Viability of a Parsing Algorithm for Context-sensitive Graph Grammars

Jeroen T. Vermeulen

August 26, 1996

Abstract

Graph Grammars describe formal languages similar to the textual ones commonly used to define computer languages, but their productions operate on graphs instead of on text. They are *context-sensitive* when productions rewrite patterns of symbols rather than single symbols. Parsing such grammars is inherently very hard because among other reasons, the input is not sequential in nature.

An algorithm for parsing a large class of context-sensitive graph grammars has been developed by Jan Rekers and Andy Schürr. This thesis describes a first implementation of this algorithm as well as several improvements, additional work, examples, theory of operation and performance characteristics. Future and existing optimizations are discussed.

1 Introduction

1.1 Graph Grammars

Although formal grammars are commonly used in computer science for the description of textual languages, relatively little progress has as yet been made towards understanding and implementing *visual* languages. A visual language defines a set of diagrams rather than a set of textual sentences. Commonly used visual languages include electrical diagrams, Petri nets, and ER diagrams; however they are only rarely defined in terms of formal grammars.

Interactive graphical editors are already available for some of these visual languages, eg. circuit-board design tools and flow-chart editors, but they are usually part of a dedicated environment and only allow syntactically correct diagrams to be generated by limiting the editing primitives to valid transformations on the underlying interpretation (or semantic value) of the diagram.

This procedure, known as *syntax-directed editing*, makes it easier on the environment but requires the user to perform a sequence of actions that is not necessarily related to his conception of the diagram he is creating. Other environments may allow more or less random editing, but provide no way of storing an unfinished diagram to disk because they don't know how to handle expressions that are not members of the formal language they implement.

It is believed that the general ability to parse visual languages after *free-form editing* with a generic graphical editor, which is accepted practice for textual languages such as C or TEX, could be used to improve user-friendliness of user interfaces and bring a host of applications that currently require expert knowledge closer to the end user.

Setting up a relational database, for instance, normally involves designing an entity-relationship architecture in ER notation, and manually converting that to a textual representation in the data-definition language included with the DBMS package. Given some kind of parser for ER diagrams this conversion

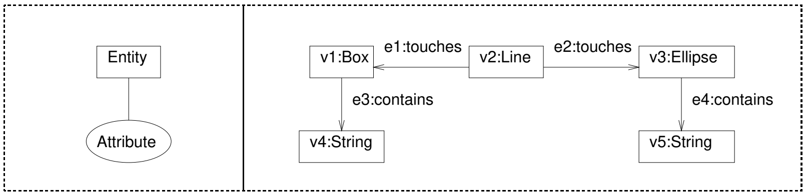


Figure 1: A simple ER diagram and its Spatial Relationships Graph

could be automated, reducing the risk of mistakes and eliminating the need to master a separate textual DDL.

To make such technology useful in practice, a generic framework is sought that may be applied to any of a reasonably large class of visual languages, so that the same parsing ability can be delivered for a new language with no more human effort than writing a formal grammar for it.

1.2 Project Goals

This project aims to evaluate an approach towards parsing visual languages developed by J. Rekers and A. Schürr. This approach attacks the parsing problem by processing graphical input in two main stages.

The first stage, a lexical scanner, reads a raw visual "expression" such as an ER diagram and abstracts it into a *spatial relationships graph* or SRG, representing the graphical elements of the input expression as vertices and their visual interrelationships (such as proximity of a string to a box, or a circle being contained in another circle) as edges between them. See figure 1 for a simple example.

The second stage, the actual parser, analyzes the resulting graph by interpreting it as an expression in a *graph grammar* using an algorithm developed by Rekers and Schürr [17, 14, 15, 13]. It is this algorithm that this report focuses on.

The aim is to determine the practical usefulness of the approach, and of the parsing algorithm in particular, by providing an implementation and studying its behaviour, as well as the "friendliness" of the language formalism used. It will become apparent that practical considerations will remain important and that there is more to defining grammars than meets the eye—but these qualities are inherent in every useful invention, starting with the wheel.

1.3 Parser Framework

1.3.1 Automated Parsing

Due to the difference in complexity between visual and textual languages, and perhaps also to the lack of standardization in the field of visual language implementations, a parser for such grammars as presented in this report will not fit into an application framework as easily as a lex/yacc or flex/bison parser. Although formalisms such as the one presented in section 1.4 may make valiant attempts to fold similar functionality into a single parser generator, there is always a cost in usability as well as a perceived improvement.

Most parsers today are created in a standard fashion, and through a highly standardized set of layers. First comes a lexical analyzer generated by a tool such as flex from a formal description tailored to the application. The generated program feeds its output into eg. an LALRparser generated by bison from yet another description file. The latter traverses a conceptual *parse tree* for the input file and interfaces tightly with user code through embedded statements.

The parser generator implemented in the scope of this project may be compared to bison: The initial abstraction from diagrammatic input to a graph is performed by an extraneous program (the lexical scanner, which can be likened to flex). The parser then proceeds to reconstruct the set of applications of language productions needed to generate the diagram starting with the grammar's start symbol.

At this point the similarity ends. As visual languages are generally speaking multi-dimensional, there is no inherent linear order in which embedded statements could be executed. Conversely, context-sensitivity creates more and more subtle dependencies between production applications, replacing the parse tree common to all context-free languages by a directed acyclic graph (DAG) of applied productions and their interdependencies, any linearization of which constitutes a valid derivation sequence and vice versa.

In contrast to bison parsers, a parser generated by our implementation does not implicitly traverse the parse DAG as it is being constructed, but instead produces it as explicit output, in the form of a graph data structure which may either be examined and traversed by application-specific code, or more likely, be processed by yet another software layer to produce more meaningful output at a higher abstraction level. In effect, an extra stage has been added to the traditional automatically-generated parser.

Although graph editors might be used to construct graph expressions directly, one will normally want to use a lexical scanner such as the one implemented by A. de Graaf [8] which reads arbitrary input drawings and translates them into spatial relationships graphs suitable for graph-grammar parsing.

1.3.2 A Day's Work

Practical use of this implementation of the Rekers-Schürr parsing approach for an end-user software product capable of interpreting a graph language is currently envisioned along the following lines. Direct graph input from an editor such as EDGE [21] is assumed for brevity as the case for a generic visual language would involve lexical analysis as well.

- The application programmer writes a grammar that generates exactly the set of all acceptable input graphs. Each production takes the form of a file containing a single graph specification in the GRL format.

- Once this is done, the *Rekers-Schürr Parser Generator* is invoked on the grammar (the program is called RSPGen) which verifies that the grammar satisfies certain requirements to be outlined in later sections of this report and, if it does, constructs a parser for the given language.

  If we were making coffee instead of a parser, this would be the equivalent of *heating water*.

- The generated parser consists of C++ source code tailored for the input grammar. In reality the larger part of the actual parser's source code, called the *skeleton* code, does not vary for different grammars so it is not generated again for each grammar. Instead it can simply be copied into a working directory.

  The skeleton code is our *instant coffee*; we just spoon the stuff into a mug without any further processing.

- The code generated by RSPGen is merged into the working directory which already contains the skeleton code. Once merged together they can be compiled with a simple make command; the resulting parser, when invoked on an input graph (an expression in the language), returns the parse DAG.

  Returning to coffee terms, we've just added hot water to the instant coffee waiting in our mug. It doesn't take much imagination to see that all we need to do now is to stir it (the programmer's equivalent of which is typing "make" to start compilation) and finally to drink it—or in this case, to integrate it with the application and put its output to good use.

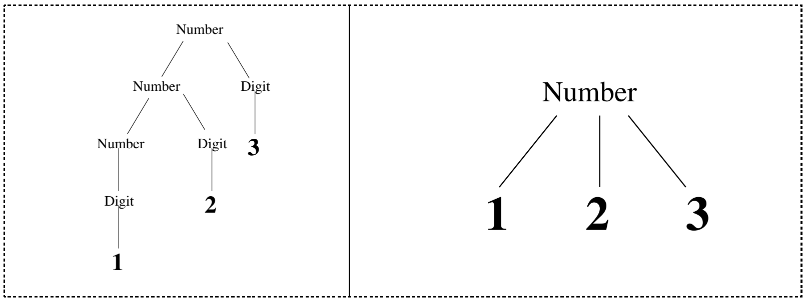


Figure 2: Parse tree & abstract syntax tree for the number 123

However, as mentioned earlier, the DAG returned by the parser is a relatively meaningless data structure and it may well be desirable to further process it before feeding it back to the application, kneading it into a more useful form. Several suggestions for such further processing (generally described as *high-level parsing*) are summarized in [15], the most interesting perhaps being that of *coupled graph grammars*. This approach could be used to formalize the translation from a parse tree to an abstract syntax tree which is usually much closer to the application's perception of the structure of its input.

Figure 2 shows a small but practical example of a parse tree translated into a syntax tree. It is based on a simple grammar that generates the natural numbers:

$$\begin{aligned} \text{Number} &::= \text{Digit}|\text{Number Digit} \\ \text{Digit} &::= 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

The parse tree for the number 123 is shown on the left-hand side of figure 2, with its abstract syntax tree on the right. The latter is obviously a lot closer to the notion of "a sequence of digits" that it can be seen to represent.

As proposed by Schürr [20], a similar notion could be exploited for graph grammars by coupling the application of a graph-grammar production in eg. a "spatial relationships graph" sentential form to the application of a "mirror" production in another, more abstract grammar in another sentential form, the corresponding *abstract relationships graph*, so that a full derivation for a given input sentence produces an equivalent sentence in a more useful language at a higher level of abstraction.

No such extra parser layer exists at this time, and the possibilities have not yet been fully explored. However the output of the parser constructed during this project is sufficiently generic to support such additions.

1.4 Related Work

A different approach to parsing a class of visual grammars called Constraint Multiset Grammars (CMG's) was devised by Kim Marriott et al. [11, 5, 6]. Their formalism works along context-free lines and implements all relationships between graphical elements as constraints.

In short, a CMG consists of a start symbol and a set of rewriting rules or *productions* of the form

$$\begin{aligned} P(t_1 a_1, \ldots, t_m a_m) \quad ::= \quad & P_1, \ldots, P_n \; where \; (\\ & exist \; P'_1, \ldots, P'_m \end{aligned}$$

$$where\ (C))$$
$$\{\ \dots\ \}$$

The meaning of this is that when generating a sentence in the grammar, any graphical element of type P may be replaced by the group of elements $P_1, \dots, P_n$ in the current sentence, provided that elements $P'_1, \dots, P'_n$ can be found such that C holds.

The optional { ... } clause contains statements in the semantic domain that compute *attribute values* a_i (of type t_i) for P, perhaps using those of $P_1, \dots, P_n$ and $P'_1, \dots, P'_m$. These are analogous to synthesized attributes in context-free textual grammars.

The CMG formalism is powerful enough to generate just about any visual grammar directly, provided that the semantic domain is sufficiently expressive. There is no clear distinction between lexical properties—such as whether the head of an arrow is "close enough" to an object to say that it is touching it—and syntactic information, eg. which object the arrow points to. Arbitrary relationships between visual elements can thus be included in the grammar by writing the necessary C++ functions.

As an example, let's look at a CMG-style grammar for a small subset of the ER diagram language. This subset only has relationships (diamond-shaped boxes) and entities (rectangular boxes). There must be at least one entity and each relationship must be connected to at least two entities (although they may actually be two connections to the same entity) by either a single line or a double line.

First, we need a rule that generates all the elements of the diagram. We do this with a special *collection production* which has been introduced into the formalism for practical reasons.

```
diagram ::= E:entities,
            R:relationships,
            C:connections.
entities ::= all S:entity where (true).
relationships ::= all R:relationship where (true).
```

Next we define what entities and relationships look like, and how they relate to each other. An `entity` derives a single rectangle (`rect`) and has an attribute `shape` containing its attributes such as position and size.

```
entity(rect shape) ::= B:rect
{
  shape = B;
}
```

Relationships are more interesting. A relationship consists of a diamond-shaped box, but it also needs to be connected to at least two entities. We introduce existentially quantified constraints to demand the presence of two such connecting lines. The existence of the utility function `touches` is presumed.

```
relationship(diamond shape, set participants) ::= D:diamond
  where (
    exist C1:connection, C2:connection
    where (
      touches(C1,D), touches(C2,D), C1.shape != C2.shape
```

```
    )
  )
{
  shape = D;
  participants = {};
}
```

We've decreed that there must be at least two connecting lines touching the diamond but the above production doesn't actually derive them; a separate production is needed for that. A connecting line has a starting point and an end point and touches the diamond of a relationship on one end and the rectangle of an entity on the other.

```
connections ::= all C:connection
  where (
    exists E:entity, R:relationship where (
      touches(C.shape,R.shape), touches(C.shape,E.shape)
    )
  )
  {
    R.participants.add(E);
  }
```

Of course a connecting line may be a single or a double line. Let's look at double lines first: These consist of two parallel lines close together. A random one of the two is picked as the double line's "shape".

```
connection(line shape) ::= L1:line, L2:line
  where (
    parallel(L1,L2), close(L1,L2), L1.shape != L2.shape
  )
{
  shape = L1;
}
```

Strangely enough, a single line is a little harder to describe. For practical reasons the grammar must be *deterministic*, meaning that any two possible derivation sequences for a single diagram must be equivalent.

As the two parallel lines in the production above could also be seen as two distinct connecting lines, and any resulting derivations could hardly be called equivalent to the intended one, we must cater for the possibility that we're looking at one of two parallel lines which should be recognized as one "double" line.

The solution offered to us by the CMG formalism is a negatively quantified constraint; the production applies only if *no* lines exist in the diagram that are both close to `L1` and parallel to it.

```
connection(line shape) ::= L1:line
  where (
    not exist L2:line
      where (
        close(L1,L2), parallel(L1,L2), L1.shape != L2.shape
      )
  )
  {
    shape = L1;
  }
```

In this example we see how a visual language may be expressed in a context-free formal grammar using semantic constraints to define the relationships between symbols and their context. The question of how to handle such relationships is known as the *embedding problem*.

The generic constraints approach taken by the CMG formalism has the obvious advantages of flexibility and expressive power, but some disadvantages are also apparent:

- *Computational costs*—the constraints on an applied production may potentially be satisfied by any combination of graphical elements of the right types, so the number of times they must be evaluated may grow dramatically in relation to the size of the input sentence.

  Example: In the ER grammar above, a `relationship` is required to be touched by at least two `connections`. Barring exceedingly complicated analysis by the implementation, every combination of any two `connections` in the entire diagram must be considered a candidate for these constraints.

  If the diagram contains a total of n `connections`, the constraints for this production may potentially need to be evaluated up to n^2 times for different candidate combinations.

- *Structural clarity*—as the grammar formalism concentrates on graphical elements exclusively, their interrelationships are degraded to semantic properties.

  Example: Generating the set of `connections` in a separate production obfuscates their real rôle in the language, which is to connect an `entity` on one end to a `relationship` on the other. Instead their place in the diagram as a whole is added almost as an afterthought.

  The grammar can of course be rewritten to generate each connection separately with constraints defining its relationship to the objects it connects. This would trade in higher grammar complexity for a more well-structured definition, but it can never promote such interrelationships to "first-class properties" of the formalism—one might say that the formalism discourages such well-structured grammar definitions.

2 Graph-grammar Formalism

2.1 Description

Describing visual languages in a single formalism can be quite awkward, as may be seen from the example in section 1.4. The parser in the Rekers-Schürr approach does not deal with visual languages directly; it operates on *graph grammars* instead, letting the lexical analyzer deal with the recognition of the `touches` relationship and such.

The language generated by a graph grammar contains only graphs, eg. all spatial relationships graphs for a particular visual language (section 1.2). Although useful visual languages rarely consist entirely of graphs, they can almost always be adequately described in terms of SRG's and these in turn may be parsed in terms of a graph grammar.

In this section, a formalism is presented to describe graph grammars suitable for parser generation by our implementation. One attractive feature compared to other formalisms such as CMG's is its context-sensitivity, meaning that productions can rewrite entire expressions instead of just single symbols.

A production may also leave some elements of the rewritten expression untouched, effectively specifying the *context* in which the production is applicable. Interrelationships between elements can thus be expressed directly and intuitively in the grammar without resorting to constraints in the semantic domain, tackling the embedding problem in a relatively natural and potentially efficient way.

2.2 Definition

A *graph grammar* in this formalism consists of a set of *productions* of the form (L, R), where L and R are non-empty graphs, and a non-empty *initial graph* S which is the grammar's start symbol. These graphs are called the productions' *left-hand side* and *right-hand side*, respectively.

Each *element* (vertex or edge) in the graphs of a graph grammar gg has a type or *label* similar to the different types of visual elements in a CMG grammar (eg. `diamond`, `entity` or `plainline`). The set of labels L_{gg} of gg is partitioned into a set of vertex labels and one of edge labels.

A graph element can be a terminal or a non-terminal symbol depending on its label, that is, the set of labels L_{gg} in a grammar gg is partitioned into a set of terminals and a set of nonterminals as well as into a set of vertex labels and a set of edge labels.

Terminal labels are the labels that may occur in expressions in the language generated by gg. They must be declared terminal by the grammar, and the parser will only accept an input graph if all of its elements have labels of terminal type. All other labels are called *nonterminal* labels. Graph elements are said to be either *terminals* or *nonterminals* depending on their label.

Graphical sentences in a graph grammar gg are generated in a context-sensitive manner: A graph G is initialized to gg's initial graph S. The following process is repeated at least until all terminals have been removed from the graph: Find a subgraph of G that matches a graph L in some production $P \equiv (L, R)$ in gg and *apply* P: Replace the matched subgraph with R, leaving the elements of G that match $L \cap R$ untouched. Thus the elements matched by $L \backslash R$ are removed and copies of the elements of $R \backslash L$ are inserted. The graphs G that can be encountered during this process are the *sentential forms* of gg. A sentence contains no nonterminals but a sentential form usually does.

As you can see, the graphs L and R in a production (L, R) may have a non-empty intersection $K \equiv L \cap R$, representing *context elements* that are referenced but not affected by the application of P. Each element of K must have the exact same label in R and in L as it does in K). K is also called the *interface graph* because it describes how new elements may be attached to existing elements of G when applying P.

Implementation Note

> Each production is encoded as a single graph, with each element colour-coded as a member of K, or of either L or R exclusively. The parser generator then extracts L and R as different "views" of this unified representation.

Naturally an edge in a production may never connect a vertex in L which is not in R (such an element is a member of the production's *exclusive left-hand side*) to one that is in R but not in L (such elements form the production's *exclusive right-hand side*), as conceptually the two vertices can not exist

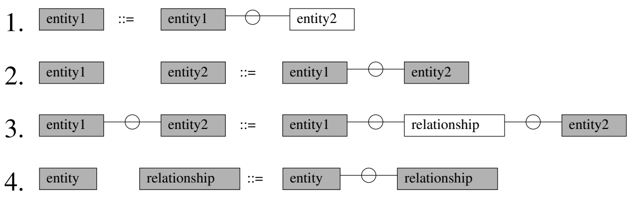


Figure 3: Main productions for "simple ER" language

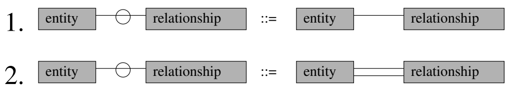


Figure 4: Productions for single and double lines

together in the same graph. Neither can an edge be in K unless both adjacent vertices are also in K. Put differently, L, R and K must be proper graphs; meaning that each edge must have one vertex on either end, which must both be in the same graph as well.

When finding a match of L in G, two elements of K may be matched by the same element of G; however the elements of $L \backslash R$ and of $R \backslash L$ must each find unique matching elements for themselves.

Example

Let's look at our "simple ER" grammar from section 1.4 again, and define a similar language in terms of a graph grammar. For simplicity, we will assume here that the diagrams are supplied in graph form directly[1] so we need not deal with their spatial relationships graphs. In fact SRG's are conceptually similar to ER diagrams.

First of all, we need an *initial graph* for our grammar. This is simply a single `entity`. Next we can simply tag the other graph elements onto this vertex and onto each other. In figure 3 you will see the four productions that can be used to build the diagram in stepping-stone fashion starting with this initial graph.

The boxes with gray backgrounds are the unaffected context elements; the white boxes and all arrows in the left- and right-hand sides of the productions are the deleted and generated elements, respectively.

Although a shorter grammar exists for the same language, none of these productions could be safely omitted from this grammar. The first is needed to generate new entities attached to the existing ER diagram by at least one relationship, the second connects pairs of entities arranging for relationships to be inserted between them by the third production, and number four is needed for creating non-binary relationships.

Figure 4 generates the two different kinds of lines. Each of the lines generated by productions 3 and 4 in figure 3 is replaced by either a single line or a double line. The graph abstraction allows us to skip over the indeterminism we encountered for the CMG version in section 1.4; we simply assume that double lines have already been recognized and abstracted into one instance of a special type of connecting line[2]

[1] By a lucky coincidence all ER diagrams take the form of graphs, with two types of vertices for entities and relationships and with single and double lines as edge labels.

[2] A grammar using a lexical scanner would of course not be able to make this assumption and would run into the same problem as the CMG formalism w.r.t. this construct. A solution is explored in section 2.3

Note how context-sensitivity is employed to specify that lines between entities must generate relationships, whereas lines connecting entities to relationships correspond to the `connection` type in the example of section 1.4.

At this point it should be noted that using these productions it is impossible to generate an unconnected ER diagram since all productions' right-hand sides are connected graphs. In fact a similar grammar without this property would be much easier to conceive, and its language would be a possibly desirable superset of the language defined by this example. The reasons for the restriction made here will become more apparent in the following sections.

2.3 Constraints vs. Context

Incorporating context into productions has the clear advantage of making structural properties a part of the grammar, which is one of the central goals of formal languages. Consider the third production in figure 3 (the ER example): It clearly formalizes the fact that each relationship is connected to entities by at least two different lines without using constraints of any kind.

An added benefit, at least potentially, of handling context in this way is one of efficiency with regard to the embedding problem: The graph-grammar formalism will only allow constraints to be imposed on adjacent graph elements, giving an implementation a clear indication of "where to look" and reducing the number of candidate elements in a context pattern considerably. Generic and hard-to-optimize expressions of the form "$\exists l \in lines : \ldots$" are avoided.

Negative quantifiers can also be expressed in the graph-grammar formalism, albeit in a less powerful and less obvious way. This flows from the fact that this type of context constraint is expressed in an implicit rather than an explicit way through the use of the *dangling-edge condition*.

The dangling edge condition states that, for a production instance to be applied to any sentential form, no edges may be left "dangling" by its application. An edge is said to be left dangling when a vertex v in the graph is deleted by the application of some production instance pi, but some edge e adjacent to v is not. In this case e is not implicitly dropped from the graph, but instead its presence blocks the application of pi until e is deleted so that the sentential form really remains a proper graph.

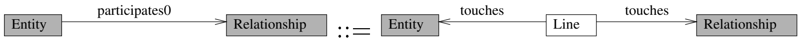
The dangling edge condition applies in both directions: Whether one is generating a sentence from the grammar's initial graph or attempting to reduce an input graph to the initial graph (parsing), no vertex may be removed unless all of its adjacent edges have been accounted for—any more than, as seen from the other direction, an edge could be created with either its source or its target vertex missing.

An interesting example of the use of the dangling edge condition to express a negatively quantified constraint is the problem we encountered in section 1.4 where a line connecting an `entity` to a `relationship` can only be recognized as a `connection` of "single line" type if there is no other line close and parallel to it. In the CMG grammar, the existence of "double line" connections gave rise to potential indeterminism that may be rather hard to deduce from a grammar definition.

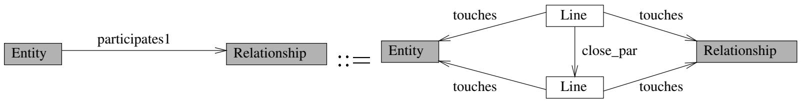
Up to now we have assumed for simplicity that an ER diagram is produced in graph form directly so that this problem was not a factor in our ER graph grammar. But consider a situation where ER diagrams are drawn in a free-form drawing program such as idraw or fig and subsequently processed by a lexical scanner to produce their spatial relationships diagrams such as the one shown in figure 1.

After lexical scanning all visual elements of the diagram (even the lines!) are represented as nodes in a graph which is then fed into a graph-grammar parser. Given any two lines in the original diagram that share both their adjacent `entity` and `relationship`, we are now faced with the problem of determining in our grammar whether the two should be read as one "double line" or whether they are really two distinct connections between a single `relationship` and two participants of the same `entity` type.

Luckily, and in accordance to the grammar writer's intuition, the case for a single line is simple. It can be reduced to a connection by the following straightforward production:



And what about double lines? Let's say that if the lexical scanner runs into two lines that are close together and parallel, it connects their objects in the spatial relationships graph with an edge labeled "close_par" (in any direction). Recognizing this case is as simple as writing a production



One may well ask how we can tell these two cases apart. The answer is surprisingly simple: We already have. If two lines share the same Entity and Relationship, either of the following must be the case:

- The two are not "close and parallel".

  There is no close_par line between their SRG vertices so only the single-line production applies.

- The two are "close and parallel".

  The SRG representing them are connected by an edge labeled close_par. In this case the single-line production can't be applied because in deleting the two Line vertices, the close_par edge would be left dangling!

The negative existential qualifier is neatly tucked away in the formalism. As can be seen here, failing to take ambiguities and indeterminism into account in defining a graph grammar will result in one that accepts a smaller language, rather than accepting too much and producing indeterministic results.

As with traditional context-free textual languages, the main effort of defining a graph grammar lies in covering the entire language rather than in identifying and excluding spurious derivations. Maintenance of graph grammars is facilitated by the fact that adding productions produces fewer unforeseen side effects and has no hidden interactions with the existing productions.

On the down side, implicit constraints may also complicate the definition of a graph grammar. There would, for instance, be no way of extending our ER grammar so that an entity could be directly rewritten to some other kind of vertex once it has been generated; this would have to be achieved by having a production delete the entity and generate the new object, and then re-tie all adjacent connections to it. This would be impossible due to the dangling-edge condition: A vertex cannot be removed from a sentential form before all adjacent edges have been deleted. The grammar would have to be restructured from the ground up to accommodate this new object type.

2.4 Language Restrictions

Now that we are equipped to express our graph grammars using graph productions, and we know how to generate a sentence in the language of such a grammar, we come to the practicalities of inverting that process, or *parsing* such a sentence by reconstructing the set of production applications (or production *instances*) which, when applied in a suitable order, will generate the given input sentence from the grammar's initial graph.

As it turns out, several restrictions must be imposed on the graph grammars we wish to parse in order to ensure that the existence or nonexistence of a set of production instances generating the sentence can be determined in finite time, ie. that the *membership problem* for these grammars is decidable.

In addition to this, the ability of any implementation of a grammar to deliver a generating set of production instances for any given sentence in its language, and to do so within a "reasonable" time span, is considered vital if it is to be of any practical use.

The set of graph grammars obeying the additional restrictions (which will be outlined below) is referred to as the class of *parsable* graph grammars. The Rekers-Schürr parsing approach is applicable only to grammars that belong to this class.

Connected right-hand sides

One restriction is that R must be a connected graph. This is justified by implementation issues, specifically the fact that the parsing algorithm uses *linear search plans* to recognize matches of productions' right-hand sides in the input graph and the intermediate graphs it builds during the parsing process. These search plans are unable to "hop" from one vertex to another without a connecting edge to traverse. It is this restriction that so complicated the ER grammar in section 2.2.

Single initial graph

As follows from the definition of a graph grammar, all productions must have non-empty left-hand sides. In other words, a grammar can have only one initial graph and nothing else "comes out of nowhere". This is a mere simplification and does not affect the generated language; for every conceivable graph grammar there exists an equivalent one obeying this rule such that the difference is at most a few added productions to generate the alternative start symbols from a single new "super-start symbol".

Layering Condition

A more fundamental restriction is the *layering condition*; this can be seen as an extension of the traditional partition into terminal and nonterminal labels. The following definitions are needed before we can express the layering condition.

Layer assignment A *layer assignment* of a graph grammar gg is a partition $\{L_0, L_1, \ldots, L_n\}$ of the set of labels L_{gg} of gg. The L_i ($0 \leq i \leq n$) are called the *layers* of L_{gg}.

Layer number The *layer number* of a label x is the number j for which $x \in L_j$ (x is in layer L_j). There is only one such number for any label x because the layer assignment is a partition of L_{gg}.

Graph restriction The *restriction* $G \cap L_i$ of a graph G to a layer L_i is the set of graph elements in G whose labels are in L_i.

Layer vector Given a graph G labeled according to label set L_{gg}, its *layer vector* $|G|$ is the vector

$$(|G \cap L_0|, \ldots, |G \cap L_n|)$$

Note that the $G \cap L_i$ are sets of elements, so the $|G \cap L_i|$ are the respective sizes of these sets ($|G \cap L_i| \in \mathbb{N}$).

Lexical comparison A vector $X \equiv (x_0, \ldots, x_n) \in \mathbb{N}^n$ is said to be *lexically smaller than* a vector $Y \equiv (y_0, \ldots, y_n)$ if and only if there is a i such that $0 \leq i \leq n$ and:

$$x_i < y_i \;\wedge\; \forall j < i : \; x_j = y_j$$

This is similar to sorting words of the same length alphabetically.

Now we come to the definition of the **layering condition**: A graph grammar gg obeys the layering condition with regard to layer assignment $L_0, \ldots, L_n$ if and only if for every production (L, R) in the grammar, $|L|$ *is lexically smaller than* $|R|$.

As a result of this condition, applying a sequence of production instances pi from gg generates a sequence of sentential forms (beginning with the initial S graph and ending in a sentence G) whose layer vectors are monotonously increasing in terms of lexical comparison. The reverse also holds: The layering condition ensures that the parsing process for input sentence G, ie. the reversed application of sequence pi to G, will lexically reduce the layer vector with each step. Hence it can easily be proven that the length of sequence pi must be finite, making the membership problem for gg decidable.

Parsable Graph Grammars

A graph grammar gg can now be defined to be *parsable* under the following conditions:

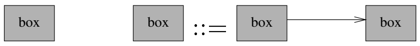
- For every production (L, R) in gg, R is a connected graph, and
- for every production (L, R) in gg, L is non-empty, and
- there is a layer assignment $L \equiv L_0, \ldots, L_n$ such that gg obeys the layering condition with regard to L.

Note that it follows from these conditions that gg is λ-free, ie. the right-hand side of a production in gg is never empty and is never a subset of the left-hand side.

2.5 Possibilities and Limitations

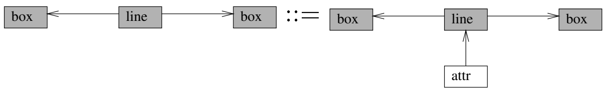
Like fire, context sensitivity can be both man's best friend and his most dangerous enemy. The following short examples will show a few, perhaps unexpected, potential pitfalls in writing context-sensitive graph grammars.

Consider a simple production

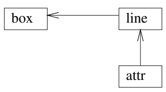


which simply connects two boxes by inserting an edge between them. In this case it is fairly obvious that the newly created edge may be a loop (ie. whenever the two context elements are mapped onto the same `box`).

Perhaps less obvious is the fact that this form of identification may also occur between two edges in the production's interface graph. Let's say we're parsing an expression in a language with the following production with a "symmetric" interface graph:



While parsing an expression in this language, the right-hand side of this production could be matched by a subgraph in the active sentential form such as:



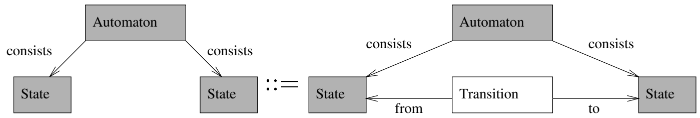


Figure 5: Production from a grammar for Finite-State Automata

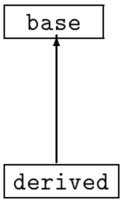
Here both arrows of the production right-hand side will be matched by the single arrow in the graph, and both boxes in the production are matched by the single box in the graph. Thus the new element may be attached to a "dead-end" line touching only a single box. While the intention of the above production may have been to match a line connecting exactly two boxes, this is not in fact what it expresses.

Hazards aside, edge identification can be as useful as vertex identification. Consider the production shown in figure 5, taken from a Finite State Automata (FSA) grammar defined in [16]. Only by identifying the two `consists` edges (as well as the `State` vertices) can loops be introduced into the Automaton. Without edge identification this would require an extra special-case production. The formalism has no way of selectively disabling identification so changing behaviour in such cases (if desired) may be as difficult as redesigning the whole grammar or adding custom C++ code to check for the unwanted identification.

2.6 Inheritance

To facilitate grammar definition, *inheritance* can be allowed between labels as a form of syntactic sugar. In many practical cases this mechanism may be used to reduce the number of near-identical productions in a grammar by exploiting commonality between them.

Semantics in graph productions are relatively simple; let's assume the following simple inheritance tree for vertex labels in a grammar gg that we're going to parse.



Now if a production $P \equiv (L, R)$ in gg contains an element $x \in L \cap R$ with label `base`, then a production instance pi of P may not only map x to a graph element of label `base` as usual, but may also identify it with a graph element of label `derived`. Inheritance does not apply to a production's exclusive left-hand and right-hand sides however, so in the case where $x \notin L \cap R$ an exact type match is required meaning that x must be mapped onto a graph element of label `base` and nothing else.

Multiple inheritance is allowed, but inheritance is restricted with regard to label layers. The restriction here is that if a label `derived` is a child of `base`, the latter may not have a lower layer number than the former. As always, the inheritance relationship may not be cyclic.

3 The Algorithm

The outline of the Rekers-Schürr graph-grammar parsing algorithm given in this section is of a descriptive nature and by no means enough to understand how it works in full detail. For a full description, see [17].

3.1 Overview

The algorithm falls into two phases of analysis: The *bottom-up phase* and the *top-down phase*. During the bottom-up phase all possible production instances are discovered and applied in reverse order (ie. starting with the input sentence and eventually arriving at potential start symbols). This phase has only limited intelligence and hence generates a set of potential production instances that contains many spurious entries. The top-down phase then scans this set for a valid derivation DAG (looking for a start symbol that generates the input sentence) and reports the first one it finds.

3.2 Bottom-Up Phase

The bottom-up phase operates on an intermediate graph of terminal and nonterminal graph elements, which starts out as a copy of the input graph and is extended with all possible elements that can be "derived" by reverse application of the generated production instances.

As briefly mentioned above, this phase of the algorithm has no other input than the grammar and an input sentence. It discovers production instances by searching for subgraphs that match right-hand sides of the grammar's productions; if one is found then a copy of the production's exclusive left-hand side is inserted into the graph relative to the part of the subgraph that corresponds to the production's interface graph.

Searching is performed by so-called *dotted rules* generated from the right-hand sides of the grammar's productions; these are essentially linear lists of *matching directives* such as "find an edge of label x starting at vertex v", plus a number i indicating that the first i directives have been executed—an imaginary "dot" is always placed between directives i and $i+1$ to mark its state, hence the name. Every vertex in the graph can be the starting point of one or more new rules called its *initial dotted rules*.

Conceptually, the dot proceeds one step through the list (ie. i is incremented) each time a new matching directive has been satisfied. This is called *propagation* of the dotted rule[3]. Once i reaches the total number of directives in the rule, the entire right-hand side of the production has been mapped onto the intermediate graph (the production instance has been *discovered* at this point) and the production's exclusive-left hand side may be added in accordance to the matching information contained in the finished dotted rule—relative to the mapping of its interface graph, called *common*. For any production instance pi, $common(pi)$ is the set of graph elements that are referenced by pi but are neither generated nor deleted by it.

To put things in a sequential perspective, whenever a new production is discovered the algorithm determines the initial dotted rules for each newly added vertex, attaches them to a waiting queue related to that vertex, and pushes them onto the *Active* stack. This stack is a kind of global back-order list of dotted rules that could still be propagated further in the current graph; any dotted rule not in this list is said to be *suspended* and is waiting for new graph elements to appear before it can propagate again.

The outer loop of the bottom-up phase grabs a dotted rule off the *Active* stack and attempts to propagate it through any candidate edges in the graph, after which the dotted rule is suspended. If

[3] As the name suggests the propagated dotted rule is really a modified copy of the propagating one, which remains in place in case other propagations may also be performed from the same vertex in the graph.

the dotted rule is in an accepting state its corresponding production instance is discovered, possibly extending the graph and adding more dotted rules to the *Active* stack in the process.

An interesting thing to note about the bottom-up phase is the fact that it only adds graph elements, extending the intermediate graph with ever more symbols. The (reverse) application of any production instance here is just one possibility, and the graph elements in its exclusive right-hand side (or $Xrhs$) remain available for other possible derivations.

When no more unfinished rules exist, the bottom-up phase exits. Its output is PPI, the set of generated production instances.

3.2.1 Terms and Relationships

Several relationships between production instances are defined, some of which have a direct counterpart in the parser's implementation. Others are only needed for discussion and understanding of the algorithm; they are described here to provide some insight in the workings and behaviour of the algorithm, and the issues arising in its implementation.

consequence A production instance pi discovered during the bottom-up phase may be a *consequence* of another production instance pi_0 (written as $pi \in \mathbf{consequence}(pi_0)$), meaning that the application of pi_0 must be followed (at some later point in the generation sequence) by the application of pi. This is the case when pi deletes a graph element that is used by the right-hand side of pi_0, ie. the exclusive left-hand side of pi overlaps with the right-hand side of pi_0.

This also means that no production instance can be a consequence of itself. The formal definition is:

$$pi \in \mathbf{consequence}(pi_0) \iff (Xrhs(pi_0) \cup common(pi_0)) \cap Xlhs(pi) \neq \emptyset$$

The expression $Xlhs(pi)$ represents pi's exclusive left-hand side, ie. the set of all graph elements deleted by pi; cf. the expression $Xrhs(pi)$ which represents pi's exclusive right-hand side or the set of all graph elements generated by pi.

consequence* This is the transitive and reflexive closure of $\mathbf{consequence}$. Thus $\mathbf{consequence}^*(pi)$ consists of pi, $\mathbf{consequence}(pi)$, and, for every $pi' \in \mathbf{consequence}(pi)$, the elements of $\mathbf{consequence}^*(pi')$. This set, too, is fully known for any production instance at the time it is discovered.

Three kinds of consequences are found in the $\mathbf{consequence}$ set of any production instance pi: The trivial one (pi itself), direct ones ($\mathbf{consequence}(pi)$) and indirect ones ($\mathbf{consequence}^* \cdot \mathbf{consequence}(pi)$)).

above A production instance pi is said to be *above* another production instance pi' ("pi $\mathbf{above}$ pi'") in the case where pi must be executed before pi' in any derivation sequence in which both occur.

For any set of graph elements S, define $V(S)$ as the set of elements of S that are vertices and $E(S)$ as the set of elements of S that are edges; for any edge e define $s(e)$ to be the source vertex of e and $t(e)$ to be the target vertex of e. Then pi $\mathbf{above}$ pi' if and only if $pi \neq pi'$ and:

- $pi' \in \mathbf{consequence}(pi) \vee$
  (any derivation that executes pi must execute pi' as well, but at a later point in the derivation sequence)
- $Xrhs(pi) \cap common(pi') \neq \emptyset \vee$
  (execution of pi' requires a graph element that must first be generated by pi)

- $\exists e \in E(Xlhs(pi)), \exists v \in V(Xlhs(pi')) : s(e) = v \vee t(e) = v$
  (before any vertex v can be deleted by pi', each of its adjacent edges must first be deleted by some production instance pi)

There is a good reason to require that pi and pi' be distinct: Otherwise the final condition would cause meaningless self-dependencies depending on whether a production (and thus all its instances) generates a vertex and an adjacent edge together.

above$_{pi}$ This is a ternary relationship. Each of its member triplets pi_1 $\mathbf{above}_{pi}$ pi_2 implies that *if pi is part of a derivation*, then so are pi_1 and pi_2, where pi must be executed before pi_1 and pi_1 must come before pi_2.

In formal logic this can be written as

$$\begin{aligned} pi_1 \ \mathbf{above}_{pi} \ pi_2 \quad \Leftrightarrow \quad & \{pi_1, pi_2\} \subseteq \mathbf{consequence}^*(pi) \ \wedge \\ & pi_1 \ \mathbf{above} \ pi_2 \end{aligned}$$

It may be enlightening to keep the set-oriented notation of this definition in mind as well, which says that $\mathbf{above}_{pi}$ is the entire $\mathbf{above}$ relationship restricted to the consequences of pi:

$$\mathbf{above}_{pi} \equiv \mathbf{above}|\mathbf{consequence}^*(pi)$$

above$^+_{pi}$ The transitive closure of $\mathbf{above}_{pi}$ is $\mathbf{above}^+_{pi}$. Note that this is *not* the same as the transitive closure of the $\mathbf{above}$ relationship ($\mathbf{above}^+$) restricted to the consequences of pi: If we have three production instances pi_1, pi_2 and pi_3 with pi_1 $\mathbf{above}^+_{pi}$ pi_2 $\wedge$ pi_2 $\mathbf{above}^+_{pi}$ pi_3 where $\{pi_1, pi_3\} \subseteq \mathbf{consequence}^*(pi)$ and $pi_2 \notin \mathbf{consequence}^*(pi)$, then $\langle pi_1, pi_3 \rangle \in (\mathbf{above}^+|\mathbf{consequence}^*(pi))$ but $\neg(pi_1 \ \mathbf{above}^+_{pi} \ pi_3)$.

As with $\mathbf{above}_{pi}$, pi_1 $\mathbf{above}^+_{pi}$ pi_2 implies that if pi is part of a derivation, then so are pi_1 and pi_2, where pi must be executed before pi_1 and pi_1 must come before pi_2.

excludes Two production instances are said to *exclude* each other if the inclusion of the one in a derivation directly prevents the application of the other, indicating a future choice point for the algorithm. The bottom-up phase however does not make such choices; it delegates them to the top-down phase instead. This relationship is symmetric.

There are two ways in which production instances pi_1 and pi_2 may exclude one another (written as pi_1 $\mathbf{excludes}$ pi_2 or vice versa): Either they depend on each other so that there is no valid linearization of the derivation, or they both claim to generate the same graph element (ie. their exclusive right-hand sides overlap).

Thus, for any two *distinct* production instances pi_1 and pi_2,

$$pi_1 \ \mathbf{excludes} \ pi_2 \iff (pi_1 \ \mathbf{above} \ pi_2 \wedge pi_2 \ \mathbf{above} \ pi_1) \vee Xrhs(pi_1) \cap Xrhs(pi_2) \neq \emptyset$$

An exception is made for the case $pi_1 = pi_2$: The $\mathbf{excludes}$ relationship is irreflexive by definition, so $\neg(pi \ \mathbf{excludes} \ pi)$ for any production instance pi. Otherwise the second condition would take the intersection of $Xrhs(pi)$ with itself, which is always nonempty by definition of the grammar formalism. That would cause all production instances to exclude themselves and we can't have that.

excludes* Appearances can be deceptive! This relationship isn't a real closure of the $\mathbf{excludes}$ relationship. It could be more aptly described as the "implied-$\mathbf{excludes}$" relationship.

For production instances pi_1 and pi_2, which need not be distinct, pi_1 $\mathbf{excludes}^*$ pi_2 means that any derivation including the one cannot include the other; it is important here to note the fact that some production instances may exclude themselves. The formal definition is

$$pi_1 \ \mathbf{excludes}^* \ pi_2 \Leftrightarrow \quad \exists \quad pi'_1 \in \mathbf{consequence}^*(pi_1), pi'_2 \in \mathbf{consequence}^*(pi_2): \; pi'_1 \ \mathbf{excludes} \ pi'_2$$

The formula reads as follows: pi_1 $\mathbf{excludes}^*$ pi_2 whenever the one has a consequence (whether trivial, direct or indirect) that prohibits the inclusion into the derivation of some consequence (trivial, direct or indirect) of the other.

inconsistent As hinted before, a production instance pi may "$\mathbf{excludes}^*$" itself which means it can never be part of any valid derivation. This does not represent a real choice point and the bottom-up phase is smart enough to deal with these cases. Another possibility is that a cyclic $\mathbf{above}^+_{pi}$ relationship may come into being with the discovery of pi. In both cases, pi is said to be *inconsistent*:

$$\mathbf{inconsistent}(pi) \Longleftrightarrow pi \ \mathbf{excludes}^* \ pi \ \vee \ \exists pi', pi'' : pi' \ \mathbf{above}^+_{pi} \ pi'' \wedge pi'' \ \mathbf{above}^+_{pi} \ pi'$$

The bottom-up phase will refuse to discover a production instance if it is $\mathbf{inconsistent}$; the proof of termination for the bottom-up phase necessitates this behaviour as it does not cover the presence of any $\mathbf{inconsistent}$ production instances. Thus detecting them in the top-down phase instead of in the bottom-up phase could be a potential eternity too late!

3.2.2 Properties of the Bottom-Up Phase

Definition 1 *A production instance pi is a* generating production instance *of a graph element e if and only if $e \in Xrhs(pi)$.*
Notation: $pi \in Xrhs^{-1}(e)$

Implementation Note

Each graph element is adorned with a set of its known generating production instances. This set may be extended at any time during the bottom-up phase.

Definition 2 *A production instance pi is a* deleting production instance *of a graph element e if and only if $e \in Xlhs(pi)$.*
Notation: $pi \in Xlhs^{-1}(e)$

Property 1 *Every nonterminal graph element e_n has a single deleting production instance; any terminal graph element e_t has at most one. To be precise, only those that were not part of the input graph have one.*

Proof The fact that $e_t \notin Xlhs(pi)$ for any production instance pi follows from the definition on page 8.

By the reverse nature of the bottom-up phase, a graph element is introduced into the graph at the time its *deleting* production instance pi_d is discovered because $Xlhs(pi_d)$ is added at that time. Should an identical production instance pi'_d be found for some reason, $Xlhs(pi'_d)$ will also be added to the graph (rather than coinciding with $Xlhs(pi_d)$).

Input graph elements cannot have a deleting production instance because they are not introduced into the graph by reverse-application of any production instance.
$\triangle$

Implementation Note

When a production instance has been discovered by the bottom-up phase and the elements of its exclusive left-hand side are added to the graph, each of these receives a reference to that production instance which will remain constant for its entire lifetime.

Definition 3 *A production instance pi is a* using production instance *of a graph element e if and only if $e \in common(pi)$.*
Notation: $pi \in common^{-1}(e)$

Property 2 *After discovery of a production instance, all elements of its exclusive left-hand side (or $Xlhs$) are added to the graph before any new production instance can be discovered.*

This property is ensured by first adding the entire exclusive left-hand side to the graph before attaching the initial dotted rules to its vertices.

Result 3 *Time of discovery is a total order on all production instances generated by the bottom-up phase of the algorithm.*

Implementation Note

Production instances are assigned unique consecutive identifying numbers according to their time of discovery. Sets of production instances be represented as bit arrays, so that the membership function is implemented as indexing the array.

Theorem 4 *For any production instance pi, the size of $\mathbf{consequence}(pi)$ is equal to or less than the number of elements in the right-hand side of pi.*

Proof The set $\mathbf{consequence}(pi)$ consists exactly of the deleting production instances of all the graph elements in the right-hand side of pi.

According to property 1, each of the nonterminal elements has exactly one deleting production instance, and each of the terminal elements has no more than one. Therefore the number of deleting production instances cannot exceed the number of graph elements in pi's right-hand side.
$\triangle$

Implementation Note

The $\mathbf{consequence}$ set of a production instance pi is derived from the final state of its dotted rule at the time of discovery and consists of $Xlhs^{-1}(Xrhs(pi) \cup common(pi))$, the set of deleting production instances of all graph elements matched by the dotted rule (ie. the production instance's entire right-hand side).

Theorem 5 *For two production instances pi_1 and pi_2 in the same graph, if $pi_1 \in \mathbf{consequence}(pi_2)$ then pi_1 is discovered before pi_2.*

Proof This property stems from the fact that the bottom-up phase works "backwards". Any graph element must have been added to the graph by reverse application of its deleting production instance before it can be part of the right-hand side of any production instance discovered at a later point in time.

Therefore no newly discovered production instance can be a consequence of a production instance discovered at an earlier time.
$\triangle$

Result 6 *When a new production instance is discovered during the bottom-up phase of the algorithm, its* $\mathbf{consequence}$ *set is fully known at that point.*

Result 7 *When a production instance is discovered during the bottom-up phase, its* $\mathbf{consequence}^*$ *set is also fully known at that point.*

Implementation Note

> The $\mathbf{consequence}^*$ set of a production instance is not explicitly stored; on the few occasions where a set $\mathbf{consequence}^*(pi)$ needs to be traversed, recursion is used.

Now we can proceed to the more complicated $\mathbf{above}$ relationship and its derivatives. These relationships can contain cycles so that their temporal qualities are less favourable than those of comparable $\mathbf{consequence}$ relationships; that is to say we can prove less aggressive bounds for the time when the truth value of eg. pi_1 $\mathbf{above}$ pi_2 can be determined than we can for, say, $pi_1 \in \mathbf{consequence}(pi_2)$. This makes it harder to optimize their implementations.

Theorem 8 *For any two production instances pi_1 and pi_2 where pi_1 is discovered before pi_2,*

$$pi_1 \ \mathbf{above} \ pi_2 \iff Xrhs(pi_1) \cap common(pi_2) \neq \emptyset$$

Proof The definition of $\mathbf{above}$ allows pi_1 $\mathbf{above}$ pi_2 by three alternative causes:

- $pi_2 \in \mathbf{consequence}(pi_1)$

  Given that pi_1 has been discovered before pi_2, this can never be the case due to theorem 5.

- $Xrhs(pi_1) \cap common(pi_2) \neq \emptyset$

  This case is allowed for by the theorem.

- $\exists e \in E(Xlhs(pi_1)), \exists v \in V(Xlhs(pi_2)) : s(e) = v \vee t(e) = v$

  This can never be the case: $v \in common(pi_1)$ implies that v must have been matched by pi_1's dotted rule, which (together with property 2) means that pi_1 could not have been discovered before pi_2—contradicting the assumption that pi_1 was discovered first.

The only possibility clearly matches the one in the theorem. This special case (where a newly discovered production instance is $\mathbf{above}$ an older one) will be referred to as an *upstream* $\mathbf{above}$ relationship.
$\triangle$

Result 9 *When the bottom-up phase discovers a production instance pi, the only already discovered production instances that are* $\mathbf{above}$ *pi must be generating production instances of the elements in pi's "common" subgraph, eg. the contents of $Xrhs^{-1}(common(pi))$.*

Implementation Note

Each production instance pi maintains a set of production instances pi' for which it is known that pi $\mathbf{above}$ pi'. Upon discovery of pi it contains all previously discovered pi' such that pi $\mathbf{above}$ pi'.

This set can later be extended, but only when a new production instance pi_n is discovered such that $common(pi_n) \cap Xrhs(pi) \neq \emptyset$, in which case pi_n is added.

Theorem 10 *For two production instances pi_1 and pi_2, where pi_1 is discovered before pi_2, the truth values of pi_1 $\mathbf{above}$ pi_2 and pi_2 $\mathbf{above}$ pi_1 are known as soon as pi_2 has been discovered by the bottom-up phase (but this does not hold for the transitive closure of $\mathbf{above}$).*

Proof It follows from theorem 8 that the truth values of pi_1 $\mathbf{above}$ pi_2 and of pi_2 $\mathbf{above}$ pi_1 depend on just four, not six, distinct conditions:

- $pi_1\ \mathbf{above}\ pi_2 \Leftrightarrow Xrhs(pi_1) \cap common(pi_2) \neq \emptyset$
- $pi_2\ \mathbf{above}\ pi_1 \Leftrightarrow$
  - $pi_1 \in \mathbf{consequence}(pi_2) \vee$
  - $Xrhs(pi_2) \cap common(pi_1) \neq \emptyset \vee$
  - $\exists e \in E(Xlhs(pi_2)), \exists v \in V(Xlhs(pi_1)) : s(e) = v \vee t(e) = v$

Hence the contents of only the following sets are involved in determining whether pi_1 $\mathbf{above}$ pi_2 or pi_2 $\mathbf{above}$ pi_1, respectively:

- $Xlhs(pi_1)$ has already been added to the graph in accordance to property 2
- $common(pi_1)$ must have been fully matched by pi_1's dotted rule in order for pi_1 to have been discovered
- $Xrhs(pi_1)$ contains exactly the rest of the graph elements matched by the dotted rule for pi_1
- $Xlhs(pi_2)$ consists of graph elements to be added based exclusively on information already contained in the finished dotted rule for pi_2. Hence the contents of this set, though not added to the graph yet, are fully known[4].
- $common(pi_2)$ must have been fully matched by pi_2's dotted rule in order for pi_2 to have been discovered
- $Xrhs(pi_2)$ contains exactly the rest of the graph elements matched by the dotted rule for pi_2
- $\mathbf{consequence}(pi_2)$ is known and complete as shown by result 6

From this it follows that all determining factors for the truth values of both pi_1 $\mathbf{above}$ pi_2 and pi_2 $\mathbf{above}$ pi_1 have been accounted for.
$\triangle$

For any two discovered production instances pi_1 and pi_2, therefore, the implementation may query the truth value of pi_1 $\mathbf{above}$ pi_2 at any time.

Implementation Note

[4] Adding $Xlhs(pi_2)$ along with its initial dotted rules may result in the immediate discovery of new production instances. While the implementation avoids this by adding the dotted rules a little later, we avoid it here by keeping to pi_2's exact moment of discovery.

The $\mathbf{above}$ set of a production instance pi (the set of pi' such that pi $\mathbf{above}$ pi') consists of the sets $\mathbf{consequence}(pi)$, $common^{-1}(Xrhs(pi))$, and for all edges $e \in Xlhs(pi)$, $Xlhs^{-1}(s(e)) \cup Xlhs^{-1}(t(e))$.

Next we will find that as new production instances are discovered (none are ever discarded by the way, so that the part of PPI that is known grows monotonously in time) the $\mathbf{above}$ relationship within any fixed known subset of PPI remains unchanged.

Result 11 *Let PI_i be the set of discovered production instances up to a point in time t_i during the bottom-up phase, and let V be a subset of PI_i. Furthermore let $\mathbf{above}_V^i$ contain the known pairs in the relationship $\mathbf{above}$ at time t_i, restricted to $V \times V$.*

Then, for every point t_j during the bottom-up phase, with $i \leq j$ (so that $V \subseteq PI_i \subseteq PI_j$):

$$\mathbf{above}_V^j = \mathbf{above}_V^i$$

As promised, we now know that the $\mathbf{above}$ relationship *restricted to a set of already discovered production instances* is fully known and valid for the rest of the bottom-up phase.

Theorem 12 *Let $(\mathbf{above}_V^i)^+$ be the transitive closure of $\mathbf{above}_V^i$; then for any two points in time t_j and t_k during the bottom-up phase,*

$$V \subseteq PI_j \cap PI_k \Longrightarrow (\mathbf{above}_V^j)^+ = (\mathbf{above}_V^k)^+$$

This need not hold for $(\mathbf{above}^+)_V^j$ and $(\mathbf{above}^+)_V^k$.

Proof There are three cases to be considered w.r.t. the relationship between t_j and t_k.

1. $j = k$ (trivial case): $\mathbf{above}_V^j \equiv \mathbf{above}_V^k$
2. $j < k$ (basic case): $V \subseteq PI_j \subseteq PI_k$ and result 11 derives $\mathbf{above}_V^j = \mathbf{above}_V^k$ (by substituting j for i and k for j)
3. $j > k$ (mirror case): $V \subseteq PI_k \subseteq PI_j$ and result 11 derives $\mathbf{above}_V^k = \mathbf{above}_V^j$ (by substituting k for i and j for j)

In all three cases, $\mathbf{above}_V^j = \mathbf{above}_V^k$ and therefore $(\mathbf{above}_V^j)^+ = (\mathbf{above}_V^k)^+$, QED.
$\triangle$

Theorem 13 *Given three production instances pi, pi_1 and pi_2, the truth value of pi_1 $\mathbf{above}_{pi}$ pi_2 is known as soon as pi has been discovered.*

Proof As by definition pi_1 $\mathbf{above}_{pi}$ pi_2 cannot hold unless $\{pi_1, pi_2\} \subseteq \mathbf{consequence}^*(pi)$, we need only consider the case where this is true.

Result 7 now tells us that $\mathbf{consequence}(pi)$ is a fully known set of discovered production instances, so the restriction of $\mathbf{above}$ to $\mathbf{consequence}(pi) \times \mathbf{consequence}(pi)$ is, according to result 11, also fully known and all production instances partaking in it have been discovered prior to discovery of pi.

From the definition it now follows that $\mathbf{above}_{pi}$ is fully known directly after discovery of pi so that it can be determined whether it holds a tuple pi_1 $\mathbf{above}_{pi}$ pi_2.
$\triangle$

Result 14 *For any production instance pi, the contents of $\mathbf{above}_{pi}$ are fully known at the time of pi's discovery and remain valid for the rest of the bottom-up phase. Therefore $\mathbf{above}_{pi}^+$ is known as well at that time.*

The $\mathbf{excludes}$ relationship, based as it is on $\mathbf{above}$, must be at least as hard to compute as $\mathbf{above}$ is. As we shall see now however, it isn't much harder either.

Theorem 15 *For any two production instances pi_1 and pi_2, the truth value of pi_1 $\mathbf{excludes}$ pi_2 is known as soon as both pi_1 and pi_2 have been discovered.*

Proof The definition for the $\mathbf{excludes}$ relationship for two production instances pi_1 and pi_2 makes use of exactly the following information:

- pi_1 $\mathbf{above}$ pi_2 and pi_2 $\mathbf{above}$ pi_1.

  These are known from the time that both pi_1 and pi_2 have been discovered, as has been shown in theorem 10.

- $Xrhs(pi_1)$ and $Xrhs(pi_2)$.

  The full contents of these sets are available from the moment that the search plans for pi_1 and pi_2 have been discovered.

All of this information is present after discovery of pi_1 and pi_2, and therefore pi_1 $\mathbf{excludes}$ pi_2 can be computed at this time; QED.
△

Theorem 16 *After discovery of production instances pi_1 and pi_2, the truth value of pi_1 $\mathbf{excludes}^*$ pi_2 is also immediately known.*

Proof According to theorems 15 and 7 all information required to determine pi_1 $\mathbf{excludes}^*$ pi_2 becomes available upon discovery of pi_1 and pi_2.
△

Theorem 17 *For any newly discovered production instance pi,*

$$\begin{aligned}\mathbf{excludes}|pi \quad \subseteq \quad & \{pi_0|pi_0 \ \mathbf{above}\ pi\} \cup \\ & Xrhs^{-1}(Xrhs(pi))\end{aligned}$$

Proof It will suffice to show that, given a production instance pi' so that pi $\mathbf{excludes}$ pi', it is implied that pi' is a member of this set. Following the definition of $\mathbf{excludes}$, suppose that

- pi $\mathbf{above}$ $pi' \wedge pi'$ $\mathbf{above}$ pi

  This condition clearly implies that pi' $\mathbf{above}$ pi so that $pi' \in \{pi_0|pi_0 \ \mathbf{above}\ pi\}$. Membership of its union with any other set follows.

- $Xrhs(pi) \cap Xrhs(pi') \neq \emptyset$

  In this case there must exist a graph element e such that both $e \in Xrhs(pi)$ and $e \in Xrhs(pi')$, ie. $\exists e :\ pi' \in Xrhs^{-1}(e) \wedge e \in Xrhs(pi) \Longrightarrow pi' \in Xrhs^{-1}(Xrhs(pi))$.

All possibilities for pi $\mathbf{excludes}$ pi' are hereby exhausted and the theorem is proven.
△

Result 18 *If pi $\mathbf{excludes}$ pi' and pi was discovered after pi', the two must either have overlapping exclusive right-hand sides or there must be an upstream $\mathbf{above}$ relationship between them as described in theorem 8 (pi' $\mathbf{above}$ pi).*

Implementation Note

When computing the $\mathbf{above}$ clause in the $\mathbf{excludes}$ relationship, the parser implementation only considers "upstream" pairs pi_1 $\mathbf{above}$ pi_2 and checks whether pi_2 $\mathbf{above}$ pi_1 holds as well. The $Xrhs$ clause is available to the implementation as the set $Xrhs^{-1}(Xrhs(pi_2))$.

Unfortunately, keeping track of $\mathbf{consequence}^*$ is slightly more involved than $\mathbf{consequence}$ itself because of the relationship's reflexivity. Newly discovered production instances may not only $\mathbf{excludes}$ older ones; they will in that case also *be excluded* by them.

The following theorem provides us with a useful cumulative property of the $\mathbf{excludes}^*$ relationship:

Theorem 19 *Given two production instances pi_1 and pi_2 with $pi'_1 \in \mathbf{consequence}^*(pi_1)$ and $pi'_2 \in \mathbf{consequence}^*(pi_2)$,*

$$pi'_1 \ \mathbf{excludes}^* \ pi'_2 \Longrightarrow pi_1 \ \mathbf{excludes}^* \ pi_2$$

Proof If pi'_1 $\mathbf{excludes}^*$ pi'_2 holds, there must be two production instances $pi''_1 \in \mathbf{consequence}^*(pi'_1)$ and $pi''_2 \in \mathbf{consequence}^*(pi'_2)$ such that pi''_1 $\mathbf{excludes}$ pi''_2.

Given the fact that $\mathbf{consequence}^*(pi'_1) \subseteq \mathbf{consequence}^*(pi_1)$ and that $\mathbf{consequence}^*(pi'_2) \subseteq \mathbf{consequence}^*(pi_2)$, the existential quantifiers in the definition of pi_1 $\mathbf{excludes}^*$ pi_2 on page 18 are satisfied by pi''_1 and pi''_2 respectively.

△

Theorem 20 *When a new production instance pi_1 is discovered during the bottom-up phase then for any previously discovered production instance pi_0:*

$$\begin{aligned} pi_1 \ \mathbf{excludes}^* \ pi_0 \iff & \\ & \exists pi'_1 \in \mathbf{consequence}(pi_1) : \ pi'_1 \ \mathbf{excludes}^* \ pi_0 \ \vee \\ & \exists pi'_0 \in \mathbf{consequence}^*(pi_0) : \ pi_1 \ \mathbf{excludes} \ pi'_0 \end{aligned}$$

Note that according to theorem 16, the truth value of pi'_1 $\mathbf{excludes}^*$ pi_0 is already known at this time.

Proof Starting off with the easiest part, the right-to-left implication:

- Because $pi'_1 \in \mathbf{consequence}^*(pi_1)$ and $pi_0 \in \mathbf{consequence}^*(pi_0)$, pi_1 $\mathbf{excludes}^*$ pi_0 follows from theorem 19.
- pi_1 $\mathbf{excludes}$ pi'_0 implies that pi_1 $\mathbf{excludes}^*$ pi_0 because $pi_1 \in \mathbf{consequence}^* pi_1$ and $pi'_0 \in \mathbf{consequence}^* pi_0$.

The rest, ie. the left-to-right part of the implication is best proven by contradiction. If it were the case that pi_1 $\mathbf{excludes}^*$ pi_0 without at least one of the two clauses of the theorem's conjunction being true, there would have to exist two production instances $\hat{pi}_1 \in \mathbf{consequence}^*(pi_1)$ and $\hat{pi}_0 \in \mathbf{consequence}^*(pi_0)$ such that $\hat{pi}_1$ $\mathbf{excludes}$ $\hat{pi}_0$, and:

- To avoid the first clause, $\hat{pi}_1 = pi_1$ otherwise there would have to be a $pi'_1 \in \mathbf{consequence}(pi_1)$ for which pi'_1 $\mathbf{excludes}^*$ pi_0.
- To avoid the second clause, $\hat{pi}_1 \neq pi_1$ because the $pi'_0 \in \mathbf{consequence}^*(pi_0)$ on the other side of the expression is unavoidable.

These two properties are clearly contradictory, so that the theorem is proven.
$\triangle$

Result 21 *The* $\mathbf{excludes}^*$ *relationship restricted to a newly discovered production instance* pi *consists of two parts, related to the two clauses in the conjunction of theorem 20 above:*

1. *A* cumulative *part:*
$$\bigcup_{pi' \in \mathbf{consequence}^*(pi)} (\mathbf{excludes}^* | pi')$$

2. *An* additional *part:*
$$\bigcup_{pi \ \mathbf{excludes} \ pi' \wedge pi' \in \mathbf{consequence}^*(pi_0)} pi_0$$

Implementation Note

When a new production instance pi is discovered, it builds up an image of $\mathbf{excludes}^*|pi$ by first computing the "cumulative" part by merging the similar images owned by the production instances in its $\mathbf{consequence}$ set. It then finds the "additional" part by, for each newly discovered direct $\mathbf{excludes}$ relationship pi $\mathbf{excludes}$ pi', visiting all pi'_0 for which $pi' \in \mathbf{consequence}(pi'_0)$ and patching their images to include pi. Then all pi_0 for which $pi'_0 \in \mathbf{consequence}(pi_0)$ are visited recursively, and so on.

Once all *known* images of excluded production instances have been patched, any production instances excluding pi that may be discovered at some later time will find pi in their cumulated images.

Perhaps the relationship most in need of optimization is $\mathbf{inconsistent}$. Its definition involves all of the $\mathbf{consequence}$, $\mathbf{above}$, and $\mathbf{above}^+_{pi}$ relations so it should be quite apparent that computing the value of $\mathbf{inconsistent}(pi)$ for any production instance pi is a very complex operation with a relatively large potential memory footprint. Properties are sought whose use can reduce these computational costs.

Theorem 22 *An equivalent definition for* $\mathbf{inconsistent}$ *is*

$$\begin{aligned}
\mathbf{inconsistent}(pi) \iff & \exists pi', pi'',\ pi' \neq pi'' : \\
& \{pi', pi''\} \subseteq \mathbf{consequence}^*(pi) \wedge \\
& (Xrhs(pi') \cap Xrhs(pi'') \neq \emptyset \vee \\
& \ pi' \ \mathbf{above}^+_{pi} \ pi'' \wedge pi'' \ \mathbf{above}^+_{pi} \ pi')
\end{aligned}$$

Proof By definition,

$$\begin{aligned}
\mathbf{inconsistent}(pi) \ \Leftrightarrow \ & pi \ \mathbf{excludes}^* \ pi \vee \\
& \exists pi', pi'' : pi' \ \mathbf{above}^+_{pi} \ pi'' \wedge pi'' \ \mathbf{above}^+_{pi} \ pi'
\end{aligned}$$

This can be expanded by the definitions of $\mathbf{excludes}^*$ and $\mathbf{above}^+_{pi}$:

$$\begin{aligned}
\mathbf{inconsistent}(pi) \ \Leftrightarrow \ & \exists pi', pi'',\ pi' \neq pi'' : \\
& \quad \{pi', pi''\} \subseteq \mathbf{consequence}^*(pi) \wedge pi' \ \mathbf{excludes} \ pi'' \vee \\
& \exists pi', pi'' : pi' \ \mathbf{above}^+_{pi} \ pi'' \wedge pi' \ \mathbf{above}^+_{pi} \ pi''
\end{aligned}$$

Seeing that neither of the relationships pi' $\mathbf{above}^+_{pi}$ pi'' and pi' $\mathbf{above}^+_{pi}$ pi'' can hold unless $\{pi', pi''\} \subseteq \mathbf{consequence}^*(pi)$, the $\mathbf{consequence}^*$ condition can be expanded to include both terms of this equation.

Since pi' $\mathbf{above}^+_{pi}$ pi'' will never hold for $pi' = pi''$, the $pi' \neq pi''$ condition can similarly be expanded. After substituting the definition of $\mathbf{excludes}$ the formula reads:

$$\begin{aligned}\mathbf{inconsistent}(pi) \quad \Leftrightarrow \quad & \exists pi', pi'',\ pi' \neq pi'' : \\ & \{pi', pi''\} \subseteq \mathbf{consequence}^*(pi) \wedge \\ & (pi'\ \mathbf{above}\ pi'' \wedge pi''\ \mathbf{above}\ pi' \vee \\ & Xrhs(pi') \cap Xrhs(pi'') \neq \emptyset \vee \\ & pi'\ \mathbf{above}^+_{pi}\ pi'' \wedge pi''\ \mathbf{above}^+_{pi}\ pi')\end{aligned}$$

Obviously, if $\{pi', pi''\} \subseteq \mathbf{consequence}^*(pi)$ then pi' $\mathbf{above}$ pi'' also implies that pi' $\mathbf{above}^+_{pi}$ pi'' (and similar for pi'' $\mathbf{above}$ pi' etc.) so that the $\mathbf{above}$ clauses above actually imply the $\mathbf{above}^+_{pi}$ conditions.

Omitting the unnecessary clause we are left with the formula given in the theorem.

$\triangle$

Put less formally, a production instance is inconsistent (and can therefore never be applied) if either its application would cause a graph element to be generated by more than one production instance, or two of its consequences must mutually "precede" one another.

Unfortunately this formula is still extremely compute-intensive. The ternary $\mathbf{above}_{pi}$ relationship can grow quite large and its transitive closure needs to be computed exactly once for each and every production instance. Even worse, the closure is computed after its restriction to $\mathbf{consequence}^*(pi)$ so there is very little gain to be had by storing its image explicitly.

For the scope of this project it was decided to leave out the implementation of the $\mathbf{above}_{pi}$ part and compute the $\mathbf{excludes}^*$ clause only. Just how much of an impediment this is to the algorithm is unknown at this time, since none of the graph grammars developed for it so far appears capable of generating such a case of inconsistency.

3.3 Top-Down Phase

After the bottom-up phase, the set PPI contains all production instances necessary to derive the given input sentence from the grammar's initial graph, if such a derivation exists.

Unfortunately it may contain a lot more than that. As was mentioned earlier, the bottom-up phase isn't very intelligent. Some of the production instances it generates may delete graph elements that are never generated, and some may delete vertices that still have edges attached which would lead to a sentential form with dangling edges (which would not be a proper graph therefore!). And finally, there may be more than one valid derivation.

It is the duty of the top-down phase to dig into PPI and deliver the first valid derivation it finds, omitting all production instances that are not part of it. This is done by trial and error: Starting with one of the potential "initial productions" discovered during the bottom-up phase, a derivation is pursued until a choice between mutually exclusive production instances becomes inevitable; when it does (a *choice point* is encountered) an arbitrary choice is made by applying some production instance pi and derivation is attempted. If this derivation fails, the algorithm is restored to the state it was in at the choice point and the alternative choice is accepted, ie. pi is rejected and derivation continues from there.

When there are no more candidate production instances, the derivation attempt is finished and either the exact input graph has been generated from the grammar's initial graph (successful derivation) or some input element has not been generated (failed derivation) and the algorithm backtracks to find some other unfinished derivation attempt. Derivation may also abort prematurely because an applied production instance turns out to be excluded by the current derivation.

Strictly speaking the notion of time as seen in the bottom-up phase does not apply in the algorithm's top-down phase. Instead of a linear progression from one situation to another, we see a tree of possibilities being traversed depth-first until a valid leaf is encountered. Nevertheless the absence of any interaction between alternative attempts allows us to view the linear path from the root of this tree to the current situation as a timeline—the difference being that we can literally rewind it and do things differently next time. The $<$ and $\leq$ operators as applied to the states of the top-down phase are partial orders, but become total ones when restricted to a single (partial) derivation or attempted derivation.

At any "time" t_i then, the set PPI is partitioned into three subsets API_{t_i}, EPI_{t_i} and PPI_{t_i} where

- API_{t_i} is the set of *applied* production instances, ie. those that are included in the current attempted derivation at time t_i.
- EPI_{t_i} is the set of *excluded* production instances; these are production instances that can either never be applied (eg. because they delete graph elements that are never really generated) or are in an $\mathbf{excludes}^*$ relationship with some element of API_{t_i}.
- PPI_{t_i} is the set of remaining (*potential*) production instances; they may be either applied or excluded at some later time.

3.3.1 Properties of the Top-Down Phase

Theorem 23 *When a production instance pi is not applied in a derivation, any graph elements in its exclusive left-hand side $Xlhs(pi)$ must never be part of any of its sentential forms.*

Proof Derivation can only succeed if the input graph is exactly reproduced. Thus any graph element that was not in the original input must eventually be deleted, which means that its deleting production instance must be applied in the derivation (theorem 1 says that pi is its only deleting production instance). Conversely, elements that must not be deleted (input elements) cannot be members of $Xlhs(pi)$ because they have no deleting production instances.

The graph element can therefore never be part of a sentential form in a valid derivation.
$\triangle$

Result 24 *If a production instance pi is not applied in the current derivation, then neither should any of the elements of $Xrhs^{-1}(Xlhs(pi))$ and $common^{-1}(Xlhs(pi))$.*

Result 25 *When a graph element's deleting production instance is excluded from the current derivation, then so can its generating and using production instances.*

Implementation Note

Exclusion of any production instance pi from the current derivation attempt also triggers the immediate exclusion of $Xrhs^{-1}(Xlhs(pi))$ and $common^{-1}(Xlhs(pi))$.

Theorem 26 *If none of a graph element's generating production instances is applied in a derivation, then neither must any of its deleting or of its using production instances (that is, any of $Xlhs^{-1}(x) \cup common^{-1}(x)$).*

Proof If none of the element's generating production instances is applied in a derivation, it is never generated therefore it can not be part of any sentential form in that derivation. It follows that no production instance that refers to it can be part of that derivation.
$\triangle$

Result 27 *If all generating production instances of a graph element are excluded from the current attempted derivation, then so can its deleting and using production instances.*

Implementation Note

> Exclusion of an element's last generating production instance from the current attempted derivation also triggers the immediate exclusion of its deleting and using production instances. A counter called `potentialGPI` in every graph element keeps track of the number of non-excluded generating production instances; it is decreased whenever one of the element's generating production instances is excluded.

Property 28 *No two production instances pi and pi' such that pi $\mathbf{excludes}^*$ pi' will be applied in the same derivation.*

Proof This property is guaranteed by excluding all members of $\mathbf{excludes}^*|pi$ when pi is applied.
$\triangle$

Theorem 29 *At a point during the top-down phase s where a successful derivation is completed,*

$$pi \in API_s \Longrightarrow \mathbf{consequence}(pi) \subseteq API_s$$

Proof Let us assume that the theorem is false: That there is a derivation, completed at s, where some production instance pi is in API_s but some $pi' \in \mathbf{consequence}(pi)$ is not. Result 25 now says that none of the elements of $Xrhs(Xlhs^{-1}(pi'))$ and $common(Xlhs^{-1}(pi'))$ can be applied in this derivation. By definition however,

$$pi' \in \mathbf{consequence}(pi) \Longleftrightarrow pi \in (Xrhs(Xlhs^{-1}(pi')) \cup common(Xlhs^{-1}(pi')))$$

…from which it follows that $pi \notin API_s$. This cannot be reconciliated with the original assumption that $pi \in API_s$ and the theorem is proven by reductio ad absurdum.
$\triangle$

Result 30 *Applied recursively, and in conjunction with the transitivity of $\mathbf{consequence}^*$, this also means that at any point during the top-down phase s where a successful derivation is completed,*

$$pi \in API_s \Longleftrightarrow \mathbf{consequence}^*(pi) \subseteq API_s$$

Property 31 *As the attempted derivation progresses, production instances are either applied or excluded. Therefore API_t and EPI_t grow monotonously while PPI_t shrinks:*

$$\begin{aligned} t_i < t_j \quad \Longrightarrow \quad & API_{t_i} \subseteq API_{t_j} \wedge \\ & EPI_{t_i} \subseteq EPI_{t_j} \wedge \\ & PPI_{t_i} \supset PPI_{t_j} \wedge \end{aligned}$$

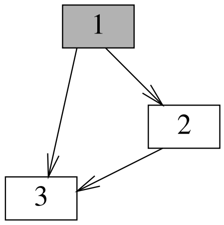


Figure 6: Above relationship between three production instances

At every step in the algorithm's top-down phase, there is a subset of production instances that may be applied next. This set is called $Candidates$; the original algorithm defines it at any point c in the top-down phase as

Definition 4

$$
\begin{aligned}
Candidates_c \equiv \quad \{ \quad & pi \in PPI_c | \\
& \exists pi' \in API_c : pi' \ \mathbf{above}\ pi \wedge \\
& \forall pi'' \in PPI : pi'' \ \mathbf{above}\ pi \rightarrow (pi'' \in API_c \vee pi'' \in EPI_c)\}
\end{aligned}
$$

The reason for this careful restriction is that the production instances must be applied by the top-down phase in a safe order, a "safe" order being one that never goes against the $\mathbf{above}$ relationship. The $\exists pi' \in API_c$ condition ensures that only those production instances are considered that are directly reachable from the set of applied production instances within the current $\mathbf{above}$ DAG whereas the $\forall pi'' \in PPI$ etc. condition makes sure that whenever there is an pi_1 $\mathbf{above}$ pi_2 relationship between any of these, the "highest" production instance pi_1 is considered first.

Figure 6, depicting three production instances and their $\mathbf{above}$ relationships, may help to explain how this works: If production instance 1 has already been applied and production instances 2 and 3 have been neither applied or excluded yet, then only production instance 2 may be a candidate. Before production instance 3 can be considered, a decision must first be made about production instance 2. It is clear that applying 3 first would make it impossible to apply 2 later without violating the $\mathbf{above}$ relationship.

Through this careful selection mechanism the Rekers-Schürr algorithm ensures that no graph element is used or deleted in the top-down phase without having been generated first. Whenever a production instance uses or deletes a graph element, all its generating production instances must have been either applied or excluded first; the $\mathbf{excludes}$ relationship is checked to make sure that no more than one of them is applied. Only after all production instances using the graph element have been either applied or excluded can its deleting production instance become a candidate for the top-down phase.

Less obvious to the eye is how the $\mathbf{excludes}^*$ relationship, by its cumulative nature w.r.t. the $\mathbf{consequence}$ relationship, also ensures that *at least* one of the graph element's generating production instances has been applied; however proving the original algorithm's correctness is beyond the scope of this report.

Theorem 32 *Definition 4 is equivalent to*

$$
Candidates_c \equiv \quad \{ \quad pi \in PPI_c |
$$

$$\begin{aligned}&\exists pi' \in API_c : pi' \ \mathbf{above}\ pi \wedge \\ &\neg\exists pi'' \in PPI_c : pi'' \ \mathbf{above}\ pi\}\end{aligned}$$

Proof By definition, PPI is the set of all production instances and PPI_c is the set of PPI members that are neither in API_c or in EPI_c; thus, by De Morgan's laws,

$$\begin{aligned} PPI_c &\equiv \{pi \in PPI | pi \notin API_c \wedge pi \notin EPI_c\} \\ &\equiv \{pi \in PPI | \neg(pi \in API_c \vee pi \in EPI_c)\}\end{aligned}$$

The valid substitution $p \rightarrow q \Leftrightarrow \neg(p \wedge \neg q)$ now allows us to rewrite the second condition in definition 4 as follows:

$$\begin{aligned} & \forall pi'' \in PPI : pi'' \ \mathbf{above}\ pi \rightarrow (pi'' \in API_c \vee pi'' \in EPI_c) \\ \Longleftrightarrow\ & \forall pi'' \in PPI : pi'' \ \mathbf{above}\ pi \rightarrow pi'' \notin PPI_c \\ \Longleftrightarrow\ & \forall pi'' \in PPI : \neg(pi'' \ \mathbf{above}\ pi \wedge pi'' \in PPI_c) \\ \Longleftrightarrow\ & \forall pi'' \in PPI : \neg pi'' \ \mathbf{above}\ pi \vee pi'' \notin PPI_c \end{aligned}$$

Since the second subcondition, $pi'' \notin PPI_c$, will always hold for exactly all members of $PPI_c \subseteq PPI$, the truth of the entire condition hinges on the first subcondition, $\neg pi'' \ \mathbf{above}\ pi$, for all $pi'' \in PPI_c$:

$$\begin{aligned} & \forall pi'' \in PPI : \neg pi'' \ \mathbf{above}\ pi \vee pi'' \notin PPI_c \\ \Longleftrightarrow\ & \forall pi'' \in PPI_c : \neg pi'' \ \mathbf{above}\ pi \\ \Longleftrightarrow\ & \neg\exists pi'' \in PPI_c : pi'' \ \mathbf{above}\ pi \end{aligned}$$

The latter expression is identical to the one used in the theorem, hence the equivalence is proven. $\triangle$

Result 33 *A production instance pi is part of $Candidates_c$ if and only if $pi \in PPI_c$ and:*

- $pi \in Candidates_{c-1} \vee$
- $|\ \{pi' \in PPI_c | pi' \ \mathbf{above}\ pi\}\ | = 0$ *and* $|\ \{pi' \in API_c | pi' \ \mathbf{above}\ pi\}\ | > 0.$

… where $c-1$ is the state directly preceding c in the same attempted derivation.

Implementation Note

A single $Candidates$ set, computed incrementally, represents the value of $Candidates_c$ at any time c during the top-down phase. Two counters in each production instance pi keep track of the number of pi', with $pi' \ \mathbf{above}\ pi$, such that $pi' \in PPI_c$ and $pi' \in API_c$ respectively. When the former reaches zero and the latter is not zero, and $pi \in PPI_c$, then pi is added to $Candidates$.

Theorem 34 *A production instance can only be removed from the set of* $Candidates$ *(ie. be part of* $Candidates_{c-1}$ *but not of* $Candidates_c$*) when it is either applied or excluded.*

Proof Of the conditions that make up the characteristic function of $Candidates_t$, only "$pi \in PPI_t$" can be true for $t \equiv c$ and false for $t \equiv c - 1$:

- $\exists pi' \in API_t : pi'$ **above** pi

  Property 31 states that once pi' is a member of API_{c-1}, it must also be a member of API_c. The truth value of pi' **above** pi will not change either; therefore this condition is true for any $c' > c$ (in the same derivation attempt) if it once holds for $t \equiv c$.

- $\neg\exists pi' \in PPI_c : pi'$ **above** pi

  Also according to property 31, once pi' has been removed from PPI_{c-1} it can never be a member of PPI_c and we're still stuck with the truth value of pi' **above** pi so this condition will continue to hold for any $c' > c$ in the same attempted derivation.

The only remaining condition is $pi \in PPI_t$, which can still evaluate to false for some value c for t.
△

Definition 5 *A production instance* pi adds n *elements if* $|Xrhs(pi)| - |Xlhs(pi)| = n$*. This is written as* $\mathbf{add}(pi) = n$*; note that* n *may be positive, negative or zero.*

A production instance pi covers n input elements *if* $|\{x \in Xrhs(pi) | Xlhs^{-1}(x) = \emptyset\}| = n$*, ie. if it generates* n *elements of the input graph. This set is the* coverset *of* pi*, or* $\mathbf{cover}(pi)$*, so that* $|\mathbf{cover}(pi)| = n$*.*

Property 35 *Any instance* pi *of a production* $P \equiv (L, R)$ *adds exactly* $|R| - |L|$ *elements.*

Proof There cannot be any dynamic variation in this number because the definition of the graph-grammar formalism used does not allow any two elements of $(L \cup R) \backslash (L \cap R)$ to be mapped to the same graph element. Identification of elements is limited to the interface graph $L \cap R$, which has no effect on the number of graph elements added by pi.
△

Property 36 *No production instance deletes ("uncovers") an input element.*

Proof This follows from property 1.
△

Theorem 37 *If the input graph contains* n *elements, a derivation attempt has exactly reproduced the input graph at time* s *if and only if:*

$$\sum_{pi \in API_s} |\mathbf{cover}(pi)| = \sum_{pi \in API_s} \mathbf{add}(pi) = n$$

Proof Derivation starts off with zero graph elements, the initial graph being generated out of void by a special production. From there all input graph elements must be generated.

Property 28 guarantees that no single graph element can be generated by more than one production instance in the same derivation; after all there would be an **excludes** relationship between them. Therefore if $\sum_{pi \in API_s} |\mathbf{cover}(pi)| = n$ then all input graph elements have been generated.

The only remaining question is the total number of elements; as many non-input elements must be deleted as are generated or there will be non-terminal "leftovers" in the generated graph. If the total number of added elements equals the number of input elements, there can be no leftovers and the generated graph must match the input graph exactly.
△

Implementation Note

Whereas the original algorithm compares the generated graph to the input graph each time a derivation attempt ends, the implementation merely keeps track of the values $n - \sum_{pi \in API_t} |\mathbf{cover}(pi)|$ and $n - \sum_{pi \in API_t} \mathbf{add}(pi)$ for every state t. The graphs match when both of these reach zero.

The top-down phase contains two functions responsible for identifying any production instances whose application can be shown to be impossible in the current derivation attempt, and excluding them. First of the two is `initial-cleanup()` which performs a first pass over the graph to find any graph elements that are never generated. Such artifacts are the result of the way the bottom-up phase works: When a dead end is reached in the "reverse derivation" performed there, graph elements in its exclusive left-hand side are added to the graph but no other production instance ever claims to have generated them.

The `initial-cleanup()` function deletes any $pi \in PPI$ for which there is a graph element $x \in (Xlhs(pi) \cup common(pi))$ such that $\neg\exists pi' \in PPI : x \in Xrhs(pi')$; this is repeated until no more such pi are found.

Theorem 38 *An alternative definition for what `initial-cleanup()` does is: For any graph element x such that $Xrhs^{-1}(x) = \emptyset$, exclude $Xlhs^{-1}(x)$ and $common^{-1}(x)$ and repeat until no more such elements are found.*

Proof

$$
\begin{aligned}
& pi : \exists x \in (Xlhs(pi) \cup common(pi)) : \neg\exists pi' \in PPI : x \in Xrhs(pi') \\
\iff\; & pi : \exists x \in (Xlhs(pi) \cup common(pi)) : Xrhs^{-1}(x) = \emptyset \\
\iff\; & pi : \exists x : (x \in Xlhs(pi) \vee x \in common(pi)) \wedge Xrhs^{-1}(x) = \emptyset \\
\iff\; & pi : \exists x : (pi \in Xlhs^{-1}(x) \vee pi \in common^{-1}(x)) \wedge Xrhs^{-1}(x) = \emptyset \\
\iff\; & pi : \exists x : pi \in (Xlhs^{-1}(x) \cup common^{-1}(x)) \wedge Xrhs^{-1}(x) = \emptyset
\end{aligned}
$$

$\triangle$

Implementation Note

Rather than checking all production instances for these conditions several times, the implementation makes a single pass over all graph elements and checks the number of generating production instances. If an element has none, its deleting and using production instances are deleted which in turn may lead to new deletions because other graph elements may have lost their deleting or generating production instances in the process.

The other cleanup function is called, unsurprisingly, `cleanup()`. This function is performance-critical as it is executed for every iteration of the main top-down loop and was thought to present a major stumbling block for the implementation. Quoting from [16]:

> 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.

Some effort has naturally been put into optimizing this function; it has eventually been made implicit to such an extent that it would be hard to point it out in the parser's source code. In the original algorithm it works like this: When executed at a point c and a sentential form G_c, the function excludes from the current derivation every production instance pi such that

$$\begin{aligned} pi \quad : \quad & \exists pi'' \in API_c : pi'' \textbf{ above } pi \wedge \\ & (x \in (Xlhs(pi) \cup common(pi)) \wedge \\ & \neg \exists pi' \in (PPI \backslash EPI_c) : x \in Xrhs(pi')) \vee \\ & (\exists v \in V(Xlhs(pi)), \exists e \in E(G_c) : \\ & (s(e) = v \vee t(e) = v) \wedge \\ & \neg \exists pi' \in PPI_c : e \in E(Xlhs(pi')))) \end{aligned}$$

Theorem 39 *Any production instance pi that is to be excluded by this function based on the condition that*

$$x \in (Xlhs(pi) \cup common(pi)) \wedge \neg \exists pi' \in (PPI \backslash EPI_c) : x \in Xrhs(pi')$$

... is already excluded by the implementation based on result 27 and the subsequent implementation note.

Proof The condition that $\neg \exists pi' \in (PPI \backslash EPI_c) : x \in Xrhs(pi')$ implies that all production instances in $Xrhs^{-1}(x)$ (all of x's generating production instances) have been excluded from the current derivation. When the last of these was excluded, all production instances in $Xlhs^{-1}(x) \cup common^{-1}(x)$ have also been excluded. As $x \in (Xlhs(pi) \cup common(pi))$, pi must have been among these and has therefore been excluded as well.
△

Implementation Note

> This part of the `cleanup()` test is hidden away in the function responsible for excluding production instances, which is in fact a set of mutually recursive functions.

Next we look at optimizing the second condition. As it turns out, the situation it describes does not require a full check.

Theorem 40 *If there are a production instance pi, edge $e \in G_c$ and vertex $v \in G_c$ such that*

$$\begin{aligned} v \in V(Xlhs(pi)) \wedge e \in E(G_c) \wedge (s(e) = v \vee t(e) = v) \wedge \\ \neg \exists pi' \in PPI_c : e \in E(Xlhs(pi')) \end{aligned}$$

... then the current derivation attempt is doomed to fail.

Proof There can be three possible causes for the absence of any $pi' \in PPI_c$ such that $e \in E(Xlhs(pi'))$:

1. There is a $pi' \in PPI$ such that $e \in E(Xlhs(pi'))$ but it has already been applied ($Xlhs^{-1}(e) \subseteq API_c$).

   In this case there is no way that $e \in G_c$ because e has already been deleted, so the condition cannot be satisfied. No falsification of the implication can be found in this case as the precondition is false.

2. There is a $pi' \in PPI$ such that $e \in E(Xlhs(pi'))$ but it has been excluded ($Xlhs^{-1}(e) \subseteq EPI_c$).

   Here theorem 23 helps out: No generating production instance for e can be applied. If $e \in G_c$ then the attempted derivation is *self-contradictory* because e clearly must have been generated by some production instance and for this reason the derivation attempt must fail.

3. There is no $pi' \in PPI$ such that $e \in E(Xlhs(pi'))$.

   According to property 1, e must be an edge in the input graph. But as the input graph can have no dangling edges this implies that v must be a vertex in the input graph which, again according to property 1, has no deleting production instances. This contradicts the subcondition that $v \in V(Xlhs(pi))$.

$\triangle$

Implementation Note

When a production instance pi_0 is excluded from the current attempted derivation, all edges e in $Xlhs(pi_0)$ have their generating production instances ($Xrhs^{-1}(E(Xlhs(pi_0)))$) excluded so e will never become part of the sentential form.

If some such e has already been generated, $API_c \cap EPI_c$ will become nonempty (the derivation attempt is self-contradictory) which is detected as a failure condition for the derivation attempt. The attempt is promptly aborted if this happens.

Result 41 *The function that is responsible for excluding production instances can take care of all conditions formerly checked by* `cleanup()` *with the same final results as the original function.*

Implementation Note

The implementation of the top-down phase does not explicitly keep track of the current sentential form, so no graph operations are used in the top-down phase at all. This is a major performance improvement.

As noted in sections 3.2.1 and 3.2.2, the implementation of the bottom-up phase does not check for cycles in the $\mathbf{above}^{+}_{pi}$ relationship. It is thus possible that the implementation delivers $\mathbf{inconsistent}$ production instances to the top-down phase! Luckily there is no risk of such a production instance ending up in a derivation, as the following theorem shows.

Theorem 42 *If there is a $pi \in PPI$ such that for some pi_1, pi_2 there is a chain $pi_1 \ \mathbf{above}^{+}_{pi} \ pi_2 \wedge pi_2 \ \mathbf{above}^{+}_{pi} \ pi_1$, then pi will not be applied in any derivation.*

Proof Note that according to result 30, any derivation that applies pi must also apply pi_1 and pi_2. Two possibilities are to be considered here for any state t:

1. All production instances in the $\mathbf{above}^{+}_{pi}$ chain are in PPI_t.

   In this case, none of the production instances in the chain can be a member of $Candidates_t$ because for each of them (say, pi_x) there is another production instance pi_y from the cycle, which in this case must be in PPI_t, such that $pi_y \ \mathbf{above} \ pi_x$. For lack of pi_1, pi_2 etc. a derivation attempt including pi can never complete.

2. The chain is "broken" because one or more of its production instances have been excluded.

   Now it is possible that one or more production instances in the chain become candidates. However because at least one of them has now been excluded, there is still no way to apply all of $\mathbf{consequence}^*(pi)$ and the attempt will not complete.

What could be called a third case is where the chain is broken because one of the production instances in it has been applied. To reach this state however, the production instance would have had to be included in $Candidates$ first. Thus for this case to occur, at least one $pi' \in \mathbf{consequence}^*(pi)$ would still have to be excluded.
$\triangle$

4 The Parser

4.1 GRL File Format

A brief explanation of the GRL graph language, which is used as the universal graph description language for all input and output by the parser framework, is in order before we can proceed with a more detailed description of the implementation. The GRL language is textual and human-readable, meaning that it uses only plain ASCII symbols and ignores unnecessary whitespace to allow the user to freely format it as any old text document. Even C++-style comments are allowed, both the traditional C "`/* ... */`" variety and the newer rest-of-line "`// ...`" kind, to add annotations in any place where whitespace is allowed.

4.1.1 Graphs, Nodes, and Edges

Extensions aside, three kinds of objects can exist within a GRL graph specification: The `graph` itself as well as, nested within it, `node` and `edge` objects (nodes and vertices are the same thing but for historical reasons this report will refer to them as "nodes" in a GRL context and as "vertices" in a theoretical context). Each object is declared by its type name followed by a colon (no whitespace between the type name and the colon[5]) after which comes a block of contents between curly braces, ie.

```
graph: {
  ...
  node: { ... }
  node: { ... }
  edge: { ... }
}
```

4.1.2 Attributes

Now for those content blocks, indicated above as "`...`". Each consists of *attribute* definitions describing properties of the object it is contained in. An attribute consists of an *attribute identifier* followed by a colon (separating whitespace is allowed in this case) and an *attribute value*. The set of attribute identifiers is defined by the GRL language and cannot be extended; as a result of this relative inflexibility there are several dialects such as GDL and its application-specific specializations. Luckily we are only interested in a standardized subset of these attributes, and the graph library developed and used for this project (more about the *Anonymous Graph Library* in section 5.7.4) will ignore any unknown or irrelevant attributes.

[5] All code delivered with this project will quietly swallow any whitespace inserted here, though!

Node	*type*	*description*	*mandatory*
title	string	Node name	yes
label	string	Node Label	no
color	enumerated	Fill colour	no
textcolor	enumerated	Text colour	no
bordercolor	enumerated	Border colour	no
info1	string	Pragmatic information	no
info2	string	Pragmatic information	no
info3	string	Pragmatic information	no

Figure 7: Main `node` attributes

Edge	*type*	*description*	*mandatory*
title	string	Edge name	no
label	string	Edge Label	no
color	enumerated	Fill colour	no
sourcename	string	Name of source node	yes
targetname	string	Name of target node	yes
linestyle	enumerated	continuous, dashed, dotted, invisible	no
thickness	integer	Line thickness	no

Figure 8: Main `edge` attributes

There can be as many attributes in any single object as desired, separated simply by whitespace; if an attribute is defined more than once within the same object, the last-defined value is remembered. Sensible default values are used for undefined attribute values (where applicable).

The different object types have different sets of attributes, some of which may be mandatory. Figures 7 and 8 show a compact overview of the relevant node and edge attributes (respectively). Some of these deserve some elaboration:

title Each node needs to have a `title` attribute simply so edges can be attached to it. For this reason the attribute is not mandatory for edges.

sourcename Title of edge's source node. This string must match the `title` of that node exactly (case-sensitive).

targetname Title of edge's target node (but otherwise similar to `sourcename`). Note that only directed graphs can be described.

info1 (plus `info2` and `info3`) are free for application-specific information. Strictly speaking these are node attributes but in this application they are sometimes used for edges as well. The vcg tool will choke over this, unfortunately; a solution is being pondered and/or negotiated as you read this.

4.1.3 Attribute Types

Note that each attribute value has a *type*, which is similar to the type of a variable in just about any conventional programming language. The atomic types (the only kind we're interested in) are:

string A sequence of free-form characters between double quotes, very similar to C strings. By using the backslash as an escape character, special characters such as newlines and double quotes (otherwise read as a string delimiter!) can be inserted just like in C. A string may not span more than a single line, ie. its opening and closing quotes must be on the same line of a file.

```
"This is a string containing newline\n and \"quotes\"... "
```

integer A simple C-style integer constant, such as `-32767` (decimal) or `0x5a60`. No fractional part may be added.

floating-point Any floating-point value, with or without a fractional part. Exponential notation is allowed, as in `65e3`. The parser framework currently does not use attributes of this type, but they are allowed.

enumerated Specialized types, similar to C++ enumerated constants; each value looks like an identifier, ie. a sequence consisting only of letters, digits, and underscores where the first character must not be a digit.

The set of legal values for an attribute of enumerated type is language-defined, eg.

- Colours: `red`, `blue`, `white` etc.
- Shapes: `box`, `rhomb`, `ellipse` etc.
- Drawing styles: `continuous`, `dashed`, `dotted` etc.

The code delivered in the scope of this project actually takes a *laissez-faire* approach to these definitions; extensions of these enumeration types (or even invention of entirely new ones) will not confuse the software.

4.1.4 Examples

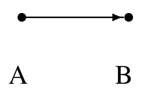
The trivial graph description is the empty one, containing no nodes and no edges:

```
graph:
{
}
```

Let's throw in a couple of nodes to get a graph looking like


```
graph:
{
  node: { title : "A" }
  node: { title : "B" }
}
```

Next we'll connect the nodes to get a graph looking thus:




```
graph:
{
  node: { title : "A" }
  node: { title : "B" }
  edge: { sourcename : "A"    targetname : "B" }
}
```

As order is unimportant in GRL, the edge may just as well be declared before one of its adjacent nodes, ie.

```
graph:
{
  node: { title : "A" }
  edge: { sourcename : "A"    targetname : "B" }
  node: { title : "B" }
}
```

In fact it may even precede both its source and its target nodes in the graph specification:

```
graph:
{
  edge: { sourcename : "A"    targetname : "B" }
  node: { title : "A" }
  node: { title : "B" }
}
```

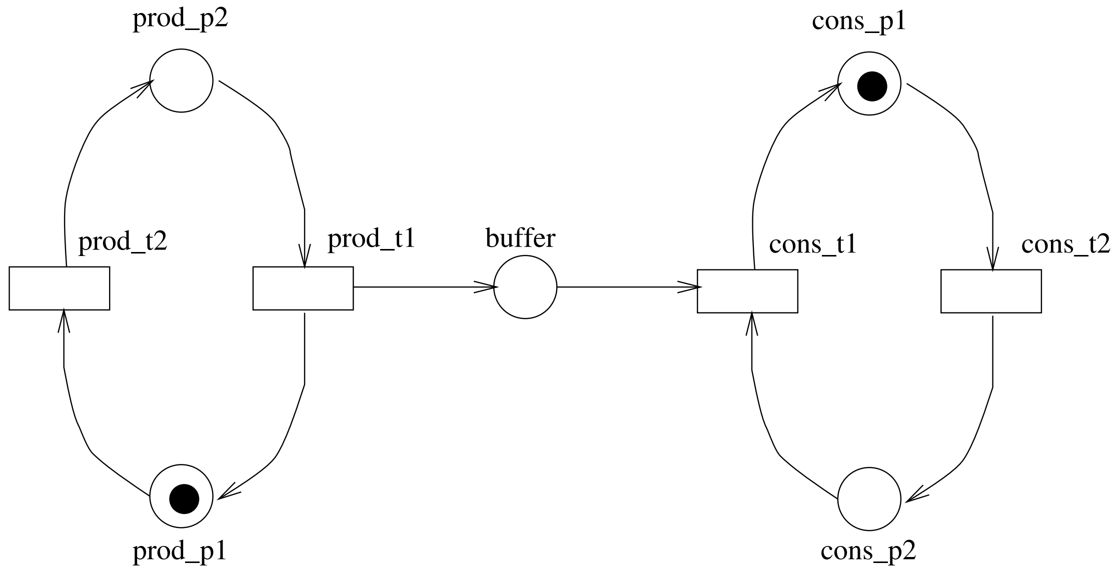


Figure 9: A producer/consumer Petri net

All these specifications are equivalent in GRL because order is not significant. For a more complicated example, here's the producer/consumer Petri net seen in figure 9, encoded in GRL. Positions and transitions are distinguished by their conventional shapes in this case: Positions are ellipsoid nodes, Transitions are rectangular (the default shape for a node). Tokens are indicated in the `info1` field.

```
graph:
{
  // Producer half:
  node: { title : "prod_p1" shape : ellipse     info1 : "token" }
  node: { title : "prod_p2" shape : ellipse }
  node: { title : "prod_t1" }
  node: { title : "prod_t2" }
  edge: { sourcename : "prod_t1"    targetname : "prod_p1" }
  edge: { sourcename : "prod_p1"    targetname : "prod_t2" }
  edge: { sourcename : "prod_t2"    targetname : "prod_p2" }
  edge: { sourcename : "prod_p2"    targetname : "prod_t1" }

  // Buffer:
  edge: { sourcename : "prod_t1"    targetname : "buffer" }
  node: { title : "buffer"  shape : ellipse }
  edge: { sourcename : "buffer"     targetname : "cons_t1" }

  // Consumer half:
  node: { title : "cons_p1" shape : ellipse     info1 : "token" }
  node: { title : "cons_p2" shape : ellipse }
  node: { title : "cons_t1" }
  node: { title : "cons_t2" }
  edge: { sourcename : "cons_t1"    targetname : "cons_p1" }
```

```
  edge: { sourcename : "cons_p1"    targetname : "cons_t2" }
  edge: { sourcename : "cons_t2"    targetname : "cons_p2" }
  edge: { sourcename : "cons_p2"    targetname : "cons_t1" }
}
```

There are also attributes for defining default values, graphical layout hints and so on. For full language definitions of GRL and GDL, see [9, 10, 12, 19, 22]. Eventually all hand-written GRL specifications used by the parser framework are to be replaced by output of more human-friendly tools; for now at least one graph editor (EDGE) is capable of producing GRL output but some textual post-processing is still required for this to work.

4.2 RSPGen

The program RSPGen generates the parser code for a particular graph grammar. It creates four source files which must be copied into the parser skeleton directory and compiled to produce the actual parser function: `setup_out.h`, `setup_out.cc`, `plan_out.h` and `plan_out.cc`. The former deal with grammar-specific initialization of label sets and such; the latter define the dotted-rule classes that are described in more detail in section 5.2.

The parser code, when compiled, can be called by user code through the `RSParse()` function, or a self-made equivalent body of code, which obtains an array of applied production instances in a valid order and adorns the input graph with any generated intermediate graph elements. The full dependency graph can also be reconstructed if need be.

The grammar that RSPGen is expected to generate a parser for needs to be provided in a number of separate files, each containing a single graph definition in the GRL format. These files are:

- The *vertex* label database, eg. as saved by the graph library's `LabelFamily` class.

  The nodes of this graph describe the label types to be used for the vertices of the graphs in the grammar; edges can be used to add inheritance between labels.

- The *edge* label database, eg. as saved by the graph library's `LabelFamily` class.

  This file is completely analogous to the vertex label database, but the nodes represent labels to be used for the edges of the graphs in the grammar.

- A number of productions. A production $P \equiv (L, R)$ is represented as a single graph where the elements of $L \backslash R$ are "coloured" as being deleted by the production and those of $R \backslash L$ are coloured as being generated by the production. The remaining elements form the production's interface graph $L \cap R$.

- Exactly one of these productions must be the *initial production*: This is the grammar's initial graph S presented in the form of a special production $I \equiv (\emptyset, S)$, ie. a production where all elements are coloured as being generated "out of thin air".

There are thus two kinds of input files for RSPGen: The *label family* and the *production graph*. In both cases the information contained in a file is encoded as a single graph, which means that they have different file formats but both are expressed as graphs in the GRL language.

4.2.1 GRL Input: Label Family

A *label family* encodes a set of labels as vertices with layer numbers as attributes, and their inheritance relationships as edges between them. Multiple inheritance is allowed, but naturally the inheritance graph

Node	*type*	*description*	*mandatory*
title	string	Label name	yes
layer	integer	Layer number	no
info1	string	`"default"`	no
info2	string	`"terminal"`	no

Figure 10: Label attributes

Edge	*type*	*description*	*mandatory*
sourcename	string	Name of *child* label	yes
targetname	string	Name of *parent* label	yes

Figure 11: Inheritance edge attributes

must be acyclic. Layer numbers are not usually needed for simple grammars, and future implementations may fill them in automatically. They default to zero.

By convention the edges representing inheritance relationships start at the derived label and point to the base label. This may seem “backwards” to some, but existing notations favour such an arrangement. The simple example label family on page 14 follows this rule and hopefully looks natural enough.

Figure 10 shows the GRL attributes that are used for the labels in the family. There may be at most one `default` label. If a vertex in a graph production or in the parser’s input sentence has no label, the default label from the vertex label family is used. Similarly the default edge label is assigned by default to all unlabeled edges.

There are no special attributes for the edges in the label family; let figure 11 remind you though that the head of an edge always points to the parent label of an inheritance relationship. The label family depicted on page 14 would look something like this in GRL:

```
graph: {
  node: { title : "base" }
  node: { title : "derived" }
  edge: { sourcename : "derived"  targetname : "base" }
}
```

We might also put them in different layers by defining the `layer` attribute:

```
graph: {
  node: { title : "base"    layer : 1 }
  node: { title : "derived" layer : 0 }
  edge: { sourcename : "derived"  targetname : "base" }
}
```

Note that the layer for `derived` should never be higher than that for `base`. Now suppose that `derived` is a terminal symbol, ie. one that may occur in the parser’s input:

```
graph: {
  node: { title : "base"    layer : 1 }
```

```
  node: { title : "derived" layer : 0  info2 : "terminal" }
  edge: { sourcename : "derived"  targetname : "base" }
}
```

As an extended example, here's a vertex label family for the Process Flow Diagram (*PFD*) grammar presented in [15, 17]. The full grammar is not reproduced here; suffice it to say that a PFD is similar to the familiar "flow charts" of yore with concurrency (`fork`/`join`) and communication (`send`/`receive`) thrown in. Note that a few labels are qualified with `"wildcard"` which is not recognized by the current implementation as a useful value for the `info1` attribute, so it is ignored and the label is assumed to be non-default. Wildcards were coined in [15] as a "poor man's inheritance" mostly because they took less explanation but provided enough extra syntactic convenience for the moment. The feature is not (or not yet) supported by the implementation but inheritance is. To achieve the same effect in implementing the PFD grammar, duplication was used combined with inheritance where possible. For the purpose of this report it is sufficient to view these "wildcard" labels as normal label types with several derived labels.

```
graph: {
  node: { title : "PFD"     layer : 2 }
  node: { title : "Stat"    layer : 1  }

  node: { title : "begin"   layer : 0  info2 : "terminal" }
  node: { title : "end"     layer : 0  info2 : "terminal" }
  node: { title : "fork"    layer : 0  info2 : "terminal" }
  node: { title : "join"    layer : 0  info2 : "terminal" }
  node: { title : "assign"  layer : 0  info2 : "terminal" }
  node: { title : "send"    layer : 0  info2 : "terminal" }
  node: { title : "receive" layer : 0  info2 : "terminal" }
  node: { title : "if"      layer : 0  info2 : "terminal" }

  node: { title : "B?"      layer : 2  info1 : "wildcard" }
  edge: { sourcename :    "begin"   targetname :    "B?" }
  edge: { sourcename :    "fork"    targetname :    "B?" }
  edge: { sourcename :    "if"      targetname :    "B?" }

  node: { title : "C?"      layer : 2  info1 : "wildcard" }
  edge: { sourcename :    "begin"   targetname :    "C?" }
  edge: { sourcename :    "fork"    targetname :    "C?" }
  edge: { sourcename :    "if"      targetname :    "C?" }

  node: { title : "S?"      layer : 2  info1 : "wildcard" }
  edge: { sourcename :    "end"     targetname :    "S?" }
  edge: { sourcename :    "assign"  targetname :    "S?" }
  edge: { sourcename :    "fork"    targetname :    "S?" }
  edge: { sourcename :    "join"    targetname :    "S?" }
  edge: { sourcename :    "send"    targetname :    "S?" }
  edge: { sourcename :    "receive" targetname :    "S?" }
  edge: { sourcename :    "if"      targetname :    "S?" }

  node: { title : "T?"      layer : 2  info1 : "wildcard" }
  edge: { sourcename :    "end"     targetname :    "T?" }
```

Node	*type*	*description*	*mandatory*
title	string	Node name	yes
label	string	Node label	no
info1	string	`"Xlhs"`, `"common"` or `"Xrhs"`	no
info3	string	Symmetry pool	no

Figure 12: Production vertex attributes

Edge	*type*	*description*	*mandatory*
label	string	Edge label	no
info1	string	`"Xlhs"`, `"common"` or `"Xrhs"`	no
info3	string	Symmetry pool	no
sourcename	string	Name of source node	yes
targetname	string	Name of target node	yes

Figure 13: Production edge attributes

```
  edge: { sourcename :    "assign"  targetname :    "T?" }
  edge: { sourcename :    "fork"    targetname :    "T?" }
  edge: { sourcename :    "join"    targetname :    "T?" }
  edge: { sourcename :    "send"    targetname :    "T?" }
  edge: { sourcename :    "receive" targetname :    "T?" }
  edge: { sourcename :    "if"      targetname :    "T?" }
}
```

For obvious reasons, unspecified attributes default to non-default, nonterminal and layer 0 respectively.

4.2.2 GRL Input: Production

Productions can be a tad awkward to write by hand in GRL, whence a number of defaulting rules are provided. Figure 12 shows the attributes used for vertices in a production, and figure 13 gives the very similar list for edges. Keep in mind that all graph elements of a production $P \equiv (L, R)$ are folded into a single graph as an elegant solution to the "matching problem", the question of how to easily identify interface graph elements in the production representation's left-hand side L with their counterparts in the right-hand side R. The approach chosen here has each graph element declared only once, regardless of whether it is in the interface graph $L \cap R$.

The `info1` attribute is used to tag graph elements as being in either the exclusive left-hand side (`"Xlhs"`), interface graph (`"common"`), or exclusive right-hand side (`"Xrhs"`) of the production. Default rules are as follows:

- Vertices default to `common`. A production graph specification without any `info1` attributes at all would not generate or delete any graph elements[6].

[6] Of course such a production would be rejected by RSPGen due to the layering condition!

- Edges default to the "lowest common denominator" of their source and target vertices, that is, they default to being in the interface graph if possible:
  - If source and target are in `common`, the edge is in `common`.
  - If either source or target vertex is in `Xlhs`, the edge is in `Xlhs`.
  - If either source or target vertex is in `Xrhs`, the edge is in `Xrhs`.

  The remaining cases are errors: An edge can never connect a vertex from the exclusive left-hand side to one from the exclusive right-hand side because application of the production deletes one of the vertices as it adds the other (section 2.2). An edge cannot be part of the interface graph if either of its source and target vertices is not, because the edge would be dangling either before or after application of the production.

Each graph element in a production must also have a label; however the default rules make the use of the `label` attribute unnecessary in most cases. These rules are actually an extended version of the relationship between the `title` and `label` attributes defined by the GRL language:

- If the `label` attribute is used, it defines the element's label.
- Otherwise, if the element is a vertex, its `title` is looked up in the label family. If it is a valid vertex label name then it is taken as a `label` definition.

  This feature is in support of existing practice in defining textual grammar productions: If a symbol is the only one of its type used in the production it is often given the same name as the type itself. The rule does not extend to edges because their `title` attribute is normally ignored.
- Failing the above rules, the default label is used. As separate label families are used for vertices and edges, they will as a rule have different default labels. If no default label has been defined for the appropriate label family, the omission of a `label` attribute is an error.

The "symmetry pools" (`info3`) can be safely ignored; they are part of an optimization feature so there is no need to concern yourself with them until you start generating parsers for your own graph grammars. Their use and purpose will be explained in section 6.1.4.

The straightforward production on page 11 to add a single line between an Entity and a Relationship in an ER diagram would look something like this:

```
graph: {
  node: { title : "Entity"       }
  node: { title : "Relationship" }
  node: { title : "Line"         info1 : "Xrhs" }

  edge: {
    sourcename : "Entity"
    targetname : "Relationship"
    label      : "participates0"
    info1      : "Xlhs"
  }
  edge: { sourcename : "Line"    targetname : "Entity"       }
  edge: { sourcename : "Line"    targetname : "Relationship" }
}
```

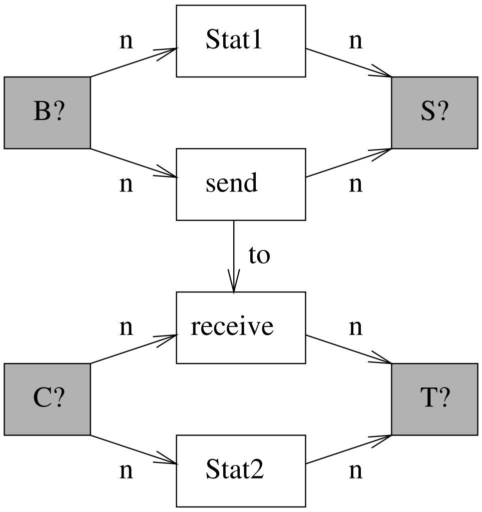


Figure 14: PFD Send/Receive production (as single graph)

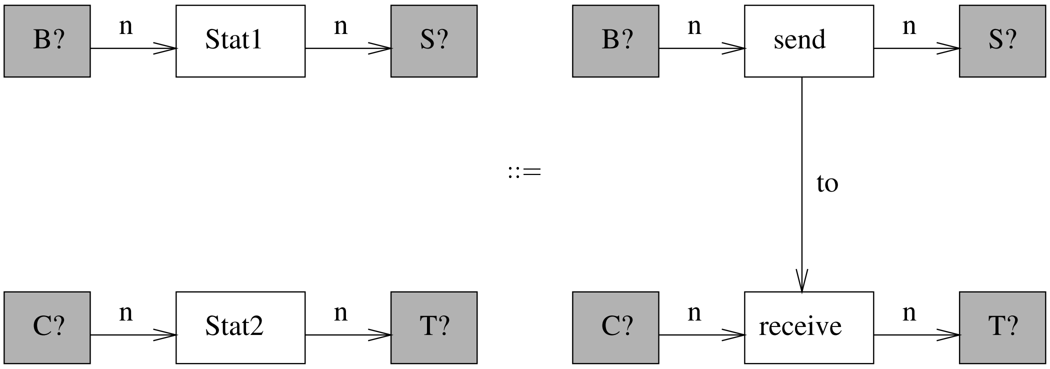


Figure 15: PFD Send/Receive production (as production)

It is relieving to note the amount of information that the parser generator infers from this specification without any need to specify them explicitly:

- The `Entity` and `Relationship` vertices default to `common`—right where we want them. No `label` attributes are needed because their titles are also the appropriate label names.
- The `Line` also naturally defaults to `Line` label.
- The `participates0` edge is explicitly specified as being deleted. Without the `info1` attribute it would have defaulted to `common`.
- Assuming that the default edge label is `touches`, the remaining two lines will default to this label. They are also forced into `Xrhs` because their source vertices are both in `Xrhs`.

Some real-life productions can still grow quite big. To give you an idea, here is a production from the PFD grammar that handles communication between processes. This is perhaps the most complex production in the grammar. The default edge label is `n` (for *next statement*). Refer to figure 14 to see what the combined graph looks like (ie. without taking the `Xlhs`/`common`/`Xrhs` specifications into account); figure 15 shows the production represented by this specification as replacing two statements by a send/receive pair with a communication being passed between them.

```
graph: {
  node: { title : "B?" }
  node: { title : "C?" }
  node: { title : "S?" }
  node: { title : "T?" }

  node: {
    title : "Stat1"
    label : "Stat"
    info1 : "Xlhs"
  }

  node: {
    title : "Stat2"
    label : "Stat"
    info1 : "Xlhs"
  }

  node: { title : "send"    info1 : "Xrhs" }
  node: { title : "receive" info1 : "Xrhs" }

  edge: { sourcename : "B?"      targetname : "Stat1"   }
  edge: { sourcename : "B?"      targetname : "send"    }
  edge: { sourcename : "Stat1"   targetname : "S?"      }
  edge: { sourcename : "send"    targetname : "S?"      }

  edge: { sourcename : "C?"      targetname : "Stat2"   }
  edge: { sourcename : "C?"      targetname : "receive" }
  edge: { sourcename : "Stat2"   targetname : "T?"      }
  edge: { sourcename : "receive" targetname : "T?"      }

  edge: {
    sourcename : "send"
    targetname : "receive"
    label      : "to"
  }
}
```

4.3 Generated Parser: GRL Input

Once a parser has been generated with RSPGen and compiled to produce a working executable, it will accept input in a very plain GRL format. As may be expected intuitively when looking at figure 14, this format is a subset of that used for productions: The labels are the same, the `title` attribute is still

Node	*type*	*description*	*mandatory*
title	string	Node name	yes
label	string	Node label	no

Figure 16: Parser input vertex attributes

Edge	*type*	*description*	*mandatory*
label	string	Edge label	no
sourcename	string	Name of source node	yes
targetname	string	Name of target node	yes

Figure 17: Parser input edge attributes

there, only the information pertaining to the application of productions and the `terminal` declarations are gone.

Figures 16 and 17 show the (now very short) lists of attributes relevant for parser input. The current implementation is mostly a proof-of-concept product though, and some attributes may be added later. There would be little point, for instance, to parsing productions that add `strings` to `boxes` when there is no way of retrieving the actual contents of that string.

The following example is a Process Flow Diagram taken from [17] with minor corrections. It is displayed in a more digestible form in figure 18. There is very little to explain here, except to say that the `label` defaults to vertex `title` or the appropriate default label in exactly the same way as it does for production specifications.

```
// Example Process Flow Diagram taken from paper
//      Defining and Parsing Visual Languages with
//      Layered Graph Grammars
// (J. Rekers & A. Schuerr).  See fig. 5 on page 11.

/* Small bug fix:  The original diagram contains four edges
 * erroneously labeled 't' (should have been 'n').
 */

graph:
{
  node: { title : "begin"   }
  node: { title : "fork"    }
  node: { title : "leftif"      label : "if"     }
  node: { title : "rightif"     label : "if"     }
  node: { title : "leftassign"  label : "assign" }
  node: { title : "rightassign" label : "assign" }
  node: { title : "send"    }
  node: { title : "receive" }
  node: { title : "join"    }
  node: { title : "end"     }
```

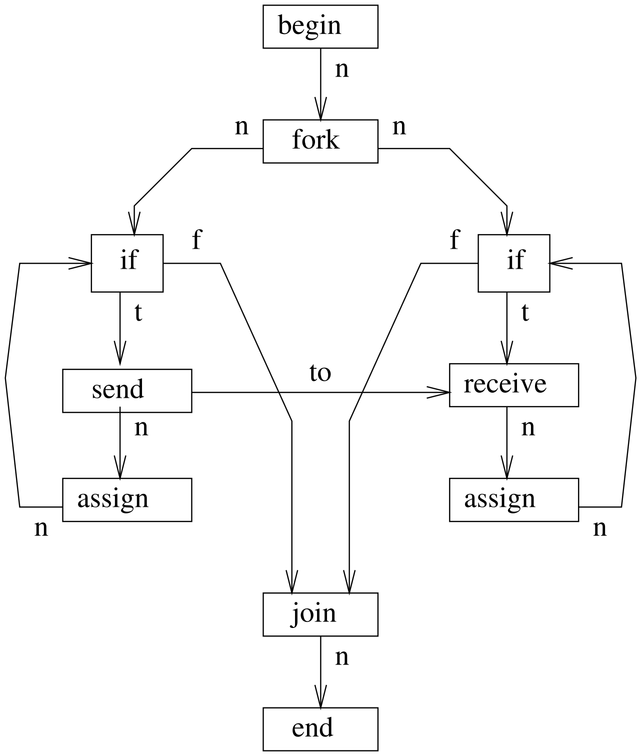


Figure 18: A sample PFD

```
edge: {
      sourcename : "begin"
      targetname : "fork"
}
edge: {
      sourcename : "fork"
      targetname : "leftif"
}
edge: {
      sourcename : "fork"
      targetname : "rightif"
}
edge: {
      sourcename : "leftif"
      targetname : "send"
      label      : "t"
}
edge: {
      sourcename : "rightif"
      targetname : "receive"
      label      : "t"
}
edge: {
      sourcename : "send"
      targetname : "leftassign"
}
edge: {
      sourcename : "receive"
      targetname : "rightassign"
}
edge: {
      sourcename : "leftif"
      targetname : "join"
      label      : "f"
}
edge: {
      sourcename : "rightif"
      targetname : "join"
      label      : "f"
}
edge: {
      sourcename : "leftassign"
      targetname : "leftif"
}
edge: {
      sourcename : "rightassign"
      targetname : "rightif"
}
```

```
  edge: {
        sourcename : "send"
        targetname : "receive"
        label      : "to"
  }
  edge: {
        sourcename : "join"
        targetname : "end"
  }
}
```

5 Implementation

5.1 Differences with the Original Algorithm

5.1.1 The `inconsistent()` function

The implementation does not follow the original algorithm by Rekers and Schürr religiously in all respects. As noted in section 3.2.1, the **inconsistent** function is not fully implemented which means that the bottom-up phase may get stuck in an infinite loop[7]. Due to the nature of this loop (self-perpetuating growth of the sentential form), the implementation will not sustain this situation forever but instead fail when the available memory is exhausted; one might say that such a loop does not continue ad infinitum but merely ad nauseam.

5.1.2 Propagation Order

Another difference in the bottom-up phase is that propagation of dotted rules is performed in a slightly different order. When reverse-applying a newly discovered production instance in the bottom-up phase, the algorithm as described by the authors [17] chooses to push any initial dotted rules for the added vertices onto the *Active* stack, and if new edges have been added, feed them to any suspended dotted rules that may want to propagate through them.

My implementation deals with the initial dotted rules immediately by treating the attachment of initial dotted rules to a vertex as a special case of dotted-rule propagation. This happens to eliminate a slight bug in the original algorithm for the case where a production's right-hand side contains only a single vertex, in which case attaching initial dotted rules to a vertex may lead to immediate discovery of a new production instance because the search plan for that production contains only a single vertex. This case was never caught by the original algorithm.

Furthermore, due to the concurrency semantics of the graph library used and the need to eliminate the possibility of duplicate dotted rules, dotted rules are suspended by the outer bottom-up loop <u>before</u> they are considered for propagation. This is used to defuse cases where propagation of a dotted rule dr which has just been taken from the *Active* stack directly causes a new edge e to be added to the graph which is also a candidate for propagation of dr. If dr were still to be considered "active" at that time, dr might be propagated through e twice (once immediately and one while considering more propagation candidates for dr).

Both the original algorithm and my implementation catch this hazard by limiting immediate propagation through newly added edges to "suspended" dotted rules. The graph library's set iteration mechanisms, however, are blind to elements that are added after iteration has started. Therefore in the imple-

[7]For a practical introduction into infinite loops see [23]

mentation the immediate propagation of dr in the booby-trap case described above must still proceed as if dr were suspended or e would never be considered as a candidate; this is achieved by marking dr as "suspended" as soon as it is grabbed off the *Active* stack instead of after it has been propagated.

5.1.3 Main Top-Down Loop

The original algorithm's top-down phase is essentially a loop centered around a stack of states; each iteration of the loop pops the top state off the stack and attempts to continue its derivation. Several possibilities arise, each with a different appropriate action:

- The stack is empty: Derivation has failed and there are no more ongoing attempts. This is the proverbial "it".
- The popped state is final; there are no more candidate production applications to be applied. If the derivation is valid, success is reported. If not, the state is simply dropped so we can pop an other and hopefully better one in the next iteration.
- The popped state contains an uncontested candidate production instance, ie. one that causes no conflicts with any other possible future candidate. The candidate is applied without further ado; the state is replaced on the top of the stack by its successor.
- Candidates are mutually exclusive; the popped state represents a choice point. Its place on top of the stack is taken by not one but two new states: One where a candidate pi is applied, and one where pi is rejected. The two can be said to be alternative realities and represent a "fork" between two derivation attempts.

My implementation makes mixed use of recursion (for choice points) and iteration (for uncontested candidates or continuing after the first leg of a fork has failed to produce a derivation) so the "stack of states" is more or less identified with the normal program stack. Pass-by-value semantics are used to implicitly copy the current state to explore a possible derivation path and to restore it if the path should fail.

The main loop is contained in a function called `derive()`. Its arguments include the algorithm's state before the choice point (API_c, EPI_c and PPI_c) and a production instance to be applied, pi. First pi is applied; then the loop applies as many production instances as possible, distinguishing the cases listed below. Note that the main function of the top-down phase calls `derive()` for every discovered instance of the grammar's special initial production until a derivation is found (if none is found, derivation has failed altogether). The following courses of action can be taken by `derive()`:

- There are no more $Candidates$; success is returned if appropriate, otherwise failure is returned and backtracking performed so the caller can make another attempt if desired.
- There is an uncontested candidate; it is applied and the loop rolls on.
- In the remaining case a choice must be made; some candidate pi' is selected and a recursive call to `derive()` is made with pi' as to-be-applied production instance. If this renders a valid derivation, success is returned.

  Otherwise, the choice point has effectively been eliminated. We now know that pi' must be excluded, and the function does so. The loop continues.

Note how restoration of an earlier state is mostly hidden away in the traditional call-by-value function call paradigm!

5.1.4 Incremental `cleanup()` and `initial-cleanup()`

As described in section 3.3.1, these functions were implemented in a way that hardly resembles the original algorithm at all. As a side effect they are not only more efficient but also more aggressive:

- The `initial-cleanup()` function deletes not only the deleting and using production instances of any never-generated graph elements, but also of any elements that may lose their deleting production instance as a result of this action.
- The `cleanup()` function catches and excludes more production instances than originally specified, in many cases preventing the offending edge from being generated in the first place and either avoiding or aborting the attempted derivation at an earlier stage.

It has been shown that these changes do not affect the correctness of the top-down phase because they will still only delete or exclude production instances that have been shown to be unusable in the current derivation attempt.

A related innovation is the concept of *self-contradictory* derivation attempts which is used to make the `cleanup()` function more efficient. When an edge's deleting production instance is excluded, the original algorithm acts differently based on whether the edge has been generated yet; the implementation recognizes the fact that the derivation can never succeed if the edge ever becomes part of the sentential form, and excludes its generating production instances whether one of them has already been applied or not. Should the edge already have been generated, this will show up as a contradiction and the derivation attempt is aborted.

As only relatively few derivation attempts are self-contradictory, later versions of the parser may defer this check until after completion of a derivation attempt to reduce its overhead which is a full pass over the shortest of API and EPI, and a partial one over the other (with the same length), for every applied production instance.

5.2 Dotted Rules

Dotted rules are the means by which the bottom-up phase of the parsing algorithm recognizes a match for (the right-hand side of) a production in the graph. Each represents a linearized search plan for a particular kind of connected subgraph and propagates along edges matching the current step in the plan until it has fulfilled all steps and a graph has been matched. For each step, it "waits" at a queue attached to one of the vertices that the expected new edge is or may be attached to. As a small optimization every vertex has separate queues for incoming and for outgoing edges.

A strongly object-oriented design was chosen for the representation of dotted rules, in spite of the fact that compiler technology has not yet developed to the point where such a programming style can be optimized as well as codes written in more traditional paradigms[8]. A single abstract base class `DottedRule` declares all functionality that is required of a dotted rule (propagation, matching graph elements, spawning production instances &c.) most of which is actually implemented by derived classes. This choice has hopefully contributed to the clarity and maintainability of the source code as well as encouraged exploitation of common functionality.

A separate dotted-rule class is derived for every production in the grammar; each object of such a class is a state machine with an active transition function that upon a successful propagation reroutes itself to another transition function that represents the next step in its search plan. These functions also declare all aspects of the graph elements that are requested by the next step in the search plan in

[8]Simple virtual function call elimination and receiver class prediction have been shown to bring potential performance increases of up to 48% (see [3]) but are not widely implemented at the time of writing.

an encoding defined and interpreted by the `DottedRule` base class: Desired edge and vertex labels (if applicable), the vertex that the rule is attached to, whether a candidate edge is to be incoming or outgoing relative to that vertex, and so on. When a new graph element is presented to the dotted rule, the base-class function `match()` is able to determine from this encoded information whether or not it is a candidate for propagation.

A dotted-rule object also contains pointers to all graph elements along its propagation path (up to its current position), divided into four arrays: Vertices in $Xrhs$, vertices in $common$, edges in $Xrhs$ and edges in $common$. If a dotted rule goes into accepting state (meaning that its search plan has been satisfied) a production instance has been discovered, and all this information is used to produce a `ProdInst` object.

In retrospect the `DottedRule` class could have been simpler because the parser doesn't actually "move" dotted rules through the graph; instead it performs a propagation by creating an updated copy of the dotted rule and attaches that to a new vertex. The old dotted rule is not deleted as there might yet be other candidate propagations for it from the same position in the graph. However little benefit (in performance or otherwise) is expected from such a simplification.

One more interesting note about the `DottedRule` classes is the fact that although the sets of matched right-hand side graph elements are members of the derived classes, most code that operates on them has been generalized to the point where they could be implemented at the base-class level. At one point this has caused a very subtle but destructive bug: When copy-constructing an object of a derived dotted-rule class, these sets (taking the form of smart-pointer arrays) where copied by the `DottedRule` copy constructor which was called from the derived-class copy constructor's base-class initializer. Being derived-class members however, they were subsequently initialized by the member initializers—calling their default constructors and thereby resetting them to *null*. No compiler warning was generated for this case.

5.3 Production Instances

A production instance, or an object of the `ProdInst` class as it is known to the compiler, is really a very static creature. Once it has been discovered and has turned out not to be $\mathbf{inconsistent}$, it hardly ever changes anymore. A good portion of its data space is occupied by pointers to the vertices and edges that make up its $Xlhs$, $common$ and $Xrhs$. Other information stored for each production instance pi includes:

- A unique identifying number assigned at creation time.
- Information about its production, eg. its full human-readable name.
- Pointers to all its members, grouped into six sets:
    1. `XlhsV`, the set of vertices deleted by pi
    2. `XlhsE`, the set of edges deleted by pi
    3. `commV`, the vertices in pi's interface graph
    4. `commE`, the edges in pi's interface graph
    5. `XrhsV`, the set of vertices generated by pi
    6. `XrhsE`, the set of edges generated by pi
- The number of original input elements generated by it.
- The set of other production instances pi' such that pi $\mathbf{above}$ pi'. This set may be extended throughout the bottom-up phase.

- The set of other production instances pi' such that $pi \in \mathbf{consequence}(pi')$. This set may be extended throughout the bottom-up phase.
- The set of other production instances pi' such that pi $\mathbf{excludes}^*$ pi'. This set may be extended throughout the bottom-up phase.

Due to its highly standardized and static nature, a single class is used to represent all kinds of production instances.

5.4 Graph Elements

The fact that vertices and edges get very similar treatment in the algorithm is reflected by the fact that graph elements are implemented as sibling classes: Both `VertexData` and `EdgeData` are derived from a single `CommonData` class. The graph template library is then used for creating graph objects with `VertexData` and `EdgeData` objects for vertices and edges respectively (the full C++ class name is `StructuredGraph<VertexData,EdgeData>`).

The `CommonData` class contains the following information:

- A unique identifying vertex or edge number.
- The graph element's *label*. Vertex and edge labels are kept completely separate but they are of the same C++ object type.
- A pointer to its *deleting production instance*, if any (`deletingPI`).
- A set of *using production instances* (`usingPI`).
- A set of *generating production instances* (`generatingPI`).
- Some additional information bits indicating whether the element was part of the original input graph, and whether it is part of the initial graph (generated by the special "initial production").

The only additional contents are defined for the `VertexData` subclass are the waiting queues for suspended dotted rules. The current implementation has only two such queues for each vertex: One for production instances awaiting an edge incoming to the vertex (`Suspi`), and one for production instances awaiting an outgoing edge (`Suspo`).

A possible future optimization is to divvy up these queues based on their expected edge label type, reducing the amount of work done when a new edge is presented as a possible propagation candidate. Instead of comparing it to each dotted rule's expectations, the parser would then only process the rules waiting in the queue of the edge's exact type. Inheritance could be simulated by inserting a rule into all queues whose types matched the expected one. A similar optimization is used by Chok and Marriott for their CMG parser (in that case the graph elements themselves are grouped by type, rather than the objects that are on the lookout for them).

This idea could be taken further by distinguishing vertex type as well; however overdoing this kind of optimization might nullify its effect by increasing memory usage, code complexity, and queue overhead; on the other hand a good portion of the "encoding" of dotted-rule information could be made implicit, reducing the size and complexity of the `DottedRule` classes. It would be interesting to see how far this idea will go. Initial tests have not been very encouraging but a thorough reconsideration of the entire design could make this idea much more profitable.

5.5 Top-Down Phase Backtracking

Result 33 and the subsequent implementation note describe an extremely efficient recipe for determining the $Candidates$ set incrementally using two counters (`higherPPI` and `higherAPI`) in every `ProdInst` object. Another counter, `potentialGPI`, is used to efficiently keep score on each graph element's excluded generating production instances.

There must be a catch somewhere, and indeed there is. Unlike the bottom-up phase, the top-down phase can't just go around modifying production instances because somewhere down the line the attempted derivation may fail and some earlier state must then be restored in the process of *backtracking* to find another ongoing derivation attempt[9]. The original information in both counters, for each and every production instance, must somehow be preserved as "time" proceeds.

Different approaches were used for preserving the counters' values:

- `higherAPI` can be restored from a global stack of applied production instances that is unwound during backtracking, "undoing" the production instances in the exact reverse order of how they were tentatively applied. At every choice point in a derivation the current top-of-stack address is remembered so the stack can be unwound exactly back to the most recent choice point.

  Every time a production instance pi is "undone" while backtracking, each production instance pi' such that pi $\mathbf{above}$ pi' (ie. each member of the $\mathbf{above}$ set of pi) has its `higherAPI` decremented by 1.

- `higherPPI` isn't decremented while backtracking, but it can be incremented by two causes: Undoing an applied production instance pi will increment the `higherPPI` for every pi' with pi $\mathbf{above}$ pi' just as it decrements its `higherAPI`, as one of its "higher" production instances is moved back from API to PPI.

  Moreover, backtracking may also move some production instances from EPI back to PPI—in which case we once again increment the `higherPPI` for all pi' with pi $\mathbf{above}$ pi'. We find the set of production instances that were excluded since the last choice point by storing the value of EPI_c at every choice point c, like we do with the top-of-stack address. Should any backtracking from c' (with $c' > c$) back to c be necessary, the set difference $EPI_{c'} \setminus EPI_c$ is computed. This set contains exactly those production instances that have been excluded since c; each of its members is a production instance that is "un-excluded" in backtracking.

- `potentialGPI` is incremented for any graph element x when the exclusion of some $pi \in Xrhs^{-1}(x)$ is undone during backtracking.

5.6 Set Representations

Throughout the parser, *sets* of objects are created and used. The performance tradeoffs involved in choosing an internal representation for this elementary data structure were evaluated for each case individually, rather than picking a single one-size-fits-all solution and sticking with it for better or for worse.

The reason for this is obvious: Different uses have different requirements. For some requirements packages it is possible to provide an interface where each atomic operation incurs $\mathcal{O}(1)$ costs. Additional operations retrofitted to the same solution will typically have much less attractive performance parameters. Or there may be a representation with very flat cost curves over a feature-rich interface—the

[9] This is the Computer Science equivalent of what is known in drama as the Weakest Plot Device of All Time: "It was only a dream!"

GNU `libg++` library offers several such data structures—which is still no match for a more restricted custom-made solution tailored to the specific required subset of operations.

The resulting arsenal of set representations is surprisingly small and unsophisticated; as it turned out the most complex code (a generic template-based hash-table implementation) was hidden deeply in the graph library where it serves to speed up operations such as loading a graph from file. Three representations are visible on the parser level:

Arrays Many pointer sets within the parser required an efficient enumeration capability, were of fixed size, and either did not need a lookup primitive or were small enough to make low-overhead operations worthwhile even at the cost of poor scalability. In these cases, standard C++ arrays were used. Addition is $\mathcal{O}(1)$ but lookup is $\mathcal{O}(n)$ for a set of n members. Full enumeration is $\mathcal{O}(n)$, or $\mathcal{O}(1)$ per member.

Memory usage for arrays is exactly one machine word for each member (densely allocated, therefore well-localized).

Bit vectors Sets of production instances were often involved in costly set operations such as union or intersection. Fast lookup was often required in sets of sizes that were unknown at compile time, and contents needed to be manipulated frequently with protection against duplicate entries.

For these cases a `bitVector` class was written, essentially a set-of-integers which represents the characteristic function of its set as an array of bits (1 for set members, 0 for non-members). The array is reference-counting to avoid unnecessary copy operations, and self-expanding so it never occupies more memory than needed to contain (the machine word of) the highest non-zero bit. Lookup is $\mathcal{O}(1)$, other operations are more complex. For a set of n members and m "potential members" (zero bits), set operations are generally $\mathcal{O}(m)$ but can often be deferred or optimized to $\mathcal{O}(1)$ and addition is $\mathcal{O}(1)$ or $\mathcal{O}(m)$ depending on whether the bit array needs to be expanded to accommodate the new member. Removal is always $\mathcal{O}(1)$.

Memory usage is one bit per potential member, but rounded upwards to an integral number of machine words. Allocation is dense and extremely well-localized.

Collection class In the more general case, where set size and contents are not fixed and enumeration is much more common than lookup, the graph library's `Collection` class was used. This has the classic linked-list interface with $\mathcal{O}(1)$ insertion and removal, $\mathcal{O}(n)$ enumeration and no lookup primitive.

Memory usage is several machine words per member, with usually little or no localization between members.

The bit vector solution was also chosen for one case, the set of using production instances for a graph element, where pointer enumeration was really required by the top-down phase. The bottom-up phase won out in implementation because of its much more intensive manipulation of the set which then remains static for the entire top-down phase. To fill this feature gap, the initialization for the top-down phase sets up a dense lookup table (an array of pointers) to find a pointer to any production instance given its unique identifying number as stored in the set.

5.7 Programming Details

5.7.1 Considerations

Three main bodies of code are involved in the parser implementation, each with different performance requirements. Hence it is not surprising that they developed as more or less independent products with similar but different coding styles:

1. Graph library: In order of priority, the objectives pursued were type safety, flexibility, robustness, and performance. This has led to defensive programming and large quantities of optional code. Later developments have rearranged this prioritization somewhat; performance has improved as a side effect of a major rewrite but robustness has not been fully restored yet.

2. Parser generator: The objectives here were correctness, maintainability, robustness and "getting the job done". Performance was *not* a consideration, and although the source is poorly readable because of its complicated nature, it leaves plenty of room for future adaptations.

3. Generated parser: Although this is subject to change, the main efforts were directed towards (1) meeting programming deadlines for a working product, (2) delivering high performance given a choice of search plans as made by the parser generator, meaning that the generated code is highly optimized without actually reordering or merging search plans; and (3) reducing dynamic memory usage to a minimum.

A common driving assumption was that current hardware trends would remain in force: That the speed gap between processors and memory would continue to grow, with increasing machine-word and cache-line sizes as well as clock speeds, and that hardware optimizations of existing superscalar CPU implementations (wide issue, branch prediction, speculative execution) would stay around for years to come. In accordance with this assumption, some in-memory data structures have been compressed in ways that would take more processor effort to extract information, but did reduce program data footprint.

5.7.2 Bit Vectors vs. Red/Black Trees

A more far-reaching choice was made on this basis in the case of the bit-vector representation of certain very sparse sets, such as the ones used to represent the $\mathbf{above}$ relationship. The use of red/black trees was suggested here (for an in-depth discussion of this data structure see [7]), which had much better theoretical performance parameters for some operations as only actual members are stored—as opposed to bit vectors where all *potential* members take up space. If n is the number of set elements and m the number of potential elements, then obviously $0 < n < m$; a sparse set is one where n is much smaller than m so $\mathcal{O}(n)$ solutions are generally preferred over $\mathcal{O}(m)$ ones.

In spite of the theoretical superiority of a red/black tree representation, bit vectors were preferred because a more practical analysis showed the tradeoff points (where a good complexity function starts winning out over low overhead) to lie outside, or at the edge of, the expected usage envelope.

Using a typical 32-bit processor with 32-byte cache lines as a comparison base, some typical cases of sparse bit vectors were examined; obviously, a single bit vector in this case would contain up to 256 "potential member" bits whereas a pointer-based tree implementation would require 12 or 16 bytes per member at the very least, taking $\mathcal{O}(\log n)$ poorly-localized sequential memory accesses to find any single member. Moreover the expected hardware developments would boost bit-vector performance due to larger cache lines, whereas a tree representation would merely take up the slack by doubling the size of each node in the transition to 64-bit computing.

Whole-set operations such as union or enumeration showed the difference more clearly: On bit vectors, they typically require $\lceil \frac{m}{w} \rceil$ memory accesses (where w is the machine word size in bits, typically 32 or 64) touching $\lceil \frac{m}{l} \rceil$ cache lines if l is the number of bits per line, typically 256 or higher. In an optimal tree representation these figures would be somewhere around $2(n+\log n)$ and $n\frac{3w}{l}$ respectively (best case), with less painstaking implementations steeply deteriorating to $2n\log n$ accesses to xn cache lines, where $\frac{3w}{l} \leq x \leq 2$.

The break-even point for such whole-set operations, in terms of total referenced cache lines is thus estimated at $n : m = \frac{1}{3w}$ or lower, meaning that the optimal tree representation only wins on the example architecture if less than 1 in 96 potential set members actually is a member. The break-even point in terms of actual word accesses ($n : m = \frac{1}{2w(1+\frac{\log n}{n})}$) comes slightly closer to equilibrium and approaches to $\frac{1}{2w}$ as n increases. In the example case it amounts to 1 in 96 potential members for $n \leq 4$ but eventually stabilizes at a ratio of 1 in 64 potential members.

Time will tell how well these tradeoffs work for real-life applications; the use of red/black trees may have to be reconsidered for very large inputs where many sets are extremely sparse. The existing `bitVector` class attempts to reduce the pathological cases by lazy expansion so that m is reduced to the highest number in the set; a similar optimization on the bottom end of the set could be beneficial if the 1 bits are not spread out too widely in the larger sets.

5.7.3 Development Environment

All programming was done with the GNU development tools including the gcc compiler. Compatibility with version 2.6.3 was required for practical reasons but the desire to anticipate the developing ANSI C++ standard in a future-proof product made support for older development tools a "second choice" from the start.

The main working environment was my trusty Amiga 4000 (still the most pleasant way to get the job done) with a Sony stereo next to it. Parts and versions of the software were also compiled and run on various Sun, Hewlett-Packard and SGI systems.

Thirty-six megabytes of RAM were acquired in the course of this project; two pairs of trousers lost due to dog bites; required disk quota have quintupled; four death threats were received; five compiler upgrades were made; approximately 4000 kilometers of travel were covered, some twenty megabytes' worth of software was updated and my advisor has handed in his resignation.

5.7.4 Graph Library

In support of our parser's development, a C++ template library for creation and manipulation of directed graphs was first developed as a preliminary project [22]. Although we had no knowledge of the C++ standard template library (STL) at the time, its design goals of flexibility and efficiency often led to similar design choices and considerations.

There was a very real need for a flexible low-level programming interface for the graph library as the exact requirements for implementing the Rekers-Schürr parsing algorithm had not been outlined in any detail and were likely to develop further as implementation of its prime application progressed. Some existing graph libraries were considered but tended to operate at higher levels of abstraction or catered to more "heavy-weight" uses such as databases.

The newly developed library also provided interoperability with existing software by supporting a load/save format reasonably compatible with the GRL graph language and the related GDL dialect used by tools like vcg [18] and EDGE [21].

Its flexibility may eventually give the library a life of its own; already it is a candidate for upcoming research in at least one foreign university. This project, the development of "universal" graph algorithms suitable for different graph representations, depends on the availability of such low-level graph template libraries which had hitherto been lacking.

The Anonymous Graph Library features robust iterators, smart pointers, multiple debug levels, call-back error handling and a flexible (if still single-format) load/save interface. The template mechanism is used to create graph structures out of almost any combination of user-defined object types (one for the vertices and one for the edges) with a minimum requirement of supported member functions.

5.7.5 Coding Standards

All code was written completely in a reasonably conservative post-AT&T C++ dialect. The Anonymous Graph Library was kept separate from the actual parser source tree, is self-documenting and features a fine-granularity `include` structure to promote data hiding. Implementation-private code and declarations are kept in separate header files that reside in a directory of their own.

The ANSI X3J16 committee's april 1995 Working Paper for the draft C++ standard [1] and gcc versions 2.7.0 up to 2.7.3 were used as references for identifying obsolescent language features. Some recent language improvements were used but only in those cases where compilation with gcc 2.6.3 remained possible with no more than minor configuration changes. Some arbitrary quantitative complexity limitations in the standard may have been exceeded; as a rule no attempts have been made to find and eliminate these cases.

Unfortunately the level of complexity does cause some problems for the compilers used:

- The `#include` nesting level is deeper than the GNU preprocessor cpp really supports. During development, most error and warning messages related to graph library code referred to non-existing line numbers or incorrect filenames which often made the errors difficult to track down.
- Compilation memory requirements are very high considering the size of the input source files. The first full compilation of the graph library with optimization enabled at debug level 20 took place in the night of june 26^{th}, 1996—some 18 months after its development started!—on an SGI Indy workstation. Until then, all attempts had resulted in the message "`virtual memory exhausted`".
- Compiler optimization is not always feasible, especially where the code for RSPGen is concerned. Several source files take more than sixty megabytes to compile with optimization enabled, which is at the time of writing an unusually high amount to have in a single workstation.
- Current versions of gcc are severely limited w.r.t. implementation of C++ templates. To work around this, by far most template code had to be inlined (especially the nested class member functions). This in turn is the main culprit for the high compilation costs.

Nevertheless all code has been compiled on these compilers with full diagnostics enabled and generated no warnings or errors. Non-optimized compilation of the entire source tree appears to use between 16 and 28 megabytes of memory on our systems. Explicit template instantiation is supported; parser-based instantiation is currently assumed.

The implementation falls short of good programming practice in two related areas: First, the `new` allocation primitives are assumed never to return *null* as they did according to the AT&T standard. If allocation fails, the language implementation is expected to handle the problem as an exception instead of returning from the operator, as suggested by the X3J16 drafts.

Second, the parser currently makes use of the knowledge that certain graph library functions can only fail due to `new` allocation failure so no error value is ever returned from those functions. In these cases the parser implementation simply neglects to check the return value and continues on the assumption that all is well. Fixing this may require a design update and should have no noticeable effect on the current implementation (apart from a minor slowdown) for which reason this work has been deferred to "mañana".

5.7.6 Compiler Problems

Some bugs in and problems with the compiler have been discovered and worked around where possible. These examples may be interesting for other gcc users; it may also answer some of the questions that people may have after reading the source code for the parser framework.

- As mentioned before, a derived-class constructor is allowed to pass (references to) its own members to base-class constructors before they have been constructed:

```
struct Base
{
  Base(int *x) { *x = 10; }
};

struct Derived : public Base
{
  int mine;

  Derived(void) : Base(&mine) {}    // No warning!
};
```

- Bogus name clashes are reported between function members of nested classes when defining them outside the class definition:

```
template<class T> struct Outer
{
  struct Inner1
  {
    void func();
  };

  struct Inner2
  {
    void func();
  };
};

template<class T> void Outer<T>::Inner1::func()
{  // "(‘template <class T> void Outer<...>::func()’
   //   previously declared here)"
}

template<class T> void Outer<T>::Inner2::func()
{
// "redefinition of ‘template <class T> void Outer<...>::func()’
//   type ‘Outer<...>’ is not a base type for type ‘Outer<...>’
//   common_type called with uncommon method types (compiler error)"
}
```

- Template arguments (other than `class`) don't require an exact type match as they should (in effect some coercions are quietly accepted):

```
template<int N> struct Foo
{
  void bar();
};

template<unsigned long N>     // No warning!
void struct Foo::bar()
```

```
{
}
```

- The most unfathomable one of all, as yet unresolved:

```
Generator.h: In method `cxxClass::Private::~Private()':
Generator.h:192: Internal compiler error.
Generator.h:192: Please submit a full bug report to
        `bug-g++@prep.ai.mit.edu'.
```

- Miscellaneous Malignant Messages (probably related to compiler port):

```
CheckProd.cc:36: sorry, not implemented: `save_expr' not supported
        by dump_decl
CheckProd.cc:36: sorry, not implemented: `gt_expr' not supported
        by dump_decl
CheckProd.cc: CheckProd.cc:36: sorry, not implemented:
        `gt_expr' not supported by dump_decl
In function `':
CheckProd.cc:36: confused by earlier errors, bailing out

{standard input}: Assembler messages:
{standard input}:60469: Fatal error:Case value 0 unexpected at line
                1748 of file "/ade-src/fsf/binutils/gas/write.c"

graphcontents.cc: In function
  `static bool ANON_StringArray::strsame(const char *, const char *)':
graphcontents.cc:32: Could not find a spill register
(insn:QI 5 3 7 (set (reg/v:SI 9 a1)
        (mem:SI (plus:SI (reg:SI 15 sp)
                (const_int 4)))) 34 {movsi+1} (nil)
    (expr_list:REG_EQUIV (mem:SI (plus:SI (reg:SI 15 sp)
                (const_int 4)))
        (nil)))
graphcontents.cc:32: confused by earlier errors, bailing out
```

6 Future Work

6.1 Research

6.1.1 Coupled Grammars

Section 1.3.2 briefly mentions the possibilities of coupling graph grammars. Implementation on top of the existing parser framework should be trivial; further research into the practicalities of such a mechanism would be desirable.

6.1.2 Connected Right-Hand Sides

Section 2.4 describes, among other limitations, the need for *connected right-hand sides* in all grammar productions. Removing this restriction would be possible with relatively minor changes to the parser, but at a cost. Propagation to non-adjacent vertices would greatly increase the numbers of propagation candidates for dotted rules. The need for such an extension should be worth investigating, as well as ways of making it as painless as possible.

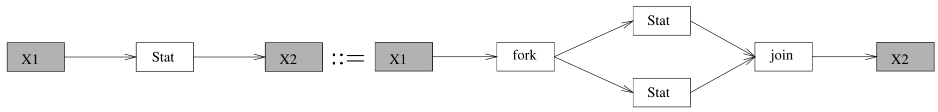


Figure 19: “Two-way fork” production in PFD grammar

6.1.3 Automatic Layering

Section 4.2.1 touches on the subject of automatic layering. Layering currently needs to be taken into account when designing graph grammars and layering information must be hand-coded into the grammars' label families. Future implementations could take this work out of the hands of the user by examining the entire grammar and deducing a maximal layer distribution much like compilers for some advanced programming languages may infer a variable's type from its use in the program.

Writing such an inference engine could be fairly difficult because not only inheritance information needs to be taken into account (which would be very simple to analyze), but the layer vectors of the individual productions as well. These layer vectors are really multisets of the number of occurrences of each layer in a production's left and right hand sides; what matters is that the difference between these is lexically positive, and the inference engine may reorder entries (in all vectors simultaneously, of course) or even merge and split layers.

Having a maximal layer distribution is so desirable because at least one future optimization, *levelling* (see section 6.2.5 for a discussion) gets performance improvements by restricting the labels under consideration by the bottom-up phase to a sliding interval of layers. Levelling only becomes effective if there are enough layers, and the more the merrier.

6.1.4 Symmetry Reduction

As yet mostly unexplored but already deplored is the occurrence of *symmetry* in graph productions: A production's right-hand side may match the exact same subgraph in several ways that are, to all intents and purposes, equal. They will give rise to distinct dotted rules and cause the discovery of distinct production instances during the algorithm's bottom-up phase, and finally, in the algorithm's top-down phase, take part in rivaling derivations that are, after all, equivalent. This is a special case of *ambiguity*, which is defined as the property that some members of the language that a grammar generates have more than one valid derivation in that grammar.

As an example, consider the production shown in figure 19; it was taken from the grammar for Process Flow Diagrams in [15, 17]. It states that any statement (`Stat`) may be expanded into a fork/join construct embracing two concurrent statements.

Now suppose we had a PFD containing a two-way fork, ie. a subgraph that looks exactly like the production's right-hand side. The search plan for this production's right-hand side would match it in two ways, as there are two ways to map the two `Stats` in the production's right-hand side onto the two `Stats` in the actual graph (they are “interchangeable”). As there are now two alternative ways to parse this single subgraph, the total number of possible derivations for the whole graph is doubled!

The formalism, adding insult to injury, allows no meaningful distinction to be made between the two ways to parse the subgraph, making one of them redundant. But so much for the bad news. The good news is that criteria can be found to prevent one of the two alternative matches for the subgraph at a very early stage—as soon as the dotted rules find the symmetric graph elements, in fact. Before we look at the actual reduction of symmetry however, let's discuss its forms and their detection.

Most symmetry situations can be detected at parser generation time for any production $P \equiv (L, R)$ by producing a copy R' of its right-hand side R and releasing the dotted rules for P on it as if it were an input graph being parsed; if n "production instances" of P are "discovered" in this way, each covering the whole of R' then P is said to have an n*-way static symmetry*. When parsing an input graph for this grammar, any match for R will be covered by at least n production instances; presumably these are fully interchangeable.

Even more symmetry may appear at run-time due to the fact that two elements in P's interface graph $L \cap R$ may be mapped onto the same graph element. In contrast to the method for detecting static symmetry described above, the possibilities for this kind of symmetry become visible as the cases where some "production instances" discovered in R' do not cover all its elements. If n of them are "discovered" that cover exactly the same subgraph of R then P is said to have an n*-way dynamic symmetry*.

Note that some cases of dynamic symmetry may still cover the whole of R' if the "interchangeable" elements are part of the interface graph $R \cap L$. In reality it is also possible that a production contains both static and dynamic symmetry, but detecting these cases requires more detailed analysis and more precise definitions to be made in the future. Separate "symmetries" can be distinguished in a single production, perhaps some static and some dynamic, and each containing one or more equivalence classes. A static symmetry can then presumably be recognized by the fact that each equivalence class contains at most one interface-graph element. These can be further divided into *horizontal* symmetry containing no interface-graph elements (symmetry within a single addition to the graph) and *vertical* symmetry between already present elements and newly added elements (symmetry across production instances).

The notion of dynamic symmetry is a bit trickier than that of static symmetry. Given a dynamically symmetric production instance, the existence of a "mirror" production instance may depend on run-time information such as whether or not two symmetric graph elements in the production have been mapped onto the same graph element in the sentential form, or some form of symmetry may only occur if two or more particular elements of the production's interface graph are mapped onto the same graph element.

Furthermore, the fact that the graph elements making up the symmetry, being part of the interface graph $L \cap R$, exist both before and after the application of P makes the assumption of equivalence a lot more dangerous. It will generally hold true if addition of the production's exclusive left-hand side $L \backslash R$ treats the symmetric elements in exactly the same way, ie. if the symmetry is somehow maintained.

Now that we have some notion of the problem, ways of attacking it may be found. Forgetting dynamic symmetry for now, static symmetry can be reduced by taking a set of "interchangeable" elements in R and imposing some total order on them (eg. in ascending order of object number), prohibiting any propagation of dotted rules in the bottom-up phase that violates this total order[10].

The aforementioned "interchangeable" elements in a static symmetry are determined by the non-singleton equivalence classes that appear in the mapping functions for its "production instances" onto R'. If two vertices v_1 and v_2 are mapped onto their own copies v'_1 and v'_2 respectively by one match in the symmetry, and onto each other's copies by another (ie. $v_1 \mapsto v'_2$ and $v_2 \mapsto v'_1$), then v_1 and v_2 form an equivalence class; imposing a total order on the mappings of v_1 and v_2 while parsing will eliminate this symmetry. Larger equivalence classes are possible but should be quite rare[11].

It stands to reason that no more than a single equivalence class per symmetry must be ordered, preferably the one that can be checked at the earliest possible point in the search plan. Once the symmetry has been eliminated, imposing an extra restriction would block arbitrary dotted rules from finishing a perfectly valid match!

Detection of symmetry has not been automated yet, but symmetry reduction based on manually inserted declarations in the production graphs is a working part of the current implementation. The user may declare graph elements to be part of an equivalence class using the `info3` attribute in the GRL production specification; the only equivalence classes currently supported are `"1"`, `"2"` and `"3"` but each name harbours two distinct equivalence classes as vertices and edges are kept completely separate.

[10] Given the symmetry, there must be another dotted rule for the same subgraph that does obey the total order.

[11] And a good thing too: Symmetry grows as a factorial function of the size of these equivalence classes!

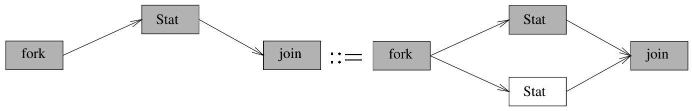


Figure 20: “Extended fork” production in PFD grammar

Derivations	2-way	3-way	4-way	5-way	6-way	7-way	8-way
No Reduction	2	6	12	20	30	42	56
Full Reduction	1	3	6	10	15	21	28

Number of derivations for n-way fork

Production Instances	2-way	3-way	4-way	5-way	6-way	7-way	8-way
No Reduction	12	30	56	90	132	182	240
Full Reduction	8	18	32	50	72	98	128
Vert. Reduction	11	27	50	80	117	161	212
Hor. Reduction	9	21	38	60	87	119	156

Number of Production Instances for n-way fork

Complexity	Production Instances	Derivations
No Reduction	$4n^2 - 2n$	$n^2 - n$
Full Reduction	$2n^2$	$\frac{1}{2}n^2 - \frac{1}{2}n$
Vert. Reduction	$\frac{7}{2}n^2 - \frac{3}{2}n$	$n^2 - n$
Hor. Reduction	$\frac{5}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$

Number of Production Instances for n-way fork

Figure 21: Effect of symmetry reduction on n-way fork

Symmetry reduction was tested on the PFD grammar; in the case of the two-way fork production from figure 19, the two generated `Stat`s were declared to be in symmetry pool `1` removing the horizontal static symmetry and eliminating the doubling effect that these forks used to have.

The PFD grammar also contains a case of vertical symmetry: There is a production for widening existing `fork`s with additional `Stat`s, similar to the recursive production in our context-free textual grammar for the natural numbers on page 4. This production is shown in figure 20: The two `Stat` vertices in the production's right-hand side are interchangeable because it does not matter which `Stat`s are taken to be produced by the original “two-way fork” production and which are read to have been added later with the “extended fork” production. This case also shows that correctness of symmetry reduction is harder to prove for vertical symmetry than it is for horizontal symmetry because interaction with other productions must be taken into account.

The combined effect of symmetry reduction for these two productions is impressive. Every n-way fork used in the original grammar produces $n(n-1)$ or $n^2 - n$ or $2\binom{n}{2}$ equivalent derivations. Intuitively this is because one `Stat` is picked for the upper `Stat` in the “two-way fork” production (n possibilities), another for the lower `Stat` ($n-1$ possibilities), and any other `Stat`s are added later without any interdependencies so there are $n(n-1)$ possibilities. In combinatorial terms, two distinct `Stat` vertices are picked from a set of n which can be done in $2\binom{n}{2} = \frac{n!}{(n-2)!}$ ways.

With symmetry reduction enabled this is halved to $\frac{1}{2}n(n-1)$ or $\binom{n}{2}$ derivations; as execution time is a worse-than-linear function of the number of possible derivations, the net performance gain is much greater than a constant factor even if only a single fork is involved. It is interesting to note that horizontal symmetry constitutes the bulk of the improvement, as can be seen in the tables of figure 21. Vertical symmetry reduction only reduces the number of production instances but not the number of derivations.

6.2 Implementation

6.2.1 Semantic Constraints

A problem encountered in 2.5 is that some useful language properties cannot be described in graph productions. Take Petri nets, for example: Writing a simple grammar for Petri-net graphs such as the example in section 4.1.4 is a matter of seconds. But the first grammar that springs to mind, at least in my case, is unable to enforce restrictions such as the one that forbids transitions from having a position as both input and output. Working around such problems is awkward and involves a complete redesign of the grammar, if possible at all.

The ability to recognize these forbidden cases is vital in cases where an input graph may have two derivations that are equivalent from the graph-grammar perspective but only one is semantically correct. An end-user program based on a graph parser might then succeed or fail for a given input, depending only on which of the two derivations the top-down phase would happen to complete first.

Two solutions that can be implemented at low costs are the inclusion of *embedded semantic conditions* similar to those in the CMG formalism, and *iterated derivations* where the program may reject a derivation if it fails to meet its semantic conditions and instruct the parser's top-down phase to continue where it left off to find another. These solutions may be combined to suit the inspection of fine-grained and structural conditions respectively.

6.2.2 Consistency Check

Section 3.2.1 notes the fact that the $\mathbf{inconsistent}$ function has not been fully implemented in the bottom-up phase due to excessive performance costs. None of the grammars investigated so far were capable of producing $\mathbf{inconsistent}$ production instances, so cases where this makes a difference are expected to be quite rare. Future research could provide a way of detecting at parser generation time whether a grammar is capable of producing the problematic variety of $\mathbf{inconsistent}$ production instances, or at least whether this can lead to an infinite loop during the bottom-up phase.

Such generation-time detection might produce failure cases that are hard to predict by reasonable human contemplation of the grammar; at this time it is hard to say which cases they would be, and this may remain so even after implementation. However this situation is comparable to the LALR class of textual grammars, whose definition is blatantly ad-hoc and un-intuitive to a reader unfamiliar with the workings of the actual parsing algorithm. Quoting from Aho, Sethi and Ullman [2]:

> The table produced by Algorithm 4.11 is called the LALR *parsing table for G*. If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar.

The bison manual has the following to say about the issue:

> In general, it is better to fix deficiencies than to document them. But this particular deficiency is intrinsically hard to fix; parser generators that can handle LR(1) grammars are hard to write and tend to produce parsers that are very large. In practice, Bison is more useful as it is now.

The popularity of LALR grammars is witness to the fact that this annoyance is not considered prohibitive by the majority of programmers. Grammars will sometimes need to be rewritten, but over time language design has also adapted itself to the limitations of the parsers used. Perhaps a similar development will in time also occur for visual grammars; alternatively, detection of the problematic variety of **inconsistent** production instances could be limited to a minimal number of checkpoints (at least one in each potential cycle).

6.2.3 Search Plan Selection

RSPGen has but two tasks in life: To construct a search plan for finding subgraphs that match a production's right-hand side, and to spew C++ code to execute that search plan. It already does a pretty good job of the latter if I may say so myself, but the current implementation just takes pot luck as far as the former is concerned.

There is room for significant statistical optimization here. Of the number of search plan instantiations (`DottedRule` objects) created in the bottom-up phase of the algorithm, only a fraction will ever complete to produce the actual production instances (`ProdInst` objects) that the subsequent top-down phase is interested in. Reducing this number could have a significant impact on overall performance of the bottom-up phase, especially if the reduction could be made without incurring too much parse-time overhead.

The most obvious optimization comes at no charge whatsoever in terms of parse-time overhead: If most search plan instantiations are going to fail anyway, it would be best to let them fail as early as possible. This can be done by reordering the search plan to start with those elements that are least likely to find a match in an arbitrary input graph; the general goal is to reduce the chance of further propagation as quickly as possible. Unfortunately this optimization assumes some degree of clairvoyance (as well as a good heuristic—minimizing the "cost function" for each production is no easy task) so realistic traceback information will probably be needed to make the optimization work.

Given a working prediction of matching frequency however, the code for RSPGen could be restructured with acceptable effort to make room for "plug-in" external optimizers. Such optimizers could find search-plan sequences (or reshuffle existing search plans) using their own particular insights—creating the possibility to run their respective sets of search plans through real-world situations and compare their output quality.

6.2.4 Merging Search Plans

Real graph grammars tend to contain copious amounts of duplication. Many productions contain right-hand sides that are similar to a certain extent, and in fact many more are created by the mechanisms of *wildcard labels* and *label inheritance*. Both of these can be viewed as kinds of macros for graph productions: Using either of these is equivalent to producing multiple productions that are isomorphous with the one original, but each with "slightly different" labels for one or more of the elements in their right-hand sides[12]. Or sometimes, as with the productions from figures 19 and 20 (the `fork` productions of the PFD grammar), two productions' right-hand sides just happen to resemble each other.

Here too we find room for optimization. With the necessary changes in the bottom-up phase, similar search plans could be merged into multi-production plans that propagate as a single `DottedRule` objects up to the point where they differ, after which they could propagate along different paths as graph elements of their respective expected label types show up. In effect the linear search plans would have their initial parts tied together to make a single decision tree where search plans could detach themselves at any stage depending on where they first differ.

[12] This set of productions spanned by the single production specification may be called a *normal form* for the production

Attractive as this technique may seem at first glance, the increased overhead in performance-critical code sections such as the matching and propagation functions may defeat its purpose in practice for any number of cases. Another major issue is that in order for any noticeable effect to be achieved, search plans must be reordered to create the longest possible matches between them. This clearly conflicts with the practice of simple search-plan optimization as discussed above: If one is given priority, the other suffers. Should this result in too few search plans being merged, then the benefit is no longer worth the additional overhead and search-plan merging becomes a liability instead of an asset.

6.2.5 Levelling

Another technique that has been proposed in [16] to speed up the bottom-up phase is that of *levelling* (the name was taken from [4] where the method is used for a CMG parser), which reduces the amount of dotted rules to be taken into account at any time during the bottom-up phase of the algorithm by prioritizing the `DottedRule` objects in the *Active* stack according to two labelling properties of the productions they belong to.

Definition 6 *For any production $P \equiv (L, R)$, its* lowest left layer $lll(P)$ *is the lowest layer number of any element in its exclusive left-hand side $L \backslash R$. If $L \backslash R$ is empty, then $lll(P)$ is defined to be* inf.

The highest right layer $hrl(P)$ *is the highest layer number of any element in P's exclusive right-hand side $L \backslash R$.*

Ordering the productions' dotted rules in the *Active* queue primarily by $lll(P)$ for each dotted rule belonging to production P and secondarily by $hrl(P)$, giving highest priority to the dotted rules of production P with the lowest $lll(P)$ or $hrl(P)$, creates a steady progression from low to high layer numbers in the bottom-up phase by always adding those new graph elements whose labels are in the lowest possible layers. This is beneficial for two reasons. First, the number of suspended dotted rules is minimized by generally bringing the moment of appearance of new graph elements labeled in a particular layer closer to the time of creation of any dotted rules that should eventually match them (low layers first, high layers later). Second, it provides a useful criterion for finding dotted rules that can no longer complete their matches.

This surprising effect is a result of the layering condition: Due to the lexical ordering of productions' left- and right-hand sides, the lowest (and as dictated by the prioritization, foremost) $lll(P)$ in the *Active* queue at any time during the bottom-up phase determines the lowest layer of labels that may yet be added to the graph. It follows that if any dotted rule d has been suspended to wait for an element of label l to be added to the graph, then d may be discarded as soon as the lowest $lll(P)$ in the *Active* queue is greater than the layer number of l.

In support of this optimization, maximal layer distributions are desired to make the best possible use of this "sliding window" effect; another desirable property of the layer distribution is to maximize the number of productions P such that $lll(P) > hrl(P)$ so that application of any instances of P will not cause further propagation of any dotted rules of that same production P as it would for instance with a recursive production.

A tertiary prioritization of the *Active* queue based on *remaining search plan costs* is also suggested; this could be used to bias the progression of the bottom-up phase towards rules that are more likely to complete soon to avoid an explosion of dotted rules.

6.2.6 Labeled Dotted-Rule Queues

Section 5.4 describes how the waiting queues for suspended dotted rules may be split up in future implementations so that a graph element's label can be used as an exact predictor to find the single queue holding any suspended `DottedRule` objects that it might match, potentially diminishing the amount of time spent in the algorithm's bottom-up phase. When inheritance is used in the grammar, the optimization is complicated by the fact that a single step in a search plan may be satisfied by graph elements of different label types. One way to tackle this problem is to insert the single `DottedRule` in all matching queues so inheritance would be removed from the parser implementation altogether.

As more information is encoded in a `DottedRule` object's queue selection, its object size may be greatly reduced which would definitely have a beneficial effect on propagation as well as matching speed. The implementation would have to be restructured on both the parser and the parser-generator level to suit this optimization, but it is likely that this will combine well with the addition of levelling.

One question remains open: Will it be possible to make the increase in `DottedRule` efficiency outweigh the deterioration in vertex creation overhead? The improvement equals a constant portion of the time spent on matching/propagating each `DottedRule` whereas the deterioration equals a constant addition to the time spent on creating/deleting each graph vertex. This is certainly a positive sign as the former overwhelmingly outnumber the latter.

6.2.7 Coverset Inspection

Some of the potential derivations explored by the top-down phase of the algorithm will, as a rule, fail to complete forcing the parser to backtrack and retry. A likely reason for this failure is that not all input graph elements have actually been generated[13]—meaning that a (nearly) complete derivation attempt has been performed before the problem is detected! Tests so far indicate that this problem can have a radical effect on the top-down phase. Performance could potentially be greatly improved if some fast way could be found to determine incrementally, perhaps during the bottom-up phase, whether an instance of the initial production and its underlying derivation attempt could be capable of producing the input graph in its entirety.

This is where context sensitivity bites us in the leg. A context-free parser such as a CMG one like [4], dealing as it does with a derivation tree instead of a DAG, could simply cumulate a count for the number of input elements covered by a production instance and its subtree during (the equivalent of) the bottom-up phase. Any instance of the initial production that does not generate exactly the right number of input elements can then be rejected. Any overlap in cover sets could still be easily handled by the $\mathbf{excludes}$ relationship.

Things are not so easy with context-sensitive parsers. How would we cumulate the input elements covered by single production instance? If the $\mathbf{consequence}$ relationship were to be used for this, what of eg. the situation where $pi_1, pi_2 \in \mathbf{consequence}(pi_0)$ and $pi_3 \in \mathbf{consequence}(pi_1) \wedge pi_3 \in \mathbf{consequence}(pi_2)$? How to avoid counting the input elements generated by pi_3 twice (once as counted by pi_1 and once as counted by pi_2)? Alas, this situation cannot be statically detected in the general case.

An obvious approach would be to take the chance anyway, counting too many input elements if we have to, and accept what little improvement we can get. As the error is always positive (input elements are being counted double), rejecting all derivation attempts whose cumulated input-element counts are too low would be perfectly safe. As the error would typically be quite large however, this would not lead to any noticeable reduction in the number of failed derivation attempts.

One way to increase the usefulness of this trick would be to keep a separate count for every label type, hoping that at least one label might escape the double-counting bonanza and come out too low in a significant number of cases. Looking in a completely different direction, whereas keeping complete bit vectors of covered input elements is going to be much too expensive, it might be feasible to pick some single label type and keep score of each production instance's coverset restricted to input elements of

[13] Note that the $\mathbf{excludes}^*$ relationship ensures that no derivation attempt can ever generate too many input elements, so all we have to worry about is the case where not all graph elements are generated.

that particular label. Following this scheme any derivation attempt could be rejected that fails to cover all input elements of that label type.

6.2.8 Partitioning Production Instances

Section 5.6 notes a weak spot in the performance of bit vectors, which are extensively used throughout the parser but particularly in the top-down phase: Operations on very sparse sets will be dominated by the number of potential elements (total bit-vector size) rather than the number of actual elements (number of '1' bits or *population count*). The main worry are the $\mathbf{above}$ sets, which do tend to be quite sparse for larger derivations.

An improvement that can be made here is to partition these by production: Each production instance will have not one single $\mathbf{above}$ set, but one for the instances of every production P in the grammar where instances of P are numbered consecutively (instead of a single global assignment of unique numbers as is currently the case). The benefit that may be expected from this is that in some cases it can be shown that the $\mathbf{above}$ relationship cannot hold between instances of particular combinations of productions: For some productions P and Q in the grammar, it is possible that there can be no instance pi_P of P and instance pi_Q of Q such that pi_P $\mathbf{above}$ pi_Q. In these cases there is no need for instances of P to have an $\mathbf{above}$ set partition for instances of Q; however there is no need for RSPGen to detect and optimize such cases while generating search plans, as the `bitVector` class handles zero-sized sets in a fast and intelligent manner.

The sparseness of the $\mathbf{above}$ sets can hopefully be ameliorated by this technique by concentrating large numbers of non-elements into unused bit vectors resulting in better data localization, higher hit rates, and potentially a more efficient structure for production-instance data.

6.2.9 Meta Grammars

Section 4.1.4 touches on the subject of too much work currently being done "by hand" in a textual editor. Eventually it should become possible to perform all work in a visual, interactive, graphical environment such as a graph editor. Support for other graph languages can be added to the implementation with relative ease, if need be, to make more existing software useful for this task.

Another alternative is to use the parser package to write software to generate its own input: The user can then draw the productions for his language in a structured drawing program such as fig or idraw. The development environment runs this picture through a lexical analyzer generated by LeVis and channels the resulting graph through a parser generated by RSPGen based on a *graph grammar for graph grammars* or *meta-grammar*. This parser distills from the input the actual graph-grammar productions as intended by the user.

The grammar, now in a format suitable for use by RSPGen, is finally streamed into the existing parser generation framework to create a working parser. In a nutshell, this level of automation would be sufficient to generate a working parser directly from a free-form picture of the grammar.

What happens after this remains an open question: Development of visual software must be made as declarative as possible, freeing the developer of tasks that would otherwise be performed in hand-written code in some general-purpose programming language. Existing attribute-grammar theory could be applied to graph grammars with minor modifications; user code could be embedded in parsing actions as they are with yacc or bison, or coupled grammars could transform input sentences into a data structure that corresponds directly to the semantics given to them by the grammar.

6.2.10 Incremental Parsing

Some attention has already been given to possibilities of using the Rekers-Schürr algorithm, perhaps based on my implementation, in an interactive visual environment. One interesting possibility that has not been discussed so far is that of on-the-fly parsing in a specialized *but free-form* graphical editor. The editor could "learn" a grammar by reading in its label families and productions much in the same way that RSPGen does, and try to parse it as the user adds or removes visual elements.

Such on-the-fly parsing has distinct advantages and disadvantages. First of all there is the advantage that work can be done in the program's main event loop, ie. while the program is waiting for the user instead of the other way around. Second there is the advantage of immediate feedback: The current view that the parser has of the existing picture can be represented to the user immediately, replacing the traditional save-compile-modify feedback loop by a continuous editing process that is finally, hopefully, rewarded by a well-formed visual expression.

The main disadvantage is the real-time nature of interactive applications. The daunting complexity of visual parsing makes significant reuse of parsing efforts imperative for the development of a practical interactive parser/editor environment. This means that *incremental* parsing becomes a necessity: Response time should not degrade anywhere near as quickly as it does in traditional batch-style parsing.

The Rekers-Schürr method shows promise in this light. The bottom-up phase lends itself well to incremental implementation where dotted rules remain in place, waiting for new graph elements to be added. The top-down phase may then be re-executed whenever desired to get the result of a full parser run on the current graph or picture.

In some cases, notably when graph elements are deleted, production instances may have to be discarded. Unfortunately there is no efficient way to scoreboard and reuse their numbers so performance will degrade slowly even when the user simply keeps adding and deleting the same visual element. A "garbage collection" pass may be needed from time to time to refresh the allocation of production instance numbers and restore response time to an acceptable level.

7 Behaviour

As was to be expected, the time taken for the parser to process an input graph is at the very least an exponential function of the size of the graph for the general case. However sheer input size does not appear to be the most useful dimension when measuring the parser's performance; most graph grammars contain inherent ambiguity resulting in multiple equivalent derivations and a great number of invalid potential derivations being found whereas some simple ones may produce only a single derivation for any valid input. In the worst cases, where there is no basic order in the input, the number of possible derivations found may explode dramatically and so will execution time. Statistical distribution of the total execution time is likely to be a function of such factors as

- Ratio of failed derivation attempts in the top-down phase
- Number of production instances discovered by the bottom-up phase
- Graph sizes at different stages of the algorithm

Quantitative performance evaluations have so far focused on the `fork`/`join` construct in the PFD grammar (which is in fact the only ambiguity in the grammar) because its results are highly regular and because it poses a major performance bottleneck. With full symmetry reduction, as it turned out, the parse time for a PFD containing a single n-way fork soon waxed intolerable for growing n. Parsing a 12-way fork took some four minutes on an SGI Indy workstation, against only 31 seconds for an 11-way one.

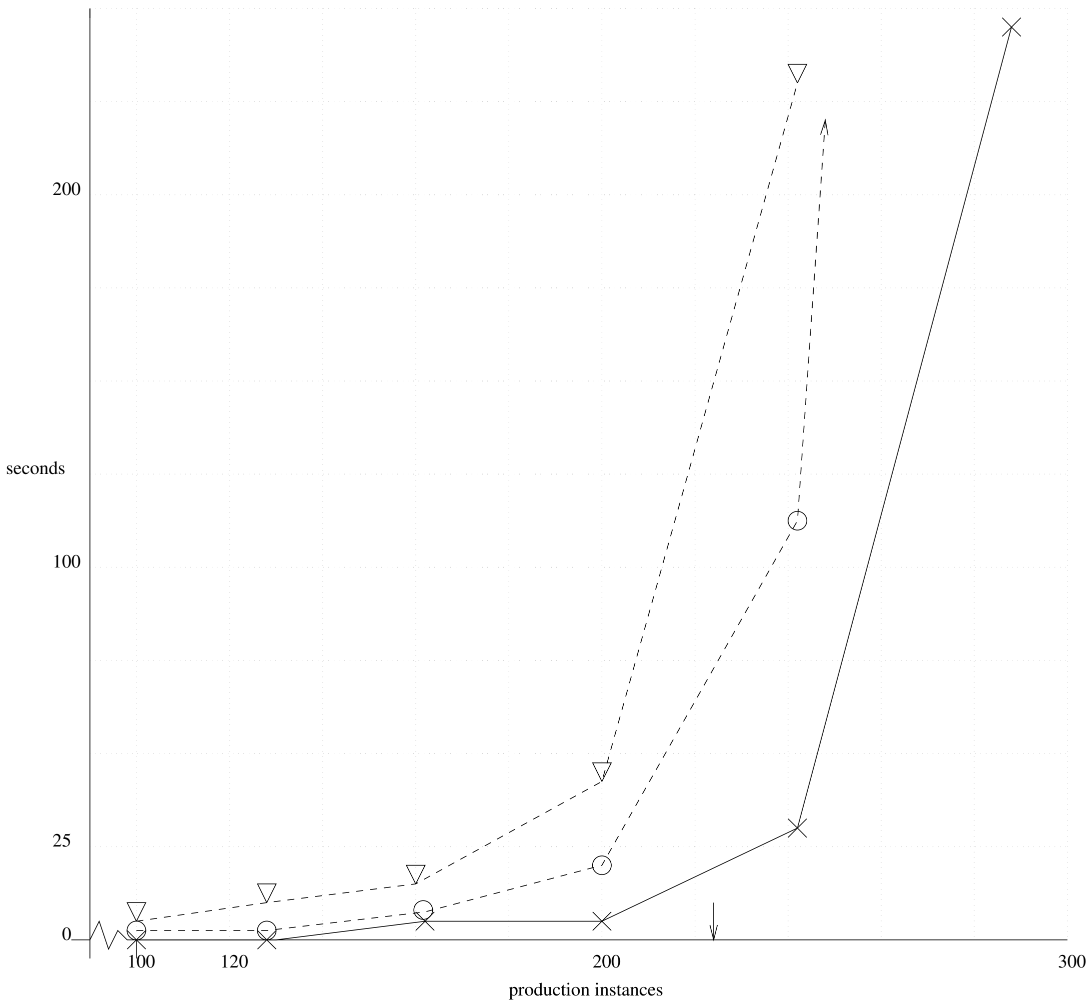


Figure 22: Parse Time vs. Number of Production Instances

Figure 22 plots execution time for these forks, ranging from a 7-way for to a 12-way one, as a function of the total number of discovered production instances (on three different computer systems). Even considering the high complexity of the parsing algorithm, the sudden steepness of the curve seems remarkable; it is not unlikely that it is caused in part by bit vectors, which are extensively used by my implementation, hitting the *cache-line barrier*. As noted in section 5.7.1 typical current 32-bit CPU architectures will squeeze up to 256 bits of a bit vector into a single L1 cache line. However the first machine word in the bit vector is used for administrative purposes, so that at most 224 useful bits will fit into the vector's first cache line. Assuming that the `new []` operator built into the compiler aligns its allocations to cache-line boundaries (a common optimization in compiler run-time systems) then a sharp increase in execution time can be expected as large quantities of bit vectors are outgrowing their first cache line and breaking into a second one. The critical point is marked in the graph by an arrow pointing at the y axis.

More importantly however the plot of figure 22 shows *worst-case performance* with regard to failed derivations. The order of attempted derivations is extremely unfortunate, leading to a violent explosion of the number of backtracks performed[14]. For this particular case, a simple reversal of the arbitrary total order imposed by the symmetry reduction engine improves the situation beyond imagination.

Best-case performance for the same inputs is shown in figure 23 for the slower two of the test systems. Any unwarranted optimism over these results may pale a little in the light of the incredible "luck" that the algorithm appears to have in these cases: All these best-case derivations hit the jackpot at first try, following the right path all the way towards completion without backtracking even once.

However the difference between these two graphs is dramatic enough to show the need for further research into ways of getting closer to best-case performance; it may well be that the symmetry-reduction order is less arbitrary than originally thought. The optimization sketched in section 6.2.7 above may also be of great help in eliminating false derivations, or a prioritization similar to the levelling technique described in section 6.2.5 could be applied to the top-down phase to increase the chance of hitting a valid derivation as soon as possible. The highest possible number of covered input elements seems a likely criterion for such prioritization.

Reversing the order in which both alternatives are explored when a choice point is encountered by the top-down phase may have the same effect as reversing the symmetry-reduction order, but would require a structural change in the implementation. More statistics will need to be gathered to show how well these transformations would work out in more diverse practical cases.

It is also interesting to note that the algorithm appears to be fairly well-balanced: Although the bottom-up phase is not as susceptible to wild variations in parse time for different inputs, its time curve generally matches the envelope described by top-down performance for these cases as can be seen in figure 24. This can be taken as an indication that there is no general trend of one phase becoming insignificant compared to the other in terms of execution time for growing inputs. The balance will rather depend on other factors such as ambiguities in the grammar and positive or negative influence of such choices in the implementation as the chosen symmetry-reduction order.

Drawing far-reaching conclusions about performance characteristics based solely on the number of discovered production instances is dangerous; at the very least a token effort should be made toward validating this intuition. I shall attempt to do this based on some simple statistics.

Figure 25 shows the effect of symmetry reduction on the performance of both phases of the algorithm (only the best-case is measured here because the parser happens to hit the best case either way when reduction is disabled). Note that the x axes of these graphs measure the total number of production instances discovered, *not* the width of the `fork` or the size of the input graph; this plot does not show the same inputs at the same x offset for the symmetry-reduced parser and the unreduced version. The

[14] The number of failed derivations for this test grows as an exponential function of the number of production instances, eg. 409114 backtrack operations are performed when parsing the 12-way fork!

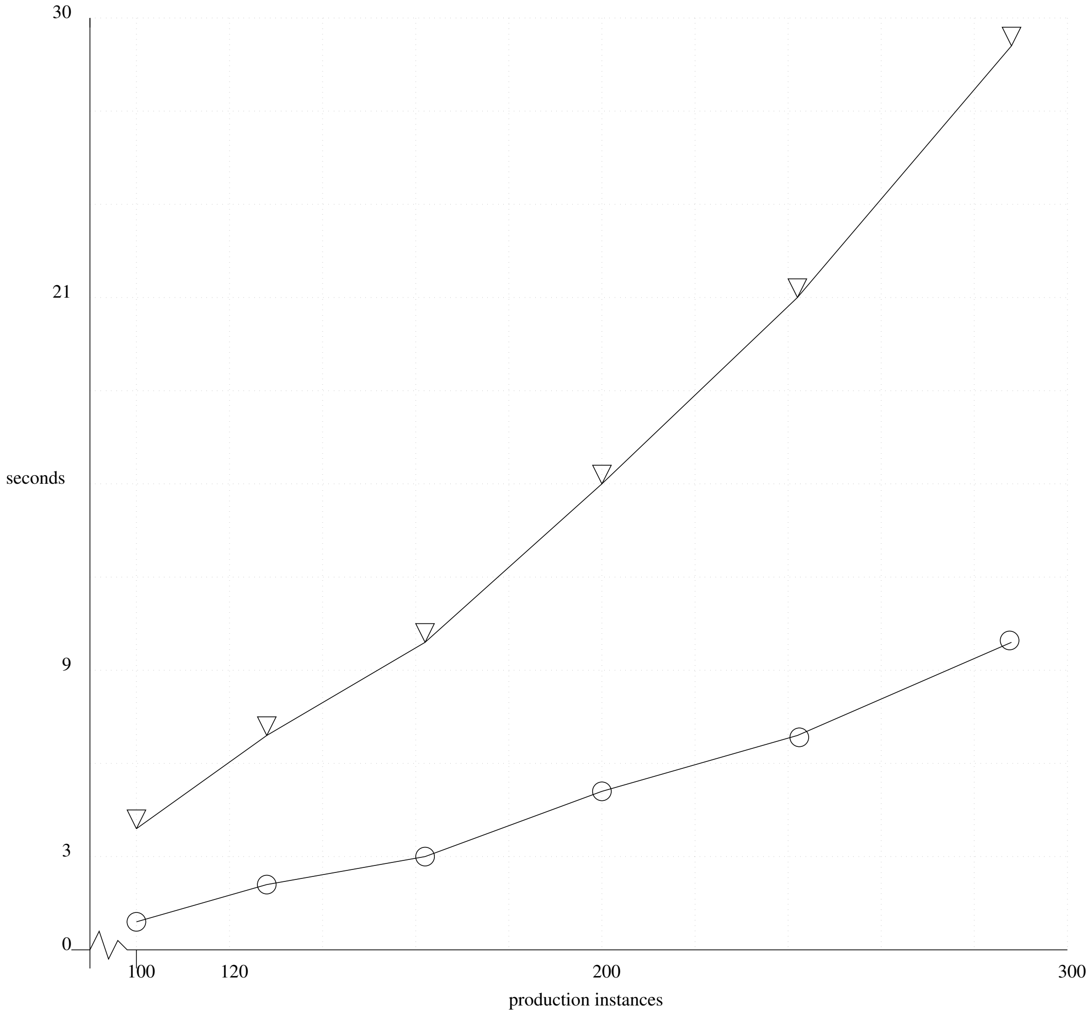


Figure 23: Parse Time vs. Number of Production Instances (best-case)

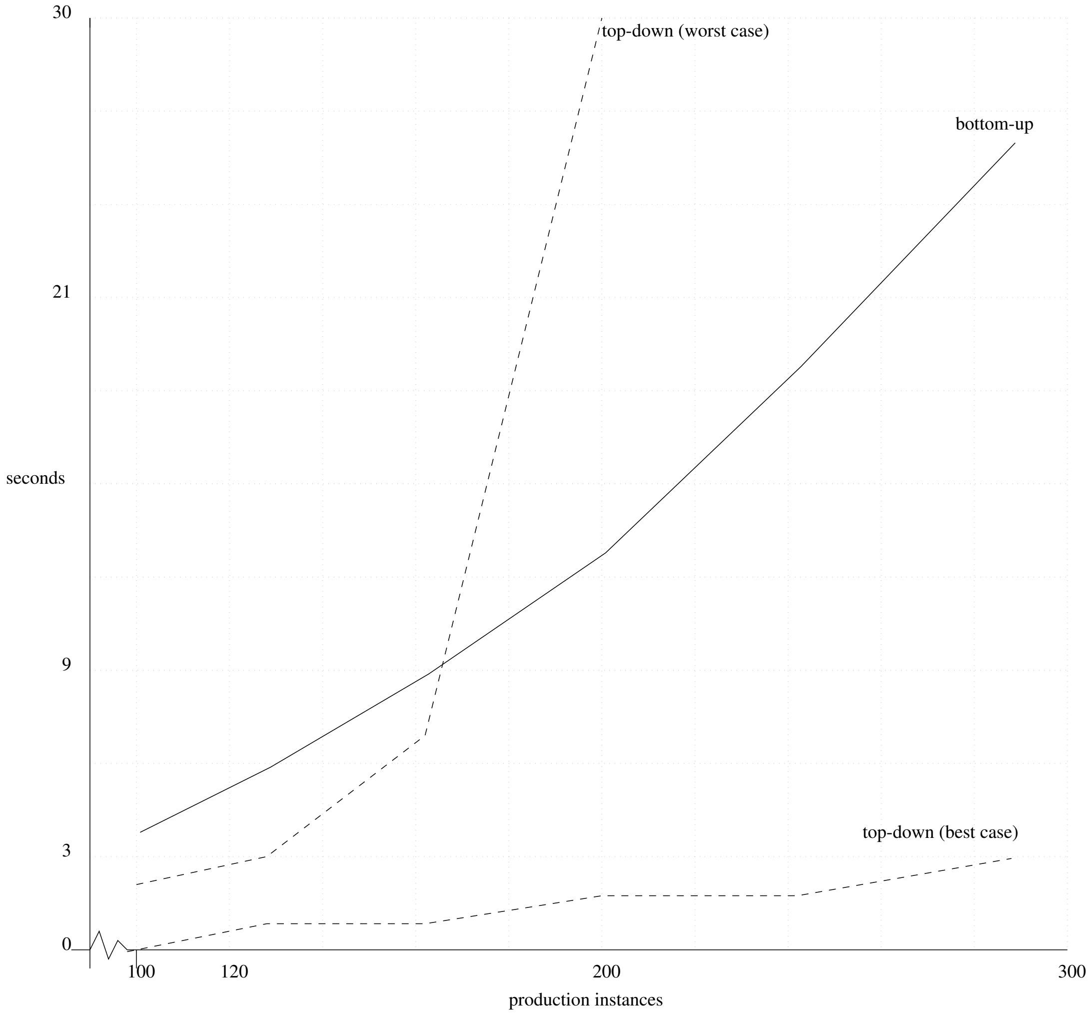


Figure 24: Comparison of Bottom-Up Phase vs. Top-Down Phase

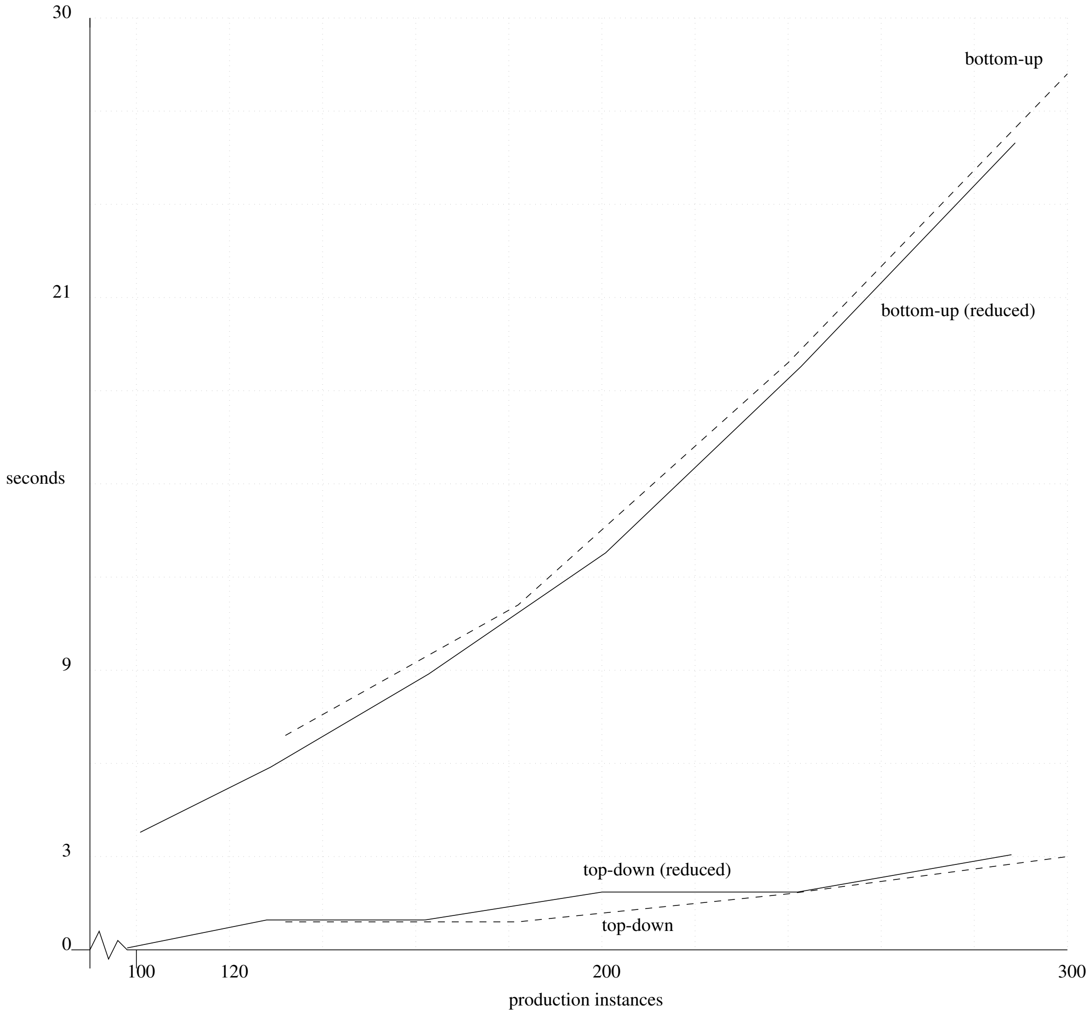


Figure 25: Curves With and Without Symmetry Reduction

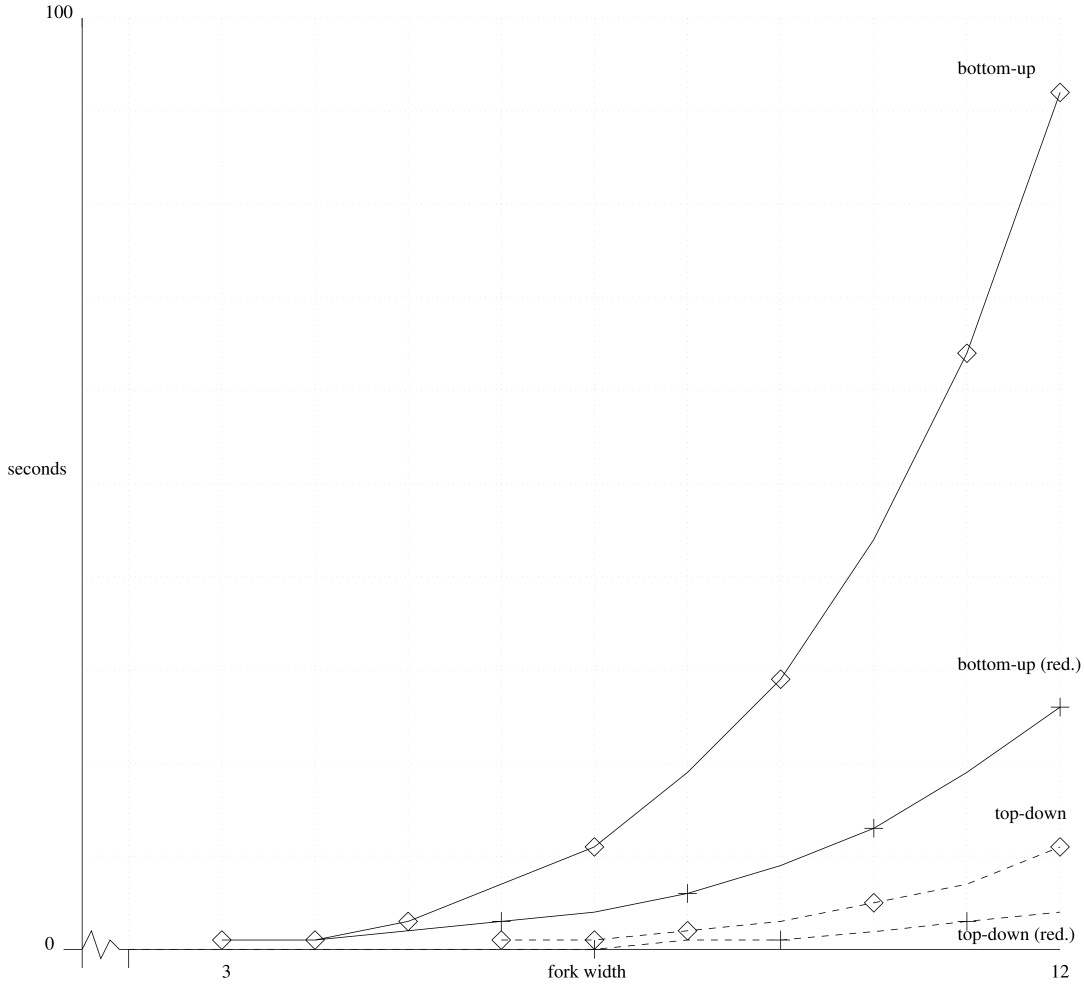


Figure 26: Effect of Symmetry Reduction

symmetry-reduced version actually performs much better for the same input file in this test as can be seen in figure 26. What the graph in figure 25 does show is that judging by the close match between the curves for the symmetry-reduced and the unreduced parser, the total number of production instances discovered by the bottom-up phase is apparently a highly useful predictor for total execution time of both phases of the algorithm in the case where no backtracking is performed.

Memory usage is currently still a problem: The 12-way fork, for instance, takes between twelve and fifteen megabytes of memory to parse (without symmetry reduction). Extensive use of a custom array allocator, already in use in the bottom-up phase, should help to improve this situation by reducing memory fragmentation. Large amounts of small memory chunks are allocated at several places in the algorithm, and in most cases the C++ run-time system provides more flexibility than required. The use of an array allocator may reduce allocation overhead per chunk (both time and memory) by several orders of magnitude in exchange for reduced opportunity for memory re-use.

8 Conclusion

8.1 General

There is no question that parsing graph grammars with the Rekers-Schürr algorithm is extremely hard. As it looks now, there will always be a performance barrier keeping it from fulfilling its full theoretical potential. However it has been shown that there is definitely a set of problems where even this first implementation can do useful work, and that there is still room for significant improvement in parser speed. Future research in this direction, especially the development of a large and diverse test set and the gathering and analysis of practical performance statistics, is paramount to creating a useful tool for visual software development based on the Rekers-Schürr approach—an important end goal that in my opinion justifies a significant investment of time and effort.

Perhaps the most important achievement of this implementation is that a testbed for future experimentation is now available. Experience has already shown the value of actual performance information and the presence of a working body of code: In the first week of testing and debugging an operational version of the parser software, minor additions such as the backtracking mechanism and careful selection of iteration directions have resulted in a ten-fold increase of performance. The noteworthy behaviour of the gap between best-case and worst-case performance of the top-down phase for the PFD grammar, which would have been hard to predict in theory, was discovered at almost literally the last minute during the analysis of unexpected timing results. The list of possible performance factors and future improvements has been growing steadily throughout the testing period.

Even after the value of these improvements has been measured or quantified, their implementation will pose new challenges as conflicts of interests between them become more restrictive. As can be seen in the field of traditional compiler construction, the resolution of such conflicts and the development of strategies that combine them in an effective way is as difficult and as important as developing the optimizations themselves; in fact they gain more importance as the field of opportunity becomes saturated with the many individual optimizations invented[15]. The development of such optimization strategies is based on the analysis of practical cases, for which the existence of an implementation testbed is invaluable.

Finally, a fringe benefit of having a working parser framework is its educational use. Combined with a graph visualization tool like vcg, animated visualizations of a large class of graph transformations, including the parsing algorithm itself, can be programmed in a very short time.

8.2 Visual Parsing

Based on the behaviour of the existing implementation of the Rekers-Schürr algorithm, it appears to be a generally viable strategy for parsing visual languages, but with certain limitations. Barring significant future breakthroughs in graph-grammar analysis, the construction of graph grammars is likely to remain a matter for expert users with some grasp of the practical implications of the choices they make during development. Restructuring grammars manually is likely to remain important for the generation of efficient parsers and semantic constraints, once implemented, will be an indispensable tool for removing from the grammar the ambiguities that could otherwise cause parse times to skyrocket.

The end user, on the other hand, will need no such expert knowledge except perhaps some experience with the relevant grammar and how visual input may be restructured to keep processing times near their minimum. When users start to work with inputs that are sufficiently large to make this necessary, it may be reasonably expected that they have gained this level of experience on the way.

[15] Due to the complexity this problem, researchers tend to develop a blind spot for it. Books such as [24] will describe conflicting transformations in great detail but shy away from the question of how to combine them effectively.

8.3 Graph Parsing

Facilitating the development of visual environments and applications is not the only potential use of the main Rekers-Schürr algorithm. A generic parser generator for graph grammars may be of use for other graph-based applications such as control-flow graph analysis, constraint solvers, circuit analysis or even the parser framework itself. Visual parsing using a lexical analyzer such as LeVis is really one case of graph parsing where the input graphs are generated automatically. This will often be the case in graph-based use.

Parser-generator users in this application area will generally have more than sufficient insight in the problems involved and should be able to produce efficient grammars. On the other hand the end user of a parser-based graph application, if human at all, will rarely be in a position to restructure his graphs to suit the parser. The mind boggles at the idea of, for instance, an entire microprocessor being redesigned to better suit one of its design or verification tools. In some similar applications a long wait is acceptable, eg. the constraint-based automated design suite used by Digital Semiconductor may run for several days on a powerful computer to produce a complete chip design but this still reduces the total turn-around time by eliminating a slow external feedback cycle between the designer and the subcontracted manufacturer.

That said, low parsing speed inevitably reduces the usefulness of the framework and makes a hand-coded parser more attractive because the latter can be optimized for the specific application. The major limiting factor is the amount of ambiguity in the grammar; if none is present, parser performance should be adequate for very large graphs, perhaps with some tailoring to suit the characteristics of the job at hand. In the more general case, symmetry reduction and coverset inspection may be able to reduce ambiguity to the point where an automatically generated Rekers-Schürr parser can be profitably used.

8.4 The Project

Although most effort in this project was focused on delivering a working implementation of the Rekers-Schürr parsing algorithm, some hopefully useful research has been added to the main work by Jan Rekers and Andy Schürr. Some improvements were made to the algorithm and many ideas were discussed, although by no means all of them survived a careful examination. Many questions remain and conclusions are open-ended; indicating several clear paths of possible future research was given priority over singling out one of these and pursuing them to the bitter end. The resulting chapter, *Future Work* which unorthodoxly precedes *Conclusion* in this thesis, will hopefully show that there are years of useful work left in this subject and that whatever results are available at this time are nowhere near definite.

Another regret is the fact that there has not been enough time for some more practical work towards integration with LeVis including extensive testing with its output—particularly for the highly complex MSC grammar that was the focus point of its implementation. A practical comparison with the CMG parser described in [4] could have added a useful perspective as well; a limiting factor in both these cases is that the programs were finished only very shortly before the conclusion of this project.

As for the practical viability of the Rekers-Schürr parsing approach, which is first and foremost performance-limited, there was not enough time after completion of the implementation to conduct as thorough a performance analysis as I would have liked; but if one conclusion must be drawn, it would be that the (to a certain extent unavoidable) poor performance of the generated parsers will hopefully be offset by the apparent fact that it is largely ambiguity and backtracking, not mere input size, that drive the performance curve.

Finally, an original contribution has been made to the algorithm in the form of symmetry reduction, which has already proven its ability to boost the performance of both phases of the algorithm. A first implementation of this technique has been included in the parser framework and possible means of automating it have been discussed.

References

[1] Accredited Standards Committee: X3, INFORMATION PROCESSING SYSTEMS. Working paper for draft proposed international standard for information systems–programming language C++. Available by ftp from `ftp.cygnus.com` and others, april 1995.

[2] Alfred J. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. also known as the Dragon Book.

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. Technical Report TRCS 95-22, University of California, december 1995. Available by ftp from ftp.cs.ucsb.edu or via WWW from `http://www.cs.ucsb.edu/TRs` or `http://www.cs.ucsb.edu/~oocsb`.

[4] Reinier Balt. Full CMG parsing. Master's thesis, University of Leiden, The Netherlands, august 1996. Email: `lbalt@wi.leidenuniv.nl`.

[5] S.S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *Proceedings 11th IEEE Symposium on Visual Languages – VL'95*, pages 242–249, 1995.

[6] S.S. Chok and K. Marriott. Parsing visual languages. In *18th Australasian Computer Science Conference*, Glenolg, South Australia, 1995.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] Arie de Graaf. LeVis: Lexical scanning for visual languages. Master's thesis, University of Leiden, The Netherlands, july 1996. Email: `A.C.J.M.de.Graaf@cpb.nl` or `adegraaf@wi.leidenuniv.nl`.

[9] Stefan Manke. Generierung von Graphbearbeitungsprogrammen aus objekt-orientierten Spezifikationen (Generation of graph manipulation programs from object-oriented specifications). Master's thesis, University of Karlsruhe, Department of Informatics, march 1990.

[10] Stefan Manke and Frances Newbery Paulisch. *Graph Representation Language: Reference Manual*. See EDGE package [21].

[11] K. Marriott. Constraint multiset grammars. In *Proceedings 10th IEEE Symposium on Visual Languages – VL'94*, pages 118–125, 1994.

[12] Frances Newbery Paulisch. *The Design of an Extendible Graph Editor*. PhD thesis, University of Karlsruhe, Department of Informatics, 1991.

[13] 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, 1994. Also available from ftp site *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1994/tr94-11.ps.gz*.

[14] J. Rekers and A. Schürr. A parsing algorithm for context-sensitive graph grammars (short version). In *Workshop handouts of the fifth international workshop on Graph Grammars and their application to computer science – GraGra94*, Williamsburg, Virginia, 1994. This paper is a shortened version of [16].

[15] J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *Proceedings 11th IEEE Symposium on Visual Languages – VL'95*, pages 195–202, 1995. Available from *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1995/tr95-15.ps.gz*.

[16] J. Rekers and A. Schürr. A parsing algorithm for context-sensitive graph grammars (long version). Technical Report 95-05, Leiden University, the Netherlands, 1995. Available via ftp from *ftp.wi.leidenuniv.nl*, file */pub/CS/TechnicalReports/1995/tr95-05.ps.gz*.

[17] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. Technical Report 96-09, Leiden University, the Netherlands, 1996. Available via www from `http://www.wi.LeidenUniv.nl/TechRep/tr96-09.html`.

[18] Georg Sander and Iris Lemke. Vcg. Available by ftp from `ftp.cs.uni-sb.de`. Email: `sander@cs.uni-sb.de`.

[19] Georg Sander and Iris Lemke. *VCG: Visualization of Compiler Graphs*. See VCG package [18].

[20] 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.

[21] University of Karlsruhe. *The generic graph browser EDGE*. Software package, available by ftp from *ftp.informatik.uni-karlsruhe.de:/pub/graphic/edge.tar.Z*.

[22] Jeroen T. Vermeulen. The Anonymous Graph Library. C++ template library, 1996. Available by ftp from `ftp://ftp.wi.leidenuniv.nl/pub/CS/misc/jvermeul/pub/anon_lib.tar.gz` Email: `jvermeul@wi.leidenuniv.nl` or `jtv@xs4all.nl`.

[23] Jeroen T. Vermeulen. Viability of a parsing algorithm for context-sensitive graph grammars. Master's thesis, University of Leiden, The Netherlands, september 1996. Email: `jvermeul@wi.leidenuniv.nl` or `jtv@xs4all.nl`.

[24] Michael Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.