A Parsing Algorithm for Context-Sensitive Graph Grammars[*]

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
e-mail: `andy@i3.informatik.rwth-aachen.de`

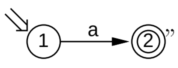
Abstract – Sentences of visual languages may often be regarded as assemblies of pictorial objects like "circles", "arrows" or "strings" with spatial relations like "above" or "contains" between them, i.e. their underlying structure is a kind of directed graph. Therefore, graph grammars are a natural means for defining the syntax of visual languages. Their main drawback until now is the lack of general enough and efficiently working parsing algorithms. All published graph grammar or —more general— visual language parsing algorithms are only able to deal with context-free graph grammars, where the left-hand side consists of a single nonterminal vertex only. This makes syntax definitions of visual languages hard to read, prohibits the use of complex pattern matching, and disallows graph-grammars which specify transformation processes.

We have developed the first parsing algorithm for context-sensitive graph grammars which allows left- and right-sides of productions to be almost arbitrary graphs. The algorithm is divided into two phases, where the first one constructs bottom-up a set of all eventually useful production applications. The second one extracts top-down viable derivations from the computed set of production applications. This separation into two phases leads to more comprehensible algorithms. Furthermore, it allows for independent optimization efforts in the form of heuristics which reduce the algorithm's exponential time and space requirements dramatically for "real world" examples.

1 Introduction

1.1 Parsing visual languages

Just as there are many different textual languages, many *visual* languages do exist. Nassi-Shneiderman diagrams or control flow diagrams are used as graphical pseudo code representations, database management systems are extended with visual data definition and query languages as front-ends, and CASE tools offer a large variety of diagrammatic languages for specifying software requirements and designs. Especially the number of already existing entity relationship diagram, data flow diagram, and finite state automaton dialects is unsurveyable. It is remarkable that almost all of these visual languages come without a formal definition of their underlying syntax (and semantics).

Everybody knows for instance that " 

" is a well-defined Finite State Automaton (FSA) as well as

[*]This is technical report 95-05 of Leiden University. Available by ftp from *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1995/tr95-05.ps.gz*.

[†]Partially supported by Netherlands Organization for Scientific Research (NWO) under grant B62-427

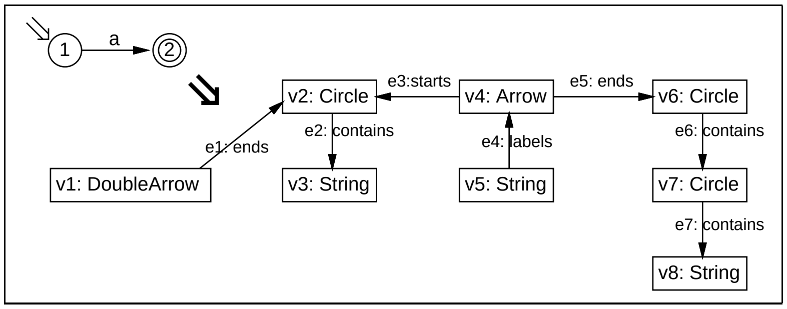
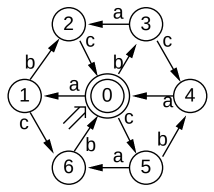
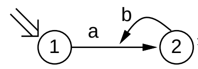


Figure 1: Translation of automaton into spatial relationships graph



whereas "  " is not a legal FSA. But where are the formalisms which allow us to write precise and computer readable definitions of this FSA language?

In the case of textual languages, various kinds of *context-free/attribute grammars* are used to define their syntax. These syntax descriptions are used to generate efficiently working parsing tools. This allows to use ordinary text-editors to create and manipulate sentences of these languages. Afterwards, generated parsing and analysis tools transform these sentences into decorated abstract syntax trees. Therefore, users are not forced to stick to syntax-directed editors, which manipulate the underlying abstract syntax tree instead of its textual representation.

In the case of visual languages, people seem to have less problems with the paradigm of syntax-directed editing. But even then, general purpose graphical editors would be welcome for any kind of restructuring processes, which introduce temporary syntactical inconsistencies in manipulated diagrams. Visual parsing is necessary after such modifications in order to rediscover the underlying abstract syntax graph. Summarizing, there are real needs for visual language definition *formalisms* and accompanying *parsing algorithms*.

In the sequel, we will focus our interest onto those visual languages, whose sentences may be seen as assemblies of *pictorial objects* (vertices) with well-defined *spatial relationships* (edges) between them. FSA diagrams contain for instance circles, arrows, and text labels as objects, and their spatial relationships are "contains", "starts at" etc. Figure contains an example of the translation of a given automaton into a spatial relationships graph. Translating arrows into vertices instead of edges, and text labels into vertices instead of vertex labels is strictly necessary. Otherwise, we would not be able to reason about relative positions of arrows and text labels, unless we would use a graph model which supports edges between edges and vertex or edge labels (cf. figure 1).

Graph grammars with their well-established theoretical background may be used as a natural and quite powerful syntax-definition formalism (cf. [15]) for languages of spatial relationships graphs. This means in turn that parsing algorithms for graph grammars may be used to check the syntactical correctness or to recover the underlying abstract syntax of "free-hand drawings".

1.2 Graph parsing algorithms

Unfortunately, even for the most restricted classes of graph grammars the membership problem is *NP-complete* [16]. As a consequence, all graph grammar parsing algorithms suggested up to now are either unable to recognize interesting languages of graphs, or tend to be hopelessly inefficient when applied to graphs with more than a few dozen nodes and edges[1].

Even worse, all currently known graph grammar parsing algorithms [10, 16, 12, 21, 3, 22, 7] deal with *context-free productions* only (where the left-hand side is a single non-terminal node). This might be sufficient from the theoretical point of view[2]. But in practice it would be quite useful to allow arbitrary graphs in the left-hand side of a production, which might even share a common subgraph with its right-hand side. Such a common subgraph allows us to identify "context elements" in a graph. Specifications of graph languages by means of these *context-sensitive graph grammars* tend to be more compact and easier to understand than their context-free counterparts.

The graph parsing algorithm proposed within this paper deals with context-sensitive graph grammars and consists of two major phases: A so-called **bottom-up phase** searches the graph for matches (redices) of right-hand sides of productions. For every redex found, it extends the given input graph with the corresponding production's left-hand side and generates a *potential production instance*. A **top-down phase** inspects the generated collection of production instances and selects a subset which together forms a viable derivation for the graph parsed.

The separation into two phases leads to more comprehensible algorithms and allows for independent optimization efforts. Therefore, we have been able to incorporate a number of *new heuristics* into our algorithm. These heuristics — hopefully — reduce the algorithm's worst-case complexity by orders of magnitude, when applied to "real-world" context-sensitive graph grammars and graph sentences.

1.3 Organization of the paper

The paper is organized as follows:

- Section 2 introduces our class of context-sensitive graph grammars, which is a slightly simplified and restricted version of the algebraic double pushout graph grammar approach [6].
- Section 3 presents our new graph grammar parsing algorithm and proves its correctness. Furthermore, it introduces the notion of search plans for finding matches of right-hand sides of productions.
- Section 4 discusses possible optimizations for the basic parsing algorithm of Section 3. Among other things, we discuss a priority queue mechanism, as well as heuristics and cost functions which help us to find good search plans for productions.
- Section 5 presents an example of a graph grammar which defines the syntax of a visual language and shows the intermediate and final data structures as would be generated by our parsing algorithm on an example sentence.
- Section 6 compares our new parsing approach with related research with respect to the expressiveness of the supported formalism and the worst-case analysis of the presented parsing algorithms.
- Section 7 summarizes the paper and discusses possible future work in this direction.

1.4 About the length of this paper

The ideas behind this paper have been conceived in the summer of 1994, while Andy Schürr visited Leiden University for four months. Unfortunately, the paper has become quite long. This lengthiness is mainly caused by the complicated nature of the developed algorithm: this has required a large

[1]For further details see Section 6.

[2]Since certain types of context-free graph grammars have the same expressiveness as their context-sensitive counterparts, cf. [13]

collection of basic definitions, has made quite some explanation next to the algorithms necessary, and has obliged us to prove a number of important lemmata. Furthermore, we felt the need to introduce a number of improvements and optimizations, to withstand the exponential complexity of the problem solved. All this has lead again to a complicated example.

In retrospect, we acknowledge that the paper is long, but do not see what could be left out without having the entire building collapse; neither do we see how to split the paper up in parts which solve interesting subproblems. We hope that readers will be interested enough in our solution to the hard problem of context-sensitive graph parsing, such that they will bear with us and will be willing to follow the entire line of thought.

2 Basic definitions

This section introduces the basic vocabulary of *graphs* and *graph grammars*, as well as some additional definitions for the parsing algorithm presented in the next section. As we will see later on, the defined class of graph grammars is equivalent to the algebraic double pushout approach [6], with modest additional requirements concerning the form of left- and right-hand sides of productions. These requirements simplify the parsing algorithm and guarantee its termination. Future extensions with respect to vertex (and edge) attributes as well as embedding rules (from algorithmic graph grammar approaches [13]) are planned. One very convenient extension is already part of this proposal: the introduction of a class hierarchy over vertex and edge labels. Such a *class hierarchy* is a prerequisite for the formulation of "generic" productions, where labeled vertices and edges match all vertices and edges in a host graph with the same or a more specific label. These generic productions do not improve the expressiveness of graph grammars (as long as label sets are finite), but they may reduce the number of necessary productions for the definition of a graph language considerably.

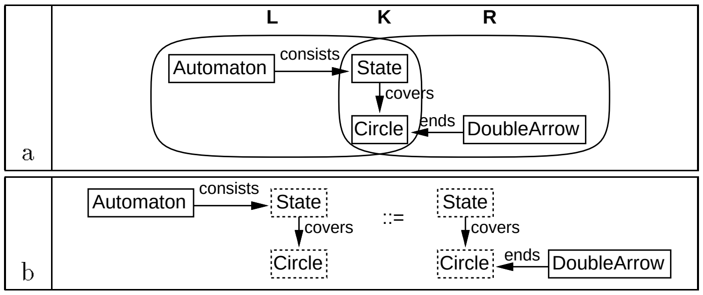
2.1 Graph grammar

Definition 2.1 Two (finite) sets L_V, L_E together with two binary relations $isa_V \subseteq L_V \times L_V$ and $isa_E \subseteq L_E \times \mathrm{L}_E$ are termed **hierarchical vertex and edge label sets** iff isa_V and isa_E are both (reflexive) partial orders. In the sequel, we will often omit the subscripts "V" or "E" if they are clear from context. □

Definition 2.2 $G := (V, E, l_V, l_E, s, t)$ is a **graph** over (hierarchical) label sets L_V, L_E with:

- $V(G) := V$ is a (finite) set of vertices,
- $E(G) := E$ is a (finite) set of edges,
- $l_V(G) : V \rightarrow L_V$ is the labeling function for vertices,
- $l_E(G) : E \rightarrow L_E$ is the labeling function for edges,
- $s(G) : E \rightarrow V$ assigns each edge its source, and
- $t(G) : E \rightarrow V$ assigns each edge its 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 will 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 $x \in V(G) \vee x \in E(G)$. □



	$L\backslash R$	$K = L \cap R$	$R\backslash L$
	$v_a : Automaton$	$v_s : State$	$v_d : DoubleArrow$
	$e_c : v_a \xrightarrow{\text{consists}} v_s$	$v_c : Circle$	$e_e : v_d \xrightarrow{\text{ends}} v_c$
c		$e_{cov} : v_s \xrightarrow{\text{covers}} v_c$	

Figure 2: A production (depicted in three ways)

The graph model defined above has as main advantage that both vertices and edges (in the sequel called *graph elements*) are identifiable objects. This allows to treat both kinds of elements in a similar manner and simplifies the following definitions and constructions considerably. Nevertheless, only minor modifications are necessary to adapt the parsing algorithm of section 3 to a data model without edge identification.

Definition 2.3 A (graph grammar) **production** is a tuple of graphs $p := (L, R)$ over the same alphabets of vertex and edge labels L_V and L_E. Graphs L and R may have a common subgraph K if the following restrictions are fulfilled:

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

Under these circumstances the common subgraph K, also called **interface graph** is defined as follows:

- $V(K) := V(L) \cap V(R)$
- $E(K) := E(L) \cap E(R)$
- The functions l_V, l_E, s and t of K are those of L and R restricted to $V(K)$ and $E(K)$. $\square$

The common subgraph defined above has the same purpose as the explicit interface graph of double pushout productions in [6]. It identifies all those *context elements* in a host graph[3] which have to be present, but are not deleted by the application of the production.

Figure 2 shows three ways to represent the same production. 2.a shows the overlap between L and R the most clearly, but, as is it is sometimes inconvenient to find a way to depict the overlap, we mainly use the notation of 2.b, which uses dotting to indicate the interface graph. The notation of 2.c is of textual nature, and is convenient if node and edge identifiers matter mostly. Do note that L, R and K are proper graphs, but that $L\backslash R$ and $R\backslash L$ may have dangling edges.

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 between two graphs over the same alphabets of vertex and edge labels L_V and L_E (which will be assumed fixed in the sequel).

[3]The "host graph" is the graph to which a production is applied.

Definition 2.4 A pair of functions $h := (h_V, h_E)$ is a **graph morphism** from a graph G to a graph G' (denoted as $h : G \to G'$) with $G := (V, E, l_V, l_E, s, t)$ and $G' := (V', E', l'_V, l'_E, s', t')$ iff:

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

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. Please note that the straightforward definition of the difference of two graphs as the difference of their vertex and edge sets may produce "dangling edges". We will use this definition nevertheless in order to be able to reason about dangling edges, while for example definition 2.5 below excludes them for the purpose of "sound" graph rewriting.

Definition 2.5 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:

  $\forall\, v \in V(L) \setminus V(R),\ e \in E(G) :$
  $\quad (s(e) = h_V(v) \vee t(e) = h_V(v)) \ \to \exists\, e' \in E(L) \setminus E(R) : h_E(e') = e.$

- Identification condition:

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

- Labeling condition:

$$\forall\, x \in L \setminus R : l(L)(x) = l(G)(h(x)).$$

Furthermore, a morphism is called a **potential redex** if the identification condition and the labeling condition are fulfilled, but the dangling edge condition is not taken into account. □

The three additional conditions for a redex are a necessary precondition for the inversion of a production $p := (L, R)$ and, thereby, for parsing graph grammars (see definition 2.6). The *dangling edge condition* prevents the deletion of vertices which have incident edges not mentioned in the production's left-hand side L. These edges would have to be guessed during parsing independent from the fact whether they connect two matched vertices or whether they connect a matched vertex with a vertex of the "surrounding" remaining host graph. The *identification condition* has a rather similar purpose. It states that context elements only are allowed to share their matches in G. This means that any element in $L \setminus R$ matches and deletes its own element in G, and reconstruction of deleted elements becomes feasible. The *labeling condition* finally ensures that deleted elements have just the labels denoted within the left-hand side of productions and not a more special label with respect to isa_V and isa_E. Without this restriction, we would have difficulties again to reconstruct deleted graph elements or more precisely, to reconstruct their labels. Finally note that the forthcoming definition of the application of a production ensures that the image of its right-hand side in the resulting graph is a redex with respect to the production's left-hand side (exchange the roles of L and R in definition 2.5 above).

2.2 Production instance

During parsing, which is more or less the inverse application of productions, we have to check whether potential matches of right-hand sides are indeed redices. Checking the "labeling condition" and the "identification condition" is possible without taking other production instances 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. As already mentioned before, our parsing algorithm of the next section will be divided into two phases. The first phase has not enough knowledge about applicable production instances for checking "dangling edge" conditions. It is only able to find a collection of *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 generate the given input graph (if existent).

Definition 2.6 A **production instance** of a production $p := (L, R)$ is a tuple $pi := (p, h, h')$ such that $h : L \rightarrow G$ and $h' : R \rightarrow G'$ 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 the interface graph of L and R, and
- $G \setminus (\ h(L \setminus R)\ ) = G' \setminus (\ h'(R \setminus L)\ )$

The application of a production p to a graph G with result G' will be denoted as $G \overset{p}{\Rightarrow} G'$.

Furthermore, the uniquely defined morphism[4] h for a given redex h' of R in G' will be denoted as **left-extend**$_p(h')$ (needed in algorithm 1.3). □

Figure 3 depicts a production instance for the production of Figure 2. It is applied to graphs G and G' with morphisms h and h'.

Definition 2.7 A **potential production instance** is a production instance (p, h, h') for which h and h' are *potential* redices, which do not necessarily conform to the "dangling edge" condition of definition 2.5. □

Please note that any pair of morphisms which stem from the application of a production p to a graph G build a potential production instance, but not the other way round. The overall idea of our parsing algorithm is to create first a dependency graph of potential production instances. Afterwards a DAG within this dependency graph will be selected such that its production instances do not conflict among each other and create indeed the given input graph.

2.3 Graph language

Based on the definition of productions and their application to graphs, graph grammars and their language are defined as follows:

Definition 2.8 A **graph grammar** gg is a tuple $(A, \mathcal{P})$, with A a nonempty initial graph (the axiom), and $\mathcal{P}$ 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 for a given graph grammar $gg := (A, \mathcal{P})$ is abbreviated with $\mathcal{PI}(gg)$. It includes morphisms from the initial graph A into another graph as an instance of an artificial production (λ, A). □

[4]Uniquely defined up to isomorphism

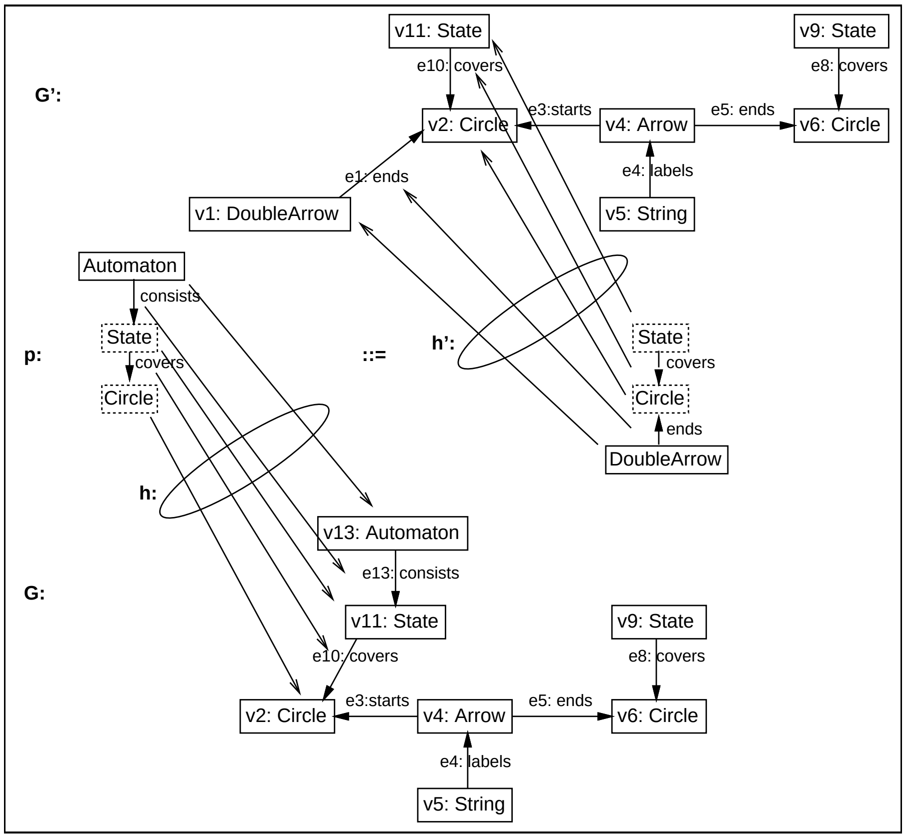


Figure 3: A production instance depicted

Definition 2.9 Let $gg := (A, \mathcal{P})$ be a graph grammar. Its **language** $\mathcal{L}(gg)$ is defined as follows:

- for graphs G and G': $G \Rightarrow G' \iff \exists p \in \mathcal{P} : G \overset{p}{\Rightarrow} G'$,
- $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$, and
- $G \in \mathcal{L}(gg) \iff A \Rightarrow^* G$. □

2.4 Layering

The above definitions of a graph grammar and its language are unusual with respect to vertex and edge labels. Up to now, we have made no distinction between *terminal* and *non-terminal* 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 set of non-terminals.

Definition 2.10 The decomposition $L_V \oplus^5 L_E = L_0 \oplus \ldots \oplus L_n$ of the vertex and edge label alphabet into n subsets is a **layered label set** iff:

$$\forall\, l \in L_i, l' \in L_j : l \; isa \; l' \rightarrow i \leq j,\; 0 \leq i, j \leq n$$

i.e. the layering is compatible with the partial orders isa_V and isa_E over vertex and edge labels. 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.:

$$\forall x \in G : layer(x) = i \iff l(x) \in L_i\,.$$

□

Definition 2.11 Given a decomposition $L_0 \oplus \ldots \oplus L_n$ of our label alphabet L_V and L_E, the language of a graph grammar gg may be decomposed into a number of **sub-languages** $\mathcal{L}_0(gg), \ldots, \mathcal{L}_n(gg)$, such that

$$\mathcal{L}_i(gg) := \{G \in \mathcal{L}(gg) \mid l(G) \subseteq \bigcup_{j \leq i} L_j\}.$$

In the case of two layers (with $n = 1$), the usual terminology may be used of $\mathcal{L}_0(gg)$ being the "real" **graph language** of gg and $\mathcal{L}_1(gg)$ the set of all **sentential forms** of gg (with $L_0(gg)$ being the set of terminals and $L_1(gg)$ the set of non-terminals). □

Using label layers, we are now able to define a rather general class of graph grammars which are "parsable". For these graph grammars we will present an algorithm which solves the memberships 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 2.12 A graph grammar $gg := (A, \mathcal{P})$ is called a **parsable** graph grammar with respect to a global **layer** assignment $L_0, \ldots, L_n$ to its labels, if $\forall p := (L, R) \in \mathcal{P}$:

- R is a connected graph.
- The left-hand side L is non-empty.
- The right-hand side R without the elements of K is non-empty. (λ-freeness[6] of the graph grammar).

[5] Disjoint union of sets
[6] also called ϵ-freeness

- $L < R$ with respect to the following order for graphs:[7]

$$G < G' \iff \exists\, i : \; |G|_i < |G'|_i \;\; \wedge \;\; \forall\, j < i : \; |G|_j = |G'|_j$$

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

These additional restrictions give us a number of desirable properties which we need in order to be able to parse according to a graph grammar:

- The connectedness of the right-hand side allows us to use a linearization of the right-hand side of a production, which can be matched step by step to elements of the graph, while following edges in this graph (see definition 3.1).
- The non-emptiness of left-hand sides guarantees that each application of a production (see definition 2.6) "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 non-emptiness 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 of section 3, as it disallows "cyclic" grammars (see lemma 2.14 below). Even better, layering of label sets and the definition of $\mathit{lll}(p)$ and $\mathit{hrl}(p)$ (definition 2.13, below) allows the parsing algorithm to organize its work in a more efficient manner (see section 4.2).

Definition 2.13 The **lowest left layer** of a production p (or $\mathit{lll}(p)$) and the **highest right layer** of a production p (or $\mathit{hrl}(p)$) are defined as:

$$\begin{array}{lll} \mathit{lll}(p) & := & \textbf{if } L\backslash R = \emptyset \textbf{ then } \infty \textbf{ else } \mathit{min}\{\mathit{layer}(x) \mid x \in L\backslash R\} \textbf{ fi} \\ \mathit{hrl}(p) & := & \mathit{max}\{\mathit{layer}(x) \mid x \in R\} \end{array}$$

Therefore, $\mathit{lll}(p)$ is defined such that p deletes only elements of $\mathit{lll}(p)$ and higher layers or creates them during parsing. $\mathit{hrl}(p)$ is defined such that all elements which are matched by the production's right-hand side during parsing belong to $\mathit{hrl}(p)$ or lower layers. □

Lemma 2.14 The reverse application (h, h', p^{-1}) of a production $p := (L, R)$ of a parsable graph grammar, the application of $p^{-1} := (R, L)$ to a given graph G, produces another graph G' which is always smaller than G with respect to the previously defined ordering of graphs.
Proof. The reverse application of the production p deletes the image of $R\backslash L$ and adds an image of $L\backslash R$, where each element of the two images has the same label as in L or R (cf. definition 2.5). The layering condition of definition 2.12 implies that

$$L < R \Longrightarrow (L\backslash(L \cap R)) < (R\backslash(LcapR)) \Longrightarrow (L\backslash R) < (R\backslash L)$$

and, thereby, that

$$G' = ((G \setminus h(R\backslash L)) \,\cup\, h'(L\backslash R)) < G$$

since h preserves labels of $L\backslash R$ elements and h' preserves labels of $R\backslash L$ elements. This is independent from the fact whether the constructed G' contains dangling edges or not, i.e. is a proper graph or not. □

[7]This is a kind of *multi-set* ordering[4, 8].

The following theorem is a direct consequence of lemma 2.14.

Theorem 2.15 The element problem for a parsable graph grammar gg is decidable due to the fact that even a naive parsing algorithm, which applies productions with exchanged left- and right-hand sides as long as possible and backtracks when necessary, *terminates*.[8]

Proof. The defined ordering of lemma 2.14 guarantees that any sequence of reverse production applications, which starts with a finite graph, has a finite length. This is due to the fact that any derived intermediate graph instance is smaller than its predecessor and has a finite size. Furthermore, graphs of finite size possess a finite number of potential matches (redices) 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 parsable graph grammars is decidable. □

Equipped with these results, we are now able to present a sophisticated parsing algorithm, which *solves the element problem* for parsable graph grammars. Practice will show whether the imposed restrictions for left- and right-hand sides of productions are weak enough, so that all graph languages, which are interesting from a practical point of view, may be generated by means of parsable graph grammars (cf. section 5).

As we will see later on, we cannot use the theorem above directly to prove the termination of the presented parsing algorithm. The problem is that the so-called bottom-up phase of the algorithm generates a *superset* of all valid (reverse) derivation sequences in the general case. Therefore, a separate proof will be necessary to show the termination of the bottom-up phase and, thereby, for the following top-down phase, too (cf. theorem 3.3).

2.5 Dependencies between production instances

Potential production instances (definition 2.7) — or better their induced applications to given host graphs — might depend on each other or exclude each other. In order to be able to reason about these dependencies, we first define some abbreviations:

Definition 2.16 We define the abbreviations **Xlhs**, **common**, **Xrhs**, **lhs**, and **rhs** for a production instance $pi := (p, h, h')$ with $p := (L, R)$ as follows:

- $Xlhs(pi) := h(L \backslash R)$: exclusive left-hand side, the set of deleted graph elements,
- $common(pi) := h(L \cap R) = h'(L \cap R)$: the set of matched but preserved graph elements,
- $Xrhs(pi) := h'(R \backslash L)$: exclusive right-hand side, the set of created graph elements,
- $lhs(pi) := h(L) = Xlhs(pi) \cup common(pi)$: the graph elements matched by the entire left-hand side, and
- $rhs(pi) := h'(R) = Xrhs(pi) \cup common(pi)$: the graph elements matched by the entire right-hand side. □

Two (potential) production instances pi and pi', which deal with common graph elements may not be applied independently of each other. It may happen that pi must be executed before or after pi', and pi may even exclude the application of pi'. Both cases are characterized by the following definitions. Furthermore, the following definitions distinguish the case that a production instance pi' *must* be executed after another production instance pi has been executed, and the case that a

[8]As a consequence parsable graph grammars cannot generate all graph languages, which may be generated by unrestricted graph grammars (for which the left- and right-hand sides of productions are arbitrary graphs). Graph grammars without any restrictions for their left- and right-hand sides are of type 0. It is well-known that the membership problem is undecidable for type 0 languages in the general case [17].

production instance pi' *may* be executed after another production instance pi has been executed. Both cases restrict the order in which pi and pi' are executed, but the second one leaves the question open whether pi' will be executed or not.

Definition 2.17 A production instance pi' is a **consequence** of *another* production instance pi, $pi' \in consequence(pi)$, iff the execution of pi must be followed by the execution of pi', i.e. $pi' \neq pi$ and:

- $Xrhs(pi) \cap Xlhs(pi') \neq \emptyset \quad \vee$
  (pi creates a graph element which is deleted by pi')
- $common(pi) \cap Xlhs(pi') \neq \emptyset$
  (pi needs a context element which is deleted by pi').

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

Definition 2.18 A production instance pi is **above** *another* production instance pi', iff pi must be executed before pi', i.e. $pi \neq pi'$ and:

- $pi' \in consequence(pi) \quad \vee$
- $Xrhs(pi) \cap common(pi') \neq \emptyset \quad \vee$

  (pi creates an element which pi' needs as context element).
- $\exists\, e \in E(Xlhs(pi)), \exists\, v \in V(Xlhs(pi')) :$
  $\quad s(e) = v \vee t(e) = v$

  (pi' deletes a vertex which is source or target of an edge which is removed by pi. In that case pi needs to be applied first in order to avoid dangling edges.)

The restriction of *above* to the consequences of another production instance is defined as follows (and needed in definition 2.20):

- $pi' \; above_{pi} \; pi'' \iff pi', pi'' \in consequence^*(pi) \wedge pi' \; above \; pi''$

The transitive closure of $above_{pi}$ is $\mathbf{above}^{+}_{pi}$. □

To summarize, $pi' \in consequence(pi)$ means that the application of pi must be followed by the application of pi'. 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 *above* pi' is weaker in that it states that if both pi and pi' are applied, then pi must be applied earlier than pi'.

We must deal with transitive closure of the *above* relation in lemma 2.14, but have to be careful here: pi' *above* pi'' only needs to hold if both pi' and pi'' are part of the same derivation sequence. We, therefore, introduced $pi' \; above_{pi} \; pi''$, which has as additional restriction that both pi' and pi'' belong to the consequences of pi, and *must* therefore both be present if pi is present.

Definition 2.19 A production instance pi **excludes** *another* production instance pi' (and vice versa) if both production instances depend on each other or if they add the same elements to a graph (cover same elements), i.e. $pi \neq pi'$ and:

$$\begin{aligned} & pi \; excludes \; pi' \iff \\ & \quad (\, pi \; above \; pi' \wedge pi' \; above \; pi \,) \vee \\ & \quad Xrhs(pi) \cap Xrhs(pi') \neq \emptyset. \end{aligned}$$

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

$$pi\ \mathit{excludes*}\ pi' \iff$$
$$\exists\, \overline{pi} \in \mathit{consequence*}(pi),\ \overline{pi}\,' \in \mathit{consequence*}(pi') : \overline{pi}\ \mathit{excludes}\ \overline{pi}\,'. \qquad \square$$

The intuition behind these definitions is the following: if pi *excludes* pi', then the choice to put pi in the DAG inhibits the addition of pi'. However, incorporating pi might not be a choice, but a necessary *consequence* of an earlier incorporated production instance pi'' (if pi'' creates an intermediate "nonterminal" graph element which must be removed by applying pi afterwards). This leads to the definition of *exclude**, which will make the "real" choice points explicit in the top-down algorithm.

Definition 2.20 A production instance pi might be **inconsistent** with respect to its own consequences:

$$\mathit{inconsistent}(pi) \iff$$
$$pi\ \mathit{excludes*}\ pi \lor$$
$$\exists\, pi', pi'' : pi'\ \mathit{above}^{+}_{pi}\ pi'' \land pi''\ \mathit{above}^{+}_{pi}\ pi'. \qquad \square$$

The definition of the *inconsistency* of a production instance pi covers all those cases, where at least one production instance in $\mathit{consequence}(pi)$ is never executable. This happens if two production instances of the consequences of pi exclude each other or if they are directly or indirectly above each other. In that case, pi creates a graph element which can never be removed by another production instance afterwards, i.e. graph elements which are not useful for the derivation of the given input graph.

2.6 Parse DAGS

Definition 2.21 A set of potential production instances $PI \subseteq \mathcal{PI}(gg)$ for a graph grammar $gg := (A, \mathcal{P})$ is a **parse DAG** for a graph G with respect to a given **completion** $\overline{G}$ of G iff:

1. PI is a DAG with respect to the *above* relation of definition 2.18.
2. The DAG contains one and only one production instance pi_0 of the axiom production (λ, A) as its root.
3. There exists a linear order $PI := \{pi_0, \ldots, pi_n\}$ with $pi_k := (p_k, h_k : L_k \to \overline{G}, h'_k : R_k \to \overline{G})$, which is compatible with the *above* relation such that:
   - $G_0 := Xrhs(pi_0) = rhs(pi_0) = A \subseteq \overline{G}$, the initial graph of gg,
   - $lhs(pi_k) \subseteq G_{k-1}$,
   - $G_k := (\, G_{k-1} \setminus Xlhs(pi_k) \cup Xrhs(pi_k)\,) \subseteq \overline{G}$ is a graph,
   - $G_n = G$, the input for the forthcoming parsing algorithm. $\square$

The definition of a parse DAG is rather complicated. Especially conditions (1) and (2) seem to be superfluous in presence of condition (3). They will be used to guide the construction process of a parse DAG. Unfortunately, these conditions are not sufficient in the general case, so that we cannot drop the additional condition (3).

The introduction of the graph $\overline{G}$ is not understandable without some knowledge about the parsing algorithm. The bottom-up phase of the algorithm does not delete any graph elements in G, but creates a completion $\overline{G}$ of the input graph G. This completion contains any intermediate graph element which might be necessary to derive G step by step. Therefore, it makes sense to require the existence of a graph $\overline{G}$ which contains G as well as the initial graph A and all intermediate derivation results as subgraphs. The graph $\overline{G}$ does not represent the derivation history of G, i.e. it is not equivalent to the parse DAG of G. This is a consequence of the fact that productions over here are not context-free, i.e. left-hand sides of productions do not consist of single vertices only. Henceforth, we do not have a one-to-one correspondence between applied production instances and replaced vertices of left-hand side matches.

Theorem 2.22 Using the vocabulary of definition 2.21 we can prove that for any (parsable) graph grammar gg and any graph G the following holds:

$$\exists\, PI \subseteq \mathcal{PI}(gg) \text{ and } PI \text{ is a parse DAG for } G \iff G \in \mathcal{L}(gg).$$

Proof.

$\Longleftarrow$ It is obvious that any derivation sequence for a given graph $G \in \mathcal{L}(gg)$ fulfills the conditions (2) and (3) of definition 2.21. Furthermore, left-hand sides of productions are never empty (cf. definition 2.12). Therefore, every production instance uses graph elements created by another production instance, and any derivation sequence of a graph forms a connected graph of production instances with respect to the *above* relation. Finally, production instances, which are part of the same graph derivation, never depend on each other. Therefore, the connected graph of production instances is acyclic, too.

$\Longrightarrow$ The condition (3) of the definition of a parse DAG defines already a derivation sequence for a graph G if all the morphisms h_k are redices of L_k in G_{k-1} (and not only potential redices) and all the morphisms h'_k are redices of R_k in G_k (and not only potential redices). But h'_k fulfills the dangling edge condition "per definition". New vertices in G_k may not have incident edges which are not created simultaneously. And h_k violates the dangling edge condition only if G_k contains dangling edges afterwards, i.e. G_k is not a graph. □

3 The graph parsing algorithm

The most important (and most time consuming) part of any graph parsing algorithm consists of searching the graph for a redex of some production. Next, when such a redex is found, the question arises whether the production should be applied or not. Definition 2.6 gave the definition of the application of a production $p := (L, R)$. During parsing this application is reversed and creation of $h'(R\backslash L)$ is replaced by creation of $h(L\backslash R)$, and deletion of $h(L\backslash R)$ is replaced by deletion (or *covering*) of $h'(R\backslash L)$. It can however be the case that the application of this production inhibits the application of another production (in the case of intersecting covers) and it might cause the entire parse to fail. This means that every production instance represents a *choice point* in the algorithm: we have — at least in the general case — the alternatives to use it as part of the constructed derivation sequence or to drop it. The overall searching for a viable derivation should not interfere with the more low-level (and time consuming) searching for redices. We, therefore, opt for an algorithm along the following lines:

Our parsing algorithm is divided into a bottom-up phase and a top-down phase.

- A *bottom-up phase* uses linearizations of the right-hand sides of productions. It searches for redices of productions by moving a *dot* through these linearizations. On the recognition of an entire right-hand side, a production instance *pi* is created, and the elements in *Xlhs*(*pi*) are added to the graph. The addition of new elements to the graph might reactivate suspended dotted rules. The result of the bottom-up phase is the collection *PPI* of all production instances discovered.

- A *top-down phase* composes afterwards subsets of *PPI* which together form correct parse DAG's for the given input graph. 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 at hand is splitted into two derivations, one for each possibility. These derivations are developed in a pseudo-parallel fashion with a preference for depth-first development.

In our approach we 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 consider individual graph elements, but only deals with dependencies between entirely matches of production instances, and combines these into viable derivation sequences. It would in theory be possible to incorporate the work of the top-down phase into the bottom-up phase, but that would complicate the algorithms considerably.

3.1 Preliminary definitions for the bottom-up phase

One of the most severe problems of any graph rewriting system or graph parsing algorithm is to keep track of all potential redices of a given set of productions, and to incrementally construct these while the graph is modified. We apply a method which constructs a *linear search plan* for every production's right-hand side, with a result that predetermines the order in which the redex must be constructed. Another approach to this problem is the RETE approach, which is discussed in Section 6.

The following definitions introduce search plans for right-hand sides of productions without stating anything about how to select a "good" search plan among the many possible search plans for a right-hand side. Section 4 considers afterwards the problem how to estimate the costs of search plans, which provides an indication for this choice.

Definition 3.1 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

- $< head(y : l) >$: 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:

- $< z : x \xrightarrow{\text{k}} (y : l) >$: 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,
- $< z : x \xleftarrow{\text{k}} (y : l) >$: 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
- $< z : x \xrightarrow{\text{k}} y >$: 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; *left*(md) is undefined for the head of a search plan. □

The following sequence of pattern matching directives is a search plan for the production of Figure 2:

$$< head(v_d : DoubleArrow) >$$
$$< e_e : v_d \xrightarrow{\text{ends}} (v_c : Circle) >$$
$$< e_{cov} : v_c \xleftarrow{\text{covers}} (v_s : State) >$$

The conditions for *consistency and completeness* of search plans are rather straightforward and have been omitted in definition 3.1. Consistency means that pattern matching directives correspond to nodes and edges of a production's right-hand side R, vertex identifiers play only once the role of a y variable, and edge identifiers play only once the role of a z variable. Completeness means that the sequence of pattern matching directives covers the whole right-hand side R. It is always possible to create such a search plan due to the requirement in definition 2.12 that R is a connected graph.

As already mentioned above, the set of consistent and complete search plans for a distinct production p is in general quite large. It is rather difficult to find a "best" search plan within this set. The quality of the choice depends to some extend on heuristics of the expected number of vertices and edges with the same label in the considered language of graphs. We will for now assume that a function $SP(p)$ selects at least a "good" search plan. A first attempt to define such a function based on estimated costs of search plans may be found in section 4.

Search plans are used by the first phase of our parsing algorithm, which step by step extends matchings of right-hand sides of productions by "pumping" *dotted rules* through the graph. A dotted rule represents a *partially executed search plan.*

Definition 3.2 A tuple $dr := (p, M, i, h, s)$ is a **dotted rule** with respect to a given graph $\overline{G}$, which is an already constructed completion of an input graph G, iff

- $p := (L, R)$ is a production of a parsable graph grammar,
- $M := SP(p)$ is a sequence $[md_0, \ldots, md_n]$ of matching directives which are a good search plan for p,
- i, with $1 \leq i \leq n$, 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,
- $h : R \rightarrow \overline{G}$ is a partial potential redex of R in $\overline{G}$ with respect to L, binding graph elements of R to already discovered graph elements in $\overline{G}$ 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 $\overline{G}$. If *suspended*, md_i can only be fulfilled when an appropriate edge is added to $\overline{G}$. □

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

3.2 The bottom-up phase of the parsing algorithm

Algorithm 1 (Main loop of bottom-up parsing phase) The bottom-up phase of our parsing algorithm extends matchings of right-hand side of productions step by step by "pumping" dotted rules through the graph. The main loop of the bottom-up parser (algorithm 1) starts with a call to *create-initial-dotted-rules* (sub-algorithm 1.2). Next, it repeatedly checks whether there are dotted rules which can extend their matches (with aid of function *ok*, sub-algorithm 1.1), and if so, calls routine *proceed* (sub-algorithm 1.3) 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* (sub-algorithm 1.4) for all edges created.

```
function CreatePotentialProductionInstances( in $G$ : graph) : set of potential production instances=
  $\overline{G} := G$
  $PPI := \emptyset$                                        (potential production instances)
  for every vertex $v \in \overline{G}$ do
    create-initial-dotted-rules($\overline{G}, v$)
  od
  while $\exists\ v \in \overline{G}$ with $dr := (p, M, i, h, active)$ attached to $v$ do
    $M = [\ldots, md_i, \ldots]$
    if $md_i$ is of the form $< z : x \xrightarrow{\text{k}} (y : l) >$ then
      for every edge $e : v \xrightarrow{\text{k'}} v' \in \overline{G}$ do
        if $ok(h, y \rightarrow v') \wedge ok(h, z \rightarrow e)$ then
          $proceed(\overline{G}, (p, M, i, h \cup \{y \rightarrow v', z \rightarrow e\}, active))$
        fi
      od
    else if $md_i$ is of the form $< z : x \xleftarrow{\text{k}} (y : l) >$ then
      for every edge $e : v \xleftarrow{\text{k'}} v' \in G$ do
        if $ok(h, y \rightarrow v') \wedge ok(h, z \rightarrow e)$ then
          $proceed(\overline{G}, (p, M, i, h \cup \{y \rightarrow v', z \rightarrow e\}, active))$
        fi
      od
    else if $md_i$ is of the form $< z : x \xrightarrow{\text{k}} y >$ then
      for every edge $e : v \xrightarrow{\text{k'}} v' \in G$ do
        if $v' = h(y) \wedge ok(h, z \rightarrow e)$ then
          $proceed(\overline{G}, (p, M, i, h \cup \{z \rightarrow e\}, active))$
        fi
      od
    fi
  change the state of $dr$ from $active$ to $suspended$
od
return $PPI$
```

Sub-algorithm 1.1 (OK — Check identification condition and labels)
Returns true iff the planned extension of the morphism $h : R \rightarrow \overline{G}$ with a binding of the variable var – a vertex or edge identifier in R – to a vertex or edge identifier id in G does not violate the identification condition of definition 2.5 (with respect to a production $p := (L, R)$). Furthermore, the label of the variable var has to be compatible with the label of id (cf. use of isa_V and isa_E in definition 2.4 and labeling condition of definition 2.5).

```
function $ok($ in $h : R \rightarrow \overline{G} : partial\ morphism,$ in $var \rightarrow id : new\ binding) :$ boolean =
  if $\exists\ var' \in R$ with $h(var')$ being defined $\wedge$                (identification condition)
                  $h(var') = id \wedge$
                  $(var' \in R \backslash L \vee var \in R \backslash L)$ then
    return $false$
  else if $var \in R \backslash L \wedge l(id) = l(var) \vee$                  (labeling condition)
        $var \in R \cap L \wedge l(id)\ isa\ l(var)$ then
    return $true$
  else
    return $false$
  fi
```

Sub-algorithm 1.2 (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 $create\text{-}initial\text{-}dotted\text{-}rules($ inout $\overline{G} : graph,$ in $v : vertex) =$
  for every production $p : (L, R) \in gg$ with
```

```
                    search plan $M = [< head(y:l) >, \ldots]$ do
    h := completely undefined (partial) morphism
    if $ok(h, y \rightarrow v)$ then
      attach $(p, M, 1, h \cup \{y \rightarrow v\}, active)$ to $v$ in $\overline{G}$
    fi
  od
```

Sub-algorithm 1.3 (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} needs to originate. Otherwise, the production $p := (L, R)$ 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 $\overline{G}$ : graph, in $(p, M, i, h', s)$ : dotted rule) =
  $M = [md_0, \ldots, md_n]$
  if $i < n$ then
    attach $(p, M, i+1, h', s)$ to $h'(left(md_{i+1}))$
  else
    $h :=$ left-extend$_p(h')$                                    (see definition 2.6)
    if not inconsistent( $(p, h, h')$ ) then                      (see definition 2.20)
      $PPI := PPI \cup \{(p, h, h')\}$
      $\overline{G} := \overline{G} \cup h(L \backslash R)$
      for every vertex $v \in V(h(L \backslash R))$ do
        create-initial-dotted-rules($\overline{G}, v$)
      od
      for every edge $e \in E(h(L \backslash R))$ do
        reactivate-dotted-rules($\overline{G}, e$)
      od
    fi
  fi
```

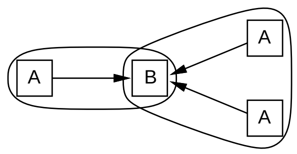
Sub-algorithm 1.4 (Reactivate suspended dotted rules) If we add a new edge e from v to v' to the graph, then it might be the case that there are suspended dotted rules attached to v or v' which can proceed their pattern matching process with this edge. These suspended rules need to be re-activated.

```
proc reactivate-dotted-rules( inout $\overline{G}$ : 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 $< z : x \xrightarrow{k} (y : l) > \wedge$
      $ok(h, y \rightarrow v') \wedge ok(h, z \rightarrow e)$ then
      proceed($\overline{G}, (p, M, i, h \cup \{y \rightarrow v', z \rightarrow e, \}$, active))
    else if $md_i$ is of the form $< z : x \xrightarrow{k} y > \wedge$
          $v' = h(y) \wedge ok(h, z \rightarrow e)$ then
      proceed($\overline{G}, (p, M, i, h \cup \{z \rightarrow e\}$, 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 $< z : x \xleftarrow{k} (y : l) > \wedge$
        $ok(h, y \to v) \wedge ok(h, z \to e)$ **then**
        $proceed(\overline{G}, (p, M, i, h \cup \{y \to v, z \to e\}, active))$
    **fi**
  **od**
```

The simplest way to reactivate a suspended dotted rule would be to change its state from *suspended* to *active* and have it eventually be reconsidered by the main loop of the parser. That would be incorrect however: say, we have a dotted rule dr attached to vertex v_1 which has its dot before an edge labeled by k. Assume furthermore that an edge labeled by k exists in the graph from v_1 to v_2. Then dr may already have been processed by the main loop of the parser, which means that it deposited an incremented version of itself at v_2 and suspended itself. Afterwards, a new edge with label k from v_1 to v_3 is added to the graph. This means that dr needs to be reactivated, but only for the new route to v_3, not for the already inspected route to v_2. That is what happens in algorithm 1.4 above.

3.3 Correctness of the bottom-up phase

Before turning our interest to the top-down phase of the parsing algorithm, we have to show that the algorithms for the bottom-up phase are *correct*. The most difficult part of this proof shows that the algorithm terminates if it is provided with a parsable graph grammar and a finite input graph. Given the layering condition for parsable graph grammars (Definition 2.12), and the fact that the above algorithms take great care not to perform double work (initial dotted rules are only created for newly created nodes, dotted rules only proceed over newly created edges), termination of the bottom-up phase should be rather straightforward. It might however also be the case that a dotted rule proceeds over results which have been generated by alternative matches of itself. This is illustrated by the following production:



Here a successful match of the right-hand side adds elements to the graph which can be used to for yet another successful match of the same right-hand side. This case is prohibited by the check for consistency in routine *proceed* (sub-algorithm 1.3). All of these considerations have to be taken into account in the proof of the termination lemma:

Lemma 3.3 The algorithm 1 *terminates* always, when it is provided with a finite graph G and a parsable graph grammar gg as input. This is equivalent to the fact that the produced set PPI of production instances is *finite*.
Proof. We assume that the produced set PPI is infinite, and we select a production instance $pi \in PPI$ for any natural number n such that:

$$consequence^*(pi) = \{pi_0, \ldots, pi_n\}$$

with

$$pi_k \ above^+_{pi} \ pi_l \ \to \ k \geq l \, .$$

i.e. the consequences of pi may be ordered with respect to the *above* relation restricted to the set of *consequences of pi* (cf. definition 2.18).

This is always possible, since:

- An infinite number of production instances in PPI requires an infinite number of completion steps of the input graph G to an infinite graph $\overline{G}$. Each of these completions relies on graph elements added by previously accomplished completions, i.e. we are able to find *consequence* chains of arbitrary length in PPI.

- The ordering of a *consequence** set of a production instance pi with respect to *above+* is always possible since:

$$pi_k \; above^+_{pi} \; pi_l \;\wedge\; pi_l \; above^+_{pi} \; pi_k$$

  for two different production instances $pi_k, pi_l \in consequence^*(pi)$ is a contradiction to the inconsistency check of sub-algorithm 1.3 and definition 2.20 of an *inconsistent* production instance.

Any sequence $\{pi_0, \ldots, pi_n\}$ defined as above is a "reverse" derivation of a completion $\overline{G}$ of the input graph $G_0 := G$ such that

$$G_{k+1} := G_k \setminus Xrhs(pi_k) \cup Xlhs(pi_k)$$

as long as "dangling edges" are ignored. This is true, since $rhs(pi_k) \subseteq G_k$:

- $\forall\, x \in rhs(pi_k)\;\; x \in G \vee \exists_1\, pi' \in consequence^*(pi_k) \setminus \{pi_k\} \subseteq \{pi_0, \ldots pi_{k-1}\}$:
  $x \in Xlhs(pi')$,

  i.e. all elements needed for the reverse application of pi_k are either already elements of G or they are created by reverse application of $pi_0, \ldots pi_{k-1}$.

- $\forall\, l < k$:
  $x \in Xrhs(pi_l) \rightarrow (pi_l \;\; above \;\; pi_k) \vee (pi_l \;\; excludes \;\; pi_k)$,
  i.e. all needed elements, which are created by reverse application of $pi_0, \ldots pi_{k-1}$ are not removed by $pi_0, \ldots pi_{k-1}$. This would be a contradiction to the initial requirement for the selected sequence of production instances that $pi_l \; above^+_{pi} \; pi_k$ does not hold and that pi_l and pi_k do not exclude each other.

Therefore, we are able to construct reverse derivation sequences of arbitrary length, which is a contradiction to the layering condition of definition 2.10 and lemma 2.14. That means that the initial assumption of infiniteness of PPI was wrong. □

Based on the lemma above and the assumption that the construction of search plans works correctly (cf. definition 3.1), we are now able to prove the *correctness* of the bottom-up phase of our parsing algorithm 1:

Theorem 3.4 Provided with a parsable graph grammar $gg := (A, \mathcal{P})$ and a finite input graph G, algorithm 1 produces a finite set PPI which *contains all potential derivation sequences* of G (up to isomorphism).
Proof. The proof is based on the construction of an inverse graph grammar

$$gg^{-1} := (G, \mathcal{P}^{-1}) \text{ with } \mathcal{P}^{-1} := \{(R, L) \mid (L, R) \in \mathcal{P}\}.$$

We know that

$$G \in \mathcal{L}(gg) \Longleftrightarrow A \in \mathcal{L}(gg^{-1}),$$

since the application of a production is symmetric with respect to its left- and right-hand side and its redices in given host graphs (cf. definition 2.6).

Therefore, it is sufficient to prove that PPI contains any potential derivation sequence of gg^{-1} starting with graph G, under the assumption that production instances of PPI remove their exclusive right-hand sides from a given input graph instead of just completing the input graph with their exclusive left-hand sides:

- *Completeness*: The execution of search plans in the parser's main loop finds all matches (potential redices) of productions' right-hand sides in the current completion $\overline{G}$ of G. The critical step of this part of the proof is to verify that $\overline{G}$ contains "enough" active dotted rules attached to vertices. This is guaranteed by sub-algorithm 1.2, *create-initial-dotted-rules*, and sub-algorithm 1.4, *reactivate-dotted-rules*.

- *Non-redundancy:* The activation and reactivation of dotted rules takes even care that any computed match of a right-hand side differs at least with respect to one graph element from an already computed match. This is a consequence of the fact that a dotted rule is only reactivated, when its next pattern matching directive corresponds to a new graph element.

- *Correctness:* The sub-algorithm 1.1, *ok*, guarantees that all computed matches (redices) for right-hand sides respect the "identification" and the "labeling" condition of definition 2.5 (both conditions for the left-hand sides are guaranteed by the completion process of G in sub-algorithm 1.3 *proceed*). Therefore, any element of PPI is a potential production instance with respect to a regarded completion $\overline{G}$.

- *Termination:* Finally, the constructed set PPI remains finite (cf. lemma 3.3). This has the consequence that the resulting completion $\overline{G}$ is finite, too. Furthermore, any finite graph contains only a finite number of partial matches for a finite number of productions (their right-hand sides). This guarantees the termination of the main loop of algorithm 1. □

Please note that the theorem above does not guarantee that PPI contains indeed a derivation sequence (parse DAG) for the given graph G. Elements of PPI may exclude each other, and G may contain graph elements which are not covered (created) by any production instance of PPI. Even worse, the theorem does not guarantee that elements of PPI, which are not excluded by other production instances, are part of at least one derivation sequence. "Dangling edges" and missing graph elements of left-hand side matches may prevent the execution of production instances. This is a consequence of the fact that the bottom-up phase does not check the "dangling edge" condition, and that a computed completion $\overline{G}$ may contain uncovered "nonterminal" graph elements (production instances of PPI do not create these elements if executed in forward direction). The following top-down phase of the parsing algorithm has to take care of these problems.

3.4 The top-down phase of the parsing algorithm

The overall idea is to start at a production instance for the axiom production and extend the derivation with production instances which do not violate condition (3) of definition 2.21. However, in doing so choices are made: once *pi* is added to the derivation, it is no longer possible to add *pi'* if it is excluded by *pi*. This means that it might happen that a choice in the past turns out to be a wrong one, and that no *pi'* can be found anymore which would not violate condition (3). There are two, quite similar, ways to deal with this situation:

- Perform a *backtrack* at the moment the derivation stucks: go back to the most recent choice and try another alternative. The main drawback of this method is that the backtrack operation itself is expensive to perform as applied production instances have to be undone again.

- It is also possible to develop alternative derivations in *parallel* and simply discard a stuck derivation and continue with an alternative right away. A consequence is that alternative derivations must be stored.

  It is open in a parallel method whether these alternatives are developed in a breadth-first or depth-first way; that just depends on the way in which the next derivation is selected to perform a step in.

It makes sense for both the backtrack method and the parallel method to postpone the application of a production instance which excludes another production instance as long as possible: in that case alternative derivations share as much work as possible before they split.

The backtrack method is most attractive if one is interested in a *single* derivation for the graph; parallel development of alternative derivations is to be preferred over the backtrack method if *all* viable derivations need to be produced, as it theoretically allows to take derivations together at the moment a conflict has been solved.[9] We are mainly interested in finding a single derivation for a graph, but will for sake of clarity of the explanation present an algorithm which creates parse DAG's in parallel. This algorithm works in a depth-first manner and returns the first parse DAG found.

The top-down phase receives the collection of potential production instances *PPI* as generated by the bottom-up phase and needs to produce a viable parse DAG which is a certain subset of *PPI*. The algorithm maintains a collection of active *derivations*:

Definition 3.5 A tuple (G_c, API_c, EPI_c) is a **derivation**. G_c is the graph as built till now by the applied production instances in API_c. The selection of certain production instances excludes other 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 sometimes refer to PPI_c as abbreviation for $PPI \backslash (API_c \cup EPI_c)$, the production instances which can still be applied. □

Algorithm 2 (create-parse-DAG) The top-down algorithm keeps its collection of active derivations in a stack, as this facilitates to pursue 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 two cleanup sub-algorithms (cf. sub-algorithms 2.1 and 2.2) for getting rid of any production instance which is useless from the very beginning of the top-down parsing phase, or which turns out to be useless later on. Furthermore, it calls *Apply* (cf. sub-algorithm 2.3) for computing the effects of a eventually selected production instance within its main loop.

```
function $CreateParseDAG($ in $G : graph,$ in $PPI : set\ of\ possible\ production\ instances) : parse\ DAG =$
  $D := emptystack$
  $PPI := initial\text{-}cleanup(PPI)$
  for every $pi := ((\emptyset, A), h, h') \in PPI$ do
    $D := push(Apply(pi, (\emptyset, \emptyset, \emptyset)), D)$
  od
  while $\neg empty(D)$ do
    $d := (G_c, API_c, EPI_c) = top(D);\ D := pop(D)$
    $d := cleanup(d)$
    $Candidates := \{\ pi \in PPI_c \mid$
        $\exists\, pi' \in API_c : pi'\ above\ pi\ \wedge$
        $\forall\, pi'' \in PPI : pi''\ above\ pi \rightarrow (pi'' \in API_c \vee pi'' \in EPI_c)\ \}$
    if $Candidates = \emptyset \wedge G_c = G$ then
      return $API_c$                                        successful derivation
    else if $Candidates = \emptyset$ then
```

[9]Such as for example done in the Generalized LR parsing algorithm of Tomita[20] for ambiguous textual grammars.

```
      do nothing                                                    dead-end derivation
    else if $\exists\, pi \in Candidates : \neg\exists\, pi' \in PPI_c : pi\ excludes^*\ pi'$ 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
```

Sub-algorithm 2.1 (initial-cleanup) Removes all production instances $pi \in PPI$ with elements in their left-hand sides which are not generated by any other $pi' \in PPI$. These pi are dead-end production instances which can never become part of any successful parse DAG. By removing them, we reduce the search space for the top-down phase.

```
proc $initial\text{-}cleanup($ inout $PPI : set\ of\ possible\ production\ instances) =$
  while $\exists\, pi \in PPI : x \in lhs(pi) \wedge \neg\exists\, pi' \in PPI : x \in Xrhs(pi')$ do
    $PPI := PPI \backslash \{pi\}$
  od
  return $PPI$
```

Sub-algorithm 2.2 (cleanup) Disables all those production instances $pi \in PPI$ with elements in their left-hand sides which are not generated by any other $pi' \in PPI \backslash EPI_c$. Furthermore, it disables those pi which have a vertex v in their left-hand side, while there are edges e in the graph which have v as source or target, and which cannot be deleted anymore by any production instance. Application of pi would otherwise lead to dangling edges.

This routine is a potential source of inefficiency as it deals with individual graph elements and is executed as part of every iteration of the top-down algorithm's main loop. We, therefore, do not consider all production instances for cleanup, but only those which are immediately reachable from the production instances in API_c. All remaining production instances are not yet considered within the forthcoming candidate selection process of the main loop.

```
function $cleanup($ in $d = (G_c, API_c, EPI_c) : derivation) : derivation =$
  while $\exists\, pi \in PPI_c, \exists\, pi'' \in API_c : pi''\ above\ pi\ \wedge$
          $(\ x \in lhs(pi)\ \wedge$
            $\neg\exists\, pi' \in (PPI \backslash EPI_c) : x \in Xrhs(pi')\ )\ \vee$
          $(\ \exists v \in V(Xlhs(pi)),\ \exists e \in E(G_c) :$
                $(s(e) = v \vee t(e) = v)\ \wedge$
                $\neg\exists\, pi' \in PPI_c : e \in E(Xlhs(pi'))\ )$ do
    $d := (G_c, API_c, EPI_c \cup \{pi\})$
  od
  return $d$
```

Sub-algorithm 2.3 (Apply) Returns a derivation d', which is the incoming derivation d on which production instance pi has been applied.

```
function $Apply($ in $pi = ((L, R), h, h') : potential\ production\ instance,$ in $d = (G_c, API_c, EPI_c) : derivation) =$
  $G_n := G_c \backslash h(L) \cup h'(R)$
  $API_n := API_c \cup \{pi\}$
  $EPI_n := EPI_c \cup \{pi' \in PPI \mid pi\ excludes^*\ pi'\}$
  return $(G_n, API_n, EPI_n)$
```

The top-down phase relies heavily on the dependency relations *above* and *excludes*. These can best be pre-computed on the basis of the entire collection of potential production instances PPI during or after the bottom-up phase.

3.5 Correctness of the top-down phase

The top-down algorithm selects the production instance pi, it applies, solely on the basis of checking *above* and *exclude* relations. Therefore, we have to prove that any selected production instance is indeed applicable, i.e. that $pi \in Candidates$ implies:

1. $\neg\exists\, pi' \in API_c : pi' \ excludes\ pi$

   It may not happen that some pi is a candidate, while there exists another $pi' \in API_c$ with $pi'\ excludes^*\ pi$. In that case pi would try to create already existent graph elements. This is trivial, since application of pi' (by sub-algorithm 2.3) is accompanied by moving all production instances which have an *excludes** relation with pi' to EPI_c, the set of already excluded production instances.

2. $\forall\, x \in lhs(pi) : x \in G_c$

   Any selected candidate pi is a production instance whose left-hand side is part of the current graph G_C. We will prove this fact in lemma 3.6.

3. $\forall\, v \in V(Xlhs(pi))$:
   $\neg\exists\, e \in E(G_c) : (s(e) = v \vee t(e) = v)\ \wedge e \notin Xlhs(pi)$

   Any selected candidate pi does not only respect the identification and labeling condition of definition 2.5 but also the required dangling edge condition. In order to ensure this condition, we rely on the fact that the definition of *above* is conscious of these dangling edges, just as the sub-algorithm *cleanup* is. We will prove this fact in lemma 3.7.

Lemma 3.6 Any chosen candidate production instance may be applied to the given host graph, i.e. $pi \in Candidates \Rightarrow \forall\, x \in lhs(pi) : x \in G_c$.
Proof. The proof consists of two parts: (1) every element in $lhs(pi)$ has once been added to G_c, and (2) no element in $lhs(pi)$ has been removed from G_c.

1. The algorithm continuously performs *cleanup*. That guarantees $\forall x \in lhs(pi) : \exists pi' \in PPI \backslash EPI_c : x \in Xrhs(pi')$. This implies in turn that $pi'\ above\ pi$. We also know from the second condition for *Candidate* that $pi' \in API_c \vee pi' \in EPI_c$, which implies that $pi' \in API_c$. The latter implies in turn that all $x \in Xrhs(pi')$ have been added to G_c at the moment pi' was applied and added to API_c.

2. Remains the question whether x is still part of G_c or whether it is already deleted by another production instance, i.e. $\exists\, pi'' \in API_c$ with $x \in Xlhs(pi'')$? In this case we know that $x \in lhs(pi) \wedge x \in Xlhs(pi'')$, which implies one of the following two conditions:

   - $x \in Xlhs(pi) \wedge x \in Xlhs(pi'')$:
     This is impossible due to the way in which the bottom-up phase works (see sub-algorithm 1.3): if the redex of a production instance pi has been recognized, pi is added to PPI and *new* graph elements are created in $\overline{G}$ for $Xlhs(pi)$. This means that two production instances can never have common elements in their $Xlhs$
   - $x \in common(pi) \wedge x \in Xlhs(pi'')$:
     This would imply $pi\ above\ pi''$. However, in that case pi'' is not yet applied; the *candidate* condition for pi'' requires that all production instances *above* have already been applied or excluded. Neither is the case for pi.

This implies that x is still present in G_c. $\square$

Lemma 3.7 None of the chosen candidate productions will cause a dangling edge in the host graph, i.e.:

$$\begin{array}{l} pi \in Candidates \Rightarrow \\ \quad \forall\, v \in V(Xlhs(pi)) : \\ \qquad \neg\exists\, e \in E(G_c) : (s(e) = v \vee t(e) = v) \ \wedge e \notin Xlhs(pi) \end{array}$$

Proof. We assume the contrary:

$$\begin{array}{l} pi \in Candidates \wedge \\ \quad \exists v \in V(Xlhs(pi)) : \\ \qquad \exists e \in E(G_c) : (s(e) = v \vee t(e) = v) \ \wedge e \notin Xlhs(pi) \end{array}$$

The considered edge $e \in G_c$ of the assumption above has a vertex v as source or target which is deleted by the production instance pi. Therefore, v is not a terminal element of the input graph G. As a consequence, e itself may not be a terminal element of G. This implies the existence of another production instance pi' which deletes this edge (introduced this edge during parsing): $\exists\, pi' \in PPI : e \in E(Xlhs(pi')) \wedge (s(e) = v \vee t(e) = v)$.

Due to the structure of the bottom-up phase, we know that only one such pi' may exist. From the definition of *above* (third condition of definition 2.18), we also know that pi' *above* pi holds.

Furthermore, we assume that $e \in G_c$. Therefore, pi' has not been applied yet, i.e. $pi' \notin API_c$. This means that $pi' \in EPI_c \ \vee pi' \in PPI_c$.

- $pi' \in EPI_c \Rightarrow pi' \notin PPI_c$, which means that sub-algorithm *cleanup* would have disabled pi altogether. This contradicts the assumption that $pi \in Candidates$.
- if $pi' \in PPI_c$ ($PPI_c = PPI \backslash (API_c \cup EPI_c)$) then pi would not be disabled by *cleanup*. However, one of the conditions for pi to become a candidate is:

  $$\forall\, pi' \in PPI : pi' \ above\ pi \rightarrow (pi' \in API_c \vee pi' \in EPI_c)$$

  Knowing that $pi' \notin EPI_c$ we can conclude $pi' \in API_c$. However, this implies that e would have been removed from G_c, at the moment pi' was added to API_c. This contradicts with the assumption that $e \in E(G_c)$.

This means that the assumption does not hold and proves the lemma. $\square$

Using the lemmata above and the previously shown correctness of the bottom-up phase, we can now prove the *overall correctness* of our two-phase parsing algorithm.

Theorem 3.8 Let G be a graph with respect to definition 2.2 and gg a parsable graph grammar with respect to definition 2.12. Then

$$\text{algorithms 1 and 2 return a parse DAG for } G \iff G \in \mathcal{L}(gg).$$

Proof. We have to show that any result of the parsing algorithm is a parse DAG for G, that the algorithm finds any existing parse DAG, and that the algorithm terminates under any circumstances. This proof consists of five steps:

1. Theorem 2.22 proves that:

   $$G \in \mathcal{L}(G) \iff \exists \text{ parse DAG } PI \subseteq \mathcal{PI}(gg) \text{ for } G.$$

2. Theorem 3.4 proves already that

$$PI \text{ is a parse DAG for } G \implies PI \subseteq PPI$$

with PPI being the set of possible production instances which is the output of the bottom-up phase.

3. The top-down phase extracts a subset PI from PPI such that PI is either the empty set or:

   - PI is a DAG with respect to the *above* relation and with an instance of the grammar's axiom as its root (cf. algorithm 2).
   - Any incorporated production instance into PI was successfully applied to an intermediate derivation result graph (see lemmata 3.6 and 3.7).
   - Finally, the algorithm's 2 "exit" condition ensures that the returned set PI derives indeed the given graph G.

   As a consequence, any nonempty set PI of the top-down phase is indeed a parse DAG for G.

4. Furthermore, we are able to conclude that algorithm 2 finds any parse DAG $PI \subseteq PPI$, since *excludes** relationships and cyclic *above* relationships are the only reasons which may prevent the incorporation of possibly useful production instances from PPI in PI. The algorithm deals with *excludes** relationships between production instances by testing all possible variations via depth-first search. Furthermore, cyclic *above* relationships may be ignored, since production instances depending on each other may never be part of a valid graph derivation(cf. definition 2.18 and 2.19). Therefore, the algorithm returns the empty set if G is not an element of $\mathcal{L}(gg)$.

5. The termination of the top-down phase is guaranteed by the termination of the bottom-up phase, which implies that its resulting set of possible production instances PPI is finite.

   Furthermore, testing the "parse DAG" property for any subset PI of PPI may be done in finite time. $\square$

4 Possible extensions to the parsing algorithm

Up to now, we explained the basic principles of our parsing algorithm, which has an exponential time and space complexity. The main reasons for its inefficiency are (beside the NP-completeness of the problem it has to solve):

- The number of active dotted rules tends to explode in the algorithm's bottom-up phase, if their underlying search plans are not carefully selected. Section 4.1 discusses a cost function for search plans.

- The number of suspended dotted rules becomes inmanageable, if processing of active dotted rules is such that a large number of dotted rules is waiting for results of other dotted rules. Section 4.2 discusses a priority queue mechanism which minimizes the number of suspended dotted rules and provides a decision criterion for discarding partially recognized dotted rules altogether.

4.1 Selecting good search plans

We need a *cost function* for search plans in order to be able to distinguish between good and bad search plans. The problem with such a cost function is that "“real" cost of processing a search plan depends on the characteristics of a given input graph. We will try to model these characteristics by introducing a number of functions which return probabilities for the existence of certain vertices and edges within a graph. These probabilities will be rough estimates only, since

- probabilities may vary from one instance of a graph language to another instance of the same language or even from one subgraph within an input graph to another subgraph considerably, and
- probabilities for the existence of certain graph elements are not independent of each other but will be treated as being independent for reasons of simplicity.

Nevertheless, experiences with a related approach for defining costs of search plans [25] show that even rather vague estimates of probabilities are very helpful and allow the selection of excellent search plans.

In order to determine the cost of a search plan, we have to determine the probability that a single *search plan directive* (cf. definition 3.1) will succeed. These probabilities can be estimated by averaging over a number of sample graphs, or by some analysis of the regarded graph grammar. Given these probabilities $p(md_i)$, the costs of a search plan is determined by:

$$costs([md_0, md_1, \ldots, md_n]) = p(md_0) * (1 + p(md_1) * (1 + \ldots))$$

This function is such that search plans which start looking for elements with low probability are assigned the lowest cost. This gives preference to search plans which fail as early as possible. For further details about heuristics for the selection of good search plans, the reader is referred to [24].

4.2 A priority queue for active dotted rules

The main loop of algorithm 1 picks the dotted rule to be processed without any preference. However, using our knowledge about label layers and occurrences of labels within productions, a significant improvement of the bottom-up phase is possible.

We extend the parsing algorithm with a priority queue *ActiveRuleQueue* which will be used to store active dotted rules. The dotted rules in *ActiveRuleQueue* are ordered such that (see definition 2.13):

a dotted rule for production p is before another one for p' $\Longleftrightarrow$
$lll(p) < lll(p') \vee (lll(p) = lll(p') \wedge hrl(p) < hrl(p'))$.

Furthermore, we propose as an additional requirement for the assignment of labels to layers that a maximum number of productions p should fulfill the condition $lll(p) > hrl(p)$. In that case, a production's right-hand side matches always graph elements of lower layers then created by its left-hand side. As a result dotted rules which produce graph elements from lower layers will be processed before those which produce elements from higher layers. Processing dotted rules in this order streamlines the parser, as it minimizes the number of dotted rules which need to be suspended in order to be activated later on.

Furthermore, if some dotted rule dr needs an element from layer L_i and the dotted rule at the head of *ActiveRuleQueue* is for production p with $lll(p) > i$, then we are sure that the required element from layer L_i will never be generated anymore and dr can be discarded safely. Checking for this condition takes little time and might reduce the number of suspended dotted rules considerably.

Finally, it seems to be useful to order dotted rules in the *ActiveRuleQueue*, which belong to the same layer, with respect to their remaining search plan costs. Rules with higher *remaining search plan costs* tend to create more dotted rule instances than others. Therefore, prefering dotted rules with minimal remaining costs is another good strategy to minimize the average number of (suspended) dotted rules over time.

5 Example

We take the language of the diagrams of Finite State Automata as an example. We will present a grammar for this language, compute reasonable choices for label layers and search plans, and activate

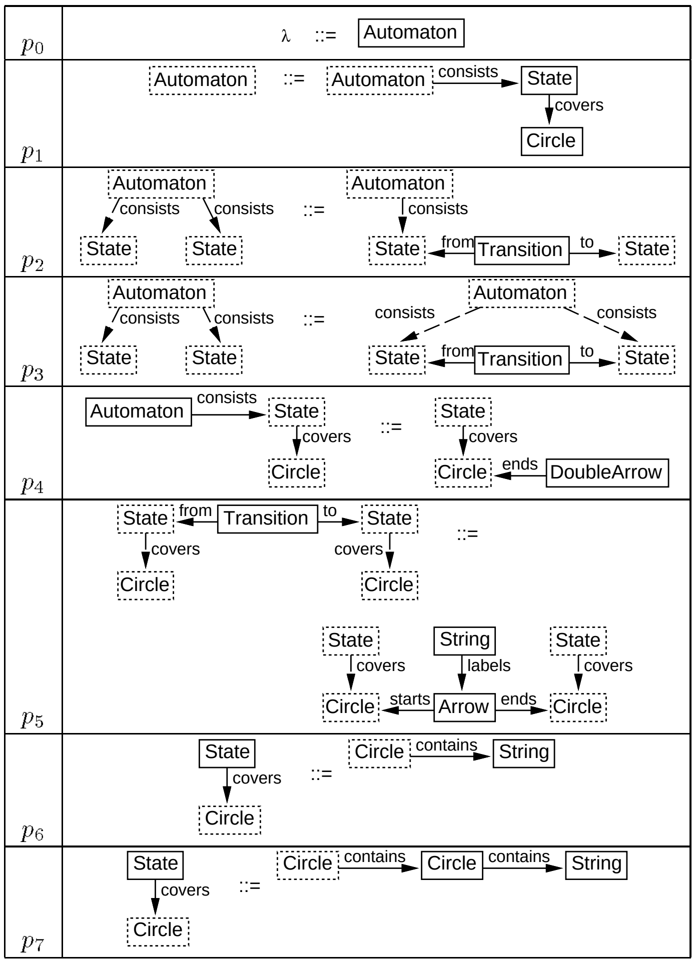

p_0
λ ::= Automaton
p_1
Automaton ::= Automaton consists State covers Circle
p_2
Automaton consists consists State State ::= Automaton consists State from Transition to State
p_3
Automaton consists consists State State ::= Automaton consists consists State from Transition to State
p_4
Automaton consists State covers Circle ::= State covers Circle ends DoubleArrow
p_5
State from Transition to State covers covers Circle Circle ::=
State covers Circle starts Arrow ends Circle covers State String labels
p_6
State covers Circle ::= Circle contains String
p_7
State covers Circle ::= Circle contains Circle contains String

Figure 4: A grammar for Finite-State Automata

the bottom-up parsing algorithm for an example sentence. Finally, we will show how the top-down algorithm finds a a valid parse DAG in the set of potential production applications returned from the bottom-up phase.

5.1 A grammar for FSA's

It turned out to be quite hard to develop the grammar for the reasonably simple language of FSA diagrams, because we had to be careful not to violate the "dangling edge" condition of definition 2.5. Moving to a graph grammar approach with embedding rules would certainly be helpful. In that case it would be possible to redirect and relabel all incident edges of vertices we are going to delete within a single graph rewrite step.

Our grammar for FSA diagrams is given in figure 4, using the same graphical notation as in figure 2 of section 2. Its overall idea is the following: States and transitions are meaningful objects of the underlying abstract syntax of FSA diagrams. Circles, strings, and arrows are just the basic elements of an actual representation of FSA diagrams on a sheet of paper or a screen. Therefore, during generation of these graphs, transition vertices need to be replaced by arrows and their strings, and state vertices by circles and strings (and the other way round during parsing). Unfortunately, we cannot delete a state vertex as long as it is the source or the target of an unknown number of transitions, and we cannot replace transitions between states by arrows between circles, as long as these circles are not already generated. Therefore, states and their circles have to coexists for a while and are connected to each other by temporarily existing *covers* edges.

Productions p_2 and p_3 are very similar. They differ in the fact that p_2 builds a spanning tree of the graph, where p_3 makes this tree into a graph by introducing additional connections between states. We will see in the sequel that various possibilities to identify spanning trees in a graph lead to parsing ambiguities which are hard to avoid.

5.2 Layers induced by the FSA grammar

First of all, we have to assign the labels of this grammar to layers such that for each production $p := (L, R)$, $L < R$ holds (see definition 2.12). We neglected the definition of an algorithm which determines the layer assignments automatically, but the assignments of figure 6 fulfill all conditions imposed by definition 2.12:

The assignment of layers to labels leads to an assignment of the *lowest left layer* $lll(p)$ and the *highest right layer* $hrl(p)$ for each production p, according to definition 2.13. Figure 5 provides the FSA grammar once more in a textual form, extended with identifiers. It also shows the lll and hrl values for all productions, given the above assignment of layers to labels.

This again results in the ordering of *before* relations between dotted rules of various productions, as defined in the context of priority queueing as proposed in section 4.2. For the FSA grammar, this leads to the following *before* relations:

$$\{p_0, p_1, p_3\} > p_2 > p_4 > p_5 > \{p_6, p_7\}.$$

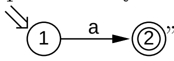
Finally note that the running toy example does not make any use of *isa*-relationships. Therefore, their definitions were omitted over here.

5.3 The chosen search plans

Next, we have to develop the search plans of definition 3.1 for each of the right-hand sides. We opt for the ones as depicted in figure 7 which might not be the most optimal ones according to the cost functions of section 4.1, but allow us to demonstrate more interesting cases in the description of the bottom-up phase.

Note that figure 7 contains two sets of productions with search plans which have a common prefix: $SP(p_2)$ and $SP(p_3)$ share a common prefix of four matching directives, just like $SP(p_6)$ and $SP(p_7)$ share their first two matching directives. It could be worthwhile to exploit this by handling these search plans simultaneously as long as possible, just like it is done in the graph pattern matching algorithm of Bunke [3] and in any LR parsing technique for textual grammars. Whether the increase in efficiency will out-weight the additional complexity of the algorithm depends on the kind of graph grammars commonly processed. Exploiting common prefixes also has consequences for the choice of search plans, which should then also maximize the number of common prefixes among productions. All this needs to be studied in more depth.

5.4 Parsing actual graphs

We will present the steps taken by our algorithms on the spatial relations graph of the following, almost trivial, automaton: " 

". If we provide this diagram to a visual scanner, it generates the graph of vertices and spatial relations between them as depicted in figure 8 (see [15] for a more detailed explanation of spatial relations).

	$Xlhs$	$Common$	$Xrhs$	lll	hrl
p_0	$\emptyset$	$\emptyset$	$v_a : Automaton$	∞	3
p_1	$\emptyset$	$v_a : Automaton$	$v_s : State$ $v_c : Circle$ $e_{cov} : v_s \xrightarrow{\text{covers}} v_c$ $e_c : v_a \xrightarrow{\text{consists}} v_s$	∞	3
p_2	$e_{c2} : v_a \xrightarrow{\text{consists}} v_{s2}$	$v_a : Automaton$ $v_{s1} : State$ $v_{s2} : State$ $e_c : v_a \xrightarrow{\text{consists}} v_{s1}$	$v_t : Transition$ $e_f : v_t \xrightarrow{\text{from}} v_{s1}$ $e_t : v_t \xrightarrow{\text{to}} v_{s2}$	3	3
p_3	$\emptyset$	$v_a : Automaton$ $v_{s1} : State$ $v_{s2} : State$ $e_{c1} : v_a \xrightarrow{\text{consists}} v_{s1}$ $e_{c2} : v_a \xrightarrow{\text{consists}} v_{s2}$	$v_t : Transition$ $e_f : v_t \xrightarrow{\text{from}} v_{s1}$ $e_t : v_t \xrightarrow{\text{to}} v_{s2}$	∞	3
p_4	$v_a : Automaton$ $e_c : v_a \xrightarrow{\text{consists}} v_s$	$v_s : State$ $v_c : Circle$ $e_{cov} : v_s \xrightarrow{\text{covers}} v_c$	$v_d : DoubleArrow$ $e_e : v_d \xrightarrow{\text{ends}} v_c$	3	1
p_5	$v_t : Transition$ $e_f : v_t \xrightarrow{\text{from}} v_{s1}$ $e_t : v_t \xrightarrow{\text{to}} v_{s2}$	$v_{s1} : State$ $v_{c1} : Circle$ $e_{cov1} : v_{s1} \xrightarrow{\text{covers}} v_{c1}$ $v_{s2} : State$ $v_{c2} : Circle$ $e_{cov2} : v_{s2} \xrightarrow{\text{covers}} v_{c2}$	$v_a : Arrow$ $v_t : String$ $e_l : v_t \xrightarrow{\text{labels}} v_a$ $e_s : v_a \xrightarrow{\text{starts}} v_{c1}$ $e_e : v_a \xrightarrow{\text{ends}} v_{c2}$	1	1
p_6	$v_s : State$ $e_{cov} : v_s \xrightarrow{\text{covers}} v_c$	$v_c : Circle$	$v_t : String$ $e_c : v_c \xrightarrow{\text{contains}} v_t$	1	0
p_7	$v_s : State$ $e_{cov} : v_s \xrightarrow{\text{covers}} v_{c1}$	$v_{c1} : Circle$	$v_{c2} : Circle$ $v_t : String$ $e_{c1} : v_{c1} \xrightarrow{\text{contains}} v_{c2}$ $e_{c2} : v_{c2} \xrightarrow{\text{contains}} v_t$	1	0

Figure 5: The grammar of figure 4 extended with identifiers and the assignment of lll and hrl

L_0	contains, String, Circle, Arrow, ends, starts, labels, DoubleArrow
L_1	State, covers
L_2	Transition, from, to
L_3	Automaton, consists

Figure 6: Reasonable definitions of label layers

$SP(p_0)$	$< head(v_a : Automaton) >$
$SP(p_1)$	$< head(v_s : State) >$ $< e_{cov} : v_s \xrightarrow{\text{covers}} (v_c : Circle) >$ $< e_c : v_s \xleftarrow{\text{consists}} (v_a : Automaton) >$
$SP(p_2)$	$< head(v_t : Transition) >$ $< e_t : v_t \xrightarrow{\text{to}} (v_{s2} : State) >$ $< e_f : v_t \xrightarrow{\text{from}} (v_{s1} : State) >$ $< e_c : v_{s1} \xleftarrow{\text{consists}} (v_a : Automaton) >$
$SP(p_3)$	$< head(v_t : Transition) >$ $< e_t : v_t \xrightarrow{\text{to}} (v_{s2} : State) >$ $< e_f : v_t \xrightarrow{\text{from}} (v_{s1} : State) >$ $< e_{c1} : v_{s1} \xleftarrow{\text{consists}} (v_a : Automaton) >$ $< e_{c2} : v_a \xrightarrow{\text{consists}} v_{s2} >$
$SP(p_4)$	$< head(v_d : DoubleArrow) >$ $< e_e : v_d \xrightarrow{\text{ends}} (v_c : Circle) >$ $< e_{cov} : v_c \xleftarrow{\text{covers}} (v_s : State) >$
$SP(p_5)$	$< head(v_a : Arrow) >$ $< e_l : v_a \xleftarrow{\text{labels}} (v_t : String) >$ $< e_s : v_a \xrightarrow{\text{starts}} (v_{c1} : Circle) >$ $< e_{cov1} : v_{c1} \xleftarrow{\text{covers}} (v_{s1} : State) >$ $< e_e : v_a \xrightarrow{\text{ends}} (v_{c2} : Circle) >$ $< e_{cov2} : v_{c2} \xleftarrow{\text{covers}} (v_{s2} : State) >$
$SP(p_6)$	$< head(v_c : Circle) >$ $< e_c : v_c \xrightarrow{\text{contains}} (v_t : String) >$
$SP(p_7)$	$< head(v_{c1} : Circle) >$ $< e_{c1} : v_{c1} \xrightarrow{\text{contains}} (v_{c2} : Circle) >$ $< e_{c2} : v_{c2} \xrightarrow{\text{contains}} (v_t : String) >$

Figure 7: The search plans chosen for the FSA grammar

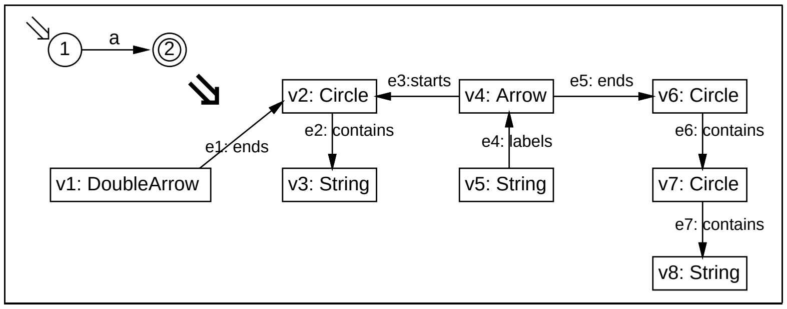


Figure 8: The initial graph G to be parsed

5.4.1 The bottom-up phase

We apply algorithm 1 to the given spatial relations graph and explain important steps of its execution.

- Initially, for all productions with matching head vertex in their search plan the following dotted rules are created in the priority queue $ActiveRuleQueue$:

$$\begin{array}{ll}
(p_4, SP(p_4), 1, \{v_d \rightarrow v_1\}, active) & \Rightarrow ppi_4 \\
(p_5, SP(p_5), 1, \{v_a \rightarrow v_4\}, active) & \Rightarrow ppi_3 \\
(p_6, SP(p_6), 1, \{v_c \rightarrow v_2\}, active) & \Rightarrow ppi_2 \\
(p_6, SP(p_6), 1, \{v_c \rightarrow v_6\}, active) & \text{fails} \\
(p_6, SP(p_6), 1, \{v_c \rightarrow v_7\}, active) & \Rightarrow ppi_1 \\
(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_2\}, active) & \text{fails} \\
(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_6\}, active) & \Rightarrow ppi_0 \\
(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_7\}, active) & \text{fails}
\end{array}$$

  In this queue, the bottom-most dotted rule is the first to be processed, and if new dotted rules are added, the new elements are inserted as low as possible without violating the conditions imposed by the before relations. The second column of the above queue indicates what the next to be described steps of the bottom-up parser will lead to. This provides us with an overview of its forthcoming results.

- The algorithm then takes the first element out of this queue, which is

$$(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_7\}, active)$$

  and for which md_1 is $< e_{c1} : v_{c1} \xrightarrow{\text{contains}} (v_{c2} : Circle) >$ with v_{c1} already bound to v_7 of $\overline{G}$. Vertex v_7 has a single outgoing edge labeled by *contains*, but this edge e_7 has vertex $v8 : String$ as target, which is not as requested by md_1. This means that the dotted rule cannot proceed and can be suspended.

  However, this dotted rule needs an edge and a vertex of layer L_0, and the next to be processed dotted rule is for production p_7. The lowest left layer of p_7 is 1, which means that vertices of layer 0 will never be generated anymore (see explanation in section 4.2). Therefore, the currently regarded dotted rule becomes useless; it can be discarded altogether.

- The next element in the queue is

$$(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_6\}, active)$$

  for which md_1 is the same as above. Now md_1 succeeds, as $\overline{G}$ has an edge $e_6 : v_6 \xrightarrow{\text{contains}} v_7$) to a *circle* node v_7. This causes *proceed* to be called. It discovers that the dot is not at the end yet, and puts the following dotted rule

$$(p_7, SP(p_7), 2, \{v_{c1} \rightarrow v_6, e_{c1} \rightarrow e_6, v_{c2} \rightarrow v_7\}, active)$$

  in the priority queue again. Next, the main loop can discard the original dotted rule for the same reason as the first dotted rule could be discarded instead of suspended.

- The main loop proceeds with the next dotted rule, which is the just added one. Matching directive md_2 of p_7 is $< e_{c2} : v_{c2} \xrightarrow{\text{contains}} (v_t : String) >$, which succeeds. Therefore, *proceed* is called with

$$(p_7, SP(p_7), 2, \{v_{c1} \rightarrow v_6, e_{c1} \rightarrow e_6, v_{c2} \rightarrow v_7, e_{c2} \rightarrow e_7, v_t \rightarrow v_8\}, active)$$

  Now the dot has reached the end which means that p_7 has been applied successfully. This has as consequence that

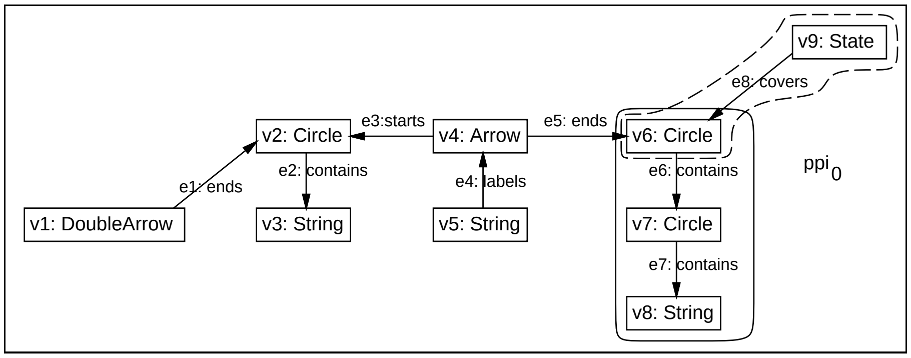


Figure 9: Graph $\overline{G}$ after the first match

- $\overline{G}$ is extended with a vertex $v_9 : \mathit{State}$ and an edge $e_8 : v_9 \xrightarrow{\text{covers}} v_6$.
- PPI is extended with a first potential production instance[10]:

$$\begin{aligned} ppi_0 = \quad (p_7, \quad & \{v_s \rightarrow v_9, e_{cov} \rightarrow e_8\}, \\ & \{v_{c1} \rightarrow v_6\}, \\ & \{v_{c2} \rightarrow v_7, v_t \rightarrow v_8, e_{c1} \rightarrow e_6, e_{c2} \rightarrow e_7\}), \end{aligned}$$

- Algorithm *create-initial-dotted-rules* is called with the new vertex $v_9 : \mathit{State}$. $SP(p_1)$ has $< \mathit{head}(s : \mathit{State}) >$ as its first pattern matching directive, which leads to a new dotted rule in the priority queue:

$$(p_1, SP(p_1), 1, \{v_s \rightarrow v_9\}, \mathit{active})$$

  As p_1 has many productions *before* it, this dotted rule floats to the top of the queue and will only be processed when all other dotted rules have been processed.

Figure 9 shows $\overline{G}$, extended with the new vertex and edge, and it depicts the matching of ppi_0 in it.

- Dotted rule $(p_7, SP(p_7), 1, \{v_{c1} \rightarrow v_2\}, \mathit{active})$ also tries to recognize a final state where there is none, so it fails and can be discarded.
- Dotted rule $(p_6, SP(p_6), 1, \{v_c \rightarrow v_7\}, \mathit{active})$ can be applied and will finally recognize an ordinary state in the inner circle v_7 and the string v_8 of the just recognized final state. This means that $\overline{G}$ is extended with vertex $v_{10} : \mathit{State}$ and edge $e_9 : v_{10} \xrightarrow{\text{covers}} v_7$, and PPI is extended with

$$ppi_1 = (p_6, \{v_s \rightarrow v_{10}, e_{cov} \rightarrow e_9\}, \{v_c \rightarrow v_7\}, \{v_t \rightarrow v_8, e_c \rightarrow e_7\}).$$

  and an initial dotted rule

$$(p_1, SP(p_1), 1, \{v_s \rightarrow v_{10}\}, \mathit{active})$$

  is added to the queue.

  The top-down phase will have to discover that production instances ppi_0 and the just created ppi_1 exclude each other, and are a *choice point* in the parse algorithm. It will also discover that ppi_1 on itself cannot be combined with any of the other production instances to become part of a parse DAG which covers the entire input.

[10]We depict the morphisms of a potential production instance in a different fashion as before, in order to avoid the repetition of the common parts of the left-hand side and right-hand side morphisms

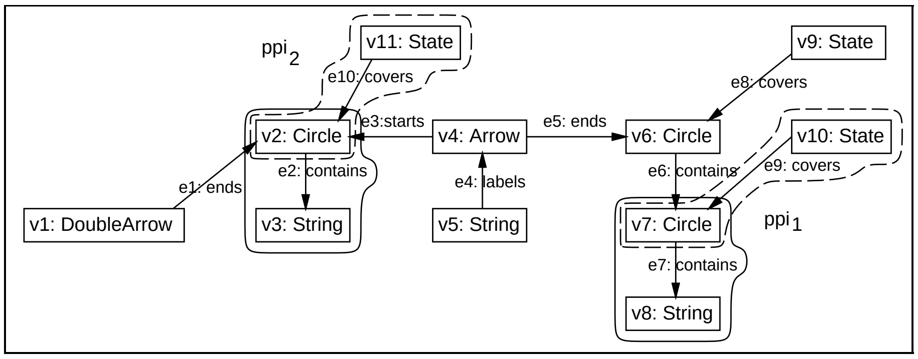


Figure 10: $\overline{G}$ after the states have been recognized

- Dotted rule $(p_6, SP(p_6), 1, \{v_c \rightarrow v_6\}, active)$ fails.
- Dotted rule $(p_6, SP(p_6), 1, \{v_c \rightarrow v_2\}, active)$ can be applied successfully and creates vertex v_{11} and edge e_{10} with Circle v_2 as target. This creates ppi_2 and leads to another dotted rule for production p_1.

  $$ppi_2 = (p_6, \{v_s \rightarrow v_{11}, e_{cov} \rightarrow e_{10}\}, \{v_c \rightarrow v_2\}, \{v_t \rightarrow v_3, e_c \rightarrow e_2\}).$$

  Figure 10 shows $\overline{G}$ after these parsing steps, and shows the matchings of ppi_1 and ppi_2 in it.
- Dotted rule $(p_5, SP(p_5), 1, \{v_a \rightarrow v_4\}, active)$ is next in queue, and it will be able to recognize Transition v_{12} and edges e_{11} and e_{12} in a number of intermediate steps. Furthermore, it extends PPI with

  $$\begin{aligned} ppi_3 = \quad (p_5, \quad & \{v_s \rightarrow v_{12}, e_f \rightarrow e_{11}, e_t \rightarrow e_{12}\}, \\ & \{v_{s1} \rightarrow v_{11}, v_{c1} \rightarrow v_2, v_{cov1} \rightarrow e_{10}, v_{s2} \rightarrow v_9, v_{c2} \rightarrow v_6, v_{cov2} \rightarrow e_8\}, \\ & \{v_a \rightarrow v_4, v_t \rightarrow v_5, e_l \rightarrow e_4, e_s \rightarrow e_3, e_e \rightarrow e_5\}) \end{aligned}$$

  "Transition" is the label of the head of the search plans for productions p_2 and p_3 and the two initial dotted rules are added to the priority queue.
- The next dotted rule to be handled is:

  $$(p_4, SP(p_4), 1, \{v_d \rightarrow v_1\}, active)$$

  which recognizes an Automaton object by matching the DoubleArrow v_1, and it creates

  $$\begin{aligned} ppi_4 = \quad (p_4, \quad & \{v_a \rightarrow v_{13}, e_c \rightarrow e_{13}\}, \\ & \{v_s \rightarrow v_{11}, v_c \rightarrow v_2, e_{cov} \rightarrow e_{10}\}, \\ & \{v_d \rightarrow v_1, e_e \rightarrow e_1\}). \end{aligned}$$

  The new Automaton vertex triggers the creation of an initial dotted rule for the Axiom production p_0:

  $$(p_0, SP(p_0), 1, \{v_a \rightarrow v_{13}\}, active).$$

As a result of the steps above, the *ActiveRuleQueue* now contains:

$$\begin{array}{ll} (p_1, SP(p_1), 1, \{v_s \rightarrow v_9\}, active) & \Rightarrow ppi_9 \\ (p_1, SP(p_1), 1, \{v_s \rightarrow v_{10}\}, active) & \text{fails} \\ (p_1, SP(p_1), 1, \{v_s \rightarrow v_{11}\}, active) & \Rightarrow ppi_8 \\ (p_3, SP(p_3), 1, \{v_t \rightarrow v_{12}\}, active) & \Rightarrow ppi_7 \\ (p_0, SP(p_0), 1, \{v_a \rightarrow v_{13}\}, active) & \Rightarrow ppi_6 \\ (p_2, SP(p_2), 1, \{v_t \rightarrow v_{12}\}, active) & \Rightarrow ppi_5 \end{array}$$

$$
\begin{aligned}
ppi_0 &= (p_7, \{v_9, e_8\}, \{v_6\}, \{v_7, v_8, e_6, e_7\})\\
ppi_1 &= (p_6, \{v_{10}, e_9\}, \{v_7\}, \{v_8, e_7\})\\
ppi_2 &= (p_6, \{v_{11}, e_{10}\}, \{v_2\}, \{v_3, e_2\})\\
ppi_3 &= (p_5, \{v_{12}, e_{11}, e_{12}\}, \{v_{11}, v_2, e_{10}, v_9, v_6, e_8\}, \{v_4, v_5, e_4, e_3, e_5\})\\
ppi_4 &= (p_4, \{v_{13}, e_{13}\}, \{v_{11}, v_2, e_{10}\}, \{v_1, e_1\})\\
ppi_5 &= (p_2, \{e_{14}\}, \{v_{13}, v_{11}, v_9, e_{13}\}, \{v_{12}, e_{11}, e_{12}\})\\
ppi_6 &= (p_0, \emptyset, \emptyset, \{v_{13}\})\\
ppi_7 &= (p_3, \emptyset, \{v_{13}, v_{11}, v_9, e_{13}, e_{14}\}, \{v_{12}, e_{11}, e_{12}\})\\
ppi_8 &= (p_1, \emptyset, \{v_{13}\}, \{v_{11}, v_2, e_{13}, e_{10}\})\\
ppi_9 &= (p_1, \emptyset, \{v_{13}\}, \{v_9, v_6, e_{14}, e_8\})
\end{aligned}
$$

Figure 11: The potential production instances created by the bottom-up phase

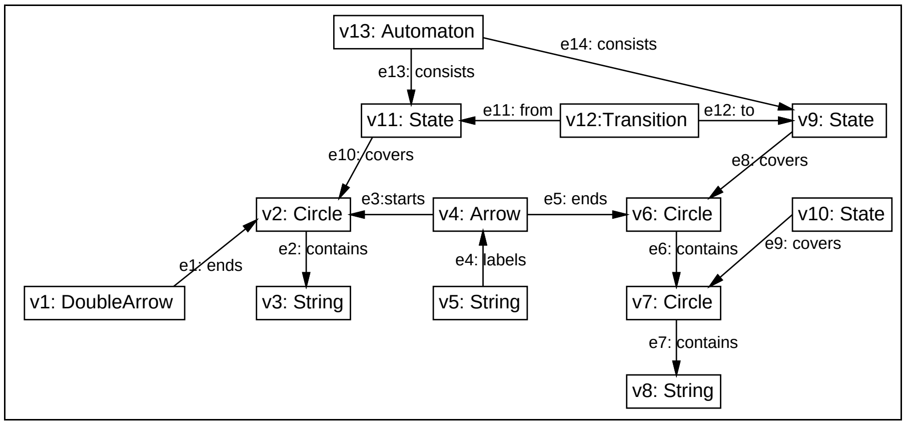


Figure 12: The final graph $\overline{G}$ created by the bottom-up phase

In order to shortcut this exhaustive description of the bottom-up phase, we now provide a complete list of potential production instances created by the continuation of the algorithm in figure 11. In this overview, we have omitted the node and edge identifiers internal to the production instances, as we do not need them anymore in the sequel of the discussion. The final graph $\overline{G}$ is shown in figure 12.

5.4.2 The dependency relations

The potential production instances of figure 11 as created by the bottom-up phase lead to the dependency relations between production instances as depicted in Figure 13. These dependencies are not manually computed but were created by a Prolog implementation of their definitions in section 2. The top-down phase needs just the relations *above* and *excludes**. Nevertheless, we provide all of others too in order to illustrate the dependencies between them and to allow the reader to check the presented results.

These dependency relations reveal that ppi_7 is *inconsistent*, and should have been discarded right away by the bottom-up phase. This clearly shows that dependency relations should be developed in an incremental manner during the bottom-up phase, instead of afterwards as we do in this example.

$conseq(ppi_3, ppi_0)$	$conseq(ppi_9, ppi_5)$	$above(ppi_7, ppi_4)$	$excludes^*(ppi_1, ppi_0)$
$conseq(ppi_3, ppi_2)$	$above(ppi_0, ppi_1)$	$above(ppi_7, ppi_5)$	$excludes^*(ppi_1, ppi_3)$
$conseq(ppi_4, ppi_2)$	$above(ppi_3, ppi_0)$	$above(ppi_8, ppi_2)$	$excludes^*(ppi_1, ppi_5)$
$conseq(ppi_5, ppi_0)$	$above(ppi_3, ppi_2)$	$above(ppi_8, ppi_3)$	$excludes^*(ppi_1, ppi_7)$
$conseq(ppi_5, ppi_2)$	$above(ppi_4, ppi_2)$	$above(ppi_8, ppi_4)$	$excludes^*(ppi_1, ppi_9)$
$conseq(ppi_5, ppi_3)$	$above(ppi_5, ppi_0)$	$above(ppi_8, ppi_5)$	$excludes^*(ppi_3, ppi_1)$
$conseq(ppi_5, ppi_4)$	$above(ppi_5, ppi_2)$	$above(ppi_8, ppi_7)$	$excludes^*(ppi_5, ppi_1)$
$conseq(ppi_6, ppi_4)$	$above(ppi_5, ppi_3)$	$above(ppi_9, ppi_0)$	$excludes^*(ppi_5, ppi_7)$
$conseq(ppi_7, ppi_0)$	$above(ppi_5, ppi_4)$	$above(ppi_9, ppi_3)$	$excludes^*(ppi_7, ppi_1)$
$conseq(ppi_7, ppi_2)$	$above(ppi_6, ppi_4)$	$above(ppi_9, ppi_4)$	$excludes^*(ppi_7, ppi_5)$
$conseq(ppi_7, ppi_3)$	$above(ppi_6, ppi_5)$	$above(ppi_9, ppi_5)$	$excludes^*(ppi_7, ppi_7)$
$conseq(ppi_7, ppi_4)$	$above(ppi_6, ppi_7)$	$above(ppi_9, ppi_7)$	$excludes^*(ppi_7, ppi_9)$
$conseq(ppi_7, ppi_5)$	$above(ppi_6, ppi_8)$	$excludes(ppi_0, ppi_1)$	$excludes^*(ppi_9, ppi_1)$
$conseq(ppi_8, ppi_2)$	$above(ppi_6, ppi_9)$	$excludes(ppi_1, ppi_0)$	$excludes^*(ppi_9, ppi_7)$
$conseq(ppi_8, ppi_4)$	$above(ppi_7, ppi_0)$	$excludes(ppi_5, ppi_7)$	$inconsistent(ppi_7)$
$conseq(ppi_9, ppi_0)$	$above(ppi_7, ppi_2)$	$excludes(ppi_7, ppi_5)$	
$conseq(ppi_9, ppi_4)$	$above(ppi_7, ppi_3)$	$excludes^*(ppi_0, ppi_1)$	

Figure 13: The dependency relations between the potential production instances of figure 11

5.4.3 The steps of the top-down phase

Input for the top-down phase are the potential production instances of figure 11 minus the inconsistent production instance ppi_7, and the dependency relations of figure 13. In the sequel, we will refer to graphs G_i, which are all depicted in figure 14.

- The top-down phase starts with the execution of *cleanup*, which removes ppi_1 from PPI, as this production instance has graph elements v_{10} and e_8 in its left-hand side, while there is no production instance which has these in its right-hand side. All remaining production instances are part of the finally constructed derivation, so that all forthcoming cleanup attempts will have no effects.

  This means that PPI, which will be used in the rest of the algorithm, contains:

  $$\{ppi_0, ppi_2, ppi_3, ppi_4, ppi_5, ppi_6, ppi_8, ppi_9\}.$$

- The only production instance for the axiom production p_0 is ppi_6, so we start with a single derivation:

  $$D := \{(G_0, \{ppi_6\}, \emptyset)\}$$

- This derivation is popped from stack. The set of candidate production instances is $\{ppi_8, ppi_9\}$, neither of which has an *excludes** relation. We choose ppi_8 to apply. This leads to the following derivation to be pushed on stack:

  $$(G_1, \{ppi_6, ppi_8\}, \emptyset).$$

- Now $Candidates = \{ppi_9\}$ which can simply be applied to create the following derivation:

  $$(G_2, \{ppi_6, ppi_8, ppi_9\}, \emptyset)$$

- For this derivation $Candidates$ is $\{ppi_5\}$. Application of ppi_5 leads to the following derivation to be pushed on stack:

  $$(G_3, \{ppi_6, ppi_8, ppi_9, ppi_5\}, \emptyset).$$

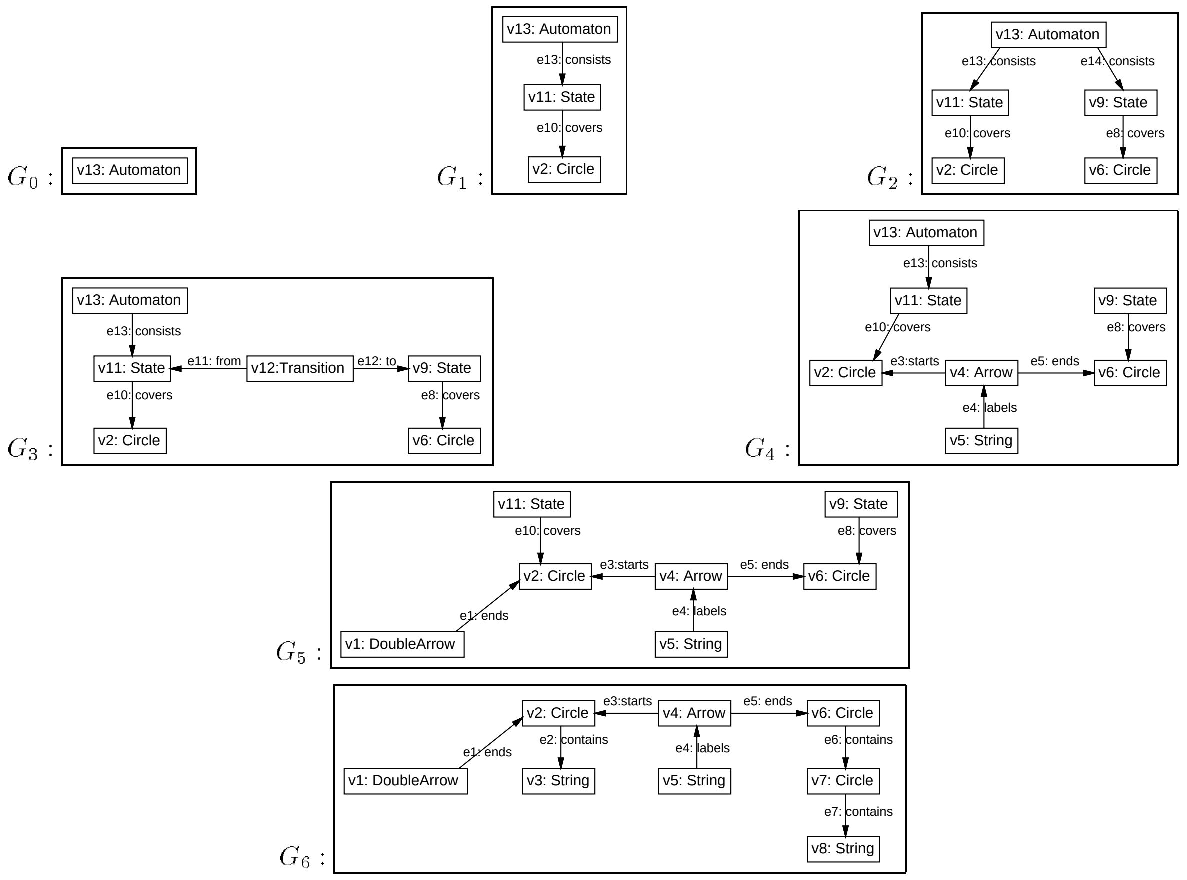


Figure 14: Various intermediate graphs of the top-down phase.

- The latter is again the first derivation to be popped, and the *above* relations indicate two candidate production instances for it, ppi_3 and ppi_4, neither of which have an *excludes** relation. We randomly choose ppi_3 to apply, which leads to the following derivation:

$$(G_4, \{ppi_6, ppi_8, ppi_9, ppi_5, ppi_3\}, \emptyset).$$

- Now ppi_0 and ppi_4 are candidates, of which we choose ppi_4 to apply:

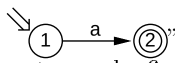
$$(G_5, \{ppi_6, ppi_8, ppi_9, ppi_5, ppi_3, ppi_4\}, \emptyset).$$

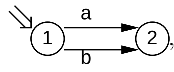
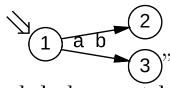
- When we pop this derivation from stack again, the candidates are ppi_0 and ppi_2, which we apply in two steps to come up with the derivation:

$$(G_6, \{ppi_6, ppi_8, ppi_9, ppi_5, ppi_3, ppi_4, ppi_0, ppi_2\}, \emptyset).$$

- For this derivation, $Candidates = \emptyset$ and $G_c \equiv G$, which means that the following sequence of production instances constitutes a successful derivation:

$$ppi_6, ppi_8, ppi_9, ppi_5, ppi_3, ppi_4, ppi_0, ppi_2$$

The generation of this parse DAG was extremely straightforward: all production instances of *PPI* have been applied, and we simply had to find the right order to do so. This was to be expected, as it is quite hard to find alternative derivations in the input graph “ ”. Still, the top-down phase did have a number of choices in which production instance to apply first, but none of these choices lead to alternative derivations, as the top-down phase knew from the *excludes** relations that these choices were non-critical.

It would be interesting to look into the handling of ambiguous FSA diagrams, such as the diagram ” ”. This diagram will be ambiguous as two possible spanning trees can be identified. Our top-down phase is able to generate both interpretations. The diagram ” ” is interesting also, as it contains a lexical ambiguity: it is unclear to which arrows the labels must be bound. Our graphical parsing algorithm is almost able to handle such ambiguities, we only have to replace the test $G_c = G$ of algorithm 2 by the less strict equality: $V(G_c) = V(G) \wedge E(G_c) \subseteq E(G)$. We will not discuss these examples however, as this section is too long already.

6 Related work

Up to now, only a “handful” of proposals are published on how to parse graph-like data structures generated by graph grammars [10, 16, 12, 21] or related formalisms like plex grammars [3], relational fringe grammars [22], or picture layout grammars [7]. These approaches fall into two classes with respect to the overall organization of the parsing algorithm. On one side, we have *Earley-style* [5] approaches [3, 22] which start at a single node of the given input graph and extend the already examined part of the graph step by step. Each extension step is guided by the structure of the input graph. It consumes one element of the input graph and updates a set of *all* partially recognized production instances. Afterwards, the current set of production instances is *completed* with new production instances, which might be needed in the future to extend already found partial matches or to process the eventually forthcoming results of the current set of production instances. In contrast, all remaining approaches – including our approach – are *parallel bottom-up* algorithms such as the

Paper	Left Side	Context	Embedding	Restrictions	Complexity
Kaul[11]	1 node	no	restricted	severe	linear
Golin[7]	1 node	terminal	no	severe	poly.
Rozenberg/Welzl[16]	1 node	no	restricted	severe	poly.
Lamshoeft[12]	1 node	no	complex	moderate	exp.
Wills[21]	1 node	no	complex	unclear	exp.
Wittenburg[22]	1 node	no	restricted	severe	exp.
Bunke/Haller[3]	1 node	no	restricted	no	exp.
our algorithm	graph	yes	no	layering	exp.

Table 1: A comparison between various graph parsing algorithms

Generalized LR algorithm by Tomita [20] and the Cocke-Younger-Kasami algorithm [23, 9] for context-free textual grammars, which process all nodes of the input graph simultaneously. They update a subset of all partial matches of a single production in each basic recognition step.

Both solutions have significant advantages and disadvantages, and further investigations are necessary to compare the effectiveness of their accompanying heuristics. Earley-style approaches start with a small set of partially recognized production instances, but this set tends to explode in the algorithm's completion phase. They have to keep track of an exponentially growing set of all eventually forthcoming partial matches. Bottom-up algorithms, on the other hand, start with a large set of production instances which is proportional to the size of the graph. It depends on the quality of selected search plans how fast the initial set of production instances grows (or shrinks). Nevertheless, it is often the case that the set of all finally created production instances has a size proportional to the number of input graph elements. Unfortunately, these bottom-up algorithms need a second phase to extract a consistent set of production instances out of the set of all production instances, created in the first phase.

When studying the above mentioned approaches in more detail we have to consider the following questions about their parsing algorithms and their underlying grammar formalisms:

- Is the left-hand side of a production restricted to a *single (nonterminal) node*, which will be replaced by its right-hand side (context-free production)?
- Allows the formalism references to additional *context-elements* which have to be present but remain unmodified during the application of a production?
- Has the proposed type of grammar 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, like "bounded edge degree" or our "layering condition"?
- Is the time and space *complexity* of the proposed algorithm linear, polynomial, or even exponential with respect to the size of the input graph?

Table 1 provides an overview of our related work studies with respect to the above mentioned questions.

Kaul [10, 11] introduced a parsing algorithm with linear time and space complexity. This algorithm is able to deal with so-called *precedence graph grammars*. Their productions have a single node as their left-hand side (and no context elements) and use a restricted form of monotonic embedding rules to get around the "dangling edge problem". The parsing process is a kind of handle rewriting, where graph handles (subgraphs of the input graph) are identified by means of three disjoint precedence relationships. These relationships are sets of (vertex label, edge label, vertex label) triples. Unfortunately, the "disjoint precedence relationship" requirement is very restrictive, and prevents the definition of many interesting graph languages by means of precedence graph grammars.

The other two remaining approaches with non-exponential time and space complexity impose also more or less severe restrictions on the form of allowed productions and their graph languages. The parsing approach for picture layout grammars by Golin [7] considers a data model with attributed objects but without explicit relationships between them. All relationships between objects (nodes) are implicitly defined by means of "matching" attribute values. The proposed parsing algorithm works if the following conditions are met:

- Two objects with the same label, created during the derivation of a language element have at least one different attribute value.
- The number of used attribute values within derivations is restricted, a requirement which is more or less equivalent to a bounded edge degree requirement.
- An additional restriction concerning the use of context elements has to be fulfilled: only terminals may be used as context elements.
- Covering conflicts due to ambiguous (sub-)derivations do not occur, since they cause the parser's immediate termination with a fatal error.

The main problem with these restrictions is that they are not checked before runtime. Even worse, their violation may lead to nonterminating computations or to wrong parsing results.

Rozenberg and Welzl [16] propose a parsing algorithm with polynomial complexity for graphs with unlabeled edges and bounded edge degree. It does not consider the question how to organize pattern matching for right-hand sides. Furthermore, its context-free productions with rather primitive embedding rules are restricted as follows: nonterminals in their right-hand sides may not be neighbors.

The approach presented by Wills [21] is the most ambitious one of those which remain for discussion. Its embedding rules are fairly general, and it even uses linear search plans in the form of dotted rules for guiding the pattern matching process. Unfortunately, the presentation of the parsing algorithm is on a very informal level and necessary restrictions for guaranteeing correctness and termination are not precisely defined.

The master thesis of Lamshoeft [12] suggests a number of extensions to the parsing algorithm of Rozenberg and Welzl [16]. By means of these extensions, the parser is able to deal with a more general class of context-free graph grammars with rather powerful embedding rules. This class of graph grammars has no restrictions for the right-hand sides of productions, but requires that the grammar is confluent with respect to the effects of embedding rules.

The following two entries in the table contain references to "Earley-style" parsing approaches. The first one by Wittenburg [22] uses dotted rules as we do, but without presenting any heuristics how to select good ones. Furthermore, it is restricted to the case of graphs, where sets of edges with the same label define partial orders, i.e. do not build cyclic paths in the graph. The second 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. Both approaches have an exponential worst case complexity, but they are claimed to be reasonably efficient in the average case.

Both approaches are somehow related so-called *RETE-network* matching techniques [2], which are the most essential part of the execution machinery of the rule-oriented language OPS-5 [1]. Such a network stores information about all potential matches of all considered productions at the same time. Its main advantages are:

- A single (sub-)network finds the matches for all isomorphic subgraphs of all relevant graph patterns[11] in parallel.
- The network contains information about all potential partial matches of all graph patterns.

[11] A "graph pattern" is a production's left-hand side for graph generation and a production's right-hand side for graph parsing

The second advantage is also the main disadvantage of this approach. It has the consequence that the network has at least in the general case a number of states, which grows exponentially with the size of its graph patterns. Even worse, all of these states carry a number of tokens which are proportional to the number of tokens of their direct predecessor states times the edge fan-out at certain vertices. Otherwise, the network would not be able to keep track of all recognized partial matches at the same time.

Our parsing approach avoids this problem by constructing a *linear search plan* instead of a network of search plans. Such a search plan is a path within a RETE-network which contains one state for each element of a considered graph pattern. The main drawback of this solution is its ignorance of partial matches which are not "compatible" with the selected search plan. On the other hand, the concentration on a distinct path in a RETE-network — instead of pursuing them all in parallel — has an invaluable advantage: we are able to get rid of bad search plans corresponding to linear paths in the RETE-network which create (too) many tokens. In practice, it is often the case that the time and space complexity of linear search plans for a fixed graph pattern of size m and a host graph of size n varies from $O(n)$ to $O(n^m)$.

To summarize, all presented approaches are either very restricted with respect to the graph languages they are able to define or have an exponential worst case complexity. Furthermore, average case complexity results with respect to suggested heuristics are more or less unknown for all approaches. Therefore, we are not able to provide the reader with more detailed information about the efficiency of discussed parsing algorithms, when they are applied to "real world" examples.

7 Conclusions and future work

Graph grammars are a very powerful tool for defining the syntax of visual languages or for defining transformation processes between instances of different visual languages in the form of pair graph grammars [19]. Their main drawback until now was the lack of nice notations and, especially, of efficiently working parsing algorithms. All algorithms, which were discussed in the previous section, are only able to deal with context-free graph grammars, where the left-hand side consists of a single nonterminal vertex only. This makes the syntax definitions of visual languages hard to read and prohibits the definition of complex pattern matching and transformation processes at all.

Our graph grammar parsing algorithm is the first one for productions which

- may delete more than one nonterminal node at the same time,
- may delete nothing at all, i.e. extend the graph only,
- take care of label hierarchies, and
- may require arbitrarily complex context graphs.

We were even able to define a class of *parsable graph grammars* together with an efficiently implementable decision criterion whether a given graph grammar is parsable or not. Furthermore, we presented a proof that *termination and correctness* of the presented parsing algorithm is guaranteed for any given parsable graph grammar and any input graph. Finally, we suggested many *heuristics* on how to reduce the overall search space of the algorithm, and how to prune failing derivation attempts as early as possible. Their effectiveness as well as the overall efficiency of the proposed parsing algorithm is unknown up to now. The algorithms need to be implemented and analyzed by feeding them with carefully selected test cases, as well as "real world" examples. Their implementation will be part of a parsing toolkit for visual languages[15], and of the already existing graph grammar programming environment PROGRES [24, 18, 14]

Finally note that our layering condition, which guarantees the algorithm's termination, imposes only very liberal restrictions for the definition of graph languages. But we have to admit that the *dangling edge condition* together with the *lack of embedding rules* makes the definition of many graph languages really awkward. Therefore, an extended version of our parsing algorithm is in preparation which deals with *embedding rules*, too. By means of these embedding rules productions will be able to redirect and relabel sets of edges from deleted nodes to created nodes.

Acknowledgements

We would like to thank Joost Engelfriet and Gabi Zinßmeister for their comments on earlier versions of this paper. We would like to thank Gregor Engels for bringing us together.

References

[1] L. Brownston, R. Farell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5 – An introduction to rule-based programming.* Addison-Wesley, 1986.

[2] H. Bunke, T. Glauser, and T.-H. Tran. An efficient inplementation of graph grammars based on the RETE matching algorithm. In *Proceedings 4th international workshop on Graph Grammars and their application to Computer Science*, LNCS 532, pages 174–189, 1990.

[3] H. Bunke and B. Haller. A parser for context free plex grammars. In M. Nagl, editor, *Proceedings 15th international workshop on Graph-Theoretic concepts in Computer Science – WG'89*, LNCS 411, pages 136–150, 1989.

[4] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[5] J. Earley. An efficient context-free parsing algorithm. *CACM*, 13(2):94–102, 1970.

[6] H. Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings international workshop on Graph-Grammars and their application to Computer Science and Biology*, LNCS 73, Berlin, 1979.

[7] E.J. Golin. *A method for the specification and parsing of visual languages.* PhD thesis, Brown University, May 1991.

[8] J.-P. Jouannaud and P. Lescanne. On multiset orderings. *Information Processing Letters*, 15(2):57–63, 1982.

[9] 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.

[10] M. Kaul. Parsing of graphs in linear time. In Ehrig, Nagl, and Rozenberg, editors, *2nd international workshop on Graph Grammars and their application to Computer Science*, LNCS 153, pages 206–218, 1982.

[11] M. Kaul. *Syntaxanalyse fuer Praezedenzgraphgrammatiken.* PhD thesis, Universität Passau, 1985. In german.

[12] Th. Lamshoeft. Ein parser für graphgrammatiken. Master's thesis, Fachbereich Informatik, Universität Passau, 1993. In german.

[13] M. Nagl. *Graph-Grammatiken.* Vieweg Verlag, Braunschweig, 1979. In german.

[14] PROGRES – *PROgramming with Graph REwriting Systems.* Software package, available by ftp from *ftp-i3.informatik.rwth-aachen.de:/pub/PROGRES*; see also *http://www-i3.informatik.rwth-aachen.de/research/progres.html.*

[15] J. Rekers. On the use of graph grammars for defining the syntax of graphical languages. In *Proceedings of the colloquium on Graph Transformation*, Palma de Mallorca, Spain, 1994. Also available from ftp site *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1994/tr94-11.ps.gz.*

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

[17] A. Salomaa. *Formal Languages.* ACM Monograph Series. Academic Press, New York, 1973.

[18] A. Schürr. Rapid programming with graph rewrite rules. In *USENIX symposium on Very High Level Languages – VHLL'94*, pages 83–100, 1994.

[19] A. Schürr. Specification of graph translators with triple graph grammars. In Mayer and Tinhofer, editors, *Proceedings international workshop on Graph-Theoretic concepts in Computer Science – WG'93*, will appear in LNCS, 1995.

[20] M. Tomita. *Efficient Parsing for Natural Languages.* Kluwer Academic Publishers, 1985.

[21] L.M. Wills. *Automated Program Recognition by Graph Parsing.* PhD thesis, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1992. Technical Report 1358.

[22] K. Wittenburg. Earley-style parsing for relational grammars. In *Proceedings IEEE workshop on Visual Languages – VL'92*, pages 192–199, 1992.

[23] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

[24] A. Zündorf. Eine entwicklungsumgebung für programmierte graphersetzungssysteme. PhD thesis, RWTH Aachen, in German, forthcoming.

[25] A. Zündorf. A heuristic solution for the (sub-)graph isomorphism problem in executing PROGRES. Technical Report AIB 93-5, RWTH Aachen, 1993.