Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation*

Paolo Bottoni,
Gabriele Taentzer†
University of Rome, Italy
{bottoni,gabi}@dsi.uniroma1.it

Andy Schürr
University of the Federal Armed Forces, Munich, Germany
Andy.Schuerr@unibw-muenchen.de

Abstract

We propose an extension of layered graph grammars to allow complex constructs in the definition of visual languages. For these grammars, the parsing process terminates. Its efficiency can be improved by critical pair analysis which allows conflicting rule matches to be computed before runtime. Using these results, decisions between conflicting rules can be delayed as far as possible. The paper illustrates the power of the approach by providing a grammar for a significant fragment of statecharts.

1 Introduction

Syntactical definition of visual languages (VLs) and visual language parsing are difficult problems due the absence of an easy to use and efficiently parsable standard syntax definition formalism. Most proposals published until today rely on grammar rules allowing replacement of a single nonterminal in the left-hand side. Using these proposals, it is difficult to produce readable syntax definitions of diagrammatic VLs, such as the hierarchical automata of Harel's statecharts [3]. Therefore, context-sensitive graph grammars have been proposed in the form of *Layered Graph Grammars (LGGs)* [6]. LGG rules are allowed to delete and create several elements and relations, represented as nodes and edges, simultaneously.

Nevertheless, defining complex VLs such as statecharts with LGGs is still difficult, due to the lack of the following mechanisms: 1) *node embeddings* which allow redirection of edge bundles of unrestricted cardinality from one node to another; 2) *negative application conditions (NACs)* excluding the existence of certain nodes and edges in the context of a rule match; and 3) *complex predicates* in the form of path expressions, which require e.g. the (non)existence of a path of typed (labeled) edges of arbitrary length between certain nodes. Furthermore, the worst-case complexity of the two-phase LGG parsing algorithm defined in [6] is exponential in time and space.

Reserved Graph Grammars (RGGs) were defined as a restricted and modified form of LGGs [10]. They support a simple form of embedding, where already existing edge bundles may be redirected from one node to another node (without any means to retype or change the direction of redirected edges), while conditions of types 2) and 3) cannot be stated. Furthermore, if RGG rules are locally confluent, the polynomial naive one-phase LGG parsing algorithm defined in [6], which exchanges left- and right-hand sides of rules and applies them exhaustively, can be used without any need for backtracking.

Unfortunately, the local confluence criterion of RGGs is rather restrictive and prohibits e.g. processing of ambiguous VL representations like the statechart example presented in the following section. Therefore, we present here a new form of LGGs, called *Contextual Layered Graph Grammars (CLGGs)*, which support node embeddings, negative application conditions, and complex predicates. As a consequence, a completely new layering condition has to be defined to guarantee termination of the parsing process, i.e. decidability of the generated graph languages. Furthermore, new static analysis techniques are used to identify a maximum set of rules which may be parsed without any need for backtrack-

*Research partly supported by the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

†On the leave of the Technical University of Berlin, Germany, gabi@cs.tu-berlin.de

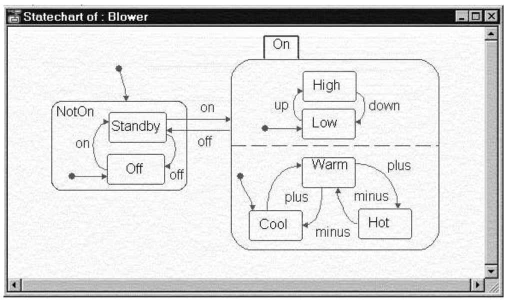


Figure 1. A sample statechart

ing. In this way, CLGGs allow one to build compact and readable definitions of complex visual languages. A parsing algorithm could then behave like the one for RGGs if applied to locally confluent sets of rules, and like the naive parsing algorithm for LGGs with backtracking, otherwise.

We first introduce a running example based on statecharts in Section 2. Section 3 defines the new contextual layered graph transformation approach and gives an example of layered rules reducing correct statechart sentences to a given axiom. Section 4 shows how critical pair analysis can be applied to graph transformation rules and illustrates it on the running example. Section 5 discusses related approaches to parsing and Section 6 draws conclusions.

2: State Charts – A Sample VL

Statecharts - as introduced by Harel in [3] - are an excellent example for testing the expressiveness of visual language grammar formalisms and the efficiency of their accompanying parsing algorithms. Supported by tools such as Statemate or Rhapsody, they are e.g. used for modeling and implementation of (embedded) reactive software systems. Unfortunately, many different dialects of statecharts exist with slightly different syntax and semantics rules. In most cases, these rules are explained by means of examples only, so that it is often a matter of debate whether a certain diagram is syntactically correct and whether it has a precisely defined meaning. Statechart users would profit from a visual language grammar formalism allowing a precise, readable, and compact definition of various statechart dialects.

Within this paper, we use a variant of statecharts, supported by the CASE tool Rhapsody. We do not consider final and history states, due to lack of space, although they are supported by Rhapsody. Moreover, we do not consider complex (Petri net-like) transitions. Figure 1 shows one example of a Rhapsody statechart modelling the behaviour of a blower with `High` and `Low` ventilation mode and three different temperature modes. Its initial state `NotOn` is a so-called *or*-state with two substates `Off` and `Standby`. State `On`, in turn, is a so-called *and*-state with two parallel subdiagrams. The initial states of these parallel subdiagrams are `Low` and `Cool`. The presented statechart contains only legal transitions between states. In the following we assume that transitions of the following kinds are not permitted, since they all break the principle of state abstraction: 1) transitions between states of different subdiagrams which are not nested inside each other (e.g. a transition from `Low` to `Warm` or from `Standby` to `Cool`), and 2) transitions between a state and one of its super states (e.g a transition from `Low` to its super state `On` or from `Standby` to `NotOn`).

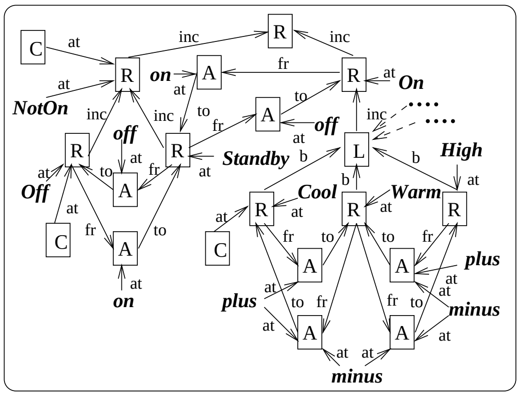
In the following we present a set of graph transformation rules which recognize charts like the one of Figure 1 and reject charts violating the above mentioned restrictions. These rules have to deal with certain local ambiguities of drawn charts. For example, in the diagram of Figure 1, it is not immediately clear whether the left-most transition label `plus` belongs to the transition from `Cool` to `Warm` or to the reverse transition from `Warm` to `Cool`. The same is true for both `minus` labels. The parsing rules resolve these inherent (local) ambiguities to find an interpretation where any transition has at most one label and any label belongs to one transition.

3: The CLGG Approach

In order to use graph grammar technology to define the syntax of a certain statechart dialect then we adopt a graph representation of statecharts based on both, a *low level graph (LLG)* and a *high level graph (HLG)* representation [6].

Example. Figure 2 presents the LLG representation for the Statechart diagram of Figure 1[1]. Its nodes represent the graphical symbols used in Figure 1, its edges encode spatial relationships such as `inc(luded in)` or `b(elow)`

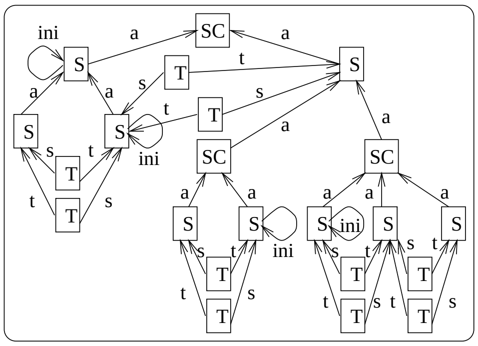
[1]We omitted the nodes and edges for the upper part of the and-state `On` due to lack of space.



Legend: R=Rectangle A=Arrow L=Line D=Double circle

u=upon b=below n=next fr=from inc=included in

Figure 2. Low level graph representation of statechart in Fig. 1



Legend: S=state T=transition SC=state chart
a=abstracts to ini=initial s=source
t=target

Figure 3. High level graph representation of statechart in Fig. 1

between them. There exists e.g. a `R(ectangle)` node with the text label `NotOn` which contains the two `R(ectangle)` elements with labels `Off` and `Standby`, respectively. It is the task of a lexical scanner to translate the internal data structure of a given graphical editor (or a low-level pattern recognizer) into the presented LLG.

The HLG in Figure 3 shows the abstract syntax of our sample statechart. A geometrical element of the LLG is recognized as a `S(tate)` or `T(ransition)`. Subcharts of *and*-states and the overall statechart are recognized as `S(tate)C(hart)`. Transitions have `s(ource)` and `t(arget)` relations to states and states `a(bstract to)` other states or charts.

As proposed in [6], parsing is done in two phases: first the LLG is transformed into an HLG by low-level parsing rules and afterwards the correctness of this graph is investigated by high-level parsing rules. Throughout this contribution, we use a new extension of layered graph rules, so-called *contextual layered graph rules*. They allow the definition of a complex VL such as statecharts in a very convenient way, offering new features to deal with the context of a rule application.

As usual, graph rules consist of a left and a right-hand side $LHS \rightarrow RHS$ over graphs with node and edge types (labels). Parts of both rule sides are related to each others (in the figures indicated by equal numbers at corresponding nodes). The related parts are preserved during a graph transformation. They function as positive application conditions as well as required context to glue newly created graph parts into the context. All non-related graph objects of a left-hand side are deleted, all non-related objects of a right-hand side are newly created.

Moreover, a rule may contain a set of *negative application conditions (NACs)* specifying exactly that fraction of a matching situation that *must not* exist for a rule to be applicable. All graph parts drawn dashed denote some negative application conditions. Throughout this paper all rules contain at most one NAC.

Rule application is performed in two steps: First applicability of a rule is checked by looking for a match m of LHS into a host graph. Then, the rule is applied at one of its matches. A match means an embedding of the whole LHS into the host graph. In general, neither existence nor uniqueness of a match into a host graph is guaranteed. A rule is applicable at a certain match, if all its NAC's and further application conditions are satisfied. In this paper, we use the following dangling condition: nodes cannot be deleted if their elimination would leave dangling edges after rule application. In this way, lots of additional application conditions can be saved.

The positive part of the rule as well as its NACs may contain *complex predicates* on edge labels. Thus, an edge of a rule having such a predicate as label may be matched to a path in the host graph such that the labels in the path satisfy the predicate in the rule. Moreover, *set nodes* may appear in the rule, meaning that they can be mapped to any number of nodes in the host graph, including zero. Set nodes have to be preserved and must not occur in NAC's. They

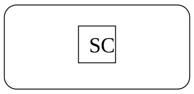


Figure 4. The axiom for a state charts transformation system

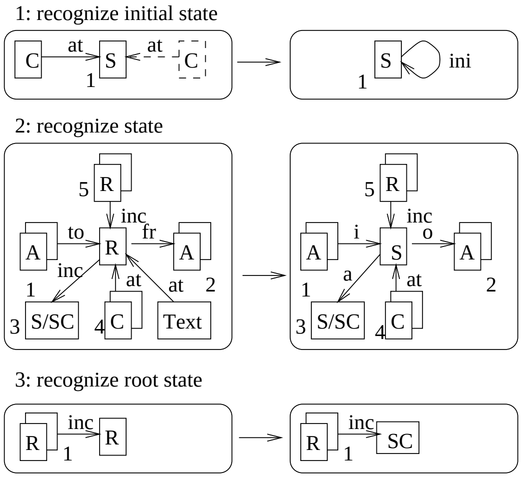


Figure 5. Low level parsing rules 1 - 3

are represented as double rectangles.

For a precise definition of the semantics of graph transformation rules with restricted negative application conditions, transitive closure as edge labels, and embedding set nodes the reader is referred to [7].

Example. We present here the graph transformation system defining the sample fragment of the language of Statecharts. Rules are presented such that a parsing process can apply exactly these rules to analyze an input graph. If the graph is correct, these rules reduce it to the axiom graph of Figure 4. If it is not correct, the parsing process will generate a graph, not isomorphic to the axiom graph, to which no rule can be applied anymore.

Low-level rules transform the low level graph into a high level graph, which is then parsed using the high-level rules. In this example, Rules 1 to 6 in Figures 5 and 6 define low-level parsing and Rules 7 to 14 in Figures 7 and 8 high level parsing. In general, rules allow the removal of elements as soon as they are no longer needed to provide context for other rules.

Rule 1 states that an initial state is one to which a circle is attached and that no two circles may be attached to the same state. Rules 2 and

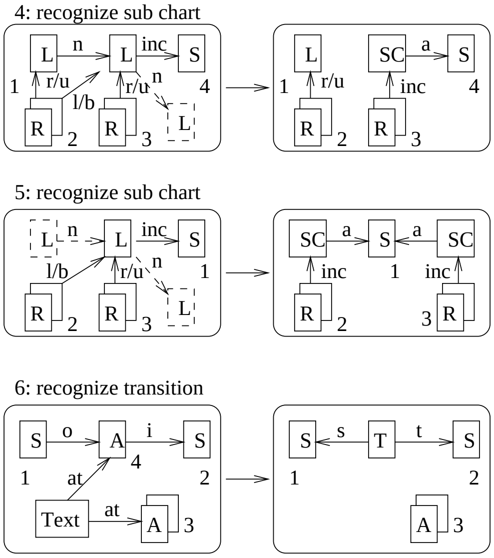


Figure 6. Low level parsing rules 4 - 6

3 together allow the parsing of simple states and *or*-states. Rule 2 recognizes that a rectangle with text inside is the representation of a state. The state is connected to all the elements the rectangle was previously connected, but higher level labels are used to indicate the relation to these elements (the node inscription `S/SC` is used as an abbreviation for two rules, where one uses the inscription `S` and the other one the inscription `SC`). While Rule 2 allows recognition of nested states, Rule 3 has to be used for the outermost Statechart rectangle. Note that, the dangling edge condition prevents conflicts on nested states to be parsed by Rule 2.

Rule 4 and 5 are used in a similar way to recognize subcharts, i.e. *and*-states separated by dashed lines. In these rules `r(ight)` and `u(pon)`, as well as `l(eft)` and `b(elow)` labels denote spatial relations to a dashed `L(ine)`, and the `n(ext)` relation (inducing a linear ordering on the set of lines within a state) is computed accordingly. Each of Rules 4 and 5 is a short hand for a pair of rules using consistent pairs of spatial relations `l/b` and `r/u`. Rule 5 recognizes that the last `L(ine)` in the `n(ext)` ordering defines a subchart, and that rectangles to its right (or upon it) are included in it, (possibly to be parsed as substates). Rule 5 then parses the last `L(ine)` and creates the two subcharts separated by it, maintaining association with rectangles.

Rules in $\{2,3\}$ and $\{4,5\}$ will be used alternately as the *and/or*-structure is parsed.

Rule 6 completes low-level parsing by associ-

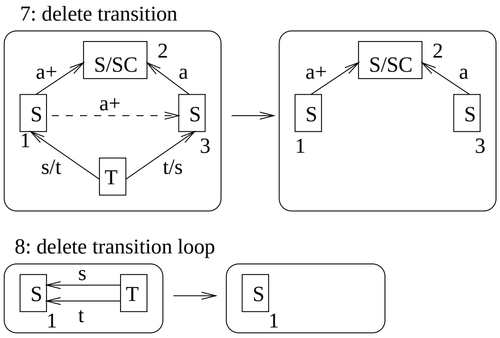


Figure 7. High level parsing rules 7 and 8

ating a T(ext) label with an arrow to identify a transition. Since the at relation is computed by the scanner based on proximity of the text to an arrow, this label could be close to several arrows. This rule makes a commitment which might later reveal wrong. Therefore, it is a source of ambiguity during the parsing process.

At high level, Rules 7 and 8 enforce constraints on transitions in a correct statechart. They use the transitive closure of the edge label a(bstract to) as a complex predicate. Rule 7 parses transitions occurring between State_1 and State_3, which are refinements of the same more a(bstract) State_2 or StateChart_2, provided that State_3 is within an immediate refinement of S_2/SC_2, and that State_1 is not in a refinement of State_3. Rule 8 parses loop transitions in the obvious way.

Rules 9 to 12 sequentially parse states appearing in subcharts (*and*-states) or in *or*-states, leaving the elimination of the ini(tial) state as the last step. When no refining state is any longer present, subcharts can be sequentially parsed by Rules 13 and 14.

Given the set L of all node and edge labels and the set R of all rules we introduce the following *layering functions*:

- $cl, dl : L \rightarrow Nat$ are total functions which assign each label a unique creation and a unique deletion layer such that $\forall l \in L : 0 \leq cl(l) \leq dl(l) \leq n$.
- $rl : R \rightarrow Nat$ is a total function which assigns each rule a unique layer.

A graph transformation system with a set R of rules over a set L of node and edge labels with layering functions as above is a *layered graph transformation system* if and only if all rules $r \in R$ with $rl(r) = k$ fulfill the following con-

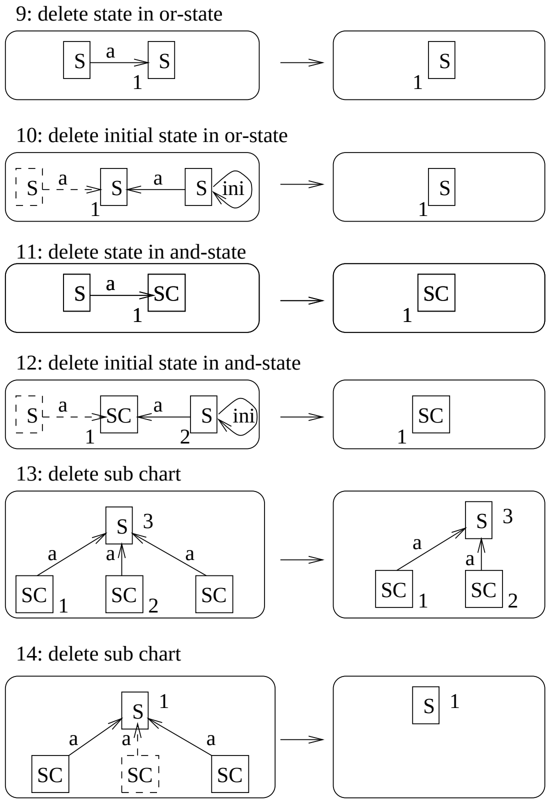


Figure 8. High level parsing rules 9 - 14

straints:

- r deletes at least one node or edge.
- r deletes only nodes and edges with labels l such that $dl(l) \leq k$.
- r creates only nodes and edges with labels l such that $cl(l) > k$.
- r uses only complex positive predicates (such as transitive closure) over nodes and edges with labels l such that $dl(l) > k$.
- r uses only complex negative predicates (such as negation of transitive closure) over nodes and edges with labels l such that $cl(l) \leq k$.

Note that edges going from or to set nodes have to fulfill the layering condition in the same way as simple edges.

The definitions above introduce a graph transformation system which reduces (parse) graphs to a given start graph (axiom). The reduction process first has to apply rules of layer 0 as long as possible, then rules of layer 1, etc. A *contextual layered graph grammar (CLGG)* which generates exactly the language recognized by a

layer	creation types	deletion types	rules
0	Text, L, R, D, C, A, to, fr, at, l, r, u, d, n, inc	Text, L, R, D, C, to, fr, at, l, r, u, d, n, inc	1 - 5
1	S, SC, i, o, a, ini	A, i, o	6
2	T, s, t	T, s, t	7, 8
3		S, SC, a, ini	9 - 14

Figure 9. Sample label and rule layers

given layered graph transformation system consists of the same set of rules with negative application conditions transfered from LHS to RHS and exchanged left- and right-hand sides. This can be shown by slightly generalizing the proof for simple layered graph transformation rules in [6]. The definition of a CLGG may be obtained by replacing any occurrence of "creates" or "cl" by "deletes" or "dl" in the definitions above and vice versa.

The parsing rules and all node and edge labels for statecharts have to be divided into four different layers to satisfy the layering conditions. Compare Figure 9 for the definition of the layering functions for our running example. Labels created by the scanning process are assumed to have $cl(l) = 0$.

The layer-wise application of rules to a given terminal graph (where each node and edge label l belongs to the creation layer 0, i.e. $cl(l) = 0$) always terminates. Indeed, rule r with $rl(r) = k$ deletes at least one element with label l_1 with $dl(l_1) \leq k$ and may create elements with label l_2 with $k < cl(l_2) \leq dl(l_2)$ only. Therefore, rules of layer k remove elements with label l with $dl(l) \leq k$ and create elements with label l with $dl(l) > k$. As a consequence, the exhaustive rule application process always terminates (either by reduction to the axiom or because no rule is applicable).

4: Critical Pair Analysis

Critical pair analysis is known from term rewriting, where it is used to check if a term rewriting system has a functional behaviour, i.e. is confluent. Critical pair analysis has been generalized to graph rewriting in [5, 4]. It can be advantageously used to make parsing by graph transformation more efficient in the sense that decisions between conflicting rule applications are delayed as far as possible. This means to apply non-conflicting rules first and reduce the graph as much as possible before starting with rule applications which are in conflict with others. At each conflict, a decision point is created which can cause backtracking in the parsing algorithm.

For critical pair analysis of contextual layered graph rules, a layer-wise analysis is sufficient, since a rule of an upper layer is not applied if rules of lower layers are still applicable. Furthermore, it is sufficient to investigate just skeletons of contextual layered graph rules. Two kinds of skeletons are defined: $skel_{NAC}(r)$ replaces all set nodes in a rule r by ordinary nodes with the same label and to eliminate all positive edges with complex predicates. $skel(r)$ is the skeleton rule $skel_{NAC}(r)$ without any NAC.

The critical pair analysis of two contextual layered graph rules $r1$ and $r2$ can be reduced to that of $skel_{NAC}(r1)$ and $skel_{NAC}(r2)$, since embedding does not cause new dependencies. If there are critical pairs where set nodes are involved, there is a critical pair where the corresponding simple nodes are involved in the same way. Matching a set node zero times there is no dependency, matching it exactly once does not differ from the skeleton rule. Matching it more than once this match can be covered by several different matches of the skeleton rule. Thus, not a real new conflict can be found, but only variants considering the skeleton rule. There are no further inferences with negative application conditions, since those rely exclusively on lower layered nodes and edges. Eliminating the edges with complex predicates in the skeleton rules is possible, since all the rules in the current layer do not delete nodes or edges of an upper layer. Thus, edges with this label cannot cause critical pairs.

Moreover, critical pair analysis for skeleton rules with NACs can be mainly restricted to that for skeleton rules without conditions. The applicability of a rule with negative application conditions is not in conflict with the application of a rule of the same layer, since the rule does not create nodes or edges of a lower creation layers and the condition is checked on edges of upper creation layers only.

Thus, the set of critical pairs can be found as follows:

- Perform critical pair analysis for each layer separately.

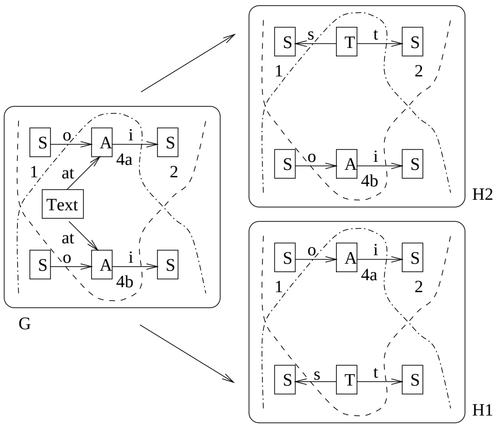


Figure 10. Two conflicting transformations

- For any two rules $r1$ and $r2$ of one layer: Consider their skeletons $skel(r1)$ and $skel(r2)$ and look for all overlapping graphs G of their left-hand sides.
- For each overlapping graph G: Test if the matches $m1$ and $m2$ of the skeleton rules $skel(r1)$ and $skel(r2)$ can be extended to matches of the rules $skel_{NAC}(r1)$ and $skel_{NAC}(r2)$. If yes, proceed with graph G, otherwise skip this graph, since it cannot cause a critical pair.
- Given a graph G, $skel(r1)$ applied at $m1$ leads to graph $H1$ while $skel(r2)$ applied at $m2$ leads to graph $H2$. $(skel(r1), m1)$ excludes $(skel(r2), m2)$ if match $m2$ cannot be completely transfered to $H1$.

Example. Our example has very few critical pairs: rule 6 is in conflict with itself, there are three critical overlappings: Nodes 3 and 4, Nodes 4 and 4, and Nodes 4 and 4 together with the overlappings of Nodes 3 and 3. Two conflicting applications of Rule 6 are shown in Figure 10 where Nodes 3 and 4 overlap mutually. This is a real critical pair in the sense that it is not possible to complete the transformation such that they are confluent. This is true also for the other critical pairs dealing with Rule 6. Furthermore, there are critical pairs if Rule 13 overlaps with itself in Nodes 1 and 4 or in Nodes 2 and 4. These critical pairs are avoidable. All conflicting transformations are confluent in the end meaning that the order of deletion is irrelevant for the subcharts of *and*-states.

5: Related Parsing Approaches

Layered Graph Grammars (LGGs) were proposed in [6] as a way to express contextual conditions, needed for the definition of several visual languages of practical use. In [6] an extensive comparison of different VL definition and parsing approaches, not reported here, is also given. The layering condition of LGGs guarantees decidability of the membership problem through the imposition of a lexicographical ordering on rules, preventing the creation of cycles during the parsing process. In the same paper, two different exponential time parsing algorithms were proposed. One algorithm combines a bottom-up parsing with a top-down parsing phase, while the other simply applies grammar rules with exchanged left- and right-hand sides. The latter one shows its exponential worst-case behavior especially if applied to incorrect input graphs, while the first one requires very complex book-keeping algorithms without being able to avoid the worst-case behavior of the one-phase algorithm completely.

In order to reduce parsing complexity, Zhang and Zhang proposed a restriction on LGGs, defining Reserved GGs [10], on which they can check a condition of local confluence. If the condition is satisfied, the naive LGG parsing algorithm can be used with no need for backtracking. However, they do not address the problem of parsing RGGs not satisfying the requirement for local confluence.

The advantage of our proposal, even if it does not reduce the implicit complexity of the parsing problem, is to reduce the need for backtracking, via identification of critical pairs, as distinguished from conflicts which do not affect the solution of the membership problem.

Static analysis of rules for context-sensitive graph grammars is also performed in the approach by Adachi *et al.* [1], which however uses a different form of embedding, namely neighbourhood embedding [7], which allows retyping only of nodes already connected to the elements substituted by the rule. In our approach instead, new edges can be created also towards previously unconnected nodes. Such a property has been argued to be necessary for example, for the definition of entity relationships diagrams [6]. Moreover, they do not allow complex and negative application conditions.

Critical pair analysis has a long tradition in term rewriting, and has also been applied to algebraic approaches to graph rewriting [5, 4], but without considering the extensions introduced in this paper.

6: Conclusion

We have presented an approach to the problem of formal definition of VLs which achieves a reasonable trade-off between expressiveness (and readability) of the grammar definition formalism and efficient parsing of visual sentences, by the use of several constructs expressing complex constraints in a synthetic way. The power of the approach has been illustrated by presenting a complete grammar for a significant fragment of statecharts. The proposed complex constructs do not cause the creation of backtracking points during the parsing process, if simplified rules obtained from them do not require so.

Critical pair analysis allows potentially conflicting rules to be identified, delaying their use in the parsing process as far as possible. While the worst-case complexity of the parsing process is not changed, its execution on actual graphs can thus be simplified. In particular, this eliminates the need for a two-phase process in parsing based on layered graph grammars. Note that achieving the effects offered by embedding or transitive closure with simpler formalisms would require an increase in the number of rules and the generation of new critical pairs, with new potential causes for backtracking during parsing. Parsing can then be realized by inverting left- and right-hand sides of grammar rules and applying these *parsing* rules according to the layering constraint. Such an algorithm has been integrated in the AGG system [9]. A similar extension of the PROGRES graph transformation system [8] is straightforward due to the fact that it does already support the execution of a superset of the presented graph transformation rules including all needed means for backtracking.

An interesting extension of this work is a more refined classification of the sources of conflicts through critical pair analysis. A considerable reduction of the set of critical pairs seems to be possible through the use of schema graphs with multiplicity constraints as introduced in [7]. It imposes overall constraints on the sentences of the visual language. Using these additional constraints we are able to identify rules with matching conflicts for incorrect graphs only. Due to the fact that parsing fails for these incorrect graphs anyway, we are able to eliminate exactly those backtracking points of the naive LGG parsing algorithm which cause its worst-case behavior if applied to incorrect input graphs.

References

[1] Y. Adachi, S. Kobayashi, K. Tsuchida, and T. Yaku. An NCE context-sensitive graph grammar for visual design languages. In *Proc. IEEE Symposium on Visual Languages (VL'99)*, 228 – 235, 1999.

[2] H. Ehrig, G. Engels, J.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools.* World Scientific, 1999.

[3] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[4] M. Löwe and J. Müller. Critical Pair Analysis in Single-Pushout Graph Rewriting. In G. Valiente Feruglio and F. Rosello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science.* Technical Report B-19, Universitat de les Illes Balears, 1995.

[5] Detlef Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M.R Sleep, M.J. Plasmeijer, and M. C.J.D. van Eekelen, editors, *Term Graph Rewriting*, 201–214. Wiley, 1993.

[6] Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *JVLC*, 8(1):27–55, 1997.

[7] A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations.* World Scientific, 1997, 479–546.

[8] A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In *[2]*, 1999, 487–550.

[9] G. Taentzer, C. Ermel, and M. Rudolf. The AGG-approach: Language and tool environment. In *[2]*, 1999, 551–604.

[10] D.-Q. Zhang and K. Zhang. Reserved graph grammar: A specification tool for diagrammatic vpls. In *Proc. IEEE Symposium on Visual Languages (VL'97)*, Los Alamitos, CA, IEEE Computer Society Press, 288–295, 1997.