Electronic Communications of the EASST
Volume 10 (2008)

Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)

Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars

Florian Brieler and Mark Minas

14 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars

Florian Brieler[†] **and Mark Minas**[†]

[†] Institute for Software Technology
Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
{florian.brieler|mark.minas}@unibw.de

Abstract: Sketching, i.e., drawing diagrams by hand and directly on the screen, is gaining popularity, as it is a comfortable and natural way to create and edit diagrams. Hand drawing is inherently imprecise, and often sloppy. As a consequence, when processing hand drawn diagrams with a computer, ambiguities arise: it is not always clear what part of the drawing is meant to represent what component. Resolution of these ambiguities is the main issue of sketching. Ambiguity can only be solved by exploring the context of ambiguous components. This paper describes ambiguity resolution by syntax analysis in DIAGEN, a generic framework for generating diagram editors. Such editors support free-hand editing (which is closely related to sketching), and allow for analyzing the created diagrams based on a hypergraph grammar. Our approach adds support for sketching to the generated editors. In order to resolve the ambiguities in sketched diagrams, DIAGEN's diagram analysis based on graph parsing is used. The necessary modifications to DIAGEN and its graph parser in particular are discussed.

Keywords: Sketching, Ambiguity Resolution, Hypergraph, Parser

1 Introduction

Nowadays diagram languages like the UML are very popular, and there is a lot of work going on to model applications and systems (or, at least, part of them) using diagram languages, instead of coding them traditionally with textual languages. Diagrams are more expressive in terms of exposing structure and coherence of the modeled system; the used diagram language can be domain-dependent, thus better focusing on the problem in question; and diagrams are – for some problems – much more suited, e.g., for expressing graph-like structures like Petri nets or class diagrams.

Tool support for processing of diagrams has evolved in the last years, with many approaches available, e.g., *Fujaba* [FNTZ00], *AToM*^{3} [LV02], DIAMETA [Min06] and DIAGEN [Min02]. Among these, the specification of syntax and semantics of a diagram language is either given by metamodels or by graph grammars.

However, creating and editing of diagrams using such tools is not very natural. Diagram components have to be selected from some graphical widget like a list, and placed on the canvas

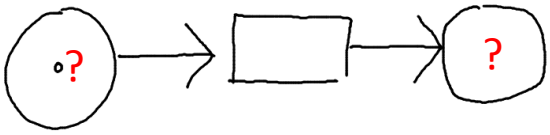

Figure 1: A simple Petri net with two ambiguities, indicated by question marks.

by one or more mouse clicks. Instead, sketching, i.e., drawing diagrams by hand on the screen, is similar to pen and paper and is much more natural, because it does not require complex user interfaces.

This paper is concerned with ambiguity resolution, which is the main issue of sketching. Ambiguities arise because drawing by hand is inherently imprecise and sloppy. Fig. 1 shows a practical example of a simple Petri net, which is also used as running example in this paper. Places and tokens are drawn as circles, transitions are drawn as rectangles. The two question marks indicate ambiguities. On the left it is unclear for the two circles whether they represent places or tokens. The component on the right could be a circle or a very deformed rectangle; possible interpretations are place, token, or transition, resp. By looking at the context of the ambiguous components, it becomes obvious that we have a place containing a token on the left (a place containing a place is not meaningful, and so is a token containing another component), and another place on the right (as the arrow connects a transition and a place, but not a transition and a token or another transition). Apparently, context of an ambiguous component has to be exploited to decide for the correct interpretation. This requires diagram analysis.

In [BM08] we present a comprehensive approach to sketching enabled diagram editors, including user interface, recognition of components in the drawing, support for text, and basic ideas for diagram analysis. There are many approaches to sketching, but most of them do not exploit the power of grammar-based approaches for ambiguity resolution. For diagram analysis we have decided for DIAGEN,

- because it supports *free-hand editing*, which is the basis of sketching,
- because it uses graph grammars for its visual language parser, which are very powerful for ambiguity resolution (discussed in this paper),
- and because DIAGEN is generic (it can be customized to any diagram language by a specification of the language). The approach in [BM08] is designed to be generic as well.

In the present paper we describe in detail how we employ DIAGEN for diagram analysis in order to resolve ambiguities. The main idea is that for each component it must be decided which of its possible interpretations fits best to the other components.

This paper is organized as follows. Sec. 2 explains DIAGEN by the example of Petri nets, and outlines the basic idea to support sketching. Sec. 3 and Sec. 4 describe the necessary modifications to DIAGEN. Sec. 5 discusses related work. Sec. 6 gives a brief summary and describes further work.

2 Hypergraph Grammars and Parsing in DIAGEN

A *diagram* is a set of diagram *components*. Each component has one or more *attachment areas*, i.e., areas where the component can be related to other components. Relationships between attachment areas depend on spatial placement. A relation is detected if two attachment areas overlap or are close to each other (since sketching is imprecise, it is not meaningful to require precise spatial placement of components). For example, places and transitions in Petri nets have one attachment area each (their full shape). Arrows have two attachment areas (their head and their tail), and can be related to places and transitions if its head or tail is close. There may be relations which are not required for a diagram type, e.g., overlapping arrow heads in Petri nets.

DIAGEN [Min02] is a generic editor generator that generates diagram editors from language-dependent specifications. Each specification describes one diagram language, and defines aspects like diagram components and attachments areas, desired relationships, reduction rules, grammar rules, and attributes for parsing (see below). Hypergraphs are used as internal models required for diagram processing. Each component is represented as a single hyperedge visiting as many unique nodes as the component has attachment areas. If a component visits more than one node, its tentacles (connecting an edge with its visited nodes) are numbered in order to be identifiable. Hyperedges are labeled; the label depends on the type of the respective diagram component. For Petri nets, we have four different types of components: places, transitions, arrows and tokens. Hence, hyperedges are labeled with `c_place`, `c_trans`, `c_token`, or `c_arrow`, resp. We call such hyperedges *component edges* in the following. Additional information about a component is stored in attributes of the representing component edge, for example, the position and radius of a place. Relationships between diagram components are binary hyperedges visiting the two nodes representing the related attachment areas. For Petri nets, we have a relationship that relates an arrow head or tail to a transition (`at_trans`), a relationship that relates an arrow head or tail to a place (`at_place`), a relationship that relates overlapping places or transitions (`touch`), and a relationship that relates a token to the place it is contained in (`inside`). Hyperedges representing relationships are called *relation edges*.

The overall system architecture of a diagram editor generated by DIAGEN is shown in Fig. 2. Rounded boxes depict data structures, rectangles depict processing units. The figure shows an editor without sketching support, called *regular editor* in the following. The *layouter* and the *transformer* are not relevant for this paper. Also, we neglect *attribute evaluation* as the final step of processing a diagram.

The *drawing tool* provides the user with a GUI, it is the actual diagram editor. As mentioned before, a hyperedge is created for each component placed on the canvas by the user. We change this behavior for sketching and create a hyperedge for each component that can be recognized in the hand-drawn diagram. Therefore we have replaced the original DIAGEN editor by another editor that allows for drawing by hand. The process of *recognition* of components from the hand drawing is described in [BM08]. Result of the recognition is a set of components. *How* these components were drawn is neither relevant nor visible to the approach shown here, but completely handled by the recognition process.

In the next step of the processing chain, the *modeler* identifies relationships between components and creates respective hyperedges. No user input or user interaction is required for this process. Relations cannot be restricted, e.g., by a condition, but depend solely on the spatial

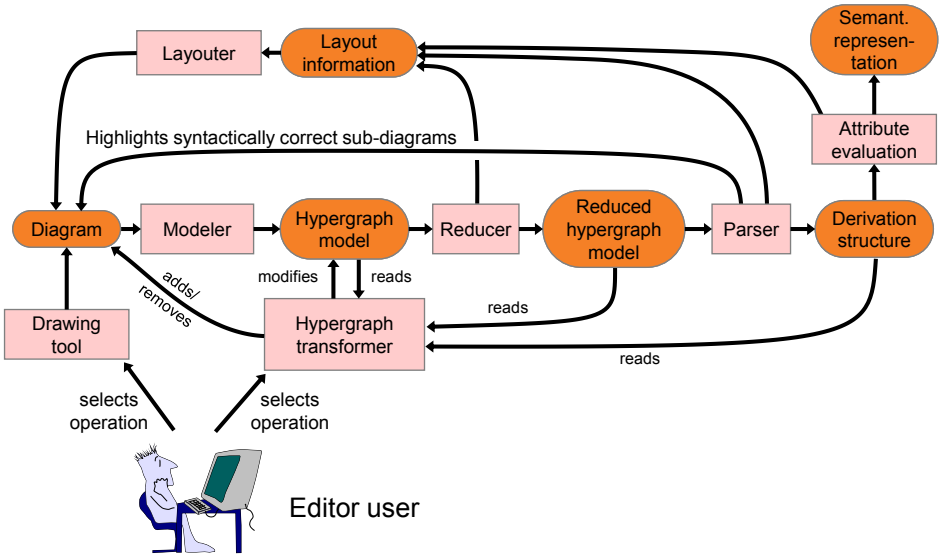

Figure 2: Architecture of a diagram editor generated with DIAGEN.

placement of the components. The result of the modeler is a hypergraph (the *hypergraph model*, or HM) containing all component edges and all respective relation edges. In case of ambiguity, a component edge is created for each possible interpretation of the ambiguous component. Such component edges are independent from each other, although they were recognized from the same strokes in the hand drawing. No information is stored that these edges actually represent the same component. For Petri nets, it is clear that only one of these edges can be valid at the same time, but for other diagram languages the situation may be different. The mechanisms of the subsequent reducer and parser are employed accordingly to account for such component edges.

The HM for the Petri net shown in Fig. 1 is depicted in Fig. 3. Relation edges are shown as arrows. As `touch` is a symmetric relation, each of the respective arrows has two arrow heads. The two ambiguities identified in the drawing are highlighted in gray: for each of the two circles on the left in Fig. 1, two component edges are created (`c_place` and `c_token`). For the single component on the right, three component edges are created. The only non-ambiguous components are the two arrows and the transition in the center. Because all component edges are independent of each other, each of the three `c_token`-edges in Fig. 3 is also related to that `c_place`-edge which represents the same circle in the drawing (the two vertically displayed `inside`-relations on the left, and the `inside`-relation on the right).

Even the `c_token`-edge representing the large circle is identified to be *inside* the `c_place`-edge representing the small circle, because their attachment areas overlap, hence an `inside` relationship can be found (the name `inside` is misleading in this case). Furthermore, all identified places and tokens in in this example overlap and have their full shapes as attachment areas, so the distance of the respective attachment areas is always 0. The same is true for the `touch`-relation between the `c_place` and the `c_trans` on the right.

Both components and relationships are rated by a positive real number. For components, the

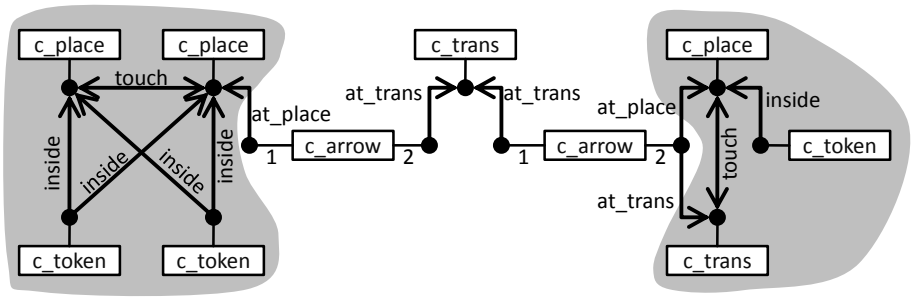

Figure 3: Hypergraph model of the Petri net shown in Fig. 1. Ambiguities are highlighted.

rating depends on the complexity of the component, and on the precision of how it has been drawn. Components being more complex, or being drawn more precisely, gain a higher rating. For relationships, the rating depends on the distance of the two respective attachment areas. A smaller distance means a higher rating. Each component edge and relation edge contains the rating of the represented component or relation in an attribute. The rating will be used by the parser.

Next, the *reducer* applies a set of reduction rules, i.e., graph transformation rules, to the HM. Such rules consist of an LHS and an RHS, each are hypergraphs. The result of the reducer is the *reduced hypergraph model* (RHM). In the first place, the RHM is newly created and therefore empty. Then, for each match of the LHS of a reduction rule in the HM, a respective match for the RHS of the matched rule is added to the RHM. The HM is not changed by the reducer.

The reducer serves two tasks: first, the RHM usually contains less hyperedges than the HM; as the RHM is the input for the parser, a smaller model containing less edges improves processing time. Second, invalid configurations which may occur in the HM due to misplaced components are not transformed, i.e., they do not occur in the RHM. Application of reduction rules can be restricted by conditions, and by negative application conditions (NACs).

The reduction rules for Petri nets are shown in Fig. 4. Corresponding nodes on the LHS and the RHS are labeled with the same letter. NACs are highlighted in gray and crossed out. The two upper rules transform places and transitions, as long as these do not overlap with any other place or transition. The third rule transforms tokens inside places, ignoring tokens not inside places. Here, a condition is applied: a token is only reduced if its radius is smaller than half the radius of the containing place. The last two rules transform arrows between places and transitions. As arrows have two attachment areas (head and tail), the attachment areas are distinguished by numbers. Note that there are no reduction rules for arrows between two places or between two transitions, as Petri nets do not allow such arrows.

Fig. 5 shows the RHM created by the modified reducer. The original reducer for regular editors would not produce the subgraphs highlighted in gray, due to the NACs. Except for the transition in the center, no transition or place would be reduced, as each of them touches another place or transition because of ambiguous interpretation of the corresponding components. Of the three tokens, only one would be reduced (the one that actually *is* a token), the other two would not, due to the condition in the third rule.

Obviously, ambiguity cannot be resolved for those components not reduced. In the depicted

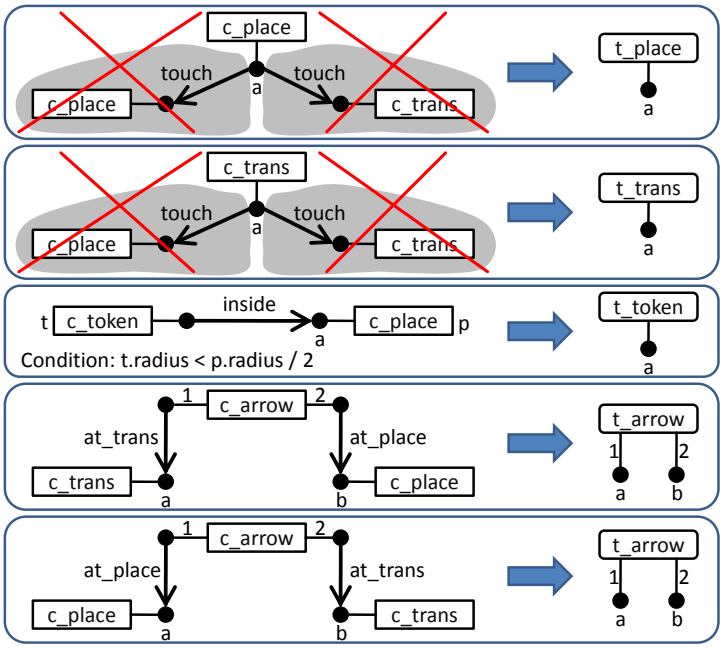

Figure 4: Reduction rules for the language of Petri nets.

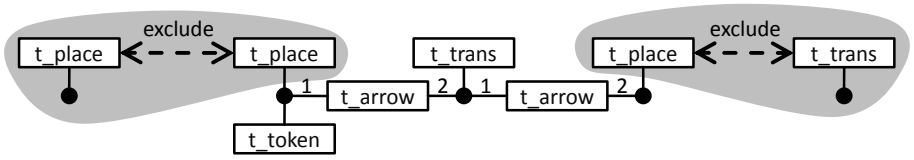

Figure 5: Reduced hypergraph model of the hypergraph model shown in Fig. 3.

case, the original reducer is too strict and discards almost all ambiguous components. We rather have to postpone ambiguity resolution to the parser, i.e., the RHM must contain complete information, even on the ambiguous components. Therefore, the highlighted subgraphs are added to the RHM in Fig. 5, together with the information about which components *exclude* each other. This information is based on the NACs. Note that the conditions (like the one for tokens) are not affected, and have to be met anyway.

Finally, the *parser* uses the hyperedges from the RHM as terminal edges and attempts to deduce the start symbol in a bottom up fashion. The production rules for Petri nets are shown in Fig. 6. Terminals from the RHM are depicted as rounded rectangles with a white background, while nonterminals have a gray background. There are two types of production rules unique in DIAGEN. The one are *set productions* and the other are *embedding productions*.

Set productions are used to *collect* all edges with the same label, not regarding any order or any subset, thus improving performance of the parsing process. In Fig. 6, set productions are depicted with a stack of edges on the RHS (the two productions with `Transitions` and `Places` on the LHS, and a stack of `Trans` and `Place` on the RHS). An alternative specification would

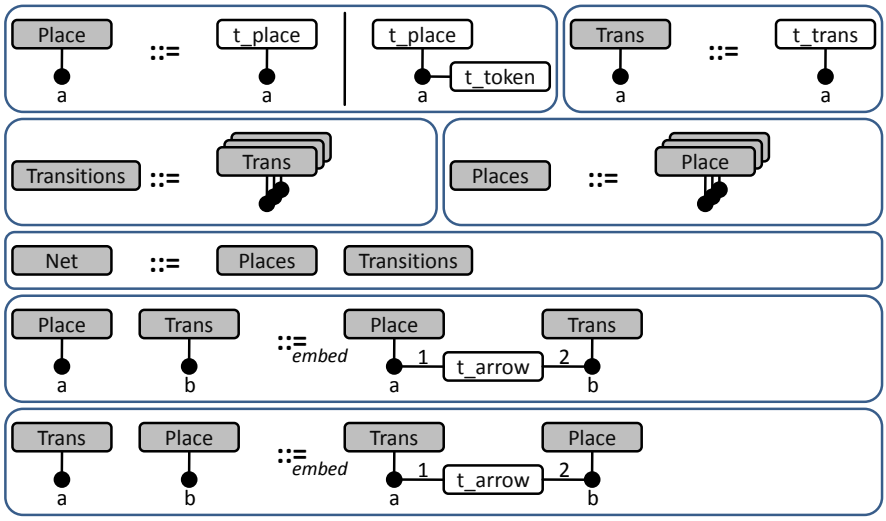

Figure 6: Production rules for the language of Petri nets.

use recursive productions like `Places ::= Place | Places Place .` However, that would require the parser to deduce any possible subset of all places, and in any possible order, which leads to a combinatorial explosion. The two set productions for Petri nets express that `Transitions` is a nonterminal edge representing the non-empty set of all `Trans` edges, and `Places` is a nonterminal edge representing the non-empty set of all `Place` edges. The start symbol for Petri nets is `Net`. The shown grammar therefore accepts a diagram as a correct Petri net if it contains at least one place and one transition (a modified grammar could also accept Petri nets without places or transitions).

Embedding productions are the other kind of production rule unique to DIAGEN. For efficiency reasons, the used graph grammar is context-free. Embedding productions are used to embed (nonterminal or terminal) edges into a context which has been derived by context-free productions. An example are arrows in Petri nets. With embedding productions they can be added to a derivation tree, resulting in a *direct acyclic graph* (DAG) as derivation structure. In the following we will always use the term *DAG*, not distinguishing between a DAG and a tree. Embedding productions consist of the same graphs on the LHS and the RHS, but with one additional edge on the RHS (the edge that is embedded). The two productions at the bottom of Fig. 6 are the two embedding productions required for Petri nets.

The parser must use the exclusion information provided by the reducer in order to deduce only those start symbols which do not contain terminal edges excluding each other in their DAG. This is described in Sec. 4, but first, it is explained how the reducer can use the NACs to collect the necessary information for the exclusion of components.

Note that the modifications to the reducer and the parser do not require the reduction rules or production rules to be modified. They are left unchanged, but are applied differently, as it is described next. Also, the architecture shown in Fig. 2 is preserved. The information necessary for ambiguity resolution is stored in additional attributes of the terminal and nonterminal edges.

3 Sketching-related Modifications to the Reducer

We require the reducer to apply the reduction rules even if they are prohibited by matched NACs, and create an exclusion relation *exclude* for the hyperedges in the RHM, i.e., the terminal edges for the subsequent parsing step. The simple example of Petri nets discussed in the previous section only shows terminal edges which mutually exclude each other. The general case is more sophisticated, as shown in the following. Such complicated situations occur when a NAC consists of more than only a single component hyperedge.

In the following we call hyperedges simply *edges*, and edges from the RHM *terminal edges* or simply *terminals*. A *match* of some pattern graph P in a host graph H is an occurrence of P in H. We regard a (sub)graph as set of edges. For a terminal t, where t is in the occurrence of the RHS of some reduction rule r, we denote by $model(t)$ the corresponding occurrence of the LHS of r, and by $nacs(t)$ a set of all matches for NACs that would have prohibited the creation of t. In the following, we omit all relation edges in $nacs(t)$ and keep only the component edges, as relation edges completely depend on component edges. We can then define a relation *exclude* between a set of terminals T and a single terminal t:

$$T \text{ exclude } t :\Leftrightarrow \exists N \in nacs(t) : N \subseteq \bigcup_{t' \in T} model(t')$$

T excludes t if the union of all $model(t'), t' \in T$ contains all edges from a NAC N in the NACs from t, $N \in nac(t)$. For example, let m_p be a component edge representing a place, and m_t be a component edge representing a transition, and both components overlap, i.e., they exclude each other (apart from the token, this is the case for the right of Fig. 3). Then, the reducer creates two terminals, t_p and t_t, with

$$model(t_p) = \{m_p\}, nacs(t_p) = \{N_1\}, N_1 = \{m_t\}$$
$$model(t_t) = \{m_t\}, nacs(t_t) = \{N_2\}, N_2 = \{m_p\}$$

Now the set with the single terminal t_t excludes t_p, because $model(t_t)$ contains all edges in N_1, which is a NAC in $nacs(t_p)$. By analogy, $\{t_p\}$ excludes t_t. If we had included the relation edges in $nacs(t_p)$ and $nacs(t_t)$, both exclusions would not hold, as the relation edges do not occur in $model(t_t)$ and $model(t_p)$.

A slight modification of this example shows that the *exclude* relation is not symmetric in general. We change the reduction rule for places, so that a place has no NACs, i.e., it can overlap with any other component (the rule for transitions is not changed). We get

$$model(t_p) = \{m_p\}, nacs(t_p) = \varnothing$$
$$model(t_t) = \{m_t\}, nacs(t_t) = \{N_2\}, N_2 = \{m_p\}$$

Here, $\{t_t\}$ still excludes t_p, but not the other way around. In the following we assume the original reduction rule for places as shown in Fig. 4.

For the general case, a NAC contains more than one component edge. Let N be a match of a NAC. By applying several reduction rules, different terminals can be reduced from the edges in N. Then, not only one, but more terminals are required to exclude a single terminal.

Based on the ratings for component edges and relation edges, a rating can be computed for a terminal t by adding up all ratings from all edges in $model(t)$. The next section describes how the parser exploits the *exclude* relation when deducing the start symbol, and how ratings are used.

4 Sketching-related Modifications to the Parser

The central data structure for the parser is the derivation DAG. Each DAG has a unique *root*: it is the node which has no incoming edges. *Leaves* of a DAG have no outgoing edges. All leaves represent terminals, all other nodes represent nonterminals. Unless embedding productions are used, each node has a unique *parent*, except for the root. Each parent node is the LHS match of a production rule, and its children are the respective match for the RHS of that rule. Nodes representing embedded edges can have more than one parent; all parents of such a node represent the context of the respective embedded edge.

The general idea for the parser is to avoid deduction of nonterminals from a set of terminals T where some terminal in T is excluded by other terminals in T. For this purpose we will define a symmetric relation *conflict*; if two nonterminals *conflict* with each other, they must not occur in the same derivation DAG.

Let nt be a nonterminal. By $term(nt)$ we denote the set of all terminals used to deduce nt, i.e., the set of all leaves in the DAG with nt as root[1]. DIAGEN applies a production rule (production, for short) if three conditions hold: (i) a match M for the RHS must be found, (ii) for all nonterminals nt in M all $term(nt)$ must be pairwise disjoint, and (iii) the condition defined for the production must hold. We leave (i)-(iii) unchanged, but add a fourth condition which regards the *exclude* relation created by the reducer. We will see in the following that this fourth condition depends on the type of the production. A *Chomsky Normal Form* can be computed for each hypergraph grammar (apart from the set productions and the embedding productions). Consequently, four different types of production rules have to be distinguished:

- *terminal productions* with exactly one terminal on the RHS.
- *nonterminal productions* with exactly two nonterminals on the RHS.
- *embedding productions* with nonterminals on both sides.
- *set productions* with an arbitrary number of nonterminals on the RHS.

Terminal productions may always be applied, no conflicts may arise here. For *nonterminal productions* we first consider a set T of terminals. We call T *conflicting* if there exists a subset $E \subset T$ and a single terminal $t \in T \setminus E$ where E excludes t. Then, two nonterminals nt_1 and nt_2 may be used on the RHS of a production rule if the union of their terminals, $term(nt_1) \cup term(nt_2)$, is not conflicting. This also implies that both $term(nt_1)$ and $term(nt_2)$ are not conflicting. We define the symmetric relation *conflict* between two nonterminals. $(nt_1, nt_2) \in conflict$ if and only if $term(nt_1) \cup term(nt_2)$ is conflicting. Then, nt_1 and nt_2 can be used on the RHS of a nonterminal production if they are not *conflicting*, i.e., $(nt_1, nt_2) \notin conflict$.

[1] $term(t)$ is only relevant during construction of the derivation *tree*, so embedded edges are never in $term(t)$.

Embedding productions are treated differently than the other types of productions. First, the DIAGEN parser identifies each match for the context of each embedding production. This is only possible if no edges of a match conflict with each other. Then, for each derivation DAG with the start symbol as root, each of the previously identified matches is checked whether all of its edges are contained in the DAG, i.e., are (direct or indirect) children of the root. Finally, for all such matches, the additional edge of the RHS of the production is embedded, i.e. added to the DAG. The edges of a match can only be contained in the DAG if none of these edges conflicts with any other edge from the DAG. Therefore, the only condition that must be checked before an edge may be embedded is whether it conflicts with the root of the DAG. If there is no conflict, there can also be no conflict with any other edge in the DAG.

The fourth type of production rule are the *set productions*. Basically, set productions can be seen as nonterminal productions with not exactly two, but one or more edges in the match of its RHS. However, the production may be applied even if nonterminals in this match are conflicting. In this case, the problem is to decide which of the conflicting edges should be omitted, because this decision may have consequences on subsequent applications of production rules and contexts for embedding productions. The former happens if the nonterminal that has not been omitted conflicts with another nonterminal in a subsequent production rule. The latter happens if the omitted nonterminal was part of a match for a context of an embedding production. Consequently, we must defer the decision, as we cannot make it when applying a set production. We temporarily ignore all conflicts, and do not omit any of the nonterminals on the RHS match. When the start symbol is reached, there can be no further productions, and we can finally decide which nonterminals to omit.

The nonterminal matching the LHS of a set production may be used in the match of a RHS of another set production itself, either directly, or as a node contained in the DAG of a nonterminal in this match. This can lead to a complex structure. An example is depicted in Fig. 7, where the top part of a derivation DAG can be seen. As before, nonterminals are depicted as rounded rectangles with a gray background. The thin arrows indicate parent-child relationship. The subtrees of nonterminals with two outgoing arrows not ending in other nonterminals are of no interest in this example. Embedding productions are not shown. A, B and C are nodes in the DAG indicating applications of set productions. The fat arrows, marked with crosses, depict conflicts between nonterminals.

When applying the nonterminal production $a_3 \rightarrow BF$ we can immediately discard b_3, as it conflicts with F. The production cannot be applied otherwise. The same is true for production $D \rightarrow EC$ and c_3. The following cases are more difficult. When applying set production $A \rightarrow a_1a_2a_3a_4$, we cannot decide for a_2 or for b_1, as we do not know about possible later consequences. There is a conflict between a_2 and c_1. The problem is that we do not know yet that A and C will be used in the same DAG. Finally, when applying $S \rightarrow AD$, we can decide for a_2 or b_1, for a_2 or c_1, for c_1 or c_2, and for a_4 or c_1, as we know that there will be no further productions.

The problem of omitting nonterminals is NP-complete. It is a slight variation of the *maximum clique problem* [Kar72]. However, we do not need the best solution; a heuristic is sufficient. In order to guide the heuristic we use the ratings assigned to each terminal. The rating $rating(nt)$ of a nonterminal nt is the sum of all ratings from the terminals in $term(nt)$. This way, the start symbol in a derivation DAG is rated. For set productions, we would like to find the non-conflicting subset of the conflicting nonterminals with the highest rating of all nonterminals. The

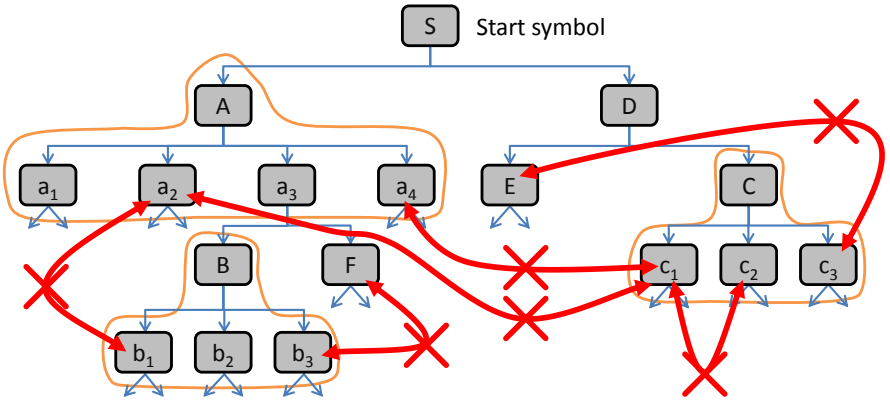

Figure 7: Top part of an exemplary derivation DAG with three set productions and root S. Conflicts between nonterminals are depicted by fat red arrows marked with crosses.

better the result is, i.e., the higher the final rating for the start symbol (which is decreased by every nonterminal omitted), the better the ambiguity resolution is, because a higher rating means more components, more complex components and more drawing precision.

The basic idea for the heuristic is to prefer nonterminals (i) with a high rating, (ii) with few conflicts, (iii) whose conflicting nodes have low ratings, and (iv) which are part of many matches of contexts for embedding productions. A ratio is calculated for each nonterminal nt. Let $embed(nt)$ be the set of all embedded edges where nt is one edge of the match of their context; the ratio r is calculated as

$$r(nt) = \frac{rating(nt) + \sum\{rating(e) | e \in embed(nt)\}}{\sum\{rating(c) | (nt,c) \in conflict\}}$$

The heuristic then works as follows: as long as there are conflicts, the nonterminal with the lowest ratio is omitted, and the ratios of its conflicting nonterminals are increased accordingly. If the start symbol cannot be deduced any more, because the last nonterminal of a set production is omitted, backtracking is applied and the next nonterminal is tried. We found that this very simple approach works well in practical cases, producing meaningful results quickly.

Therewith it is explained how the symmetric *conflict* relation must be exploited to generate derivation DAGs which are correct, i.e., which do not use terminals that must not occur in the same DAG due to the *exclude* relation introduced in the previous section. The ambiguities shown in the introductory example in Fig. 1 now can be solved in the desired manner. The RHM and a schematic representation of its DAG is shown in Fig. 8. The thin dashed arrows depict the matches for the contexts for the two arrows. Note that, due to the CNF, the one nonterminal `Place` cannot be the parent of the two terminals `t_place` and `t_token`. However, for the sake of clarity, we do not show the correct representation, which requires artificially generated nonterminal symbols.

Given that places and transitions in Petri nets are similarly rated, the decision about which components to omit depends on the embedded arrows. For Fig. 1, the leftmost place and the rightmost transition will be omitted, which is exactly the result we initially described. In general,

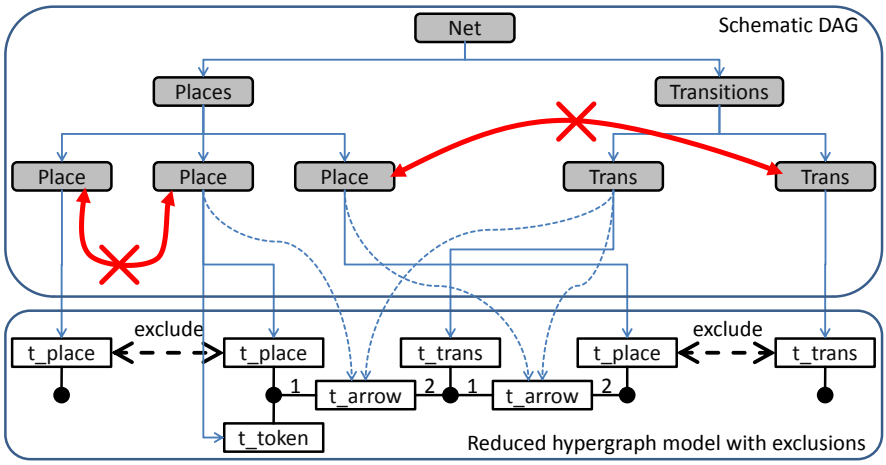

Figure 8: Reduced hypergraph model as shown in Fig. 5, and a schematic representation of the corresponding derivation DAG.

there may be a number of correct derivation DAGs. We then choose the one with the highest rated start symbol.

5 Related Work

Using context allows the machine to automatically decide for an interpretation in case of ambiguity. The alternative is to have the user explicitly make a choice. This is called *mediation* [MHA00]. Various strategies are conceivable, e.g., providing the user with a list of possible interpretations and let him decide, or requiring the user to redraw an ambiguous symbol. Especially the latter is limited in applicability. In our case, places and tokens in Petri nets are both drawn as a circle (cf. Fig. 1). Redrawing this circle obviously cannot resolve this ambiguity. Providing a list would help here, but is very uncomfortable for the user.

Some approaches decide for a possible interpretation without explicit user interaction, but neglect context information for this decision. For example, *LADDER* compares only the ambiguous components, not regarding context, and uses simple rules to prune alternatives, at the risk of preserving a wrong interpretation [HD05].

An approach by [AD06], limited to the domain of mechanical drawings, merges automatic decisions and user interaction. The system collects evidence from the drawing, based on rules. Additionally, each component is scored. Based on the evidence and the score, alternatives are pruned by a greedy algorithm. The algorithm is not able to undo its decisions, thus the result may not be optimal. In case of a wrong decision, the user can indicate that another possible interpretation is to be taken.

We are aware of only one other approach to sketching that treats ambiguity resolution with a grammar-based approach. It is based on so-called *sketch grammars* [CDPR04, CDR06]. Unlike our approach, diagram analysis is not separated into a reducer and a parser. Instead, the parser is directly applied. Parsing is directed by probabilities and rankings (similar to the ratings we

use) in order to avoid processing of unlikely interpretations. The used grammar is not based on hypergraphs, but extends positional grammars, which themselves extend traditional string grammars by more general relations than concatenation. For the actual parse process, only little detail is published. The concept of NACs is not employed. Result of the parser is a forest of ranked derivation trees, each representing a valid interpretation of the drawing. The user can then choose the desired representation.

Both recognition and ambiguity resolution works different for (handwritten) text, which we do not want to cover with our approach. Various methods are reported for this issue. For example, characters can be disambiguated by use of vocabularies, words can be disambiguated by statistical methods like Hidden Markow Models, language models, or specialized statistical grammars like PCFG (probabilistic context-free grammar) [PS00, ZCB06, HLB00].

6 Conclusion and Future Work

In this paper we explained an approach to ambiguity resolution in sketched diagrams using a hypergraph-grammar-based approach. Omission of components due to NACs is deferred as long as possible, until the parser needs a decision to go on. Using Petri nets as example we motivated the basic ideas. Applicability is restricted because of an NP-complete problem. We have implemented the presented approach as an extension of DIAGEN. Practical results suggest that the heuristic we apply works well, and quickly computes a result. We will inspect this restriction further, both from a theoretical and practical point of view. Graph grammars without set productions do not suffer from this issue.

The shown approach is incremental, i.e., modifications to the diagram do not require analysis from scratch. When components are added to the diagram, respective components edges are created, and all possible relationships of these components edges are checked. The reducer then only regards the newly created edges, and so does the parser, which modifies existing derivation DAGs. When diagram components are removed, the situation is similar.

Although we use the approach for ambiguity resolution in the context of sketching, regular graph parsers may benefit from the shown approach as well, as feedback in case of misplaced components (leading to a syntactically incorrect diagram) may be improved. The very strict behavior of the DIAGEN reducer, as shown in Sec. 2, suggests that feedback is very coarse and does not aid very much in finding misplaced components. What happens is that maximum sized subdiagrams with correct syntax are highlighted. Using our approach, the size of these subdiagrams may be increased.

Metamodel-based approaches have gained popularity in recent years. As future work, we would like to find out how DIAMETA, as a metamodel-based approach and an extension of DIAGEN, can benefit from the shown results as well, and how and to what extent they can be applied. Then we can compare the graph-based and the metamodel-based approaches.

Bibliography

[AD06] C. Alvarado, R. Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*. P. 24.

ACM, New York, NY, USA, 2006.

[BM08] F. Brieler, M. Minas. Recognition and Processing of Hand Drawn Diagrams Using Syntactic and Semantic Analysis. In *Proc. AVI '08*. ACM, 2008.

[CDPR04] G. Costagliola, V. Deufemia, G. Polese, M. Risi. A Parsing Technique for Sketch Recognition Systems. In *Proc. VL/HCC '04*. Pp. 19–26. IEEE Computer Society, Washington, DC, USA, 2004.

[CDR06] G. Costagliola, V. Deufemia, M. Risi. A Multi-layer Parsing Strategy for On-line Recognition of Hand-drawn Diagrams. In *Proc. VL/HCC '06*. Pp. 103–110. IEEE Computer Society, Washington, DC, USA, 2006.

[FNTZ00] T. Fischer, J. Niere, L. Turunski, A. Zündorf. Story Diagrams: A New Graph Grammar Language Based on the Unified Modelling Language and Java. In Ehrig et al. (eds.), *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*. Volume 1764, pp. 296–309. Springer, 2000.

[HD05] T. Hammond, R. Davis. LADDER, a sketching language for user interface developers. *Computers & Graphics* 29(4):518–532, 2005.

[HLB00] J. Hu, S. G. Lim, M. K. Brown. Writer independent on-line handwriting recognition using an HMM approach. *Pattern Recognition* 33(1):133–147, 2000.

[Kar72] R. Karp. Reducibility among Combinatorial Problems. In Miller and Thatcher (eds.), *Complexity of Computer Computations*. Plenum Press, New York, 1972.

[LV02] J. de Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Metamodelling. In *Proc. FASE '02*. Pp. 174–188. Springer-Verlag, London, UK, 2002.

[MHA00] J. Mankoff, S. E. Hudson, G. D. Abowd. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proc. UIST '00*. Pp. 11–20. ACM Press, New York, NY, USA, 2000.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Journal of Science of Computer Programming* 44(2):157–180, 2002.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. In *Electronic Communications of the EASST, Proc. GraBaTs '06*. September 2006.

[PS00] R. Plamondon, S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22(1):63–84, 2000.

[ZCB06] M. Zimmermann, J.-C. Chappelier, H. Bunke. Offline Grammar-Based Recognition of Handwritten Sentences. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(5):818–821, 2006.