Hypergraph Representation of Diagrams in Diagram Editors

Mark Minas
Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
email: 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 relations between components. Using such an internal model requires a method for translating diagrams into their hypergraph model. Such a graphical scanning method is proposed in this paper. The scanner makes use of a specification of the diagram language. Since the scanner also considers how diagram components are embedded into their context, it is applicable to a wide range of diagram languages and their hypergraph models.

Introduction

Diagrams are a powerful means to represent complex situations since they directly support visualizing multidimensional relations. The field of visual languages is only one example of where diagrams are in use. Originally, diagrams have served as a visualisation concept only (e.g., connection diagrams in electrical engineering) and have been produced on paper. Nowadays they are created on a computer using a graphical editor. Furthermore, they are used as a graphical way to express data for further processing and for reasoning about data. When using diagrams on computers, the visual representation of diagrams which yields better perception and reasoning with data is inappropriate for computers; diagrams have to be represented internally 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 (Marriott 1994), attributed symbols (Bottoni *et al.* 1995), and different kind of graphs and hypergraphs. Among the graphs, typical examples are graphs where nodes represent tokens and edges represent relations between tokens (Rekers & Schürr 1996), special graphs where nodes have distinct connection points which are then used by edges for representing connections (Zhang & Zhang 1997), and hypergraphs in our *DiaGen* project where visual tokens (diagram components) are represented by hyperedges and connections between them by nodes (Minas & Viehstaedt 1995; Minas 1997). (Hyper) Graphs have the advantage that they are a formal and yet visual concept. Therefore, even the internal model is often—but not always—quite easily perceived by humans. Furthermore, there are powerful mechanisms like graph transformation theory and the existence of (hyper) graph parsers for syntactic analysis.

The work described in this paper is continued work on *DiaGen*, a framework together with a generator for creating graphical editors from formal specifications. Each generated editor is specialized for the diagram class described by the specification. However, this paper does not describe all of *DiaGen*'s mechanisms like parser and layout approach, but rather considers a particular issue: the representation of diagrams by hypergraphs and how such hypergraphs are created and maintained during an editing process. This paper presents new results as compared to earlier descriptions of *DiaGen* (Minas 1997): On the one hand, the hypergraph model is extended beyond previous versions such that a larger number of diagram types can be modelled by hypergraphs. On the other hand, this paper presents a solution to the question of how to create such a hypergraph when a diagram is edited. When using *syntax-directed editing*, prespecified diagram operations modify the hypergraph model as the primary data structure. The diagram is modified according to the specification how hyperedges are visualized as diagram components Therefore, diagrams behave like views of their hypergraphs. However, when providing *freehand editing*, diagrams are modified directly, and the hypergraph representation has to be adjusted accordingly. The latter requires diagrams to be scanned graphically in order to create, delete, and (re)connect hyperedges. A solution to this problem is also presented within this paper.

The rest of the paper is organized as follows: The next section gives a brief overview of *DiaGen* that forms the environment for this work. *DiaGen* uses hypergraphs for internally representing diagrams. Then, three diagram classes are explained which are used as demonstration examples in the rest of the paper: Nassi-Shneiderman diagrams, Message Sequence Charts, and VEX expressions. The hypergraph model of *DiaGen* with extensions in expressibility is represented in the next section. The remaining sections propose a

method for creating such a hypergraph model for a diagram which is edited in a diagram editor offering free hand editing as in *DiaGen* and briefly discuss related work.

DiaGen

DiaGen as described in (Minas & Viehstaedt 1993; Viehstaedt & Minas 1994; Minas & Viehstaedt 1995; Minas 1997) consists of an editor framework and a generator. A formal specification of a diagram class serves as input for the generator which creates custom components that build—together with the framework—a graphical editor customized for the specified diagram class. Main features supported by this approach are:

- Diagrams are internally represented by hypergraphs; a diagram class is thus a hypergraph language together with a mapping from hypergraphs to their visual representation as diagrams. Either a context-free or a restricted kind of context-sensitive hypergraph grammar (Minas 1997) is used to describe the hypergraph language.
- Nodes and hyperedges carry attributes, and each grammar productions is augmented by layout constraints on attributes accessible in the production. A constraint-solver provides automatic, user-adjustable layout for context-free diagram shares (Minas & Viehstaedt 1993; Viehstaedt & Minas 1994). The framework also provides a mechanism to plug in external layout modules (e.g., programmed manually) for advanced layout which cannot be obtained by constraint satisfaction.
- Diagrams can be edited in a syntax-directed way using transformations on derivation trees for their context-free share and on hypergraphs described in the specification. To hide those details from the user, interactions of the user and the editor are described by certain interaction automata thus offering editing diagrams by direct manipulation.
- Free-hand editing is also supported. The user can arbitrarily add, delete, move, or modify parts of the diagram as with a drawing tool. The underlying hypergraph model is modified accordingly, a hypergraph parser distinguishes correct diagrams from incorrect ones by keeping the underlying hypergraph's syntactic meta-structure up-to-date. Free hand editing with parser support relaxes the need to specify a full set of transformations on diagrams for syntax-directed editing since free hand editing can be used for (yet) unspecified diagram operations. Therefore, this editing mode enhances usability of editors and also makes rapid prototyping of diagram editors possible because—as an extreme case—specification of diagram operations can be omitted completely. This and the parsers are described in (Minas 1997).

Diagram class examples

For showing the concepts of hypergraph representation of diagrams in the following section, we will use three kinds of diagrams: *Nassi-Shneiderman diagrams*, *Message Sequence Charts* (MSC), and *VEX* expressions. This section briefly describes these diagram classes.

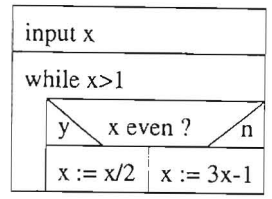


Figure 1: A Nassi-Shneiderman diagram.

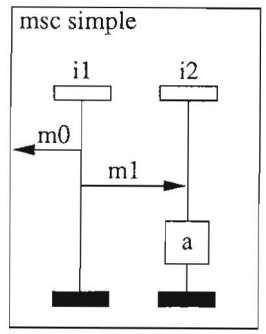


Figure 2: A Message Sequence Chart.

Nassi-Shneiderman diagrams (NSDs) are used for representing structured programs (Nassi & Shneiderman 1973). Each program statement is represented by a rectangle containing the statement. Sequences of statements are translated to stacks of corresponding rectangles. Loops are displayed as rectangles containing the loop control expression as well as the loop body. Moreover, particular graphical constructs are used for alternatives. Text contained in a box is used to describe operations and conditions. Figure 1 shows an example of a sequence of two statements, the second being a while loop. The body of the while loop consists of an alternative of two atomic statements.

MSC is a language for the description of interaction between entities (ITU 1993). A diagram in MSC describes which messages are interchanged between process instances, and what internal actions they perform. Figure 2 shows a sample diagram for MSC.

VEX (Citrin, Hall, & Zorn 1995) is a visual language for expressing λ-expressions: in VEX each variable identifier is represented by an empty circle that is connected by a line to a so-called *root node*. A root node is again an empty circle with one or more lines touching it, leading to all identifiers representing the same variable. A root node may either be internally tangential to another circle, it then represents a parameter of a λ-abstraction, or it is not included by any other circle, it then denotes a free variable. A circle representing a λ-abstraction contains its parameter circle and a VEX (sub) diagram as its body. An application of two expressions is depicted by two externally tangential circles with an arrow at the tangent point. The head of the arrow lies inside the argument circle. Figure 3 shows VEX expressions for $(\lambda x.x)y$ and $\lambda x.xx$.

In general each diagram consists of a set of atomic dia-

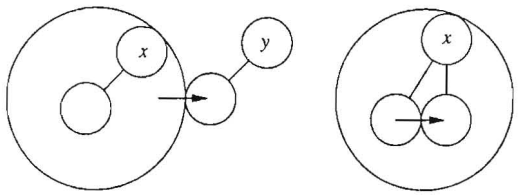


Figure 3: Two VEX expressions for $(\lambda x.x)y$ and $\lambda x.xx$.

gram components which are spatially related. The following shows that NSDs, MSC, and VEX expressions use different concepts which hypergraphs can represent in a uniform and straightforward way.

For NSDs, components are rectangular boxes, condition boxes, and polygons for loop control. MSC provides surrounding boxes, start and end boxes, vertical lifelines, message arrows, action boxes, and labeling text as diagram components. For VEX expressions, finally, we have circles, lines, and arrows. Spatial relationships used for composing a diagram from its components are very different when comparing NSDs as well as MSC with VEX expressions: NSD components are simply combined by stacking them one atop another or placing them side by side beneath alternative boxes. Components have to fit together at their corners. MSCs are mainly graph-like structures consisting of lifelines with connected line segments, action boxes, and message arrows. Components have to be connected at specific points, too. However, the ways of relating components in VEX expressions is more versatile: a circle can touch others or can contain an arbitrary number of others, there are lines and arrows connecting circles' circumferences, and arrows can also "connect" areas of circles. Another major difference to NSDs is the number of components that may be directly spatially related to a single component: for NSDs, each box can be related to four other boxes at most (to the top, bottom, left, and right), however in VEX, each circle can contain an arbitrary number of circles.

The next section shows that all these diagram types can use hypergraphs as internal model in spite of these differences.

Hypergraph representation of diagrams

Hypergraphs have proved to be an intuitive means for internally representing diagrams (Viehstaedt & Minas 1994; Minas & Viehstaedt 1995; Minas & Shklar 1996; Minas 1997). A hypergraph is a generalization of a graph, in which edges are *hyperedges*, i.e., they can be connected to any (fixed) number of nodes (Berge 1989). Each hyperedge has a type and a number of connection points that determine how many nodes the hyperedge is connected to. 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 specification of a diagram class based on such a hypergraph model consists of a mapping between hyperedges and their diagram counterparts and a description of all valid hypergraph models. The latter is usually specified by some

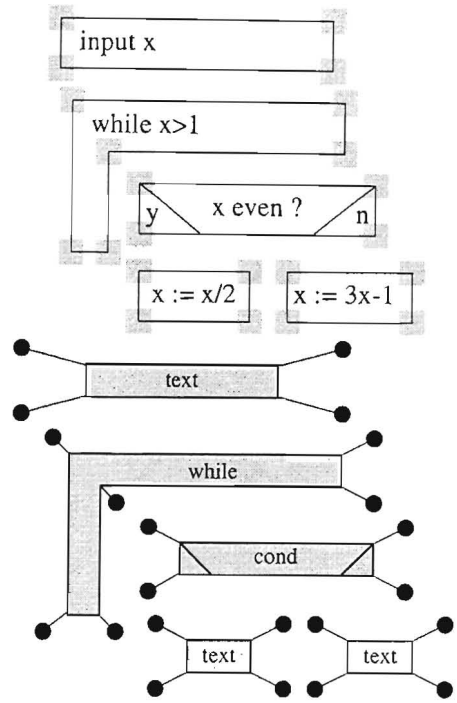


Figure 4: Diagram components represented by hyperedges. Hyperedges are shown as gray areas connected to nodes that are represented by black dots.

(hyper)graph grammar which is discussed in detail in (Minas & Viehstaedt 1995; Minas 1997). The specification of mappings between hyperedges and diagram components is quite obvious: Each atomic diagram component is simply represented by a hyperedge; nodes represent the component's "connection areas", i.e., the areas which can actually connect to other components' "connection areas". Figure 4 shows the diagram components of the NSD shown in Figure 1. The gray squares represent the components' connection areas. Figure 4 also shows the representation of each of these components by a single hyperedge visiting an appropriate number of nodes.

The representation of a complete diagram, i.e., the representation of connections between hyperedges, however, has to depend on the kind of spatial relations. The easiest way of connecting hyperedges representing related diagram components is to let them visit the same nodes if the corresponding diagram components' connection areas are connected. This representation can be used for NSDs: hyperedges visit the same nodes as soon as connection areas overlap (e.g., after moving components in the graphical editor). Figure 5 shows the resulting hypergraph for the NSD shown in Figure 1 after "unifying" the appropriate nodes of Figure 4.

MSCs are similarly represented by hypergraphs: A surrounding box has the borderline, an area for lifelines, and a subarea for labeling text as connection areas. A message's connection areas are head and tail of the arrow as well as an area for the labeling text. The other diagram components have similar connection areas. Special consideration

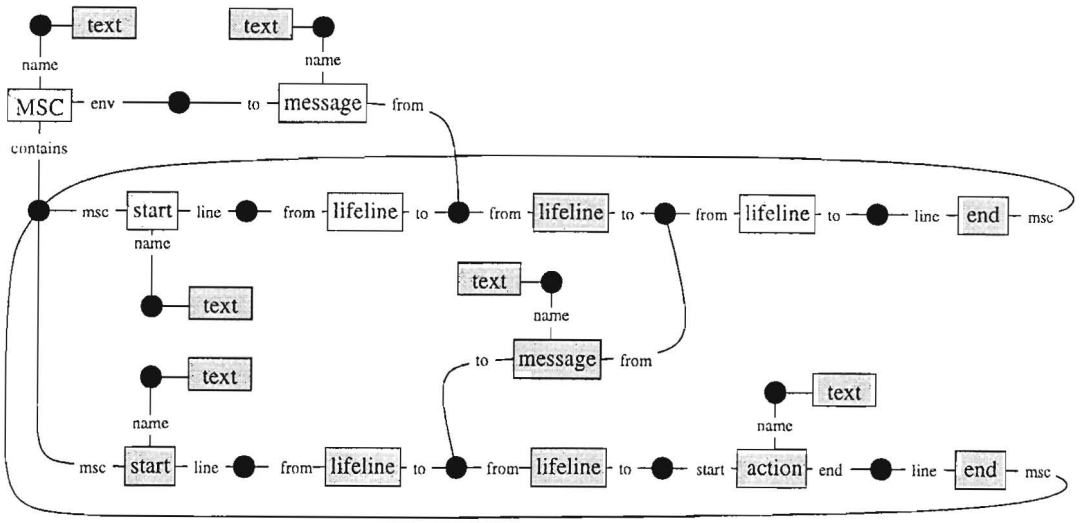


Figure 6: Hypergraph model of the NSD depicted in Figure 2. Nodes are drawn as circles, hyperedges as ovals being connected to its visited nodes.

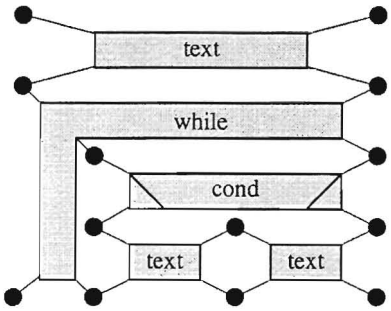


Figure 5: Hypergraph model of the NSD depicted in Figure 1. Nodes are drawn as circles, hyperedges as ovals being connected to its visited nodes.

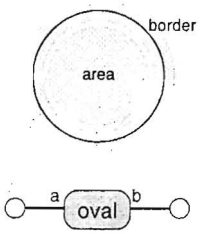


Figure 7: Hyperedge representation of a circle. "a" stands for "area", "b" for border.

is needed for lifelines: A natural representation would be a hyperedge representing the whole lifeline. This hyperedge would need as many tentacles as there are actions, incoming, and outgoing messages plus two additional ones for beginning and end of the line. However, dealing with hyperedges with variable number of tentacles makes problems with syntax definition. Therefore, we represent lifeline segments between each two events or actions instead of the whole lifeline. Figure 6 shows the hypergraph representing the MSC shown in Figure 2.

Unfortunately, this easy way of representing component connections is not sufficient for more general spatial relationships between diagram components. As an example, we consider VEX again. VEX diagrams consist of circles, lines, and arrows which are represented by corresponding hyperedges. Actually, plain edges can be used, hyperedges visiting more than two nodes are not required for VEX diagrams. The edges representing arrows and lines simply connect nodes representing the corresponding end points ("connection area" of lines and arrows). In VEX, a circle has two "connection areas": its borderline and its inner area. Therefore, circles are represented by (directed) edges connecting the two nodes representing these "connection areas" (see Figure 7).

VEX's main spatial relations relate two circles which may be internally or externally tangential, or they relate arrows with circles where arrow head and tail lie inside of two circles. The latter situation is represented similar to connecting two circles by a line: the arrow's hyperedge simply connects the circle nodes which represent the circles' areas. However, a situation where one circle is contained in another one cannot be described by simply visiting the same area node. It would not be clear which circle is the inner one. Furthermore, if there were a third circle contained in the outer one, we would have to visit the same area node again, loosing all the information of how the second and the third circle are related (Figure 8).

In order not to loose information by representing a diagram by an internal hypergraph, additional hyperedge types *touch* and *inside* are used. The first one denotes an (undirected) hyperedge connecting the borderline nodes of tangential circles, the latter one denotes a directed edge from the area node of a circle to the area node of another circle

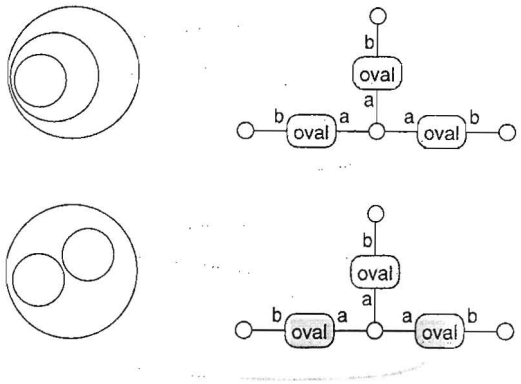


Figure 8: Inadequate representation of spatial relations between circles. Both diagrams have the same hypergraph representation.

which contains the first one (cf. Figure 9). We do not need a direct representation of circles being internally or externally tangential. This is expressed by a *touch* edge between the border nodes of both circles and an *inside* edge between the area nodes if the circles are internally tangential and without such an *inside* edge if they are externally tangential. In order to make detection of invalid diagrams possible on the level of their internal diagrams, we also need an additional *intersect* edge which connects the border nodes of intersecting circles. The existence of such an edge is an indicator for an invalid VEX diagram. Figure 10 shows the according hypergraph representations for the VEX diagrams of Figure 3.

As a result of this section, the hypergraph model of a diagram consists of hyperedges representing diagram components and nodes representing the components' connection areas. Spatial relations between a diagram's components are either modelled by visiting the same nodes or by inserting additional hyperedges directly representing the spatial relationship.

Creating a diagram's hypergraph model

In diagram editors offering free hand editing, diagram components are directly manipulated by the user, i.e., created, deleted, dragged around etc. The internal hypergraph model has to be kept up-to-date. This a graphical scanner's task. This section describes a method for scanning and how to specify such a scanner for a specific diagram class. As in the previous sections, we use NSDs and VEX expressions as demonstration examples.

Intersecting connection areas

The input of the scanning process is a set of atomic diagram components each with a specific set of "connection areas". These are used for connections, i.e., spatial relations between diagram components. For each component type (e.g., a circle in VEX), there are different connection area types with different shapes (e.g., the borderline and the inner area of circles in VEX). Connection areas got connected if they intersect in a specific way. Given two connection areas A

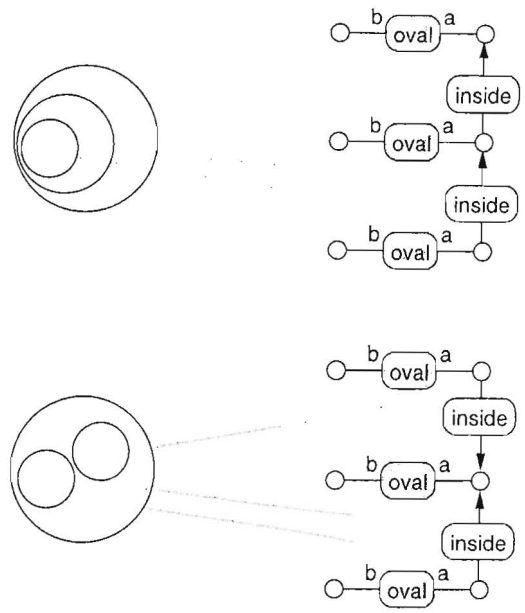


Figure 9: Representation of spatial relations using hyperedges explicitly representing spatial relations.

and B, basically three different kinds of intersection are possible (the case where A and B do not intersect is omitted, i.e., $A \quad B \quad \emptyset$ holds for all three cases):

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

In VEX, the differences between the intersection types are essential: If one circle contains another one, we have type "C" for the connection areas of the circles' areas. Two circles touching have an "S" intersection of the connection areas of the circles' borderlines. Finally, two circles intersecting (which is a syntactic error in a VEX expression) can be detected by an "M" intersection of these connection areas.

A first idea of how to create the hypergraph model from the set of diagram components together with their connection areas is as follows: For each diagram component together with its connection areas, we have a hyperedge which visits a set of nodes, one node for each connection area. Now check each intersection between any pair of connection areas. Depending on the types of connection areas and the type of intersection, unify the corresponding nodes (i.e., the former two distinct nodes become a single one), connect them with an appropriate additional hyperedge (like an *inside* edge), or do nothing (e.g., if the circle area contains another circle's inner area).

This method is effective for simple connection schemes as for NSDs. However, it is not sufficient for more complex spatial relations as with VEX: Consider functional application of expressions which are represented by two touching

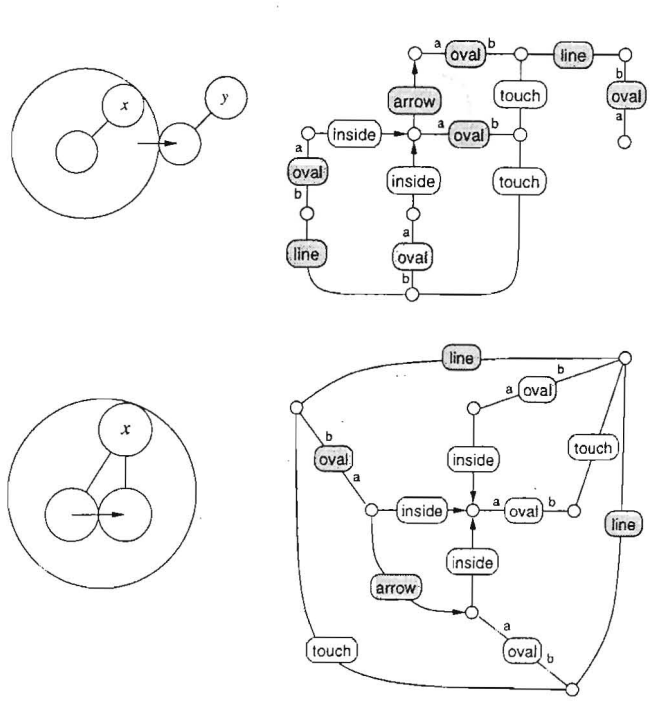


Figure 10: Hypergraph representation of the VEX diagrams of Figure 3. (Hyper)Edges for diagram components have a gray label, spatial relationship edges have a white one. Gray lines show which VEX circle is represented by which "oval" hyperedge.

circles with an arrow at the tangent point. E.g., Figure 11 shows the hypergraph model of the right VEX diagram of Figure 3. The gray part of the hypergraph demonstrates the problem which cannot be created by the scanning method described above: Each of the inner circles contains a connection area of the circle which is expressed by unifying the arrows' nodes with the circles' area nodes. However, the outer circle contains the arrow's connection areas, too. This would result in a unification of the area nodes of theses three circles which is not desired. Obviously, the scanning method's problem is not to check any context.

A similar problem arises in VEX expressions with many nested circles. 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.

A two-step scanning method

An obvious solution to the problems described above is to consider nesting level: if there is an intersection with the connection areas of some circle, all circles 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

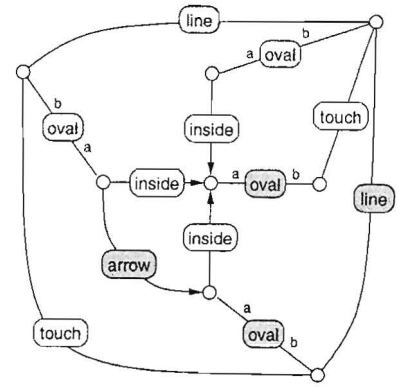


Figure 11: Hypergraph representation of the right VEX diagram of Figure 3 as in Figure 10, but with a highlighted subgraph.

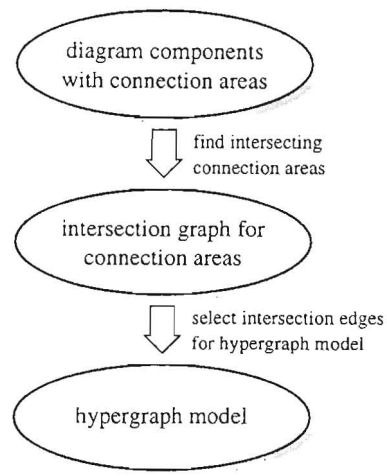


Figure 12: The two-step scanning method

the hypergraph, see Figure 12. The intermediate graph simply consists of all connection areas as nodes and all detected intersections as explicit edges labeled with the intersection type.[1] This *intersection graph* makes it possible to consider context before unifying nodes or adding hyperedges to the hypergraph.

The scanning method now works as follows: Create the intersection graph by adding all connection areas of the diagram as nodes. For each pair of intersecting connection areas, draw an edge between those 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 corresponding hypergraph nodes, connect them with an appropriate additional hyperedge, or do nothing.

The remaining task is how to specify the action modifying the hypergraph depending on edge label and edge context.

[1] C-edges are directed from the containing to the contained node; S- and M-edges are undirected.

As for the previous method, we can make use of the intersection type (i.e., the intersection edge's label) and the types of the intersecting connection areas for determining the right action. However, the action is prohibited if the intersection edge has a certain context. In the following, this context is represented as a graph.

More formally, the scanning specification is a function

$$F : T \quad I \quad T \quad Action \quad G$$

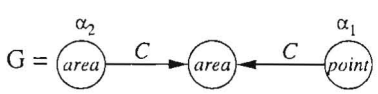
where T is the set of connection area types, $I \quad C\ S\ M$ the set of intersection types, *Action* the set of actions for hypergraph modifications, and G is the set of context graphs. For each pair $t_1 \ t_2 \quad T$ of connection area types and for each intersection type $i \quad I$, $(A \ G) \quad F(t_1 \ i \ t_2)$ consists of an action $A \quad$ representing the null operation or an λ-expression $\lambda x.\lambda y.action(x \ y)$. *action* is an action referring 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*". Finally, G is either an empty graph or a graph of connection areas whose edges are labeled with intersection labels and with two of its nodes being labeled with α_1 and α_2, resp.

With this function F, the scanning phase following after the creation of the intersection graph is as follows: 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 connection areas having some types t_1 and t_2, resp. Compute $(A \ G)$ $F(t_1 \ i \ t_2)$. If $A \quad$, do nothing. Else try to match G with the intersection graph such that G's nodes α_1 and α_2 have to match c_1 and c_2, resp. If such a match does exist and G is not the empty graph, do nothing. Else determine the hypergraph nodes n_1 and n_2 representing c_1 and c_2, resp., in the hypergraph model and use $A(n_1 \ n_2)$ in order to modify the hypergraph model.

For VEX, connection area types are T *area border point* representing a circle's inner area, its borderline, and the lines' and arrows' end points. F has non-null operations for six pairs in $T_1 \quad T_2$. For brevity, we show only the specification for the case which has caused problems in the previous scanning method, i.e., an arrow's end point is contained in a circle:

$$F(point \ C \ area) \qquad \lambda x.\lambda y.unify(x \ y) \ G)$$

where $unify(x \ y)$ means "Unify nodes x and y" and



Graph G detects the situation where the arrow ends in a nested circle. Therefore, the arrow will get connected to the inner circle only, solving the problem of the first scanning method.

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 connection areas. When doing preprocessing by searching for intersecting rectangles which serve as bounding boxes for the connection areas, this phase has complexity $O(n \log n \quad k)$ where n is the number of connection areas and k the number of (bounding box) intersections since a well know Plane-sweep-algorithm can be used (Mehlhorn 1984). The worst case of the second phase is to match graphs for each intersection edge. If e is the maximum number of edges in context graphs, the worst case of graph matching is $O(k^e)$. Therefore, the worst case complexity of the scanning procedure is $O(n \log n \quad k^{e+1})$. However, the intersection graph is normally sparse which reduces complexity of graph matching a lot.

Related work

Graphs are frequently used as an internal model for representing diagrams in a visual environment. When creating the internal model from the visual representation, some kind of graphical scanning is necessary. Examples of related concepts are the visual programming environment for implementing visual languages VLCC by Costagliola et al. (Costagliola *et al.* 1997) and the proposed visual environment by Rekers and Schürr (Rekers & Schürr 1996).

VLCC is distinguishes between *connection-based* and *geometric-based* diagrams. For connection-based ones, the user specifying the visual language can define connection points for each visual token, i.e., atomic diagram component. When using the generated editor, the user can connect such connection points by lines. No scanning is necessary. For geometric-based diagrams, tokens have to fit together in a specified way. This is also a special case of the more general case discussed in this paper.

The approach by Rekers and Schürr 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 relations. Additionally, an *abstract syntax graph* (ASG) is kept up-to-date with the SRG representing the diagram's logical structure. For free-hand editing, they propose to use a scanner for extracting low level spatial relations from the diagram. However, they do not describe in detail how such a scanner would work. Instead they claim that—as long as they do not use low level graphics editor—their spatial relationship graph can be kept up to date by the user interface.

Furthermore, this work is related to 9-intersection, a comprehensive model for binary topological relations among point-, line-, and area-objects (Egenhofer & Franzosa 1991; Egenhofer & Herring 1991). However, the problem of determining and representing spatial relationships in the context of diagram editors as in this paper is more restricted: We assume that diagrams which are edited in specialized diagram editors do not use arbitrary spatial relationships among any diagram components, but that each component has specific connection areas of certain types. We consider only those spatial relationships that can be detected by checking overlapping connection areas of certain types which happens when diagram components are moved on the editor's canvas.

Conclusions and future work

In this paper we have reconsidered modelling diagrams by hypergraphs as it is done by diagram editors in *DiaGen* (Mi-

nas & Viehstaedt 1995; Minas 1997). In a diagram editor, the internal (hypergraph) model is then further processed for syntax checking, semantic evaluation, etc., but this was outside 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 class which describes how relations between diagram components have to be modelled by edges in the hypergraph depending on the diagram components' contexts. This approach makes hypergraphs a flexible modeling concept suitable for modelling a large number of different diagram classes.

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 deals with the hypergraph model itself. So far, hyperedges are only allowed to visit a fixed number of nodes depending on the hyperedge type. However, there are diagrams where visiting an arbitrary number of nodes makes sense, e.g., lifelines in *Message Sequence Charts* as described in this paper. The "natural" model is a hyperedge representing the whole time line. However, this hyperedge has to visit all the nodes which model the points of time.

References

Berge, C. 1989. *Hypergraphs*. Amsterdam: North-Holland.

Bottoni, P.; Costabile, M.; Levialdi, S.; and Mussio, P. 1995. Formalising visual languages. In *VL'95* (1995), 45–52.

Citrin, W.; Hall, R.; and Zorn, B. 1995. Programming with visual expressions. In *VL'95* (1995), 294–301.

Costagliola, G.; Lucia, A. D.; Orefice, S.; and Tortora, G. 1997. A framework of syntactic models for the implementation of visual languages. In *VL'97* (1997), 58–65.

Egenhofer, M., and Franzosa, R. 1991. Point-set topological spatial relations. *International Journal of Geographic Information Systems* 5(2):161–174.

Egenhofer, M., and Herring, J. 1991. Categorizing binary topological relationships between regions, lines, and points in geographic databases. Technical report, Department of Surveying Engineering, University of Maine.

ITU-T, Geneva. 1993. *Recommendation Z.120: Message Sequence Chart (MSC)*. See also: *http://www.win.tue.nl/win/cs/fm/sjouke/msc.html*.

Marriott, K. 1994. Constraint multiset grammars. In *VL'94* (1994), 118–125.

Mehlhorn, K. 1984. *Data Structures and Algorithms 3 – Multi-dimensional Searching and Computational Geometry*. Berlin: Springer-Verlag.

Minas, M., and Shklar, L. 1996. A high-level visual language for generating web structures. In *VL'96* (1996), 248f.

Minas, M., and Viehstaedt, G. 1993. Specification of diagram editors providing layout adjustment with minimal change. In *VL'93* (1993), 324–329.

Minas, M., and Viehstaedt, G. 1995. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *VL'95* (1995), 203–210.

Minas, M. 1997. Diagram editing with hypergraph parser support. In *VL'97* (1997), 230–237.

Nassi, I., and Shneiderman, B. 1973. Flowchart techniques for structured programming. *SIGPLAN Notices* 8(8):12–26.

Rekers, J., and Schürr, A. 1996. A graph based framework for the implementation of visual environments. In *VL'96* (1996), 148–155.

1997. *1997 Workshop on Theory of Visual Languages (TVL'97), Capri, Italy*.

Viehstaedt, G., and Minas, M. 1994. Interaction in really graphical user interfaces. In *VL'94* (1994), 270–277.

1993. *1993 IEEE Symp. on Visual Languages, Bergen, Norway*, IEEE Computer Society Press.

1994. *1994 IEEE Symp. on Visual Languages, St. Louis, Missouri*, IEEE Computer Society Press.

1995. *1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, IEEE Computer Society Press.

1996. *1996 IEEE Symp. on Visual Languages (VL'96), Boulder, Colorado*, IEEE Computer Society Press.

1997. *1997 IEEE Symp. on Visual Languages (VL'97), Capri, Italy*, IEEE Computer Society Press.

Zhang, D.-Q., and Zhang, K. 1997. Reserved graph grammar: A specification tool for diagrammatic VPLs. In *VL'97* (1997), 288–295.