In *Proc. 6th International Workshop on Theory and Application of Graph Transformations* (TAGT'98) at Paderborn, Germany, Nov. 1998. Appears in LNCS, 2000 ©Springer-Verlag

Hypergraphs as a Uniform Diagram Representation Model

Mark Minas

Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
minas@informatik.uni-erlangen.de

Abstract. When working with diagrams in visual environments like graphical diagram editors, diagrams have to be represented by an internal model. Graphs and hypergraphs are well-known concepts for such internal models. This paper shows how hypergraphs can be uniformly used for a wide range of different diagram types where hyperedges are used to represent diagram components as well as spatial relationships between components. This paper also proposes a procedure for translating diagrams into their hypergraph model, i.e., a graphical scanner, and a procedure to check the hypergraph against a hypergraph grammar defining the diagrams' syntax, i.e., a parsing procedure. Such procedures are necessary to make use of such a hypergraph model in visual environments that support free-hand editing where the user can modify diagrams arbitrarily.

1 Introduction

Diagrams play an important role whenever complex situations have to be represented. In computer science, which is only one field of diagram applications, Nassi-Shneiderman Diagrams [17], Message Sequence Charts [10], and Entity-Relationship Diagrams are only some examples. When used on computers, e.g., for editing and interactions, diagrams have internally to be represented by a formal model which abstracts from diagrams' redundant visual information and which makes informations about the diagram readily available.

Several concepts have been used as internal models. Among others, multisets of tokens [12], attributed symbols [5], and different kind of graphs and hypergraphs. Typical graph models are graphs where nodes represent tokens and edges represent relationships between tokens [20], special graphs where nodes have distinct connection points which are then used by edges for representing connections [22], and hypergraphs where visual tokens (diagram components) are represented by hyperedges and connections between them by nodes [14, 16]. Graphs and hypergraphs have the advantage that they are a formal and yet visual concept. Furthermore, there are powerful mechanisms like graph transformation theory and the existence of (hyper) graph parsers for syntactic analysis.

This paper extends the use of hypergraphs in the context of graphical diagram editors offering *free-hand editing*. Free-hand editing—in contrast to *syntax-directed editing*—allows the user to arbitrarily arrange and modify diagram components on the computer screen. It is then the editor's task to distinguish syntactically correct diagrams from incorrect ones and to (re)construct the diagram's syntactic information. Previous work [14] has described how diagrams are internally represented by hypergraphs and how a diagram language is specified by a hypergraph grammar. However, this approach was limited due to two reasons:

1. Hypergraph grammars were restricted to context-free ones (with optional *embedding productions* which add single edges to a certain context) in order to allow for efficient parsing.
2. Hypergraphs could not be used to represent diagrams that make use of arbitrary spatial relationships like *inside* or *above*.

This paper describes how hypergraphs can now represent diagrams using arbitrary spatial relationships, too. Hypergraphs become thus a uniform representation model for a wide range of diagram languages. In order to use this internal model, this paper describes two specific tasks of a diagram editor, *scanning* and *parsing*:

1. The user somehow arranges a set of diagram components. The editor has to create resp. update a hypergraph which represents this arrangement of components ("Scanning step").
2. After creating resp. updating the hypergraph, the editor has to check whether the hypergraph is syntactically correct according to some hypergraph grammar ("Parsing step").

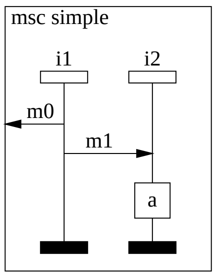
The rest of the paper is organized as follows: The next section describes two diagram languages which are used as demonstration examples throughout the paper. Section 3 then outlines how hypergraphs can be used to represent diagrams which use arbitrary spatial relationships, too. Sections 4 and 5 describe the scanning and parsing methods which create the internal hypergraph model and check its syntactic correctness. Related work is discussed in Sect. 6, Sect. 7 concludes.

2 Two Diagram Language Examples

Throughout this paper, we will use two kinds of diagrams: *Message Sequence Charts* (MSC) and *Visual Logic Diagrams* (VLD). This section briefly describes these two diagram languages.

MSC is a language for the description of interaction between entities [10]. A diagram in MSC describes which messages are interchanged between process instances, and what internal actions they perform. Figure 1a shows a sample diagram for MSC.

Visual logic diagrams (VLDs) [1] have been developed as an alternative visual notation for Horn clauses, a subset of first order logic. The following paragraph roughly sketches the idea of VLD syntax. For a complete description, see [1, 18].

(a) 

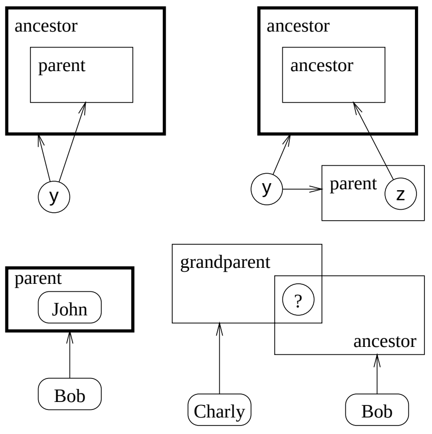
(b) 

Fig. 1. A Message Sequence Chart (a) and four visual logic diagrams (b). The letters y and z in (b) are not part of the diagram; they are used for reference in the text.

Terms in a VLD are visually represented by directed acyclic graphs (DAGs) where circles stand for variables and ovals for constants and functions. *Predicates* are visually represented by labeled boxes, meaning the set comprehension over one of its arguments, i.e., each predicate $P(x_1, x_2, \ldots, x_n)$ is defined by a box whose area represents a set $\mathbf{P}(x_2, \ldots, x_n) = \{x_1 \mid P(x_1, x_2, \ldots, x_n)\}$. Arguments $x_2, \ldots, x_n$ are explicitly represented by DAGs which are connected by arrows with the box's border. A VLD is composed of various predicates and terms put together in a Venn-diagram-like fashion. Each diagram has a special predicate (the predicate which is defined by the diagram, i.e., the Horn clause) which is highlighted by a thick border. Figure 1b shows some examples: The top-left diagram represents $\mathbf{parent}(y) \subseteq \mathbf{ancestor}(y)$, i.e., $\text{parent}(x, y) \Rightarrow \text{ancestor}(x, y)$. The top-right diagram visually represents $z \in \mathbf{parent}(y) \Rightarrow \mathbf{ancestor}(z) \subseteq \mathbf{ancestor}(y)$, i.e., $\text{parent}(z, y) \wedge \text{ancestor}(x, z) \Rightarrow \text{ancestor}(x, y)$. The other diagrams represent a fact that John is a parent of Bob and a query who (represented by the variable with the question mark) is a grandparent of Charly and an ancestor of Bob.

Each diagram generally consists of a set of atomic diagram components which are spatially related. The following shows that MSC and VLDs use different concepts which hypergraphs can represent in a uniform and straightforward way.

For MSC diagrams, components are surrounding boxes, start and end boxes, vertical lifelines, message arrows, action boxes, and labeling text. For VLDs, we have circles, ovals, arrows, rectangular boxes, and labeling text. Spatial relationships which are used for composing a diagram from its components are very different for MSCs and VLDs: MSC components are simply 2-dimensional shapes (boxes, text) which are related by connection objects (lines and arrows). MSC components are combined by attaching lines and arrows to boxes in a graph-like manner and by putting text near arrows and boxes. However, the ways of relating components in VLDs are more versatile: boxes can intersect or contain each other, circles and ovals may lie inside of boxes. Arrows can connect circles, ovals, and boxes.

3 Hypergraph Representation of Diagrams

Hypergraphs have proved to be an intuitive means for internally representing diagrams [14–16, 21]. A hypergraph is a generalization of a graph, in which edges are *hyperedges* which can be connected to any (fixed) number of nodes [8]. Each hyperedge has a type and a number of connection points ("tentacles") that attach to nodes. We say the hyperedge *visits* these nodes. The familiar directed graph can be seen as a hypergraph in which all hyperedges visit exactly two nodes.

A hypergraph-based specification of a diagram language has to consist of a mapping between diagrams and their hypergraph representation. A hypergraph grammar (see Sect. 5) specifies the set of all hypergraphs that represent valid diagrams. Mappings between diagrams and hypergraphs are specified in two steps. First, it is specified how each atomic diagram component is represented, and second, how these diagram components resp. their hyperedge representations are linked together. Each atomic diagram component is modeled by a hyperedge; hyperedge tentacles represent the component's "attachment areas", i.e., the areas which can actually connect to other components' attachment areas. However, only compatible attachment areas may connect. Such connections generally are established by overlapping attachment areas. For certain diagram languages, it is sufficient to represent all connections by "common" nodes: two tentacles are connected to the same node if the corresponding attachment areas are connected. This idea has been used in previous work [14–16, 21] and is applicable to MSC as described in the next paragraph. However, this representation is not sufficient for other diagram languages like VLDs which will be described afterwards.

MSC components are surrounding boxes, start and end boxes, vertical lifeline segments between events and actions, message arrows, action boxes, and labeling text. They are modeled by hyperedge types *msc*, *start*, *end*, *lifeline*, *message*, *action*, and *text*. A surrounding box's attachment areas are its borderline (*env*), its area for lifelines (*contains*), and its subarea for labeling text (*name*). A message's attachment areas are head and tail of the arrow (*to* resp. *from*) as well as an area for the labeling text (*name*). The other diagram components have similar attachment areas. E.g., a message's *to* attachment area is compatible to attachment areas representing end points of lifeline segments (*from* resp. *to*), i.e., corresponding hyperedges visit a common node if these attachment areas overlap. Figure 2 shows the resulting hypergraph which represents the MSC of Fig. 1a. Tentacles carry the names of their corresponding attachment areas.

This easy way of representing component connections is not sufficient for more general spatial relationships among diagram components as in VLDs. VLDs consist of ovals, circles, labeling texts, arrows, and rectangular boxes which are represented by corresponding hyperedges. *Text* hyperedges, which visit a single node (the text's area), represent text labels. Arrow hyperedges visit two nodes which represent the arrow's end points. In a VLD, circles, ovals, and boxes have two attachment areas: their borderlines and their area. Therefore, circles, ovals, and boxes are represented by (directed) edges which connect the two nodes that represent these attachment areas.

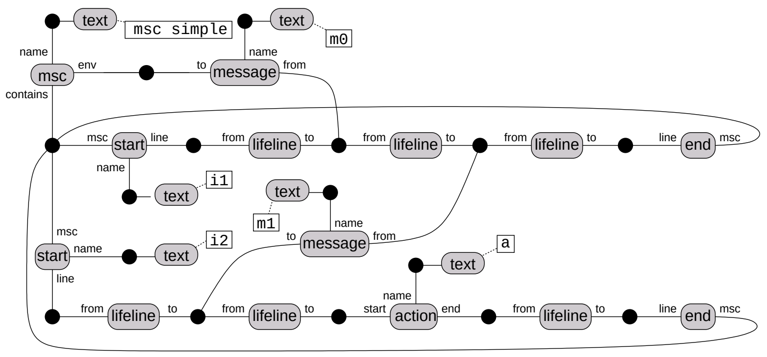


Fig. 2. Hypergraph model of the MSC diagram depicted in Figure 1a. Nodes are drawn as black dots, hyperedges as ovals, and tentacles as labeled lines connecting hyperedges with their visited nodes. m0, m1, . . . are textual attributes containing labeling text represented by *text* hyperedges.

VLDs' main spatial relationships are inclusion and intersection: Two boxes may overlap, one may contain another one, circles and ovals may lie inside of boxes. The situation where one box contains another one cannot be described by simply visiting the same area node. It would not be clear which box is the inner one. In order not to loose information by representing a diagram by an internal hypergraph, additional hyperedge types are necessary which directly represent spatial relationships. For VLD, *overlap* and *inside* hyperedges are used. The first one denotes an undirected edge (which is internally represented by two directed edges with opposite directions) connecting the area nodes of overlapping boxes, the latter one denotes a directed edge from the area node of a box to the area node of another box which contains the first one. Figure 3 shows the according hypergraph representations for the top-right and bottom-right VLD of Fig. 1b.

As a result of this section, the hypergraph model of a diagram consists of hyperedges which represent diagram components and nodes which represent the components' attachment areas. Spatial relationships between a diagram's components are either modeled by visiting the same nodes or by inserting additional hyperedges which represent the spatial relationships explicitly.

4 Scanning

In diagram editors which offer free-hand editing, diagram components are directly arranged by the user, i.e., created, deleted, dragged around etc. The internal hypergraph model has to be kept up-to-date. This is a graphical scanner's task. This section describes a scanning method and how it is adapted to specific diagram languages. As in the previous sections, we use MSC and visual logic diagrams as demonstration examples.

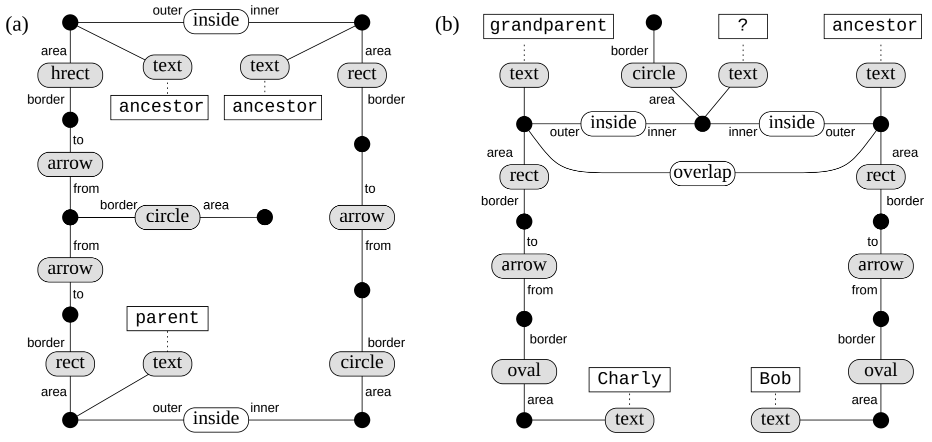


Fig. 3. Hypergraph representations of the top-right (a) and bottom-right (b) VLD of Fig. 1b. Hyperedges for diagram components have a gray label, spatial relationship edges have a white one. 'rect' specifies rectangular boxes, 'hrect' highlighted ones. `ancestor`, `parent` etc. are textual attributes containing labeling text represented by *text* hyperedges.

4.1 Intersecting Attachment Areas

The input of the scanning process is a set of atomic diagram components, each of them with a specific set of attachment areas. These are used for connections, i.e., spatial relationships between diagram components. For each component type (e.g., a box in VLD), there are different attachment area types with different shapes (e.g., the borderline and the inner area of boxes in VLDs). Attachment areas get connected if they intersect in a specific way. Given two attachment areas A and B, basically three different kinds of intersection are possible (the case where A and B do not intersect is omitted, i.e., $A \cap B \neq \emptyset$ holds for all three cases):

- $A \subset B$ or $B \subset A$ (Type "C" for *containing*).
- $A \not\subset B \wedge B \not\subset A$ and $A \cap B$ is not separated into disconnected pieces (Type "S" for *single intersection*).
- $A \not\subset B \wedge B \not\subset A$ and $A \cap B$ is separated into disconnected pieces (Type "M" for *multiple intersection*)

In VLDs, the differences between the intersection types are essential: If one box contains another one, we have type "C" intersection of the attachment areas of the boxes' areas. An arrow ending at a box's border causes an "S" or "C" intersection of the arrow's end point and the box's borderline (Requiring a "C" intersection exclusively would be too restrictive if the arrow does not exactly end at the borderline). Finally, two intersecting boxes can be detected by an "M" intersection of their borderlines.[1]

In [14], the following scanning procedure has been used to create the hypergraph model from the set of diagram components: For each diagram component together with its attachment areas, we have a hyperedge which visits a set of nodes, one node for each attachment area. Now check each intersection between any pair of attachment areas. Depending on the types of attachment areas and the type of intersection, do nothing or unify the corresponding nodes, i.e., the former two distinct nodes become a single one. This method works for the MSC example, but has to be extended for the extended hypergraph model of this paper: In some cases, additional hyperedges which represent general spatial relationships have to be inserted. However, additional context also has to be taken into account as the following example with VLD shows:

Consider the top-left diagram in Fig. 1b with the 'ancestor' box containing the 'parent' box. The 'parent' text belongs to the inner box which has to be represented by a single node that is visited by the text and the box hyperedge. The extended scanning procedure has to contain the rule to unify a box's area node with a text's node if the text's attachment area lies inside of the box's area. However, the outer box contains the text's attachment area, too. The same rule would require to unify the text's node, which is already unified with the inner box's area node, with the area node of the outer box, too. Obviously, this rule must not be applied in this context.

A similar problem arises in VLDs with many nested boxes. The method described above would create a large number of *inside* edges. Actually, the result would be the transitive closure of the set of actually required *inside* edges.

4.2 A Two-Step Scanning Method

An obvious solution to these problems would be to consider nesting level: if there is an intersection with the attachment areas of some box, all boxes containing this one are left out from further considerations. In the following, we present a more general solution which allows for specification of exactly this behavior.

For a correct scanning method, we do not create the hypergraph directly as in the previous method. Instead, we first create an intermediate graph which is then used for creating the hypergraph. The intermediate graph simply consists of all attachment areas as nodes and all detected intersections as explicit edges which are labeled with the intersection type. This *intersection graph* makes it possible to check the context for forbidden context graphs before unifying nodes or adding hyperedges to the hypergraph.

The scanning method now works as follows: Create the intersection graph by adding all attachment areas of the diagram as nodes. For each pair of intersecting attachment areas, draw an edge between the corresponding nodes. Use the intersection type as the edge's label. When this graph is completed, check each edge in the intersection graph. Depending on its label and its context, unify the

[1] This is equivalent to an "S" intersection of the boxes' areas.

corresponding hypergraph nodes, connect them with an appropriate additional hyperedge, or do nothing. This is specified by function

$$F : T \times I \times T \to \mathit{Action} \times 2^{\mathcal{G}}$$

where T is the set of attachment area types, $I = \{C, S, M\}$ the set of intersection types, *Action* the set of actions for hypergraph modifications, and $\mathcal{G}$ is the set of all graphs. $2^{\mathcal{G}}$ denotes the powerset of $\mathcal{G}$. For each pair $t_1, t_2 \in T$ of attachment area types and for each intersection type $i \in I$, $(\mathit{act}, \mathit{context}) = F(t_1, i, t_2)$ consists of an action and a set of forbidden context graphs. Either $\mathit{act} = \perp$ which represents the null operation ("do nothing"), or $\mathit{act} = \lambda x.\lambda y.\mathit{action}(x, y)$ where *action* is an action which refers to two hypergraph nodes x and y. Possible actions are "*Unify the nodes x and y*" or "*Draw an inside edge from node x to node y*", i.e., actions are actually transformation rules with two distinguished nodes. Finally, each element $G \in \mathit{context}$ is a (forbidden context) graph of attachment areas whose edges are labeled with intersection labels and with two of its nodes being labeled with α_1 and α_2, resp.

Function F describes how the scanning phase works after the intersection graph has been created: For each edge e of the intersection graph, determine the label i of edge e, source node c_1, and target node c_2 which are attachment areas that have some types t_1 and t_2, resp. Compute $(\mathit{act}, \mathit{context}) = F(t_1, i, t_2)$. If $\mathit{act} = \perp$, do nothing. Else try to match at least one graph $G \in \mathit{context}$ with the intersection graph such that G's nodes α_1 and α_2 match c_1 and c_2, resp. If such a match does exist, do nothing. Else determine the hypergraph nodes n_1 and n_2 which represent c_1 and c_2, resp., in the hypergraph model and use $\mathit{act}(n_1, n_2)$ in order to modify the hypergraph model.

For VLDs, attachment area types are $T = \{\mathit{area}, \mathit{border}, \mathit{point}, \mathit{text}\}$ which represent a box's, oval's, or circle's inner area, its borderline, an arrow's end points, and a labeling text's area. For brevity, we only show the specification of F for the case which has caused problems in the previous scanning method, i.e., a text is contained in a box:

$$F(\mathit{text}, C, \mathit{area}) = (\lambda x.\lambda y.\mathit{unify}(x, y), \{G\})$$

where $\mathit{unify}(x, y)$ means "Unify nodes x and y" and

$$\mathrm{G} = \begin{tikzcd} \overset{\alpha_1}{\textit{text}} \arrow[r, "C"] & \textit{area} \arrow[r, "C"] & \overset{\alpha_2}{\textit{area}} \end{tikzcd}$$

Graph G detects the situation where the text lies in a nested box. Since the intersection graph contains the transitive closure of "C" edges, the text will get connected to the innermost box only.

4.3 Complexity Issues

In order to get an idea of the scanning method's speed, we will now estimate its complexity.

The first phase of the scanning method checks for each intersection of attachment areas. When doing preprocessing by searching for intersecting rectangles which serve as bounding boxes for the attachment areas, this phase has complexity $O(n \log n + k)$ where n is the number of attachment areas and k the number of (bounding box) intersections since a well know Plane-sweep-algorithm can be used [13]. The worst case of the second phase is to match graphs for each intersection edge. If e is the maximum number of edges in (forbidden context) graphs, the worst case of graph matching is $O(k^e)$. Therefore, the worst case complexity of the scanning procedure is $O(n \log n + k^{e+1})$. But the intersection graph is normally sparse which reduces complexity of graph matching a lot.

5 Parsing

Diagrams are represented by hypergraphs. In order to define a diagram language in terms of a hypergraph language, *hypergraph grammars* are an appropriate means. Hypergraph parsers are used to check syntactic correctness of diagrams in terms of their hypergraphs and to (re)construct their syntactic structure. Hypergraph grammars and restricted classes of hypergraph grammars which allow for efficient parsing are briefly introduced in the following. However, these restrictions turn out to be too rigid for many diagram languages, e.g., VLD. This section outlines a parsing procedure which is applicable to a wider range of hypergraph grammars.

Hypergraph grammars are special cases of algebraic graph grammars [7]: Each hypergraph grammar HG consists of two disjoint finite sets N, T of nonterminal resp. terminal hyperedge types, a finite set of hypergraph productions, and a starting hypergraph (*axiom*) S which contains only nonterminal hyperedges. Each production $p = (L \overset{p_\mathrm{l}}{\leftarrow} G \overset{p_\mathrm{r}}{\rightarrow} R)$ is an algebraic hypergraph transformation rule with hypergraphs L, G, R called left-hand side (LHS), interface, and right-hand side (RHS), resp., and hypergraph morphisms $p_\mathrm{l}, p_\mathrm{r}$ where p_l is injective, i.e., G is a sub-hypergraph of L. Production p is applied to a hypergraph H by finding L as a subgraph (*redex*) of H, removing the sub-hypergraph $L \setminus G$ from H and merging in $R \setminus G$ instead where G and p_r exactly describe how the RHS has to fit in. The resulting hypergraph H' is called *derived* from H, $H \rightarrow H'$, in one step. As usual, the hypergraph language $L(HG)$ is the set of all hypergraphs H which contain terminal hyperedges only and which can be derived from S in a finite number of steps, $S \overset{*}{\rightarrow} H$.

There exist efficient parsers for restricted classes of hypergraph grammars only. [14] has discussed efficient parsers for context-free hypergraph grammars with optional embedding productions: Similar to context-free string grammars, context-free hypergraph grammars are defined in terms of their productions: A context-free production consists of a single nonterminal hyperedge as LHS, a discrete interface containing each node of the LHS, and injective morphisms $p_\mathrm{l}, p_\mathrm{r}$ [8]. Embedding productions add hyperedges to a certain context. Figure 4 shows the productions of a context-free grammar with two embedding productions (the two productions at the bottom) for MSC.

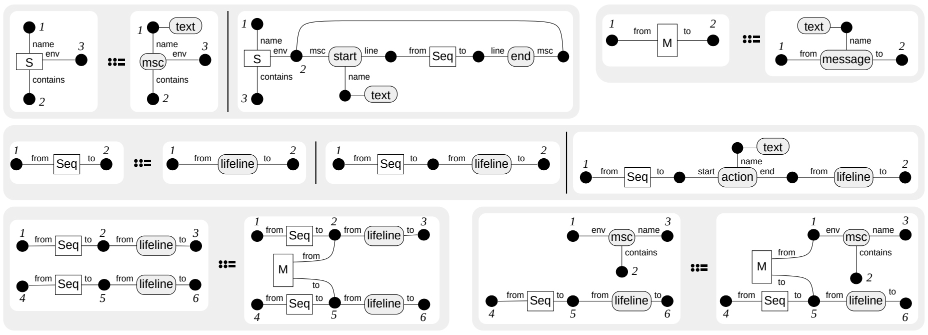


Fig. 4. Productions of a hypergraph grammar for MSCs. Ovals depict terminal hyperedges, rectangles nonterminal ones. Interface hypergraphs are implicitly defined by the labeled nodes and the hyperedges contained in the LHS and the RHS at the same time. One production which is the same as the last depicted one with a reversed message is omitted here.

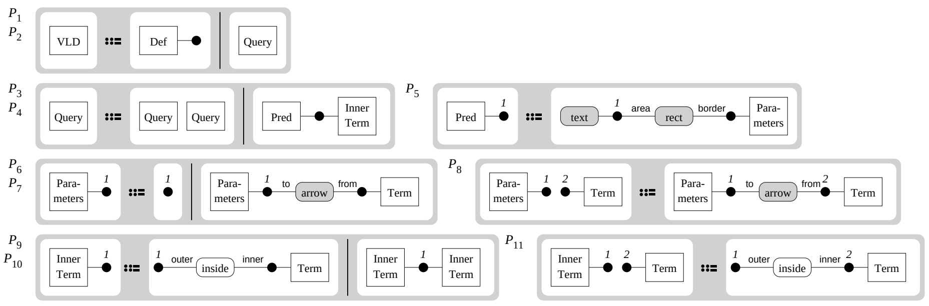


Fig. 5. Some selected hypergraph productions of the VLD grammar. The representation of productions is the same as in Fig. 4.

For VLD and the hypergraph model described in Sect. 3, we have not found such a context-free hypergraph grammar with embeddings. However, a grammar according to the general hypergraph grammar definition is easily created. Figure 5 shows a selection of eleven productions. The complete grammar consists of 32 productions. All shown productions except P_8 and P_{11} are actually context-free ones. P_8's interface hypergraph consists of the nodes *1* and *2* and of the 'Term' and 'Parameters' hyperedges as well. P_{11} is similar to P_8.

For a diagram editor with free-hand editing, a parsing algorithm has to try to reconstruct a derivation from the starting hypergraph to the internal hypergraph model of the current diagram. The diagram is valid if the parser succeeds, otherwise it is not. For context-free hypergraph grammars with optional embedding productions, this problem is more or less efficiently decidable [14]. For the general grammar, this problem is no longer decidable. For certain restricted

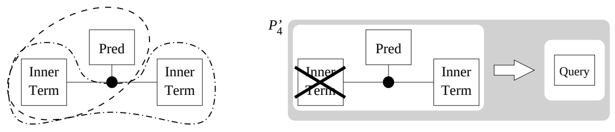


Fig. 6. Hypergraph with overlapping redexes (dashed borders) and appropriately extended reversed P_4 (see Fig. 5) with negative application condition (indicated by "X").

graph grammars[2], e.g., *layered graph grammars* [19], there are special parsing algorithms, however they are quite inefficient. Recently, the class of *reserved graph grammars* (RGGs) [22] has been proposed which allows for a straight-forward way to parse hypergraphs.[3] The derivation for a hypergraph H is reconstructed by exchanging LHS and RHS of each production ("reversed productions"), by starting a derivation at H, and using reversed productions until the derivation stops. If the resulting hypergraph is the starting hypergraph, H is valid. For RGGs, parsing is always terminating since their is a well-founded ordering on hypergraphs which is decreasing for each derivation step during parsing [22]. Furthermore, the resulting system of reversed productions is confluent, i.e., for all derivations $H \overset{*}{\to} H_1$ and $H \overset{*}{\to} H_2$, there is always a hypergraph H' with $H_1 \overset{*}{\to} H'$ and $H_2 \overset{*}{\to} H'$. Therefore we can conclude that the simple RGG parser finds a derivation from any hypergraph H to the starting hypergraph if and only if there exists such a derivation. The parser cannot run into a dead end.

However, this confluence property is the crucial property which is frequently, e.g., for our VLD grammar, hard to fulfill. We propose a simple but effective extension of graph grammars: if the system of reversed productions is not confluent, we extend the productions by appropriate context and—if this is not yet sufficient—add appropriate *negative application conditions* (NACs) to affected productions. NACs have been motivated by Habel et al. [9]: a production with matching LHS is not applicable if one of its NACs is satisfied. A NAC is simply a hypergraph that is connected to the LHS of the reversed production. The NAC is satisfied for an embedding of the LHS into the host hypergraph if the NAC hypergraph can be embedded also.

Of course, additional contexts and NACs modify and add further information to hypergraph grammars; the original grammar is no longer the only description of the hypergraph language's syntax. Tools may help here to find critical pairs and to assist in making the system of reversed productions confluent.

This paper does not present a fully fledged set of reversed productions for VLD due to its size (32 productions). Instead we give an example why we need NACs for VLDs. Figure 6 shows a hypergraph for which the reversed productions of P_4 as well as P_{10} are applicable. Reversed P_4 must not be applied since the

[2] In the literature, mainly graphs as the simple form of hypergraphs are used. However, most results for graphs also apply to hypergraphs.

[3] *Reserved graph grammars* have been introduced by Zhang and Zhang for a special kind of graphs which are actually hypergraphs.

inner terms have to be combined to a single inner term first. Otherwise, the second inner term could not be reduced. The correct behavior can be guaranteed by forbidding the existence of a second inner term if reversed P_4 is going to be applied. Figure 6 shows the appropriately extended reversed production P_4'.

6 Related Work

This paper is related to other work in the field of tools for creating diagram editors and in the fields of scanning algorithms and (hyper) graph parsing. The following list selects only some approaches.

In the field of frameworks for diagram editors there are several related approaches; the most closely related ones, which also allow to explicitly represent spatial relationships, are VLCC by Costagliola et al. [6], the proposed visual environment by Rekers et al. [2, 20], and GenGEd by Bardohl and Taentzer [3]:

Costagliola et al. use an object-oriented hierarchy for representing diagrams according to their syntactic models instead of a uniform representation by hypergraphs as in this work. For connecting visual components, their VLCC system uses attachment points which can be connected by lines. This paper presents a similar, but more general approach which allows to represent arbitrary spatial relationships between components.

The approach by Rekers et al. actually uses two kinds of graphs as internal representations of diagrams: the *spatial relationship graph* (SRG) abstracts from the physical diagram layout and represents higher level spatial relationships. Additionally, an *abstract syntax graph* (ASG) represents the diagram's logical structure and is kept up-to-date with the SRG. Two different but connected context-sensitive graph grammars are used to define the syntax of SRGs and ASGs. Free-hand editing of diagrams is planned to modify the first graph, syntax-directed editing is going to modify the second. The actually other graph is modified accordingly. In this paper and in [14–16, 21], we use hypergraphs instead of graphs which allows for a more "natural" diagram model. Furthermore, they restrict their discussions to graph-like diagrams. This work also considers diagrams with arbitrary spatial relationships.

GenGEd is an interactive tool for creating syntax-directed editors for graph-like diagram languages based on a powerful graph transformation system. However, free-hand editing based on some graph parser is not supported.

The ideas presented in Sect. 4 are also related to work of Blostein and Grbavec [4] on mathematics-recognition. They describe a scanning method which uses graph rewriting to create graphs as a representation of mathematical formulas. Application of graph transformation rules is controlled by spatial coordinates of recognized symbols. In contrast to our approach, they do not depend on overlapping attachment areas, but allow to take into account even distant components. Therefore, their approach is more general than ours at the expense of efficiency which is crucial in the context of interactive applications like diagram editors.

Furthermore, this work is related to other (hyper) graph parsing approaches. As an example, Lutz has presented a chart parser for flowgraphs [11] as a special kind of diagrams. Flowgraphs can be considered as a special case of hypergraphs used in this paper. The chart parser can be used top-down as well as bottom-up for context-free "*flow grammars*" only. Rekers and Schürr have proposed a graph parser for more general grammars, so called *layered graph grammars* [19]. Their parsing algorithm uses a bottom-up and then a top-down phase to (re)construct derivation sequences of the graph given. Finally, Zhang and Zhang have proposed an efficient parser even for context-sensitive *reserved graph grammars* based on Rekers and Schürr's one [22]. However, the properties required for these (hyper) graph grammars are quite restrictive. The hypergraph parsing procedure discussed in this paper actually uses the same parser, but relaxes these restrictions. However, additional information (*negative application conditions*) has to be provided.

7 Conclusions and Future Work

In this paper we have reconsidered modelling diagrams by hypergraphs as it is done by diagram editors in *DiaGen* [14, 16]. In a diagram editor, the internal (hypergraph) model is then further processed for syntax checking, semantic evaluation, etc., but this was beyond the scope of this paper. The paper has discussed different ways of how to model diagrams and how to obtain diagrams by connecting and combining their components. When using such a model in a diagram editor which supports free-hand editing, a graphical scanner is needed which creates the hypergraph model of the diagram currently edited. Such a scanning method has been presented in the paper. The scanning method makes use of a specification of the diagram language which describes how relationships between diagram components have to be modeled by edges in the hypergraph depending on the diagram components' contexts. This approach makes hypergraphs a flexible modelling concept suitable for modelling a large number of different diagram languages.

The paper has also outlined an efficient method for syntax-checking. Diagrams' syntax is specified by a hypergraph grammar. Correctness of a diagram is checked by trying to reduce the diagram's hypergraph to the grammar's axiom by using reversed grammar productions as transformation rules. Additional negative application conditions are used to avoid dead ends. This simple yet efficient syntax-checking procedure is applicable to a wide range of hypergraph grammars not restricted to context-freeness or context-sensitiveness. However, negative application conditions have to be provided as further information.

So far, the scanner does not work incrementally, i.e., the whole hypergraph model is (re) created from scratch even if only small diagram parts are modified. Current work on this problem tries to obtain such an incremental scanner. Other work has to deal with tool support. As already pointed out, tool support for detecting critical pairs in reversed productions would simplify the task of creating a confluent system.

References

1. J. Agusti, J. Puigsegur, and D. Robertson. A visual syntax for logic and logic programming. *Journal of Visual Languages and Computing*, 9:399–427, 1998.
2. M. Andries, G. Engels, and J. Rekers. How to represent a visual program? In *Proc. 1996 Workshop on Theory of Visual Languages, Gubbio, Italy*, May 1996.
3. R. Bardohl and G. Taentzer. Defining visual languages by algebraic specification techniques and graph grammars. In *Proc. 1997 Workshop on Theory of Visual Languages, Capri, Italy*, Sept. 1997.
4. D. Blostein and A. Grbavec. Recognition of mathematical notation. In H. Bunke and P. Wang, editors, *Handbook of Character Recognition and Document Image Analysis*, chapter 21, pages 557–582. World Scientific, 1997.
5. P. Bottoni, M. Costabile, S. Levialdi, and P. Mussio. Formalising visual languages. In *[24]*, pages 45–52, 1995.
6. G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora. A framework of syntactic models for the implementation of visual languages. In *[26]*, pages 58–65, 1997.
7. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, *LNCS* 73, pages 1–69, 1979.
8. A. Habel. *Hyperedge Replacement: Grammars and Languages*, *LNCS* 643, 1992.
9. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4), 1996.
10. ITU-T, Geneva. *Recommendation Z.120: Message Sequence Chart (MSC).*
11. R. Lutz. Chart parsing of flowgraphs. In *Proc. 11th Int. Conf. on Artificial Intelligence (IJCAI'89), Detroit, Michigan*, pages 116–121, Aug. 1989.
12. K. Marriott. Constraint multiset grammars. In *[23]*, pages 118–125, 1994.
13. K. Mehlhorn. *Data Structures and Algorithms 3 – Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.
14. M. Minas. Diagram editing with hypergraph parser support. In *[26]*, pages 230–237, 1997.
15. M. Minas and L. Shklar. A high-level visual language for generating web structures. In *[25]*, page 248f, 1996.
16. M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *[24]*, pages 203–210, 1995.
17. I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8):12–26, Aug. 1973.
18. J. Puigsegur, W. M. Schorlemmer, and J. Agusti. From queries to answers in visual logic programming. In *[26]*, pages 102–109, 1997.
19. J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *[24]*, pages 195–202, 1995.
20. J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *[25]*, pages 148–155, 1996.
21. G. Viehstaedt and M. Minas. Interaction in really graphical user interfaces. In *[23]*, pages 270–277, 1994.
22. D.-Q. Zhang and K. Zhang. Reserved graph grammar: A specification tool for diagrammatic VPLs. In *[26]*, pages 288–295, 1997.
23. *1994 IEEE Symp. on Visual Languages, St. Louis, Missouri*, Oct. 1994.
24. *1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, Sept. 1995.
25. *1996 IEEE Symp. on Visual Languages, Boulder, Colorado*, Sept. 1996.
26. *1997 IEEE Symp. on Visual Languages, Capri, Italy*, Sept. 1997.