

6 References

[1] H. Bunke and B. Haller. *A Parser for Context-free Plex Grammars*. In M. Nagl, editor, Proc. 15th Int. Workshop on Graph-Theoretic Concepts in Computer Science – WG’89, LNCS 411, Springer Verlag, pages 136–150, 1989.

[2] J. Earley. *An Efficient Context-free Parsing Algorith*m. *CACM*, 13(2), ACM Press, pages 94–102, 1970.

[3] H. Ehrig, H.J. Kreowski, and G. Rozenberg, editors, *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Springer Verlag, 1991.

[4] F. Ferruci, G. Tortora, M. Tucci, G. Vitiello: *A Predictive Parser for Visual Languages Specified by Relation Grammars*. In Proc. IEEE Symposium on Visual Languages -- VL’94, IEEE Computer Society Press, pages 245-252, 1994.

[5] E.J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

[6] T. Kasami. *An Efficient Recognition and Syntax Analysis Algorithm for Context-free Languages*. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford Mass., 1965.

[7] M. Kaul. *Parsing of Graphs in Linear Time*. In Ehrig, Nagl, and Rozenberg, editors, 2nd Int. Workshop on Graph Grammars, LNCS 153, Springer Verlag, pages 206–218, 1982.

[8] M. Kaul. *Syntaxanalyse für Präzedenzgraphgrammatiken*. PhD thesis, Universität Passau, 1985. In german.

[9] F. Newbery Paulisch. *The Design of an Extendible Graph Editor*. PhD thesis, University of Karlsruhe, Germany, LNCS 704, Springer Verlag, 1991.

[10] K. Marriott. *Constraint Multiset Grammars*. In Proc. IEEE symposium on Visual Languages -- VL’94, IEEE Computer Society Press, pages 118-125, 1994.

[11] *PROGRES – PROgramming with Graph REwriting Systems*. Software package, see world wide web page *http://www-i3.informatik.rwth-aachen.de/research/progres.html*.

[12] J. Rekers. *On the Use of Graph Grammars for Defining the Syntax of Graphical Languages*. In Proc. of Colloquium on Graph Transformation, Palma de Mallorca, 1994. Also available from ftp site *ftp.wi.leidenuniv.nl*, file */pub/CS/Technical-Reports/1994/tr94-11.ps.gz*.

[13] J. Rekers and A. Schürr. *A Parsing Algorithm for Context-sensitive Graph Grammars (long version)*. Technical Report 95-05, Leiden University, 1995. Available by ftp from ftp-server *ftp.wi.leidenuniv.nl,* file */pub/CS/TechnicalReports/1995/tr95-05.ps.gz*.

[14] G. Rozenberg and E. Welzl. *Boundary NLC Graph Grammars – Basic Definitions, Normal Forms, and Complexity.* Information and Control, 69, pages 136–167, 1986.

[15] A. Schürr. *Specification of Graph Translators with Triple Graph Grammars*. In Mayer and Tinhofer, editors, Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science – WG’93, will appear in LNCS, 1995.

[16] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer

Academic Publishers, 1985.

[17] K. Wittenburg. *Earley-style Parsing for Relational Grammars*. In Proc. IEEE Workshop on Visual Languages – VL'92, pages 192–199, 1992.

[18] D. Younger. *Recognition and Parsing of Context-Free Languages in Time n^3*. Information and Control, 10(2), pages 189–208, 1967.