Improving the graph grammar parser of Rekers and Schürr

Luka Fürst[1], Marjan Mernik[2], Viljan Mahnič[1]

[1] *University of Ljubljana, Faculty of Computer and Information Science, Tržaška cesta 25, SI-1000 Ljubljana, Slovenia*
E-mail: {luka.fuerst, viljan.mahnic}@fri.uni-lj.si

[2] *University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, SI-2000 Maribor, Slovenia*
E-mail: marjan.mernik@uni-mb.si

Abstract

Graph grammars and graph grammar parsers are to visual languages what string grammars and parsers are to textual languages. A graph grammar specifies a set of valid graphs and can thus be used to formalise the syntax of a visual language. A graph grammar parser is a tool for recognising valid programs in such a formally defined visual language. A parser for context-sensitive graph grammars, which have proved to be suitable for formalising real-world visual languages, was developed by Rekers and Schürr. We propose three improvements of this parser. One of them enlarges the class of parsable graph grammars, while the other two increase the parser's computational efficiency. Experimental results show that for some (meaningful) graph grammars, our improvements can enhance the parser's performance by orders of magnitude. The proposed improvements will hopefully increase both the parser's applicability and the interest in visual language parsing in general.

1 Introduction

Graph grammars consist of rules (called *productions*) for matching and replacing subgraphs in an arbitrary host graph and can thus be regarded as a generalisation of 'ordinary' (i.e., string) grammars. Analogously to string grammars, each graph grammar defines its *language* as a set of all graphs that can be generated by its productions. Graph grammars have thus been used for specifying the syntax of visual languages [1, 2, 3], but they have also been applied to software systems modelling [4], pattern recognition [5, 6], mathematical formula recognition [7], generation of metamodel instances [8], and to many other domains [9, 10]. While context-free string grammars are employed far more frequently than context-sensitive ones in the domain of textual languages, visual languages can often be more naturally described by context-sensitive graph grammars than by context-free ones [1].

The formalisation of a textual or visual language by means of a grammar is not an end in itself. A grammar is the basis for the automated recognition of valid sentences of the language (i.e., valid strings in the case of a textual language and

valid graphs in the case of a visual language) and for their translation into some other meaningful form. (Consider, for instance, the translation of Java language sentences – i.e., Java programs – into bytecode.) A *parser* is a tool that determines whether a given sentence belongs to a given grammar and, if this is indeed the case, derives the sentence using the grammar's productions. The parser's output can be used for various purposes, including translation.

In this paper, we present an improved version of the graph grammar parser developed by Rekers and Schürr [11, 1, 12]. This parser recognises the languages of *layered graph grammars* (LGGs), which can be regarded as a natural generalisation of standard context-sensitive string grammars (CSGs). Like CSGs but unlike context-free grammars, every LGG production may specify its *context* in addition to a replacement rule. When applying an LGG production p to a host graph G, the elements forming the context have to be present in G but are not affected by the application of p. A further similarity between CSGs and LGGs is a rule stipulating that for every production, its left-hand side (LHS) has to be 'smaller' than its right-hand side (RHS); however, in the case of LGGs, the meaning of 'smaller' is determined by a fairly complex condition.

Since the parsing problem is NP-complete even for very limited classes of graph grammars [13], the Rekers-Schürr parser generally takes exponential time to parse (i.e., to recognise and derive) a given graph. Even more, experiments revealed that the parser is, in some cases, impractical even for small-sized graphs and graph grammars. For this reason, we decided to improve the parser's efficiency.

In this paper, we present three improvements of the Rekers-Schürr parser. As a first improvement, we abolished the requirement that the productions' RHSs be connected and thus enlarged the class of parsable graph grammars. The other two improvements enhance the parser's performance. Although they do not make the parser run in a polynomial time for an arbitrary input (this is actually impossible unless P = NP), they can reduce the parse time by orders of magnitude for some graph grammars. We experimentally evaluated both the original parser and our improvements using meaningful graph grammars.

Our work may benefit the software engineering community in the following ways:

- An efficient parser can serve as a building block in various visual programming tools, such as compiler, debugger, profiler, etc.

- Besides several other application domains, graph grammars have found their use in UML-based modelling [8]. Specifically, *triple graph grammars* (TGGs) [14, 15] were invented for the purpose of solving certain modelling-related tasks (such as model transformation, model integration, etc.) within a graph grammar framework. Some TGG-related problems, such as finding a correspondence between two models, involve operations similar to graph parsing. Efficient parsing techniques may thus benefit the UML-based modelling community, too.

- The goal of *graph grammar inference* [16, 17] is to induce a graph grammar from a set of sample member graphs, e.g., a grammar of a visual language from a set of user-provided valid graphs [18]. This problem can be solved by generating candidate graph grammars according to some search strategy and selecting the grammar that can generate the maximum number of sample graphs. Javed

et al. [19] took a similar course for inferring string grammars [20]. Whether we infer a string grammar or a graph grammar, an efficient parser is required for testing which input samples belong to individual candidate grammars.

The rest of this paper is structured as follows: In Section 3, we define basic concepts associated with graph grammars and parsing. Section 4 presents the graph grammars that will serve as our test examples. Section 5 briefly describes the original Rekers-Schürr parser. Our improvements are presented in Section 6 and experimentally validated in Section 7. Section 8 brings the paper to a conclusion.

2 Related work

The Rekers-Schürr parser was chosen as the basis of our work because the LGG formalism seems appropriate and natural for many different graph languages. Our test grammars, presented in Section 4, support this claim. However, many other graph grammar formalisms and corresponding parsers have been developed so far. Some of them are briefly presented in this section.

Graph grammars can be classified with respect to (at least) two distinct criteria [21]:

- The first criterion is the form that individual productions are allowed to take. This criterion leads to a hierarchy similar to the well-known Chomsky hierarchy of string grammars. According to this hierarchy, LGGs belong to the *context-sensitive* class, which is more powerful than the *context-free* class (such grammars comprise only productions of the form $Vertex ::= Graph$) but less powerful than the *unrestricted* class (productions of such grammars are completely unrestricted).

- The second criterion has no counterpart in the domain of string grammars. It classifies graph grammars with respect to the degree of *embedding* that they support. Embedding rules specify how the graph elements created by a production have to be connected with the elements that are already present in the host graph. While LGGs do not support embedding, some graph grammar classes, such as *edNLC* [22], support complex embedding rules. In contrast to LGGs, however, edNLC grammars are context-free.

Many graph grammar parsers are limited to context-free graph grammars without embedding or with simple embedding rules [23, 24, 25]. Minas [26], however, devised a *Cocke-Younger-Kasami*-style parser for *adaptive star grammars*. These grammars are somewhat more powerful than context-free ones, since their productions may have star graphs, not only single vertices, on their LHSs. Flasiński and Myśliński [6] developed a parser for a subclass of the aforementioned edNLC grammars and used it for recognising sign language gestures.

To reduce backtracking as much as possible, the Rekers-Schürr parser employs a complex scheme of searching a given input graph and organising the collected information. By contrast, the approaches of Zhang et al. [2] and Bottoni et al. [27], both of which parse graph grammar classes similar to LGG, are based on the naive parsing algorithm, which simply tries to reduce the input graph to the initial graph of the grammar by applying productions in the reverse direction and

using backtracking if necessary. To reduce the computational complexity of the naive algorithm, Zhang et al. preclude backtracking by restricting the grammar formalism significantly, whereas Bottoni et al. employ *critical pair analysis* to delay backtracking when it cannot be prevented. The method of Bottoni et al. thus parses similar grammars as that of Rekers and Schürr, but achieves this goal in an entirely different way. To our knowledge, a study to compare the performance of these two parsers has not yet been published.

Relational grammars [28] are essentially context-free string grammars augmented with predicates specifying (spatial) relationships between grammar symbols. Tucci et al. [18] developed a parser for relational grammars and showed their relationship to selected graph grammar classes.

3 Basic definitions

A *graph* $G = (V, E)$ consists of *vertices* (V) and *edges* ($E \subseteq V \times V$), which will be collectively called *(graph) elements*. Let $s(e)$ and $t(e)$ denote the source and the target vertex, respectively, of the edge $e \in E(G)$. We will assume that each element $x \in G$ is identified with a unique integer, denoted $\mathit{index}(x)$. Let $\mathit{label}(x)$ denote the label of the element $x \in G$. If x has no label, then let $\mathit{label}(x) \equiv \phi_V$ if $x \in V(G)$ and $\mathit{label}(x) \equiv \phi_E$ if $x \in E(G)$, where ϕ_V and ϕ_E are special labels reserved for this purpose. Let $\mathit{Labels}(G) \equiv \bigcup_{x \in G} \mathit{label}(x)$.

The graph with no elements (the *null graph*) will be denoted λ. A graph H is a *subgraph* of a graph G (denoted $H \preceq G$) if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Let $\mathit{copy}(G, x_{i_1} \colon L_1, \ldots, x_{i_k} \colon L_k)$ denote a copy of the graph G in which the elements $x_{i_1}, \ldots, x_{i_k}$ are relabelled as $L_1, \ldots, L_k$, respectively.

A *morphism* $h \colon G_1 \to G_2$ is a vertex-to-vertex and edge-to-edge mapping from the graph G_1 to the graph G_2 such that $\mathit{label}(h(x)) = \mathit{label}(x)$, $s(h(e)) = h(s(e))$, and $t(h(e)) = h(t(e))$ for all $x \in G_1$ and $e \in E(G_1)$. Unless explicitly called *partial*, a morphism $h \colon G_1 \to G_2$ will be assumed to be *total*, i.e., defined for all elements of the graph G_1. Given a morphism $h \colon S \to T$, the graph $h(S) \preceq T$ will be called an *occurrence* of the graph S in the graph T. Graphs G and H are *isomorphic* (denoted $G \simeq H$) if there exists a bijective morphism $h \colon G \to H$.

A *layered graph grammar* (called just 'grammar' in the sequel) is a set of *productions* of the form $p \colon \mathit{Lhs}(p) ::= \mathit{Rhs}(p)$, where all of $\mathit{Lhs}(p)$, $\mathit{Rhs}(p)$, $\mathit{Common}(p) \equiv \mathit{Lhs}(p) \cap \mathit{Rhs}(p)$, and $\mathit{Union}(p) \equiv \mathit{Lhs}(p) \cup \mathit{Rhs}(p)$ are proper graphs, while the sets $\mathit{Xlhs}(p) \equiv \mathit{Lhs}(p) \setminus \mathit{Common}(p)$ and $\mathit{Xrhs}(p) \equiv \mathit{Rhs}(p) \setminus \mathit{Common}(p)$ may contain edges that have one endpoint (or both) in $\mathit{Common}(p)$. Figure 1 presents these sets for a sample production.

Every grammar has to fulfil the so-called *layering condition* (see Definition 3.10 in [1]), which can be seen as a generalisation of the LHS-smaller-than-RHS rule for CSGs. The layering condition ensures the grammar's parsability, since it rules out cyclic productions or groups of productions (such as $A ::= B$, $B ::= C$, $C ::= A$).

To *apply* a production p to a graph G, the following steps have to be performed:

1. Find an occurrence of $\mathit{Lhs}(p)$ in G. Let $h \colon \mathit{Lhs}(p) \to G$ denote a morphism that determines the occurrence.

2. Remove the elements $h(\mathit{Xlhs}(p))$ from the graph G.

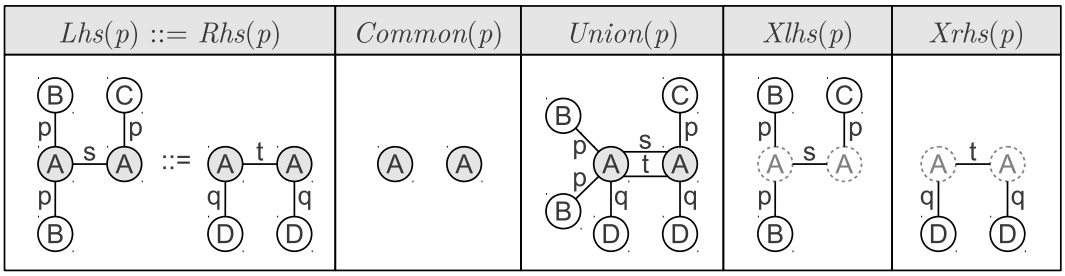


Figure 1: *A production and its sets Common, Union, Xlhs, and Xrhs. The vertices labelled 'A' do not belong to the sets Xlhs(p) and Xrhs(p).*

3. Attach a copy of $Xrhs(p)$ to the graph $h(Common(p))$ in the same way as $Xrhs(p)$ is attached to $Common(p)$ in $Rhs(p)$.

The result of an application of a production p to a graph G is thus the graph $G' \equiv G \setminus h(Xlhs(p)) \cup h(Xrhs(p))$, where the morphism $h\colon Lhs(p) \to G$ is extended to $h\colon Union(p) \to G$ as follows (consider that $Union(p) = Lhs(p) \cup Xrhs(p)$):

- For all vertices $v \in Xrhs(p)$: $h'(v) = v'$ such that $label(v') = label(v)$.
- For all edges $e \in Xrhs(p)$: $h'(e) = e'$ such that $label(e') = label(e)$, $s(e') = h(s(e))$, and $t(e') = h(t(e))$. In other words: if an edge e connects the vertices v and w in the graph G, then the corresponding edge $h(e)$ connects the vertices $h(v)$ and $h(w)$ in the graph G'.

An example of a production application is shown in the dotted frame in Figure 2.

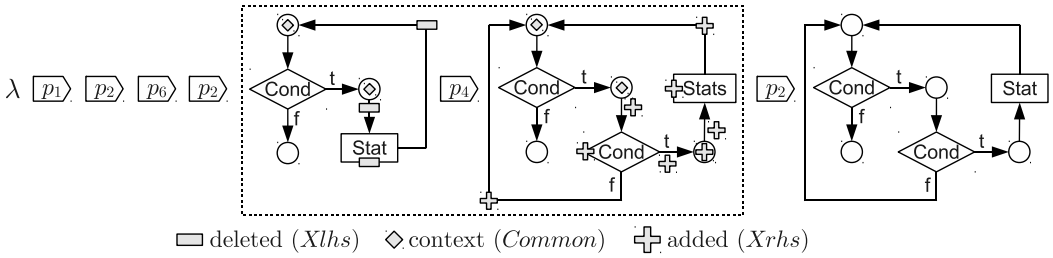


Figure 2: *A derivation of a graph in the grammar GG_{FC} of Figure 4. The application of the production p_4 is shown in detail.*

A *derivation* of a graph G in a given grammar is a sequence of production applications that transforms the null graph into the graph G (Figure 2). The *language* of a grammar GG (denoted $L(GG)$) is a set of all graphs that have some derivation in GG. A *parser* is an algorithm that, given a graph G and a grammar GG, determines whether $G \in L(GG)$ and produces a derivation of G if $G \in L(GG)$.

In the well-known case of context-free string grammars, the parser is usually expected to produce a *parse tree*, which is a data structure that contains all possible derivations of a given string. The Rekers-Schürr parser does not generate a structure analogous to the parse tree (it is questionable whether such a structure can even be defined for general LGGs), but it is able to produce, by backtracking, all derivations of a given graph $G \in L(GG)$ in some order. In this paper, however, we consider a

particular parsing problem solved after the parser has produced a single derivation (or after it has halted without producing any), so we do not deal with finding multiple derivations of a single graph.

4 The test grammars

Figures 3 and 4 display our test grammars. The grammars from Figure 3 will be henceforth called GG_{AHC} and GG_{HC}, and those from Figure 4 will be called GG_{ER} and GG_{FC}. The elements of the *Common* sets of individual productions are greyed. The small circles and squares that mark some elements will be explained in Section 6.2.

The grammar GG_{AHC} defines the language of acyclic hydrocarbon graphs. The grammar GG_{HC} generates all (both acyclic and cyclic) hydrocarbon graphs. (Hydrocarbons are chemical compounds composed of carbon (C) and hydrogen (H) atoms in which each carbon atom forms four connections and each hydrogen atom forms one.) The purpose of the indirect creation of non-labelled $\mathrm{C}-\mathrm{H}$ edges is to comply with the layering condition.

The grammar GG_{ER} (copied from [1]) defines the language of entity-relationship diagrams. The grammar GG_{FC} generates flowcharts with conditional clauses, loops, and forks.

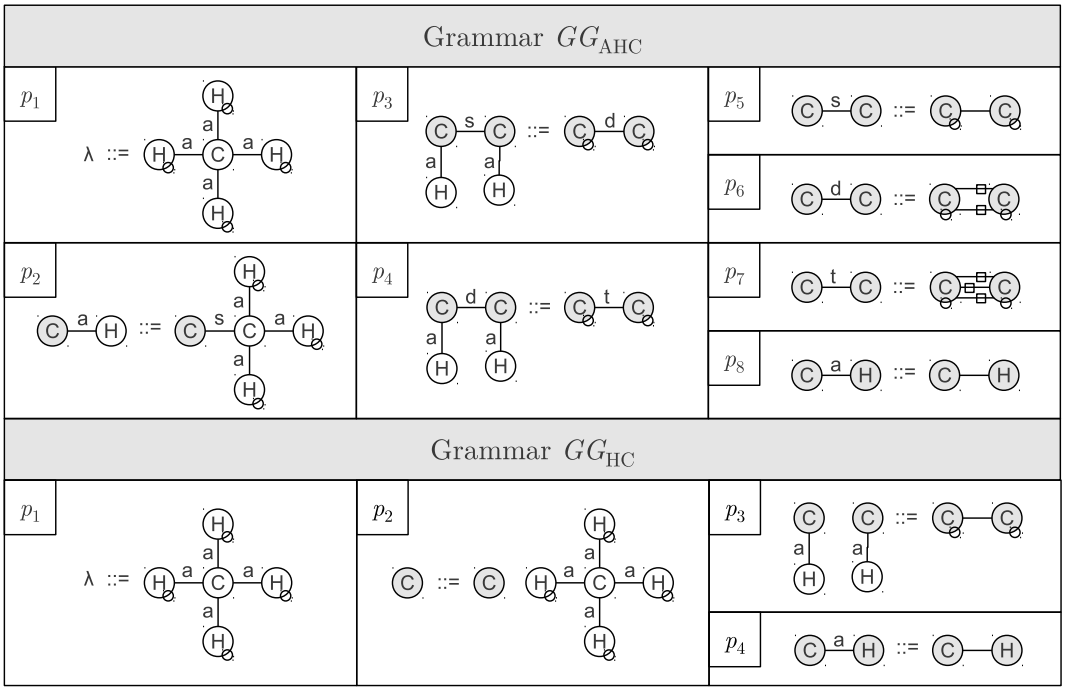


Figure 3: *Grammars* GG_{AHC} *and* GG_{HC}.

5 The original Rekers-Schürr parser

This section presents the Rekers-Schürr parser to the extent required to understand our improvements. Many important details are omitted. For a thorough description,

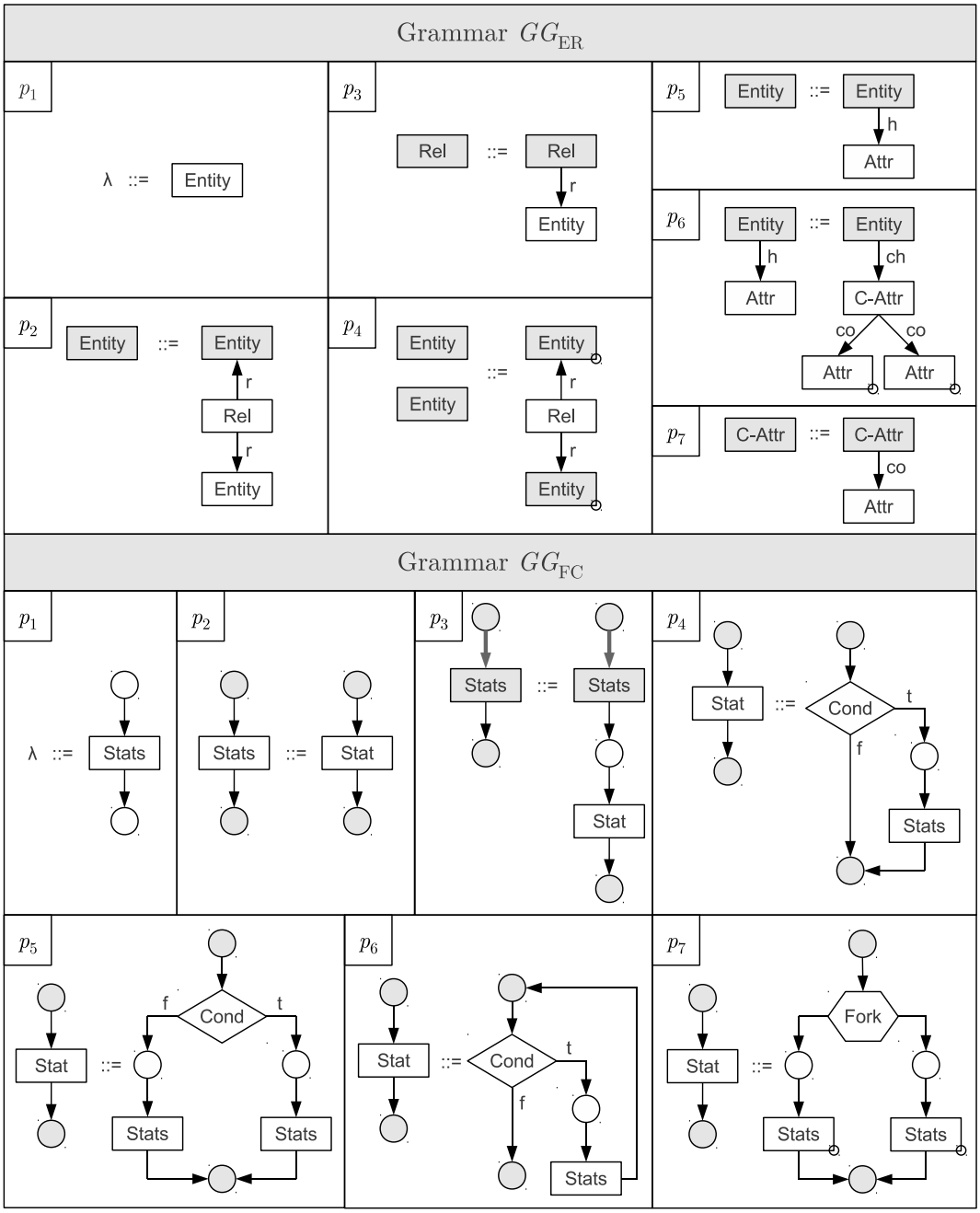


Figure 4: *Grammars* GG_{ER} *and* GG_{FC}.

the reader is referred to [11].

The Rekers-Schürr parser finds a derivation of an input graph G in a given grammar GG via two separate stages. Stage 1 produces a complete but redundant information for building any derivation of the graph G. Based on this information, Stage 2 either constructs an actual derivation of the graph G or declares that G does not belong to $L(GG)$. Stage 2 is based on backtracking, so it can produce all derivations, if desired. The rest of this section is devoted to Stage 1, because all our improvements pertain to it.

At the beginning, Stage 1 creates a graph $\overline{G}$ as a copy of the input graph G. Then it systematically searches the graph $\overline{G}$ for occurrences of the RHSs of individual productions. Whenever it discovers a graph $R' \preceq \overline{G}$ such that $R' = h(Rhs(p))$ for some production p and some morphism h, it augments the graph $\overline{G}$ by attaching copies of the elements $Xlhs(p)$ to $h(Common(p))$. The attached elements and the elements of R' jointly form an *instance* of the production p in the graph $\overline{G}$. Stage 1 repeats this discover-and-augment cycle until no more RHS occurrences can be found in the graph $\overline{G}$. Figure 5 visualises the graph $\overline{G}$ after the first two discover-and-augment steps when parsing the rightmost graph from Figure 2 against the grammar GG_{FC}. The corresponding production instances and their sets *Xlhs*, *Xrhs*, and *Common* are also highlighted.

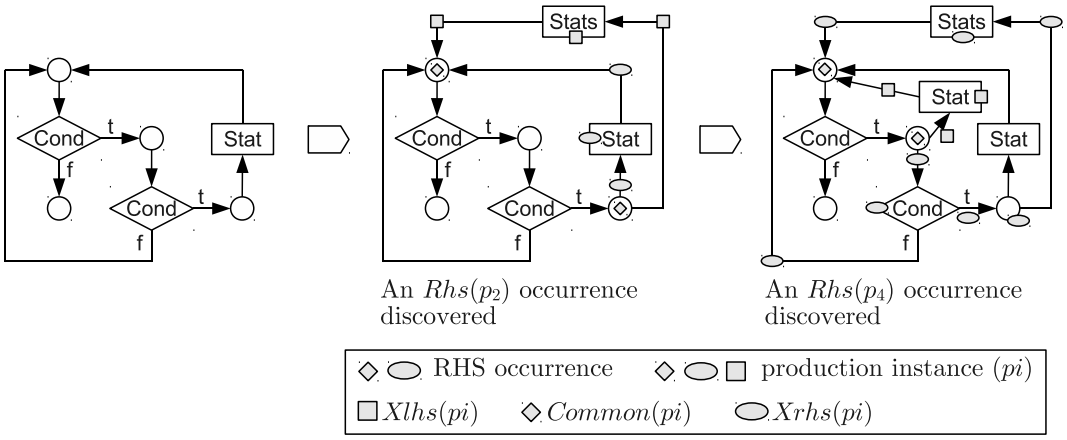


Figure 5: *The graph $\overline{G}$ after the first two discover-and-augment steps when parsing the rightmost graph of Figure 2 against the grammar GG_{FC}.*

To find an occurrence of an RHS in the graph $\overline{G}$, Stage 1 sequentially follows the RHS's *search plan*, which is a list of directives for visiting the RHS's elements. A sample search plan is given in Table 1. Stage 1 regards each directive as an instruction for finding a not-yet-discovered vertex and/or edge of the RHS in $\overline{G}$. Once Stage 1 has successfully fulfilled all directives of the RHS's search plan, it has discovered an occurrence of the entire RHS in $\overline{G}$ and simultaneously established a morphism that maps the elements of the RHS to those of the occurrence. Stage 1 executes each search plan in all possible ways. If several vertices and/or edges of $\overline{G}$ match a given directive, Stage 1 will eventually visit all of them. This implies that by the end of Stage 1, all possible RHS-to-$\overline{G}$ morphisms will have been established. This strategy ensures that Stage 1 eventually discovers all occurrences of the RHS in $\overline{G}$, regardless of the RHS's search plan (provided that it is correct, of course).

Table 1: *A search plan for the RHS of the production p_6 of the grammar GG_{FC}.*

Step	Directive
1	Find an unlabelled vertex and call it v_1.
2	Start at the vertex v_1, follow an unlabelled edge in the reverse direction to a vertex labelled 'Stats', and call this vertex v_2.
3	Start at the vertex v_1, follow an unlabelled edge to a vertex labelled 'Cond', and call this vertex v_3.
4	Start at the vertex v_3, follow an edge labelled 'f' to an unlabelled vertex, and call this vertex v_4.
5	Start at the vertex v_3, follow an edge labelled 't' to an unlabelled vertex, and call this vertex v_5.
6	Check the existence of an unlabelled edge from the vertex v_5 to the vertex v_2.

The main output of Stage 1 is a set of all production instances created in the discover-and-augment process. It is guaranteed that if $G \in L(GG)$, then there exists a subsequence (i.e., an ordered subset) of this set that determines a correct derivation of G. To find such a subsequence, Stage 2 relies on certain relationships between the production instances. These relationships are defined in Table 2. The relationship *above* partially orders the set of production instances: if pi_1 *above* pi_2, then the production instance pi_1 has to occur before the production instance pi_2 in any derivation that contains both pi_1 and pi_2. If pi_1 *excludes** pi_2, then the production instances pi_1 and pi_2 cannot both be part of the same derivation. If *inconsistent*(pi), then the production instance pi cannot be part of any derivation. Other relationships in Table 2 are not employed directly by Stage 2, but they serve as building blocks for computing the relationships *above*, *excludes**, and *inconsistent*.

Inconsistent production instances have to be removed immediately upon creation, or else Stage 1 might enter an infinite discover-and-augment loop. For this reason, all relationships have to be updated incrementally during Stage 1.

Figure 6 shows the parsing process for a graph G against the grammar GG_{FC}. Part (a) presents how the graph $\overline{G}$ grows as Stage 1 discovers production RHSs and creates the corresponding production instances. Part (b) presents the created production instances in detail: each of them is given as $pi = (p, \mathit{Xlhs}(pi), \mathit{Common}(pi), \mathit{Xrhs}(pi))$, where p denotes the production of which pi is an instance. Part (c) lists all relationships (except the ternary above_{pi} and above^+_{pi}) that hold for the created production instances. None of these instances is inconsistent. Based on the *above* and *excludes** relationships, Stage 2 builds a derivation of the graph G. The constructed derivation is shown in part (d).

6 Our improvements

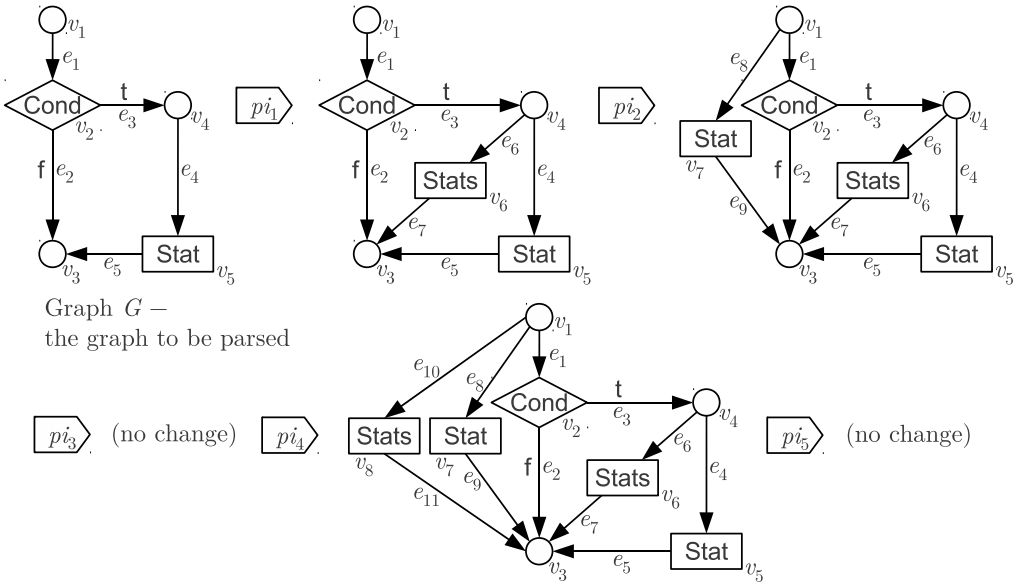
In this section, we present our improvements of the Rekers-Schürr parser. The improvement of Section 6.1 enlarges the class of parsable grammars, while those of Sections 6.2 and 6.3 enhance the parser's efficiency.

Table 2: *Relationships between production instances.*

Relationship[a]	Definition
conseqOf	$pi_1\ conseqOf\ pi_2 \iff Xlhs(pi_1) \cap Rhs(pi_2) \neq \emptyset$.
$conseqOf^+$	The transitive closure of *conseqOf*.
above	$pi_1\ above\ pi_2 \iff pi_1 \neq pi_2 \wedge$ ($pi_2\ conseqOf\ pi_1 \vee$ $Xrhs(pi_1) \cap Common(pi_2) \neq \emptyset \vee$ $\exists e \in E(Xlhs(pi_1)), v \in V(Xlhs(pi_2))\colon s(e) = v \vee t(e) = v$)
$above_{pi}$	$pi_1\ above_{pi}\ pi_2 \iff pi_1\ above\ pi_2 \wedge$ ($pi_1 = pi \wedge pi_2\ conseqOf^+\ pi \vee$ $pi_2 = pi \wedge pi_1\ conseqOf^+\ pi \vee$ $pi_1\ conseqOf^+\ pi\ \wedge\ pi_2\ conseqOf^+\ pi$)
$above_{pi}^+$	The transitive closure of $above_{pi}$.
excludes	$pi_1\ excludes\ pi_2 \iff pi_1 \neq pi_2 \wedge$ $((pi_1\ above\ pi_2 \wedge pi_2\ above\ pi_1) \vee Xrhs(pi_1) \cap Xrhs(pi_2) \neq \emptyset)$
$excludes^*$	$pi_1\ excludes^*\ pi_2 \iff pi_1\ excludes\ pi_2 \vee$ $(\exists pi'\colon pi'\ conseqOf^+\ pi_1 \wedge pi'\ excludes\ pi_2) \vee$ $(\exists pi''\colon pi''\ conseqOf^+\ pi_2 \wedge pi_1\ excludes\ pi'') \vee$ $(\exists pi', pi''\colon pi'\ conseqOf^+\ pi_1 \wedge pi''\ conseqOf^+\ pi_2 \wedge\ pi'\ excludes\ pi'')$
$excludes^*_{\text{self}}$	$excludes^*_{\text{self}}(pi) \iff pi\ excludes^*\ pi$
inconsistent	$inconsistent(pi) \iff excludes^*_{\text{self}}(pi) \vee$ $\exists pi', pi''\colon pi'\ above^+_{pi}\ pi'' \wedge pi''\ above^+_{pi}\ pi'$

[a]The relationship *conseqOf* is related to *consequence* of Rekers and Schürr [11] via the following equivalence: $pi_1\ conseqOf\ pi_2 \iff pi_1 \in consequence(pi_2)$.

(a) The growth of the graph $\overline{G}$ in Stage 1



(b) Production instances created during Stage 1

$pi_1 = (p_2, \{v_6, e_6, e_7\}, \{v_3, v_4\}, \{v_5, e_4, e_5\})$
$pi_2 = (p_4, \{v_7, e_8, e_9\}, \{v_1, v_3\}, \{v_2, v_4, v_6, e_1, e_2, e_3, e_6, e_7\})$
$pi_3 = (p_1, \emptyset, \emptyset, \{v_3, v_4, v_6, e_6, e_7\})$
$pi_4 = (p_2, \{v_8, e_{10}, e_{11}\}, \{v_1, v_3\}, \{v_7, e_8, e_9\})$
$pi_5 = (p_1, \emptyset, \emptyset, \{v_1, v_3, v_8, e_{10}, e_{11}\})$

(c) Relationships between the created production instances

pi_1 *conseqOf* pi_2	pi_1 *conseqOf*$^+$ pi_2	pi_2 *above* pi_1	pi_2 *excludes* pi_3	pi_2 *excludes** pi_3
pi_1 *conseqOf* pi_3	pi_1 *conseqOf*$^+$ pi_3	pi_3 *above* pi_1	pi_3 *excludes* pi_2	pi_3 *excludes** pi_2
pi_2 *conseqOf* pi_4	pi_1 *conseqOf*$^+$ pi_4	pi_3 *above* pi_2	pi_3 *excludes* pi_5	pi_3 *excludes** pi_4
pi_4 *conseqOf* pi_5	pi_1 *conseqOf*$^+$ pi_5	pi_3 *above* pi_4	pi_5 *excludes* pi_3	pi_3 *excludes** pi_5
	pi_2 *conseqOf*$^+$ pi_4	pi_4 *above* pi_2		pi_4 *excludes** pi_3
	pi_2 *conseqOf*$^+$ pi_5	pi_5 *above* pi_1		pi_5 *excludes** pi_3
	pi_4 *conseqOf*$^+$ pi_5	pi_5 *above* pi_2		
		pi_5 *above* pi_4		

(d) A derivation of G as produced by Stage 2

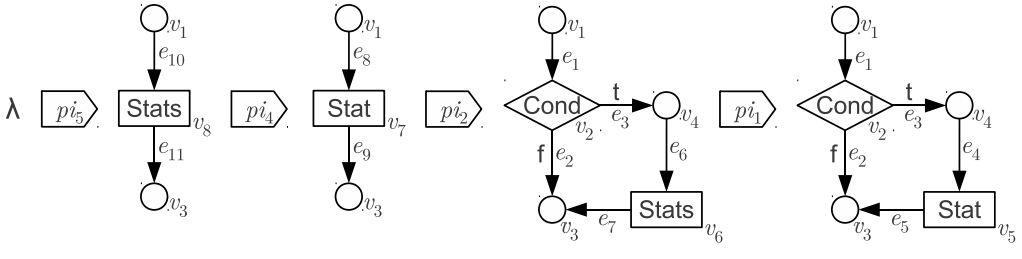


Figure 6: *Parsing a graph G against the grammar GG_{FC}.*

6.1 Improvement 1: Allowing productions with disconnected RHSs

Recall that Stage 1 searches for occurrences of an RHS in graph $\overline{G}$ by following the RHS's search plan. Each non-initial directive of a search plan either instructs Stage 1 to discover an edge between two already discovered vertices or to discover a new vertex by following an edge from an already discovered vertex. If search plans can comprise only such directives, they cannot describe how to visit the elements of a disconnected RHS. Consequently, Stage 1 cannot discover occurrences of RHSs such as that of production p_2 of grammar GG_{HC}, and the parser cannot parse such grammars as GG_{HC}.

To make the parsing of grammars with disconnected RHSs possible, we introduced another type of search plan directives, the so-called *jump* directives. A jump directive simply takes the form "jump to a vertex labelled A and call it v_i" and can be used anywhere in a search plan. If, in a given step-by-step execution of the search plan on the graph $\overline{G}$, Stage 1 comes across such a directive, it tries to find any vertex labelled A in $\overline{G}$ that has not yet been visited in *that* search plan execution (but which, of course, might have already been visited in some other execution of the search plan).

Table 3 presents one of many possible search plans for the RHS of the production p_2 of the grammar GG_{HC}. Let us assume that Stage 1 has to find an occurrence of this RHS in the graph $\overline{G}$ shown in Figure 7. To execute Step 1 of the search plan, Stage 1 can map the search plan's vertex v_1 to any vertex 'H' in the graph $\overline{G}$. If v_1 is mapped to w_2, w_3, w_4, or w_5, then the vertex v_2 of Step 2 is mapped to the vertex w_1 of $\overline{G}$; otherwise, v_2 is mapped to w_6. In the former case, the vertex v_4 of Step 4 can only be mapped to w_6; in the latter case, v_4 is mapped to w_1. Since Stage 1 executes the search plan in all possible ways, half of the discovered occurrences will cover the vertices w_1, w_2, w_3, w_4, w_5, and w_6, and the other half will cover the vertices w_1, w_6, w_7, w_8, w_9, and w_{10}.

Table 3: *A search plan for the RHS of the production p_2 of the grammar GG_{HC}.*

Step	Directive
1	Find a vertex labelled 'H' and call it v_1.
2	Start at the vertex v_1, follow an edge labelled 'a' to a vertex labelled 'C', and call this vertex v_2.
3	Start at the vertex v_2, follow an edge labelled 'a' to a vertex labelled 'H', and call this vertex v_3.
4	Jump to a vertex labelled 'C' and call it v_4.
5	Start at the vertex v_2, follow an edge labelled 'a' to a vertex labelled 'H', and call this vertex v_5.
6	Start at the vertex v_2, follow an edge labelled 'a' to a vertex labelled 'H', and call this vertex v_6.

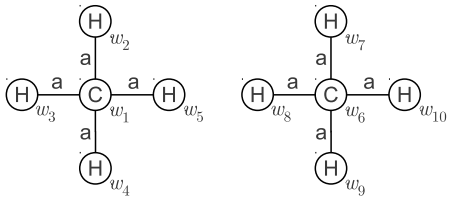


Figure 7: *A sample graph $\overline{G}$.*

6.2 Improvement 2: Preventing multiple discoveries of the same RHS occurrence

This section is divided as follows: Section 6.2.1 introduces the problem of multiple discoveries through an example. Section 6.2.2 defines a property called *interchangeability*, and Section 6.2.3 presents a redundancy elimination scheme based on this property. Section 6.2.4 presents an algorithm to determine interchangeability. Section 6.2.5 'applies' this algorithm to our test grammars.

6.2.1 Problem description with an example

Recall that Stage 1 finds all possible RHS-to-$\overline{G}$ morphisms for each RHS. For a given RHS, each such morphism determines one of its occurrences in $\overline{G}$. However, a single RHS occurrence may be determined by multiple equivalent morphisms. In such cases, Stage 1 discovers the same occurrence multiple times (once per morphism). Our first performance improvement, described in this section, is concerned with preventing such multiple discoveries. This kind of redundancy was first detected by Vermeulen [12], but his solution required the manual labelling of RHSs. By contrast, we propose an automatic solution based on formally defined conditions.

Let us illustrate the multiple-discovery redundancy by an example. Figure 8 displays two RHSs and their occurrences in some graph $\overline{G}$, and Table 4 lists all corresponding RHS-to-occurrence morphisms. In this case, Stage 1 would discover the occurrence of Rhs_1 four times, and that of Rhs_2 six times.

Table 4: *All $Rhs_1 \to \overline{G}$ and $Rhs_2 \to \overline{G}$ morphisms for the example of Figure 8.*

RHS	RHS-to-$\overline{G}$ morphisms
Rhs_1	$\{1 \mapsto 5', \ 2 \mapsto 1', \ 3 \mapsto 4', \ 4 \mapsto 11', \ 5 \mapsto 6', \ 6 \mapsto 7', \ 7 \mapsto 9' \ \}$,
	$\{1 \mapsto 5', \ 2 \mapsto 1', \ 3 \mapsto 4', \ 4 \mapsto 11', \ 5 \mapsto 6', \ 6 \mapsto 9', \ 7 \mapsto 7' \ \}$,
	$\{1 \mapsto 5', \ 2 \mapsto 1', \ 3 \mapsto 11', \ 4 \mapsto 4', \ 5 \mapsto 6', \ 6 \mapsto 7', \ 7 \mapsto 9' \ \}$,
	$\{1 \mapsto 5', \ 2 \mapsto 1', \ 3 \mapsto 11', \ 4 \mapsto 4', \ 5 \mapsto 6', \ 6 \mapsto 9', \ 7 \mapsto 7' \ \}$.
Rhs_2	$\{1 \mapsto 10', \ 2 \mapsto 8', \ 3 \mapsto 12', \ 4 \mapsto 13', \ 5 \mapsto 2', \ 6 \mapsto 14', \ 7 \mapsto 15'\}$,
	$\{1 \mapsto 10', \ 2 \mapsto 8', \ 3 \mapsto 13', \ 4 \mapsto 12', \ 5 \mapsto 2', \ 6 \mapsto 15', \ 7 \mapsto 14'\}$,
	$\{1 \mapsto 10', \ 2 \mapsto 12', \ 3 \mapsto 8', \ 4 \mapsto 13', \ 5 \mapsto 14', \ 6 \mapsto 2', \ 7 \mapsto 15'\}$,
	$\{1 \mapsto 10', \ 2 \mapsto 12', \ 3 \mapsto 13', \ 4 \mapsto 8', \ 5 \mapsto 14', \ 6 \mapsto 15', \ 7 \mapsto 2' \ \}$,
	$\{1 \mapsto 10', \ 2 \mapsto 13', \ 3 \mapsto 8', \ 4 \mapsto 12', \ 5 \mapsto 15', \ 6 \mapsto 2', \ 7 \mapsto 14'\}$,
	$\{1 \mapsto 10', \ 2 \mapsto 13', \ 3 \mapsto 12', \ 4 \mapsto 8', \ 5 \mapsto 15', \ 6 \mapsto 14', \ 7 \mapsto 2' \ \}$.

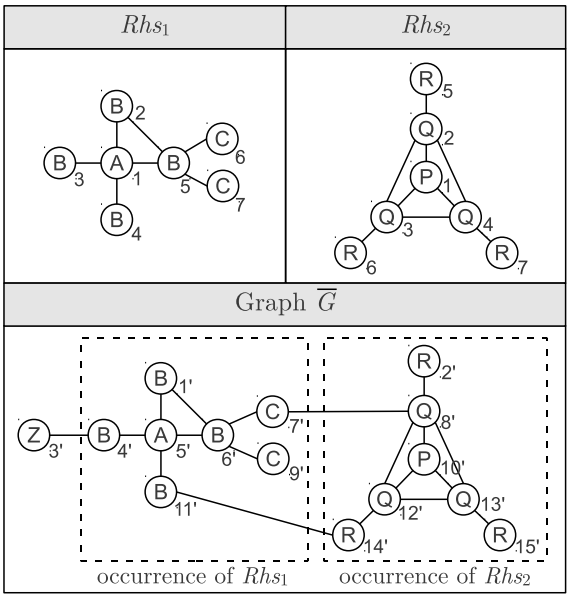


Figure 8: *Two production RHSs and their occurrences in a graph* $\overline{G}$.

6.2.2 RHS automorphisms and interchangeable elements

Multiple RHS-to-occurrence morphisms, which cause multiple discoveries of the same RHS occurrence, are due to *automorphisms* of the RHS, i.e., isomorphisms within the RHS itself. For example, the four RHS-to-occurrence morphisms for Rhs_1 of Figure 8 are due to the following automorphisms of Rhs_1 (cf. Table 4):

$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 3,\ 4 \mapsto 4,\ 5 \mapsto 5,\ 6 \mapsto 6,\ 7 \mapsto 7\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 3,\ 4 \mapsto 4,\ 5 \mapsto 5,\ 6 \mapsto 7,\ 7 \mapsto 6\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 4,\ 4 \mapsto 3,\ 5 \mapsto 5,\ 6 \mapsto 6,\ 7 \mapsto 7\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 4,\ 4 \mapsto 3,\ 5 \mapsto 5,\ 6 \mapsto 7,\ 7 \mapsto 6\}.$$

The six RHS-to-occurrence morphisms for Rhs_2 are a consequence of the following automorphisms of Rhs_2:

$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 3,\ 4 \mapsto 4,\ 5 \mapsto 5,\ 6 \mapsto 6,\ 7 \mapsto 7\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 2,\ 3 \mapsto 4,\ 4 \mapsto 3,\ 5 \mapsto 5,\ 6 \mapsto 7,\ 7 \mapsto 6\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 3,\ 3 \mapsto 2,\ 4 \mapsto 4,\ 5 \mapsto 6,\ 6 \mapsto 5,\ 7 \mapsto 7\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 3,\ 3 \mapsto 4,\ 4 \mapsto 2,\ 5 \mapsto 6,\ 6 \mapsto 7,\ 7 \mapsto 5\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 4,\ 3 \mapsto 2,\ 4 \mapsto 3,\ 5 \mapsto 7,\ 6 \mapsto 5,\ 7 \mapsto 6\},$$
$$\{1 \mapsto 1,\ 2 \mapsto 4,\ 3 \mapsto 3,\ 4 \mapsto 2,\ 5 \mapsto 7,\ 6 \mapsto 6,\ 7 \mapsto 5\}.$$

We will now define *interchangeability*, a concept closely associated with automorphisms. The subsequent definitions will not be justified immediately. Their use will become apparent in Section 6.2.3, where we describe how to use interchangeability to eliminate the multiple-discovery redundancy.

Definition 1 (Graph-level interchangeability)**.** *Vertices* $v_1, \ldots, v_k$ *of a graph* G *are* interchangeable with respect to G (denoted $inter_G(v_1, \ldots, v_k)$) *if for each permutation* σ *of* $\{1, \ldots, k\}$ *there exists an automorphism* $h\colon G \to G$ *such that* $h(v_1) = v_{\sigma(1)}$, $\ldots$, $h(v_k) = v_{\sigma(k)}$.

Definition 2. *An* interchangeability class (IC) *of a graph G is a set of vertices that are interchangeable with respect to G.*

For example, the graphs Rhs_1 and Rhs_2 of Figure 8 contain the following ICs:

- Rhs_1: $\{3, 4\}$ and $\{6, 7\}$.
- Rhs_2: $\{2, 3, 4\}$ and $\{5, 6, 7\}$.

Definition 3. *Let a graph G contain ICs with k_1, k_2, ..., k_s vertices, respectively. These ICs are* mutually independent *if G has an automorphism for each of the $k_1!\, k_2! \ldots k_s!$ permutations of the vertices composing the ICs.*

The two ICs of Rhs_1 are mutually independent, since there exists an automorphism of Rhs_1 for each of the four ($= 2!\, 2!$) permutations of the vertices $\{3, 4\}$ and $\{6, 7\}$, namely $(3, 4, 6, 7)$, $(3, 4, 7, 6)$, $(4, 3, 6, 7)$, and $(4, 3, 7, 6)$. By contrast, the ICs of Rhs_2 are not independent, since each permutation of $\{2, 3, 4\}$ corresponds to a unique permutation of $\{5, 6, 7\}$, and vice versa. The graph Rhs_2 is therefore considered to contain only one IC: either $\{2, 3, 4\}$ or $\{5, 6, 7\}$.

If the RHS of a production contains s independent ICs with k_1, k_2, ..., k_s vertices, respectively, then Stage 1 will discover each occurrence of that RHS $k_1!\, k_2! \ldots k_s!$ times, since each automorphism corresponds to an RHS-to-occurrence morphism.

Definition 4 (Production-level interchangeability)**.** *Vertices $v_1, \ldots, v_k \in Rhs(p)$ are* interchangeable with respect to a production p (denoted $inter_p(v_1, \ldots, v_k)$) *if the following three conditions are satisfied:*

1. *Either $v_1, \ldots, v_k \in Xrhs(p)$ or $v_1, \ldots, v_k \in Common(p)$.*
2. *The vertices are interchangeable with respect to the graph $Rhs(p)$, i.e., $inter_{Rhs(p)}(v_1, \ldots, v_k)$.*
3. *The vertices $v_1, \ldots, v_k$ are interchangeable with respect to the graph $Union(p) = Lhs(p) \cup Rhs(p)$, i.e., $inter_{Union(p)}(v_1, \ldots, v_k)$.*

Definition 5. *An* interchangeability class (IC) *of a production p is a set of vertices of $Rhs(p)$ that are interchangeable with respect to p.*

In the production of Figure 1, the two RHS vertices 'A' are interchangeable with respect to the RHS but not with respect to the entire production (cf. the graph $Union(p)$). The same goes for the vertices 'D'.

Although we focus on interchangeable vertices throughout Section 6, the concept of interchangeability can be naturally extended to edges:

Definition 6. *Edges e_1, ..., e_k are interchangeable with respect to a production p if the following three conditions are met:*

1. *Either $e_1, \ldots, e_k \in Xrhs(p)$ or $e_1, \ldots, e_k \in Common(p)$.*
2. *All edges have the same label, i.e., $label(e_1) = \ldots = label(e_k)$.*
3. *All edges connect the same two vertices in the same direction, i.e., $s(e_1) = \ldots = s(e_k)$ and $t(e_1) = \ldots = t(e_k)$.*

For example, all edges in the RHSs of the productions p_6 and p_7 of the grammar GG_{AHC} are interchangeable.

6.2.3 Eliminating the multiple-discovery redundancy

The knowledge of the ICs of individual productions can be directly employed to eliminate the multiple-discovery redundancy. Suppose that a production p contains a single IC, which comprises vertices $v_1, \ldots, v_k \in \mathit{Rhs}(p)$. In the original parsing algorithm, Stage 1 explores the space of partial and total RHS-to-$\overline{G}$ morphisms exhaustively and thus establishes a total RHS-to-occurrence morphism for each permutation of the vertices $v_1, \ldots, v_k$, which results in $k!$ discoveries of every RHS occurrence. This redundancy can be eliminated by allowing Stage 1 to establish only those partial and total morphisms $h\colon \{v_{i_1}, \ldots, v_{i_r}\} \to \overline{G}$ (with $1 \leq i_1 < \ldots < i_r \leq k$) for which $\mathit{index}(h(v_{i_1})) < \ldots < \mathit{index}(h(v_{i_r}))$. There is *exactly one* total RHS-to-occurrence morphism that meets the condition $\mathit{index}(h(v_1)) < \ldots < \mathit{index}(h(v_k))$, so each RHS occurrence will be discovered exactly once. If a given RHS contains several mutually independent ICs, then such a condition has to be met for each IC separately.

To illustrate this idea by an example, consider again Figure 8. Suppose that the three vertices 'Q' of Rhs_2 (indexed 2, 3, and 4) constitute a production-level IC, not just an IC of Rhs_2. Suppose further that Stage 1 is in the midst of finding occurrences of Rhs_2 in the graph $\overline{G}$ and that it has already established a partial Rhs_2-to-$\overline{G}$ morphism $h\colon \{1 \mapsto 10', 2 \mapsto 12', 3 \mapsto 13'\}$. Since $\mathit{index}(h(2)) < \mathit{index}(h(3))$, this partial morphism is allowed to 'grow' towards a total morphism. This, however, is impossible, since the vertex 4 of Rhs_2 can now be matched only with the vertex $8'$ of $\overline{G}$, giving a forbidden partial morphism $\{1 \mapsto 10', 2 \mapsto 12', 3 \mapsto 13', 4 \mapsto 8'\}$. Therefore, Stage 1 tries to find another way to match the elements of Rhs_2 with those of $\overline{G}$. By the end of its search process, Stage 1 will have discovered many partial morphisms but only one total, namely $\{1 \mapsto 10', 2 \mapsto 8', 3 \mapsto 12', 4 \mapsto 13', 5 \mapsto 2', 6 \mapsto 14', 7 \mapsto 15'\}$. Therefore, Rhs_2 will be discovered only once in $\overline{G}$.

A proof that the proposed redundancy elimination scheme preserves the correctness of the parser is given in Appendix A.

6.2.4 An algorithm to determine the ICs

The production-level ICs in a given RHS could be determined using Definition 4 directly. While condition 1 is easy to check, this is not the case with conditions 2 and 3. In this section, we give an algorithm for computing the relationship $\mathit{inter}_G(v_1, \ldots, v_k)$ for an arbitrary graph G and vertices $v_1, \ldots, v_k \in G$. Using this relationship, the conditions of Definition 4 can be easily verified.

Proposition 1. *Let u and v be two vertices of a graph G, and let # and \$ be special vertex labels that are not present in G. Let $S_1 = \mathit{copy}(G, u\colon \#, v\colon \$)$ and $S_2 = \mathit{copy}(G, u\colon \$, v\colon \#)$. Then the following equivalence holds:*

$$\mathit{inter}_G(u, v) \iff \mathit{label}(u) = \mathit{label}(v) \wedge S_1 \simeq S_2. \tag{1}$$

Proof. ($\Longrightarrow$) If $\mathit{inter}_G(u, v)$, then there exists an automorphism h of G such that $h(u) = v$ and $h(v) = u$. This is only possible if the two vertices have the same label. Since the graphs S_1 and S_2 are copies of G except for the vertices labelled '#' and '\$' (which correspond to the interchangeable vertices u and v), the automorphism h directly corresponds to an isomorphism between the graphs S_1 and S_2. The graphs

S_1 and S_2 are hence isomorphic. ($\Longleftarrow$) If $S_1 \simeq S_2$, then every isomorphism between S_1 and S_2 maps the vertices '#' and '$' of S_1 to the vertices '#' and '$' of S_2, respectively, since the labels # and $ are unique. These two mappings correspond to the mappings $u \mapsto v$ and $v \mapsto u$ in G, respectively, and the entire isomorphism corresponds to an automorphism in G, provided that $label(u) = label(v)$. □

Proposition 2. *Let $v_1, \ldots, v_k$ be vertices of a graph G, and let $\#1, \ldots, \#k$ be special vertex labels not present in G. For $i \in \{1, \ldots, k!\}$, let σ_i denote the i-th permutation of the set $\{\#1, \ldots, \#k\}$, and let $S_i = copy(G, v_1\colon \sigma_i(\#1), \ldots, v_k\colon \sigma_i(\#k))$. Then the following equivalence holds:*

$$inter_G(v_1, \ldots, v_k) \Longleftrightarrow (label(v_1) = \ldots = label(v_k)) \wedge (S_1 \simeq \ldots \simeq S_{k!}). \quad (2)$$

This proposition is a generalisation of Proposition 1 and can be proved in a similar way. Figure 9 shows the graphs S_i for the example of Figure 8.

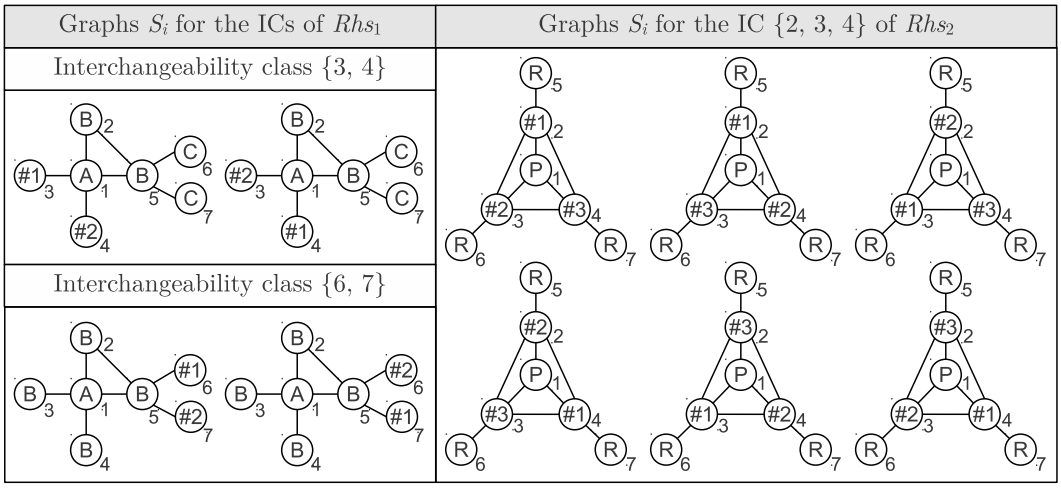


Figure 9: *The graphs S_i for individual ICs of the RHSs of Figure 8.*

Instead of checking whether all $k!$ graphs S_i are isomorphic, we determine $inter_G(v_1, \ldots, v_k)$ for $k > 2$ in a recursive way:

$$inter_G(v_1, \ldots, v_k) \Longleftrightarrow inter_G(v_1, \ldots, v_{k-1}) \wedge inter_G(v_1, v_k) \wedge inter_G(v_2, v_k) \wedge \ldots \wedge inter_G(v_{k-1}, v_k). \quad (3)$$

To find independent ICs in a given graph, we use the procedure findICs of Algorithm 1. This procedure accepts a graph and returns a set of independent ICs, denoted $\mathcal{I}$ in the procedure's body. The set Q contains the vertices that have not yet been assigned to any IC. At the beginning, Q contains all vertices of G. After discovering an IC, the vertices from that IC are removed from Q, and the IC itself is added to the output set $\mathcal{I}$.

The auxiliary procedure enlargeIC tries to enlarge a given IC composed of vertices u and v by checking condition (3) for each available vertex. Line 8 assigns a unique label to each constituent vertex of the IC C to ensure that only those ICs that are independent of C will be subsequently discovered. Without Line 8, the algorithm would regard the vertex sets $\{2, 3, 4\}$ and $\{5, 6, 7\}$ from Rhs_2 of Figure 8 as two independent ICs, which would be incorrect.

Algorithm 1: *The algorithm to find independent ICs in a given graph.*

```
1  procedure findICs(G)
2      I := ∅;
3      Q := V(G);
4      i := 1;
5      while ∃u, v ∈ Q: inter(u, v) do
6          C := enlargeIC(u, v, Q);
7          I := I ∪ {C};
8          foreach z ∈ C do label(z) := $_i;  i := i + 1 end;
9          Q := Q \ C
10     end;
11     return I
12 end
13
14 procedure enlargeIC(u, v, Q)
15     C := {u, v};
16     foreach z ∈ Q do
17         if for all t ∈ C: inter(t, z) then
18             C := C ∪ {z}
19         end
20     end;
21     return C
22 end
```

6.2.5 The ICs in our test grammars

The ICs of our test grammars are displayed with small circles and squares in Figures 3 and 4. The elements marked with the same symbol belong to the same IC.

In general, Improvement 2 should pay off whenever at least one RHS contains interchangeable elements. Since the number of RHS automorphisms grows factorially with the size of individual ICs, Improvement 2 should make an especially noticeable difference for grammars such as GG_{AHC} and GG_{HC}. As we shall see in Section 7, this really happens.

6.3 Improvement 3: Reducing the need to compute the *inconsistent* relationship

To avoid an infinite discover-and-augment cycle, Stage 1 has to determine the *inconsistent* relationship for each production instance. Since this relationship depends on the ternary $above^{+}_{pi}$ relationship, it takes $\Omega(n^3)$ time to determine the value of $inconsistent(pi)$ for n production instances. However, if we knew in advance that a given production cannot occur as an inconsistent instance, we could omit the computation of $inconsistent(pi)$ (and also of $excludes^{*}_{\mathrm{self}}(pi)$ and $above^{+}_{pi}$) for all instances of that production. In this section, we describe the conditions under which such savings are possible.

Let us adopt the following notational convention: p, p', etc. denote productions, and pi, pi', etc. denote production instances. Let $Inst(p)$ denote the set of all possible instances of production p created during Stage 1 (for any input graph), and let $inconsistent(p)$ represent the possibility of production p occurring as an inconsistent instance:

$$inconsistent(p) \iff (\exists pi \in Inst(p) \colon inconsistent(pi)). \tag{4}$$

If $inconsistent(p)$, nothing can be said in advance about the inconsistency of individual production instances, so $inconsistent(pi)$ has to be computed for each $pi \in Inst(p)$. However, if $\neg inconsistent(p)$ has been determined with certainty, then $\neg inconsistent(pi)$ is guaranteed for every $pi \in Inst(p)$.

Unfortunately, $inconsistent(p)$ cannot be accurately determined for all possible productions, since many complex interactions between production instances may occur in Stage 1. We thus assume $inconsistent(p)$ unless our conditions guarantee that $\neg inconsistent(p)$. This approach might not result in maximum possible savings, but it can never cause an inconsistent production to be missed during Stage 1.

The relationship $inconsistent(p)$ will be (indirectly) computed from the relationships $conseqOf(p, p')$, $above(p, p')$, etc. in a similar manner as $inconsistent(pi)$ is computed from $conseqOf(pi, pi')$, $above(pi, pi')$, etc. The relationships $conseqOf(p, p')$, $above(p, p')$, etc. are defined analogously to (4), e.g.:

$$p' \ conseqOf \ p \iff (\exists pi \in Inst(p), pi' \in Inst(p') \colon pi' \ conseqOf \ pi). \tag{5}$$

Like $inconsistent(p)$, the relationships $conseqOf(p, p')$, $above(p, p')$, etc. will also be assumed to hold unless our conditions imply the opposite.

The computation of $inconsistent(pi)$ is based on the following fact: If we determine that neither $excludes^{*}_{\mathrm{self}}(pi)$ nor $(\exists pi', pi'' \colon pi' \ above^{+}_{pi} \ pi'' \wedge pi'' \ above^{+}_{pi} \ pi')$ can

hold for any $pi \in Inst(p)$, then we can conclude that $\neg inconsistent(p)$ holds. This is equivalent to:

$$\neg excludes^{*}_{\text{self}}(p) \wedge \neg\exists p', p'' \colon p' \ above^{+}_{p}\ p'' \wedge p'' \ above^{+}_{p}\ p' \Longrightarrow \neg inconsistent(p). \quad (6)$$

The conditions for $\neg(p' \ above_p \ p'')$ and $\neg excludes^{*}_{\text{self}}(p)$ are constructed analogously to (6) and to the corresponding definitions in Table 2. For example:

$$(\neg\exists p' \colon p' \ conseqOf^{+}\ p \wedge p \ excludes\ p') \wedge \quad (7)$$
$$(\neg\exists p', p'' \colon p' \ conseqOf^{+}\ p \wedge p'' \ conseqOf^{+}\ p \wedge p' \ excludes\ p'') \Longrightarrow$$
$$\neg excludes^{*}_{\text{self}}(p).$$

The relationships $above_{pi}(p, p')$ and $excludes^{*}_{\text{self}}(p)$ indirectly depend on $conseqOf(p, p')$. The condition for $\neg(p' \ conseqOf\ p)$ is constructed as follows:

Proposition 3. *For any two productions p and p', it holds that*

$$Labels(Rhs(p)) \cap Labels(Xlhs(p')) = \emptyset \Longrightarrow \neg(p' \ conseqOf\ p). \quad (8)$$

Proof. If $Rhs(p)$ and $Xlhs(p')$ have no labels in common, then it is impossible for $Rhs(pi)$ and $Xlhs(pi')$, where $pi \in Inst(p)$ and $pi' \in Inst(p')$, to overlap. For any pair (pi, pi'), it thus holds that $Rhs(pi) \cap Xlhs(pi') = \emptyset$ and hence $\neg(pi' \ conseqOf\ pi)$. This implies $\neg(p' \ conseqOf\ p)$. □

The conditions for $\neg(p \ above\ p')$ and $\neg(p \ excludes\ p')$ are constructed in an analogous fashion. The relationships $(p' \ conseqOf^{+}\ p'')$ and $(p' \ above^{+}_{p}\ p'')$ are computed as the transitive closures of $(p' \ conseqOf\ p'')$ and $(p' \ above_p\ p'')$, respectively.

If Improvement 2 is in effect, we use the following additional condition:

Proposition 4. *For any production p, it holds that*

$$\neg(p \ above\ p) \wedge Xrhs(p) = \{e\} \wedge Common(p) = \{s(e), t(e)\} \wedge inter_p(s(e), t(e)) \Longrightarrow \neg(p \ excludes\ p). \quad (9)$$

Proof. If $Xrhs(p)$ contains only an edge, then for any instances pi and pi' of production p, $Xrhs(pi) \cap Xrhs(pi')$ can contain only a corresponding edge in graph $\overline{G}$. Since $Rhs(p) = Common(p) \cup Xrhs(p)$ contains only the edge's endpoints besides the edge itself, $Xrhs(pi) \cap Xrhs(pi')$ will contain the edge only if $Rhs(pi)$ and $Rhs(pi')$ coincide. If $inter_p(s(e), t(e))$ and if Improvement 2 is in effect, $Rhs(pi)$ and $Rhs(pi')$ can never coincide, for this would imply a double discovery of the same occurrence of $Rhs(p)$. Therefore, $Xrhs(pi) \cap Xrhs(pi') = \emptyset$, which, together with the assumption $\neg(p \ above\ p)$, implies $\neg(p \ excludes\ p)$. □

Table 5 lists those productions p of our test grammars for which $inconsistent(p)$ holds according to our conditions (i.e., the conditions do not imply $\neg inconsistent(p)$). In the case of the grammar GG_{HC}, analogous information is provided for all relationships. For example, since our conditions imply $\neg(p_2 \ excludes\ p_3)$, the pair (p_2, p_3) is not listed in the corresponding row of Table 5.

In general, Improvement 3 should prove beneficial when the input grammar contains many productions p for which $Xlhs(p) = \emptyset$. For such a grammar, the relationship $p_1 \ conseqOf\ p_2$ is false for many pairs (p_1, p_2). Since the *inconsistent* relationship directly depends on $above^{+}_{pi}$ and $excludes^{*}_{\text{self}}$, which in turn depend on $conseqOf$, it is reasonable to expect that many cases of $\neg(p_1 \ conseqOf\ p_2)$ will lead to many cases of $\neg inconsistent(p)$.

Table 5: *Productions or pairs of productions for which individual relationships hold according to our conditions.*

Grammar	Relationship	Productions p or pairs (p', p'') satisfying the relationship
GG_{AHC}	$inconsistent(p)$	p_1, p_2, p_3, p_4, p_8
GG_{HC}	p'' $conseqOf$ p'	(p_1, p_3), (p_1, p_4), (p_2, p_3), (p_2, p_4), (p_4, p_3)
	p'' $conseqOf^+$ p'	(p_1, p_3), (p_1, p_4), (p_2, p_3), (p_2, p_4), (p_4, p_3)
	p' $above$ p''	(p_1, p_2), (p_1, p_3), (p_1, p_4), (p_2, p_2), (p_2, p_3), (p_2, p_4), (p_4, p_3)
	p' $above^+_{p_1}$ p''	(p_1, p_3), (p_1, p_4), (p_4, p_3)
	p' $above^+_{p_2}$ p''	(p_2, p_3), (p_2, p_4), (p_4, p_3)
	p' $above^+_{p_3}$ p''	(none)
	p' $above^+_{p_4}$ p''	(p_4, p_3)
	p' $excludes$ p''	(p_1, p_1), (p_1, p_2), (p_2, p_1), (p_2, p_2)
	p' $excludes^*_{\mathrm{self}}$ p''	(none)
	$inconsistent(p)$	(none)
GG_{ER}	$inconsistent(p)$	p_5, p_6, p_7
GG_{FC}	$inconsistent(p)$	p_1, p_2, p_3, p_4, p_5, p_6, p_7

7 Experimental results

We estimated the value of our contribution by comparing the performance of the original Rekers-Schürr parser with the performance attained by our improvements. To obtain a reliable estimate, we constructed several graph sequences for our test grammars (five sequences for GG_{AHC}, six for GG_{HC}, one for GG_{ER}, and one for GG_{FC}) and ran the original and the improved parser on individual graphs of these sequences. The sequences are defined by rules presented in Figure 10. The symbol G_n denotes the n-th graph in each sequence. Individual graphs G_i are constructed by the help of auxiliary graphs (labelled S_i and T), recursion, and numbered connectors, the use of which is explained at the bottom of Figure 10. For concreteness, Figure 11 shows the graphs G_2 for some sequences.

Tables 6, 7, and 8 display the time, in milliseconds, required to parse a given graph from a given sequence against a given grammar using a given combination of improvements. The labels I2 and I3 denote Improvements 2 and 3, respectively. The label n denotes graph size, e.g., $n = 5$ represents the graph G_5 from a given sequence. For example, the time to parse the graph G_{10} from the sequence Seq_{AHC2} against the grammar GG_{AHC} using both performance improvements amounts to 287 milliseconds (Table 6, row $n = 10$ / I2 + I3, column Seq_{AHC2}). The parse times were measured on a 1.86 GHz Intel Core 2 Duo machine, and they are accurate to three digits. Each experiment was time-limited to 10^7 milliseconds (ca. 2.8 hours).

Table 9 shows how derivation length (D) and the number of production instances created during Stage 1 ($\#PI_{\mathrm{Original}}$ and $\#PI_{\mathrm{I2}}$) grow with input graph size (n denotes graph size as in Figure 10 and Tables 6, 7, and 8). Derivation length, i.e., the number of production applications required to transform the graph λ into the

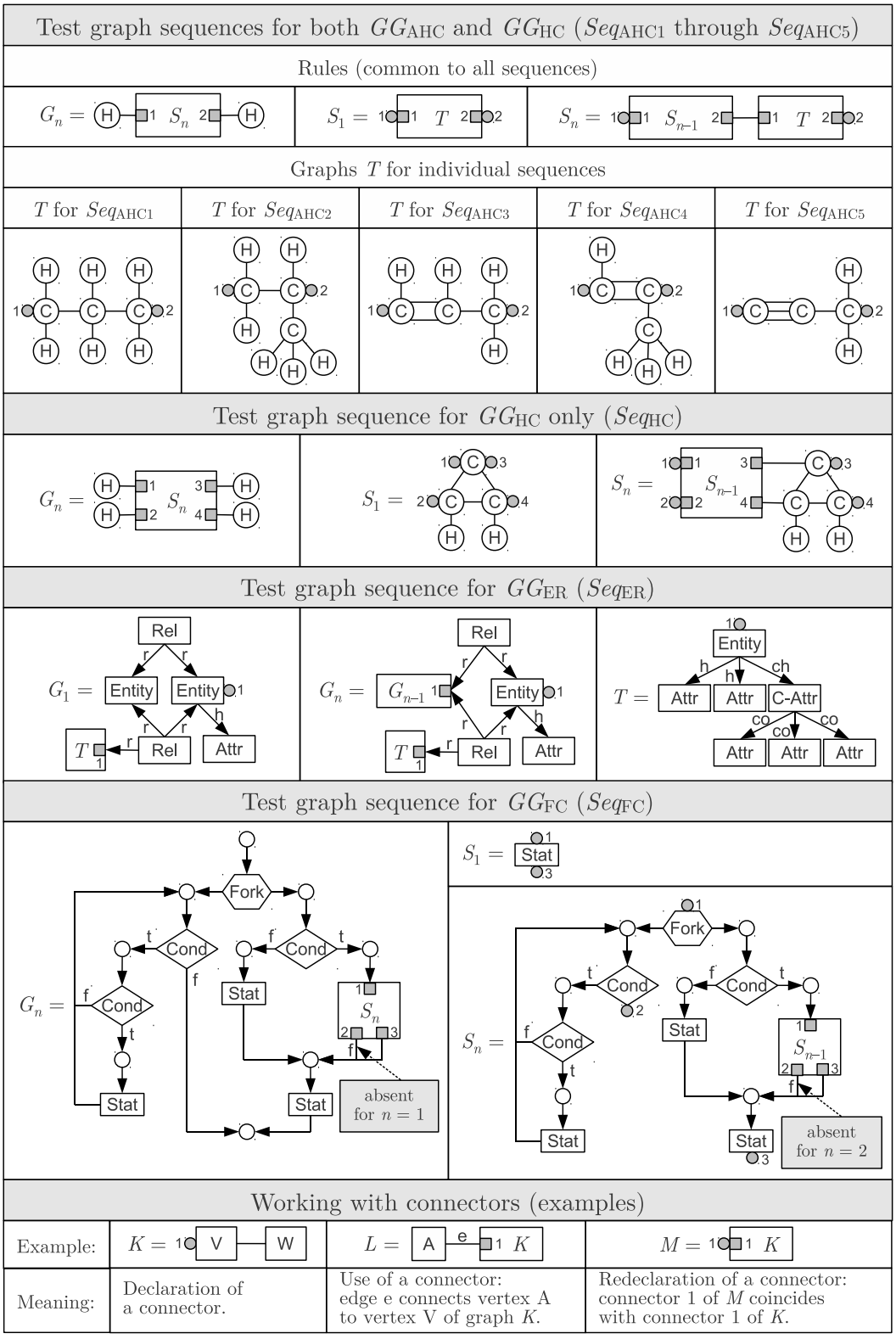


Figure 10: *Test graph sequences.*

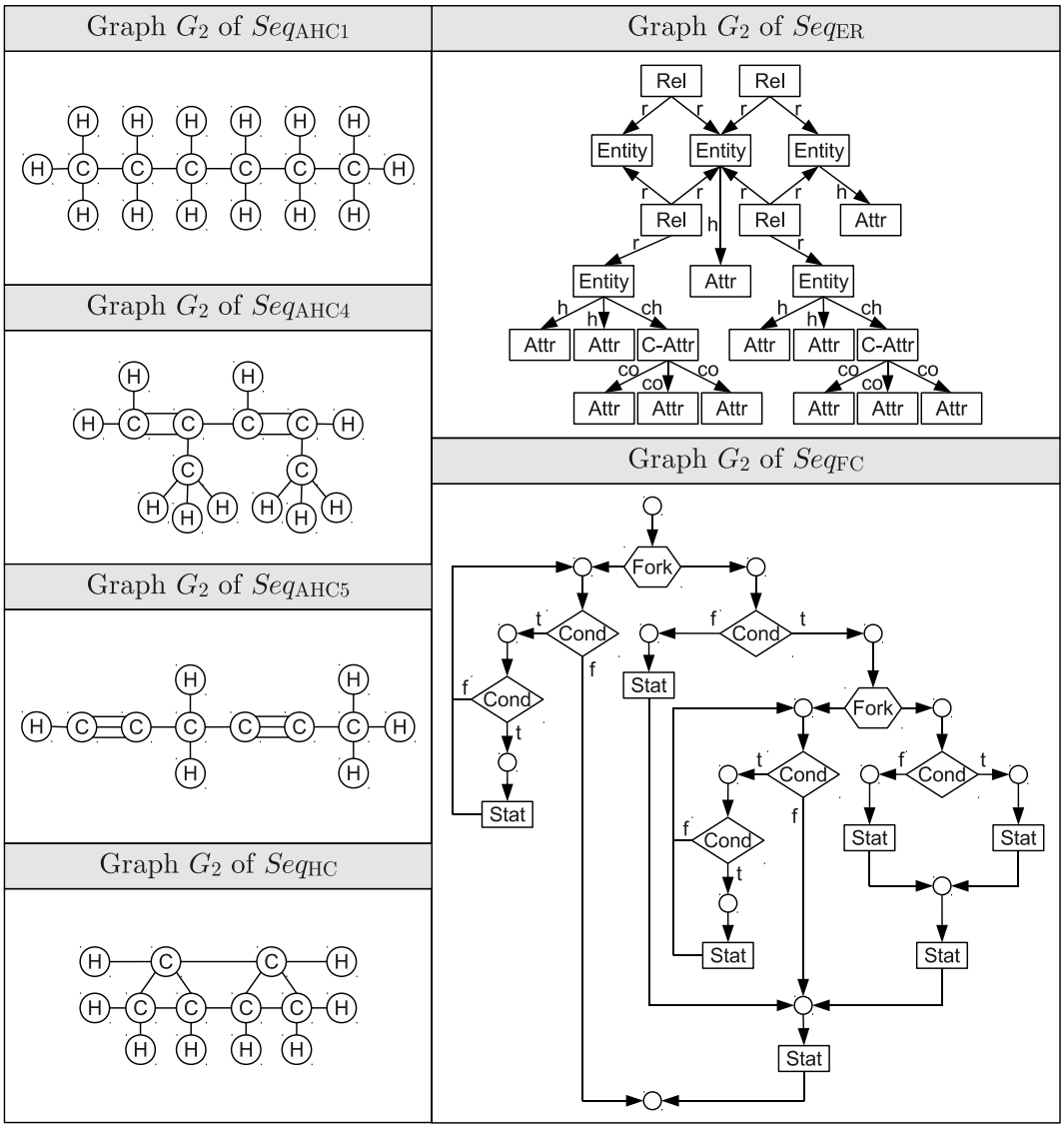


Figure 11: *The graphs G_2 for some test sequences (cf. Figure 10).*

Table 6: *The time, in milliseconds, required to parse individual* $Seq_{\mathrm{AHC}i}$ *graphs against the grammar* GG_{AHC}.

n	Parser	Seq_{AHC1}	Seq_{AHC2}	Seq_{AHC3}	Seq_{AHC4}	Seq_{AHC5}
1	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	5	5	6	6	37
	I2 + I3	5	5	6	6	37
5	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	48	46	60	58	338
	I2 + I3	47	46	60	58	334
10	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	310	289	376	349	1 390
	I2 + I3	310	287	375	347	1 370
15	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	1 170	985	1 370	1 180	4 040
	I2 + I3	1 160	977	1 350	1 150	3 960
20	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	3 050	2 630	3 550	3 090	9 370
	I2 + I3	3 030	2 590	3 530	3 050	9 130
25	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	6 930	5 740	7 970	6 570	18 700
	I2 + I3	6 880	5 690	7 880	6 500	18 400

Table 7: *The time, in milliseconds, required to parse individual* $Seq_{\mathrm{AHC}i}$ *and* Seq_{HC} *graphs against the grammar* GG_{HC}.

n	Parser	Seq_{AHC1}	Seq_{AHC2}	Seq_{AHC3}	Seq_{AHC4}	Seq_{AHC5}	Seq_{HC}
1	Original	3 230	3 260	18 000	18 000	119 000	10 500
	I2	5	5	4	5	4	5
	I2 + I3	5	5	4	4	4	4
5	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	240	240	230	228	220	222
	I2 + I3	91	90	87	88	83	85
10	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	7 490	7 380	7 150	7 090	6 890	6 890
	I2 + I3	710	704	676	683	667	665
15	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	71 800	71 700	70 500	69 900	69 000	68 900
	I2 + I3	3 810	3 840	3 740	3 720	3 670	3 690
20	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	381 000	378 000	373 000	373 000	367 000	367 000
	I2 + I3	15 900	15 400	15 800	15 300	15 600	15 900
25	Original	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$	$> 10^7$
	I2	1 400 000	1 400 000	1 380 000	1 380 000	1 370 000	1 360 000
	I2 + I3	51 200	50 300	50 000	50 100	50 000	50 600

Table 8: *The time, in milliseconds, required to parse individual* Seq_{ER} *and* Seq_{FC} *graphs against the grammars* GG_{ER} *and* GG_{FC}*, respectively.*

n	Parser	$GG_{\mathrm{ER}} + Seq_{\mathrm{ER}}$	$GG_{\mathrm{FC}} + Seq_{\mathrm{FC}}$
1	Original	4	10
	I2	4	6
	I2 + I3	3	6
5	Original	83	7 490
	I2	41	352
	I2 + I3	26	353
10	Original	561	$> 10^7$
	I2	261	9 330
	I2 + I3	141	9 310
15	Original	1 790	$> 10^7$
	I2	796	80 600
	I2 + I3	399	80 500
20	Original	4 150	$> 10^7$
	I2	1 850	407 000
	I2 + I3	891	407 000
25	Original	8 020	$> 10^7$
	I2	3 500	1 470 000
	I2 + I3	1 650	1 470 000

input graph G, is independent of the parsing algorithm. A theoretically optimal algorithm would produce exactly D production instances during Stage 1. The actual number of production instances depends on the algorithm: the original version of the Rekers-Schürr parser (column $\#PI_{\text{Original}}$) typically creates many more production instances than the improved one (column $\#PI_{\text{I2}}$). Note that Improvement 3 can only affect the parse time, not the number of production instances.

Table 9: *The growth of derivation length (D) and the number of created production instances ($\#PI$) with respect to input graph size (n).*

Grammar + sequence	D	$\#PI_{\text{Original}}$	$\#PI_{\text{I2}}$
$GG_{\text{AHC}} + Seq_{\text{AHC1}}$	$12n+1$	$\Omega(6^n)$	$36n-7$
$GG_{\text{AHC}} + Seq_{\text{AHC2}}$	$12n+1$	$\Omega(6^n)$	$36n-7$
$GG_{\text{AHC}} + Seq_{\text{AHC3}}$	$11n+1$	$\Omega(6^n)$	$53n-7$
$GG_{\text{AHC}} + Seq_{\text{AHC4}}$	$11n+1$	$\Omega(6^n)$	$53n-7$
$GG_{\text{AHC}} + Seq_{\text{AHC5}}$	$10n+1$	$\Omega(6^n)$	$655n-38$
$GG_{\text{HC}} + Seq_{\text{AHC1}}$	$12n+1$	$3240n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{HC}} + Seq_{\text{AHC2}}$	$12n+1$	$3240n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{HC}} + Seq_{\text{AHC3}}$	$11n+1$	$6120n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{HC}} + Seq_{\text{AHC4}}$	$11n+1$	$6120n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{HC}} + Seq_{\text{AHC5}}$	$10n+1$	$11160n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{HC}} + Seq_{\text{HC}}$	$10n+2$	$10080n^2+\Theta(n)$	$9n^2+\Theta(n)$
$GG_{\text{ER}} + Seq_{\text{ER}}$	$9n+1$	$41n+1$	$31n+1$
$GG_{\text{FC}} + Seq_{\text{FC}}$	$11n+2$	$42\cdot 2^{n-1}+\Theta(n^2)$	$6n^2+\Theta(n)$

Improvement 2 significantly reduces the computational effort for all four grammars, since all of them contain productions with interchangeable elements. For the grammars GG_{AHC} and GG_{FC}, the number of production instances is reduced even from an exponential to a polynomial function of the input graph size. Improvement 3 substantially accelerates parsing against the grammar GG_{HC}, since none of its productions can occur as an inconsistent instance in the graph $\overline{G}$ in Stage 1 (cf. Table 5). The grammar GG_{ER} is also favourable in this regard, but when parsing against the grammars GG_{AHC} and GG_{FC}, the costly *inconsistent* relationship has to be computed for almost every production instance.

The grammar GG_{HC} is more expressive than GG_{AHC}, since it can generate both acyclic and cyclic hydrocarbon graphs. However, the improved parser parses acyclic graphs more efficiently against GG_{AHC} than against GG_{HC} (cf. Table 9, column $\#PI_{\text{I2}}$). Let us explain why parsing against the grammar GG_{HC} leads to a quadratic, rather than linear, growth of the number of production instances. When parsing a hydrocarbon graph with k vertices 'C' against the grammar GG_{HC}, Stage 1 discovers all occurrences of $Rhs(p_3)$ (i.e., all C – C edges together with the incident vertices 'C') in the graph $\overline{G}$. For each C – C edge, a new vertex 'H' is attached to each of the two incident vertices 'C'. Ultimately, this process makes each vertex 'C' in $\overline{G}$ connected with four vertices 'H'. The graph $\overline{G}$ now contains $k(k-1) = k^2 + \Theta(k)$ occurrences of $Rhs(p_2)$, one for each pair of vertices 'C'. Stage 1 discovers all these occurrences and creates the same number of production instances. The quadratic

growth is thus an undesirable side-effect of allowing productions with disconnected RHSs.

In contrast to the grammar GG_{AHC}, the grammar GG_{HC} is completely insensitive to the arity of C – C edges; the parse time depends solely on the number of vertices 'C'. The reason is that in GG_{HC}, both single and multiple edges are created with the production p_3, while GG_{AHC} has separate productions for creating double and triple edges. Regarding the number of production instances, there is no difference between linear hydrocarbons (the sequences Seq_{AHC1} and Seq_{AHC3}) and their branched counterparts (the sequences Seq_{AHC2} and Seq_{AHC4}) for either of GG_{AHC} and GG_{HC}. However, owing to the search plan for the production p_2 of the grammar GG_{AHC}, a branched hydrocarbon is parsed against the grammar GG_{AHC} slightly faster than its linear counterpart.

8 Conclusion

We presented three original improvements of the Rekers-Schürr graph grammar parsing algorithm and showed their value by testing them on four meaningful graph grammars. Our first improvement abolished the requirement that all productions have connected RHSs and thus made it possible to parse grammars such as GG_{HC}. The second improvement eliminated the redundancy due to automorphisms of individual RHSs, which considerably reduced the parse time for all four test grammars. The third improvement determined the conditions under which the costly computation of the *inconsistent* relationship could be omitted. This improvement proved successful when parsing against the grammars GG_{HC} and GG_{ER}. However, our improvements are not specific to any of the presented test grammars, nor to the test suite as a whole.

There are several possible improvements and applications of our work. Improvement 3 could most probably be improved further. For example, 'ad hoc' reasoning and experiments indicate that the grammar GG_{ER} cannot generate inconsistent production instances at all. However, our present rules of Improvement 3 cannot reject the possibility that the productions p_5, p_6, and p_7 might occur as inconsistent instances (cf. Table 5).

Some potential applications of the improved parser were already noted in Section 1. The parser could be employed to parse or translate programs of real-world visual languages. Graph grammar inference is another domain where an efficient parser could find its use. The ideas expressed in this paper could also lead to a more efficient implementation of Triple Graph Grammars.

Appendix A: Correctness of Improvement 2

We shall now prove that the redundancy elimination scheme of Section 6.2.3 preserves the correctness of the parser, i.e., that it never causes any derivation to be lost because of its suppression of morphisms. Before proceeding to the main proof, however, we shall prove the following lemma:

Lemma 1. *Let* $v_0 : b_0 \xrightarrow{e_1 : a_1} v_1 : b_1 \xrightarrow{e_2 : a_2} \ldots$ *denote a walk that starts in a vertex* $v_0 \in G$ *with* $label(v_0) = b_0$ *and then passes through an edge* e_1 *labelled* a_1*, a vertex*

v_1 *labelled* b_1*, an edge* e_2 *labelled* a_2*, etc. If vertices* $v_0 \in G$ *and* $w_0 \in G$ *are interchangeable with respect to the graph* G*, then the following holds: If there exists a walk* $v_0 \colon b_0 \xrightarrow{e_1 \colon a_1} v_1 \colon b_1 \xrightarrow{e_2 \colon a_2} \ldots$*, then a walk* $w_0 \colon b_0 \xrightarrow{f_1 \colon a_1} w_1 \colon b_1 \xrightarrow{f_2 \colon a_2} \ldots$ *exists as well.*

Proof. If the vertices v_0 and w_0 are interchangeable in G, then (by Definition 1) there exists an automorphism h in G such that $h(v_0) = w_0$ and $h(w_0) = v_0$. Let us assume that v_0 is connected with a vertex v_1 labelled b_1 through an edge e_1 labelled a_1. Since the automorphism h, just like any other morphism, preserves both adjacency and labels, it maps the edge e_1 and the vertex v_1 to an edge f_1 labelled a_1 and a vertex w_1 labelled b_1, respectively, such that w_1 is a neighbour of w_0 through the edge f_1. Such a vertex w_1 and an edge f_1 must exist; otherwise, the automorphism h would not exist either. Therefore, h maps a walk $v_0 \colon b_0 \xrightarrow{e_1 \colon a_1} v_1 \colon b_1$ to a walk $w_0 \colon b_0 \xrightarrow{f_1 \colon a_1} w_1 \colon b_1$. By the same argument as above, a neighbour v_2 of v_1 through an edge e_2 is mapped to a neighbour w_2 of w_1 through an edge f_2 such that $\mathit{label}(w_2) = \mathit{label}(v_2)$ and $\mathit{label}(f_2) = \mathit{label}(e_2)$. A walk $v_0 \colon b_0 \xrightarrow{e_1 \colon a_1} v_1 \colon b_1 \xrightarrow{e_2 \colon a_2} v_2 \colon b_2$ is thus mapped to a walk $w_0 \colon b_0 \xrightarrow{f_1 \colon a_1} w_1 \colon b_1 \xrightarrow{f_2 \colon a_2} w_2 \colon b_2$. Continuing in this direction, both walks can be extended indefinitely. □

Proposition 5. *The proposed redundancy elimination scheme preserves the correctness of the parser.*

Proof. We have to show that our scheme eliminates only purely redundant work and that it has no effect on the parsing algorithm apart from a (significant) speedup. For simplicity, let us assume that a given production p contains a single IC, which comprises k vertices from $\mathit{Rhs}(p)$. The proof can be straightforwardly extended to the case that p contains several mutually independent ICs, since the ICs can be treated independently from each other.

Let us pick two vertices from the IC, say v_0 and w_0, and let $v_0 \colon b_0 \xrightarrow{e_1 \colon a_1} v_1 \colon b_1 \ldots \xrightarrow{e_r \colon a_r} v_r \colon b_r \xrightarrow{e_{r+1} \colon a_{r+1}} v_{r+1} \colon b_{r+1} \ldots \xrightarrow{e_s \colon a_s} v_s \colon b_s$ be a walk where v_0, e_1, v_1, …, e_r, $v_r \in \mathit{Rhs}(p)$ and e_{r+1}, v_{r+1}, …, e_s, $v_s \in \mathit{Union}(p) \setminus \mathit{Rhs}(p) = \mathit{Xlhs}(p)$. Since, by Definition 4, the vertices v_0 and w_0 are interchangeable in both $\mathit{Rhs}(p)$ and $\mathit{Union}(p)$, there must, by Lemma 1, also exist a walk $w_0 \colon b_0 \xrightarrow{f_1 \colon a_1} w_1 \colon b_1 \ldots \xrightarrow{f_r \colon a_r} w_r \colon b_r \xrightarrow{f_{r+1} \colon a_{r+1}} w_{r+1} \colon b_{r+1} \ldots \xrightarrow{f_s \colon a_s} w_s \colon b_s$ such that w_0, f_1, w_1, …, f_r, $w_r \in \mathit{Rhs}(p)$ and f_{r+1}, w_{r+1}, …, f_s, $w_s \in \mathit{Xlhs}(p)$.

When Stage 1 discovers an occurrence of $\mathit{Rhs}(p)$ in the graph $\overline{G}$ via a morphism h, it maps the walk $v_0 \colon b_0 \xrightarrow{e_1 \colon a_1} v_1 \colon b_1 \ldots \xrightarrow{e_r \colon a_r} v_r \colon b_r$ in $\mathit{Rhs}(p)$ to a corresponding walk $v'_0 \colon b_0 \xrightarrow{e'_1 \colon a_1} v'_1 \colon b_1 \ldots \xrightarrow{e'_r \colon a_r} v'_r \colon b_r$ (where $v'_i = h(v_i)$ for $i \in \{0, \ldots, r\}$ and $e'_i = h(e_i)$ for $i \in \{1, \ldots, r\}$) in the occurrence, and analogously for the walk $w_0 \colon b_0 \xrightarrow{f_1 \colon a_1} w_1 \colon b_1 \ldots \xrightarrow{f_r \colon a_r} w_r \colon b_r$. Stage 1 then augments the occurrence by attaching a copy of $\mathit{Xlhs}(p)$ to it. The walk $(v_r \colon b_r) \xrightarrow{e_{r+1} \colon a_{r+1}} v_{r+1} \colon b_{r+1} \ldots \xrightarrow{e_s \colon a_s} v_s \colon b_s$ from $\mathit{Xlhs}(p)$ corresponds to attaching a chain of elements with labels a_{r+1}, b_{r+1}, …, a_s, b_s to the vertex $v'_r = h(v_r)$, and the walk $(w_r \colon b_r) \xrightarrow{f_{r+1} \colon a_{r+1}} w_{r+1} \colon b_{r+1} \ldots \xrightarrow{f_s \colon a_s} w_s \colon b_s$ corresponds to attaching a chain of elements with labels a_{r+1}, b_{r+1}, …, a_s, b_s to the vertex $w'_r = h(w_r)$. Both chains are isomorphic and attached to the vertices at the same relative position from $h(v_0)$ and $h(w_0)$, respectively, since

the walk from $h(w_0)$ to $h(w_r)$ (and further to $h(w_s)$) has the same length and passes through the same labels as that from $h(v_0)$ to $h(v_r)$ (and further to $h(v_s)$). Since this observation holds for any corresponding pair of walks starting from v_0 and w_0, the occurrence is augmented from the perspective of the vertex $h(v_0)$ in the same way as from the perspective of $h(w_0)$. This property can be generalised to all k vertices of the IC, since the vertices v_0 and w_0 were selected arbitrarily from the IC.

Since the IC comprises k vertices, each occurrence of $Rhs(p)$ is discovered $k!$ times. For each discovery, the occurrence is augmented in the same way from the perspective of any vertex composing the IC; as a group, the elements attached to the occurrence 'look' the same to all IC vertices. Since the IC vertices of $Rhs(p)$ are, as a group, always mapped to the IC vertices of the occurrence, each of the $k!$ discoveries augments the occurrence in the same way with respect to a fixed group of vertices, i.e., those of the IC. The work is thus merely duplicated $k!$ times. This redundancy can be safely eliminated, which is what our scheme does. □

References

[1] Rekers, J. and Schürr, A.: 'Defining and parsing visual languages with layered graph grammars'. Visual Languages and Computing, 1997, 8, pp. 27–55.

[2] Zhang, D.-Q., Zhang, K., and Cao., J.: 'A context-sensitive graph grammar formalism for the specification of visual languages'. The Computer Journal, 2001, 44, (3), pp. 186–200.

[3] Hermann, F., Ehrig, H., and Taentzer, G.: 'A typed attributed graph grammar with inheritance for the abstract syntax of UML class and sequence diagrams'. Electronic Notes in Theoretical Computer Science, 2008, 211, pp. 261–269.

[4] Rafe, V., Rahmani, A. T., Baresi, L., and Spoletini, P.: 'Towards automated verification of layered graph transformation specifications'. IET Software, 2009, 3, (4), pp. 276–291.

[5] Lin, L., Wu, T., Porway, J., and Xu, Z.: 'A stochastic graph grammar for compositional object representation and recognition'. Pattern Recognition, 2009, 42, (7), pp. 1297–1307.

[6] Flasiński, M., and Myśliński, S.: 'On the use of graph parsing for recognition of isolated hand postures of Polish Sign Language'. Pattern Recognition, 2010, 43, (6), pp. 2249–2264.

[7] Lavirotte, S., and Pottier, L.: 'Optical formula recognition'. Proc. 4th Int. Conf. on Document Analysis and Recognition, Ulm, Germany, August 1997, pp. 357–361.

[8] Ehrig, K., Küster, J.M., and Taentzer, G.: 'Generating instance models from meta models'. Software and System Modeling, 2009, 8, (4), pp. 479–500.

[9] Blostein, D., and Schürr, A.: 'Computing with graphs and graph transformations'. Software – Practice and Experience, 1999, 29, (3), pp. 197–217.

[10] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (Eds.): 'Handbook of graph grammars and computing by graph transformation. Vol. 2: Applications, languages, and tools' (World Scientific, 1999).

[11] Rekers, J., and Schürr, A.: 'A parsing algorithm for context-sensitive graph grammars'. Technical Report 95-05, Leiden University, Leiden, The Netherlands, 1995.

[12] Vermeulen, J.T.: 'Viability of a parsing algorithm for context-sensitive graph grammars'. Master's thesis, Leiden University, Leiden, The Netherlands, 1996.

[13] Rozenberg, G., and Welzl, E.: 'Boundary NLC graph grammars – basic definitions, normal forms, and complexity'. Information and Control, 1986, 69, (1–3), pp. 136–167.

[14] Schürr, A.: 'Specification of graph translators with Triple Graph Grammars'. Proc. 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany, June 1994, pp. 151–163.

[15] Königs, A., and Schürr, A.: 'Tool integration with Triple Graph Grammars – a survey'. Electronic Notes Theoretical Computer Science, 2006, 148, (1), pp. 113–150.

[16] Flasiński, M.: 'Inference of parsable graph grammars for syntactic pattern recognition'. Fundamenta Informaticae, 2007, 80, (4), pp. 379–413.

[17] Kukluk, J., Holder, L., and Cook, D.: 'Inferring graph grammars by detecting overlap in frequent subgraphs'. Int. J. of Applied Mathematics and Computer Science, 2008, 18, (2), pp. 241–250.

[18] Tucci, M., Vitiello, G., and Costagliola, G.: 'Parsing nonlinear languages'. IEEE Trans. on Software Engineering, 1994, 20, (9), pp. 720–739.

[19] Javed, F., Mernik, M., Bryant, B.R., and Sprague, A.: 'An unsupervised incremental learning algorithm for domain-specific language development'. Applied Artificial Intelligence, 2008, 22, (7–8), pp. 707–729.

[20] Dubey, A., Jalote, P., and Aggarwal, S.K.: 'Learning context-free grammar rules from a set of programs'. IET Software, 2008, 2, (3), pp. 223–240.

[21] Blostein, D., Fahmy, H., and Grbavec, A.: 'Practical use of graph rewriting'. Technical Report 95-373, Queen's University, Kingston, Ontario, Canada, January 1995.

[22] Janssens, D., and Rozenberg, G.: 'Graph grammars with node-label controlled rewriting and embedding'. Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science, Osnabrück, Germany, October 1982, pp. 186–205.

[23] Kaul, M.: 'Parsing of graphs in linear time'. Proc. 2nd Int. Workshop on Graph Grammars and Their Application to Computer Science, Osnabrück, Germany, October 1982, pp. 206–218.

[24] Bunke, H., and Haller, B.: 'A parser for context free plex grammars'. Proc. 15th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Castle Rolduc, The Netherlands, June 1989, pp. 136–150.

[25] Seifert, S., and Fischer, I.: 'Parsing string generating hypergraph grammars'. Proc. 2nd Int. Conf. on Graph Transformations, Rome, Italy, September 2004, pp. 352–367.

[26] Minas, M.: 'Parsing of adaptive star grammars'. Proc. 2nd Int. Workshop on Graph and Model Transformation, Brighton, UK, September 2006, pp. 1–14.

[27] Bottoni, P., Taentzer, G., and Schürr, A.: 'Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation'. Proc. 2000 IEEE Int. Symposium on Visual Languages, Seattle, WA, USA, September 2000, pp. 59–60.

[28] Crimi, C., Guercio, A., Nota, G., Pacini, G., Tortora, G., and Tucci, M.: 'Relation grammars and their application to multi-dimensional languages'. Visual Languages and Computing, 1991, 2, (4), pp. 333–346.