Graphical Representation and Manipulation of Complex Structures Based on a Formal Model

G. Viehstaedt M. Minas

Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany

Abstract

Complex information structures can often be represented by diagrams. Diagrams (e.g., trees for hierarchical structures, or graphs for finite state machines) are useful as part of user interfaces of information systems and CASE tools. Beyond representation, manipulation by interactively editing diagrams should be possible. The implementation of editors for diagrams should be supported by a tool and based on a formal model. This paper gives an overview of our generator for diagram editors. An editor for a certain kind of diagrams is generated from a specification, which includes a grammar to describe the structure of diagrams. The user of a diagram editor, however, doesn't have to be concerned with the grammar, but can manipulate diagrams in a very convenient way.

1 Introduction

An easily comprehensible representation of complex structures is getting more and more important for the users of information systems as well as for software engineers using a CASE tool. A significant and increasing share of the effort for implementing these systems goes into the user interface. The available interface builders for so-called Graphical User Interfaces (GUIs), however, mainly focus on widgets like buttons, menus, etc., and aren't very graphical. This assessment was also made in a recent *CACM* issue on GUIs [7]:

What the market considers a GUI is little more than a glorified menu system, having no graphics. This leaves the graphical representation of the application domain as an exercise for the developer, ...

Diagrams usually have a semantical meaning for the application. Thus, changes to a diagram have to be transformations from one consistent state into another. The set of valid diagrams, called a *diagram class*, should be described

by a formal model. A diagram class can, e.g., be the set of graphs for entity-relationship diagrams or finite state machines, or the set of all Nassi-Shneiderman diagrams (NSDs). Editing is done in a syntax-directed way, which ensures that the result of a change is again a valid diagram. Such an editor for diagrams is refered to as *diagram editor*.

Many user interface tools provide no help for diagrams. Some tools, e.g., Garnet [8], support simple kinds of diagrams. The structure of valid diagrams, however, isn't specified formally, but more or less hidden in the code. There are only very few systems for generating a diagram editor that are based on a formal model. In the *PAGG* system [4] layout of diagrams is troublesome and editing inconvenient. The *Constraint* system [9] uses a grammar model based on context-free grammars and constraints for automatic layout of diagrams. A disadvantage is that context-free grammars don't permit direct representation of multidimensional relationships, as needed for the layout of diagrams. Furthermore, the editing capabilities are very restricted.

DiaGen, a generator for diagram editors, will be outlined in this paper. The very natural and convenient way of editing will be illustrated in the next section. Section 3 gives an overview of the system. The grammar model for specifying the structure of diagrams and layout are the topic of Section 3.1. User interaction with diagrams will be briefly addressed in Section 3.2.

2 Editing diagrams

A diagram editor for Nassi-Sheiderman diagrams (NSDs), which was generated with *DiaGen*, serves for demonstrating the way of editing.[1] An NSD mainly consists of lines and text blocks for statements and conditions. These diagram elements can be selected in different ways, which are stated in the specification and thus adapted to the particular editor. For our sample editor it has been specified that subsequent selected statements (which may be complex) are automatically combined into a single entity called a *group*. In Fig. 1 a group of two subsequent statements is selected, as indicated by the handles.

As an example, we now want to remove this group from its position and insert it into the upper NSD as the first statements of the *while*. Fig. 2 shows the result. One way to achieve this is to press the *Cut*-button, and then select the line immediately below the *while*'s condition to indicate the position for insertion. Finally, the *Insert*-button has to be pressed.

[1] A graphical language for database queries would be a similar example, but NSDs are more familiar and don't require further explanations.

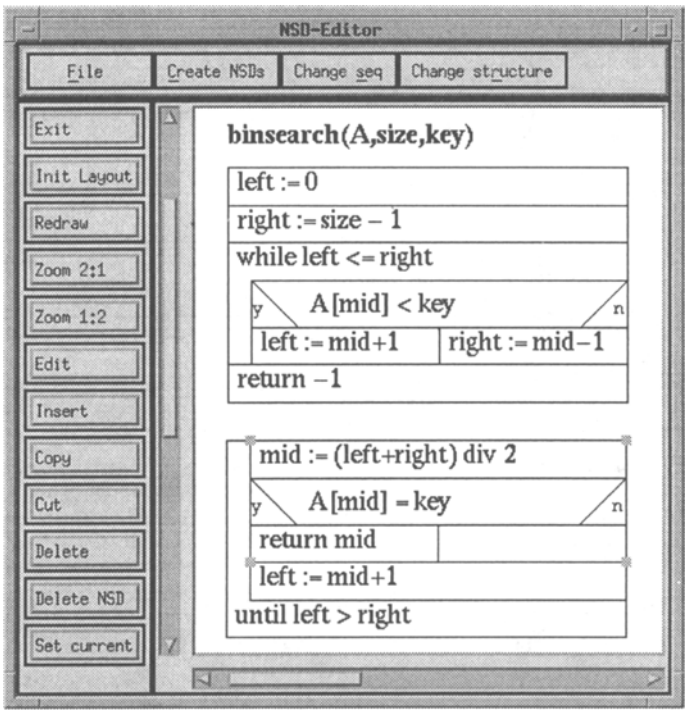

Fig. 1. *Editing state with two NSDs. A group of two subsequent statements (consisting of a simple statement and an if) is selected in the lower NSD.*

This editing style is solely based on selection and pressing buttons or choosing menu items. It is the only way of editing possible in diagram editors which can be generated with other tools like PAGG [4] or Constraints [9].

With *DiaGen* diagram editors with much more convenient editing operations can be specified. E.g., the modification from Fig. 1 to Fig. 2 can also be made by simply pressing the mouse button while the mouse pointer is located over the selected group, dragging this group to its destination, and releasing the mouse button over the line below the *while*'s condition. We will refer to this editing style as *direct manipulation* of diagrams.

In the same way any part of a diagram can be removed and inserted at any other spot. A diagram part can be made a new NSD of its own in a similar fashion by moving it around and releasing the mouse button outside any other NSD. Of course, an entire NSD can also be inserted into another diagram.

A number of other editing operations by direct manipulation have been specified for the NSD editor. E.g., the *then*- and *else*-branch of an *if* can be switched

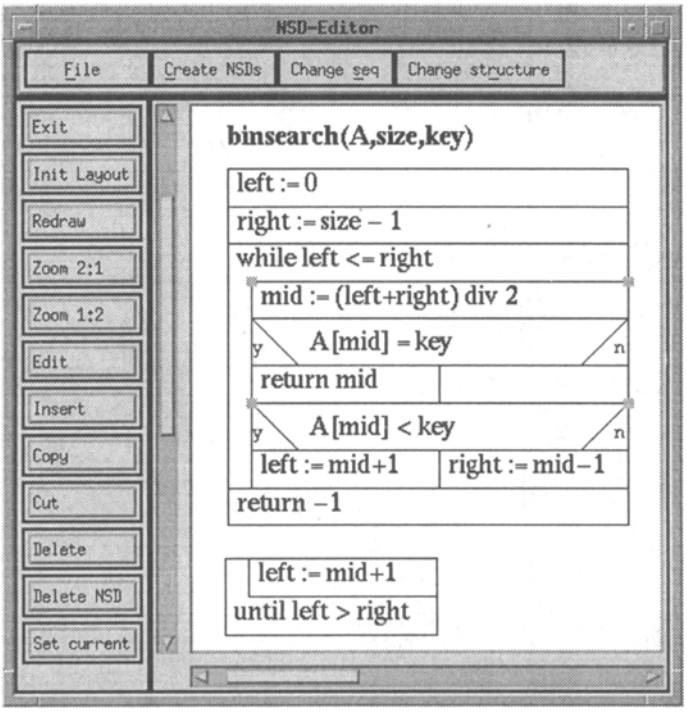

Fig. 2. *Situation after moving the group selected in Fig. 1 to the beginning of the while.*

by pressing the mouse button over the *if*'s triangle with '*y*' inside, dragging it to the right, and releasing it over the corresponding '*n*' (or the other way around). Another example of direct manipulation is to change a *while* into an *until* by moving the *while*'s condition down to the end of the *while* and releasing the mouse button there. Similarly, an *until* can be modified into a *while*.

3 The generator *DiaGen*

DiaGen generates a diagram editor for some diagram class (e.g., NSDs) from a specification, which consists of the parts shown in Fig. 3. A major part is the grammar for the diagram class. Section 3.1 will describe a sample grammar. Some particular diagram shown on the screen is internally represented in the diagram editor by a derivation graph, which is the main data structure (see Fig. 3). Layout conditions are attached to grammar productions in the specification. They constrain the values of layout attributes and thus determine a diagram's layout. The terminal symbols in a derivation graph are mapped on the

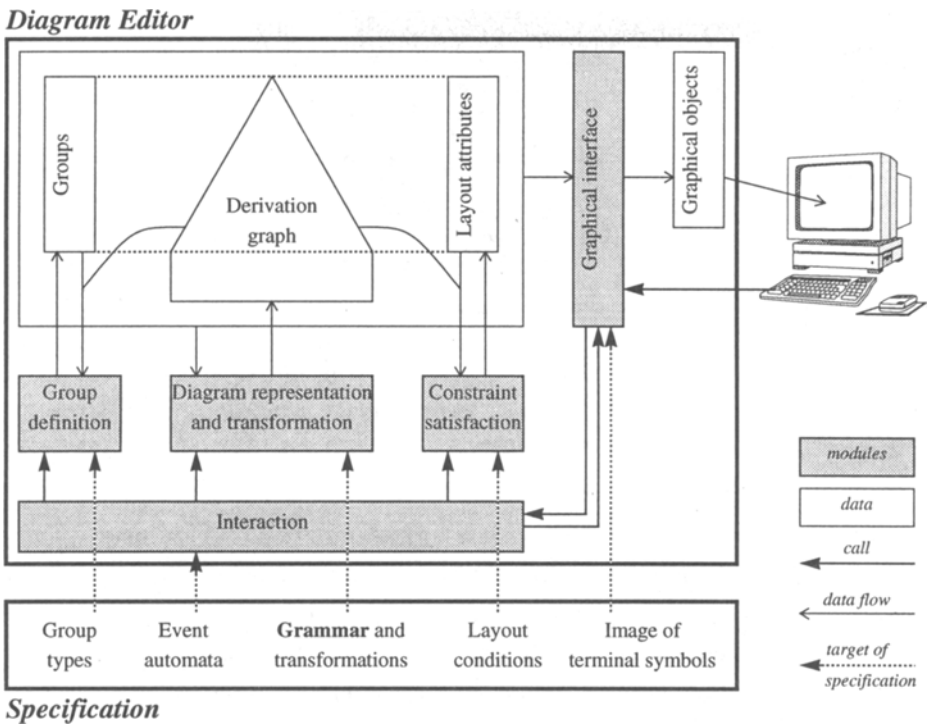

Fig. 3. *Overview of a diagram editor generated with DiaGen and its specification.*

screen. Their image is composed of primitive elements (lines, text, etc.) and also part of the specification. The remaining parts of a diagram editor's specification in Fig. 3 are needed for user interaction.

3.1 Diagram structure and layout

A diagram class, i.e., the syntactic structure of diagrams which will be edited, is specified by a hypergraph grammar. 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 [1]. Each (hyper)edge has a type and a number of connection points that determine how many nodes the hyperedge is connected to. We say the edge *visits* these nodes. The familiar directed graph can be seen as a hypergraph in which all (hyper)edges visit exactly two nodes.

Context-free hypergraph grammars are analogously defined as context-free (string) grammars and have similar properties [3]. Terminal and nonterminal hyperedges are used in hypergraph grammars instead of alphabet symbols in

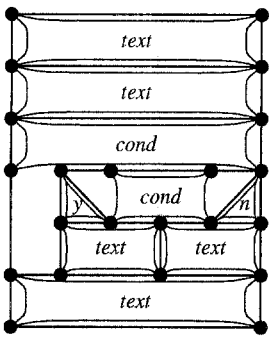

Fig. 4. *Hypergraph corresponding to the upper NSD in Fig. 1. Nodes are shown as black circles. Lines, 'text', 'cond', 'y', and 'n' are terminal hyperedges. Lines have arity 2, 'text' and 'cond' arity 4, 'y' and 'n' arity 3.*

context-free string grammars. In contrast to context-free string grammars, however, hypergraphs can represent multidimensional relationships directly. Nodes in a hypergraph stand for points (e.g., in the plane), hyperedges are diagram elements whose position is given by the nodes being visited by the edge. This is illustrated in Fig. 4, which depicts the hypergraph corresponding to the upper NSD in Fig. 1 (in a slightly simplified manner). The representation in the PAGG and Constraint systems [4, 9], which don't use hypergraphs, would be significantly more complicated.

Each production in a context-free hypergraph grammar consists on its left hand side (lhs) of a hypergraph with a single nonterminal edge and the nodes visited, see Fig. 5. The lhs of production P_1 is the *starting graph* of the grammar. The right hand side (rhs) of every production is an arbitrary hypergraph of terminal and nonterminal hyperedges. Application of a production to a hypergraph is similar to context-free string grammars, too: if the lhs is a subgraph of the hypergraph, this subgraph is removed and replaced by the rhs. The resulting hypergraph is said to be *derived* from the first hypergraph. In order to specify which rhs node replaces which lhs node, corresponding nodes of the lhs and the rhs are labeled with the same letters. A sample derivation sequence is shown in Fig. 6. Context-free hypergraph productions are sufficient for the NSD example. There are also context-sensitive productions for specifying diagram structure, but they aren't addressed here.

The diagram class given by a (context-free) hypergraph grammar is the set of derivation graphs that are derivable from the starting graph and consist of terminal hyperedges only. Just these derivation graphs are mapped on the screen such that the user of a diagram editor only sees valid diagrams. Furthermore,

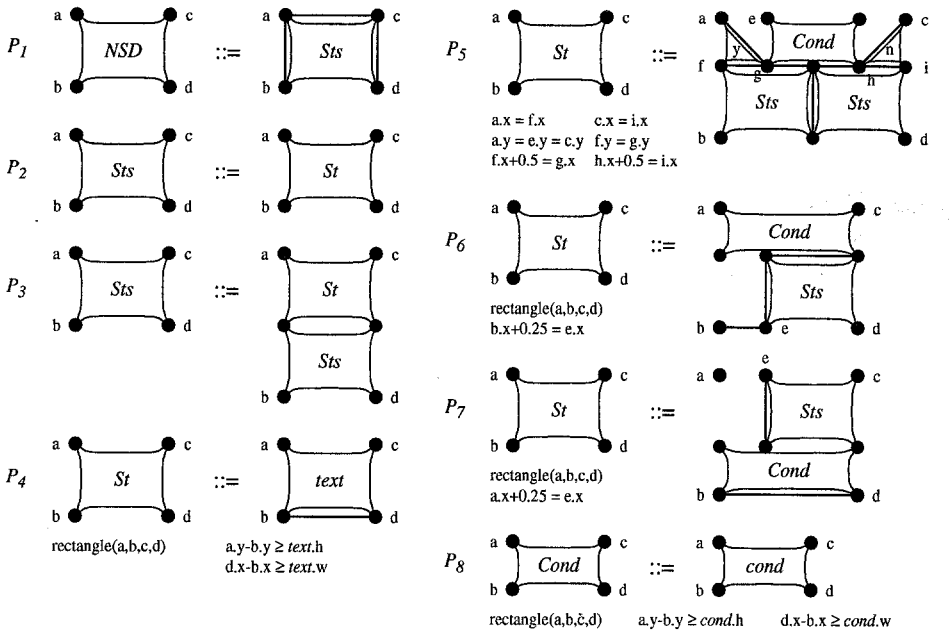

Fig. 5. *Context-free hypergraph grammar for NSDs. Terminal hyperedges are the same as in Fig. 4. Nonterminals 'Sts' and 'St' stand for statement sequence resp. statement, 'Cond' for condition. 'rectangle(a,b,c,d)' is used as shortcut for 'a.x = b.x, c.x = d.x, a.y = c.y, b.y = d.y'.*

the user isn't aware of the grammar productions and deals with diagrams in an application-oriented way, as shown in Section 2.

Hypergraph grammars can describe the syntactic structure of diagrams, but layout requirements aren't included. For context-free (string) grammars this problem can be solved by *attribute grammars* [5], i.e., by assigning attributes to grammar symbols and functional dependencies to productions that determine the attribute values. *Constraints* were made popular by Borning [2] for layout in interactive environments. Constraints are conditions automatically maintained by the system. They permit a very high level specification of layout. The advantage of constraints is that they have a multidirectional nature, whereas in attribute grammars changes are propagated only into one direction. Vander Zanden [9] combined constraints with context-free (string) grammars by assigning attributes to grammar symbols, and by adding constraints, which have to be equations, to each production in order to determine the attributes' values.

Hypergraph grammars are attributed in a similar way. However, not only edges, which correspond to alphabet symbols in context-free (string) grammars,

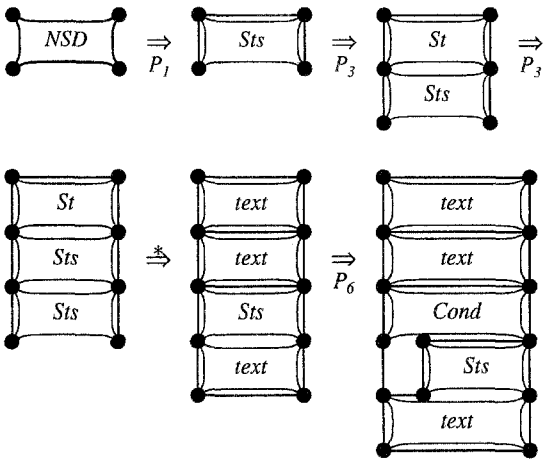

Fig. 6. *Part of the derivation for the hypergraph in Fig. 4 using the productions from Fig. 5.*

carry attributes, but also nodes. Since any number of hyperedges can be connected to a node, node attributes are common attributes for all hyperedges visiting this node. The advantage is that with using hypergraphs only few constraints are needed compared to [9]. Furthermore, constraints can be linear inequalities in *DiaGen*. Equations determine relations between attribute values in a definite way, whereas inequality constraints permit a whole range of values as solutions. This is a convenient way to combine automatic layout of diagrams provided by the system with user-defined modifications. More details on layout and the incremental algorithm for constraint satisfaction built into our generator can be found in [6].

The constraints for layout of NSDs are shown in Fig. 5. Every node has attributes x and y for the node's position. They are refered to by $n.x$ and $n.y$ for a node labeled n. The only edge attributes needed are *text.h*, *text.w*, *cond.h*, and *cond.w* for the mimimal height and width of text blocks and conditions. The values of the edge attributes are determined by the size of the text actually contained in the terminal symbol. There are no constraints needed for productions P_1, P_2, and P_3. Just those in Fig. 5 are sufficient for layout due to node attributes, which are shared among all visiting hyperedges.

3.2 User interaction and transformations

The main feature concerning user interaction is that a diagram can be modified very conveniently in a direct manipulation style by just moving diagram ele-

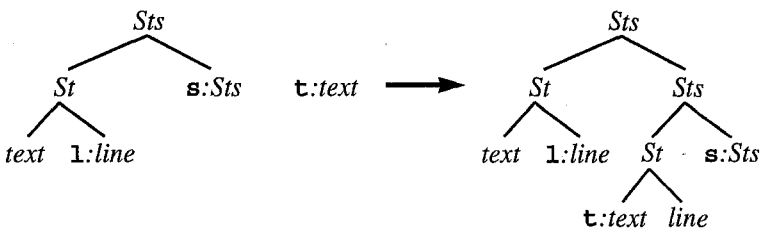

Fig. 7. *Sample transformation rule in the NSD editor for inserting text* `t` *at line* `1`. *For applying this rule the lhs has to match the diagram's current structure. The rhs describes the modification.*

ments around on the screen. This isn't the case for other systems like PAGG or Constraints [4, 9], in which editing operations have to be chosen from a menu. In a diagram editor generated with *DiaGen* the interaction module (see Fig. 3) is responsible for the reaction on a user's actions with mouse and keyboard. They can, e.g., cause selection of diagram elements, layout changes, or transformations of the diagram's structure.

A transformation is a transition from a set of diagrams to another set of diagrams. The structure of one or more diagrams can be changed, but the user of a diagram editor only sees valid diagrams, i.e., diagrams belonging to the specified diagram class. In contrast to [9], transformations in DiaGen can be complex and consist of several primitive steps. This enables more powerful transformations and reusage of primitives. E.g., many tranformations in the NSD editor are based on primitives for deleting or inserting a sequence of (compound) statements.

An example of a simple transformation rule in the NSD editor is given in Fig. 7. Transformation rules describe the modification of the derivation which stands behind each diagram being displayed on the screen. In these rules a derivation (like that in Fig. 6) is written as a tree. A transformation consists of a sequence of rules. The user of a diagram editor indicates by mouse and keyboard actions which transformation shall be applied and the location where this shall be done. In the above example the user would indicate the *text* `t` and the *line* `1` where `t` has to be inserted.[2] The system tries to apply the transformation by matching the left hand sides of the transformation's rules in the sequence given. The first matching lhs determines the modification to be made. Different rules in a transformation take care of different contexts of the positions indicated by the user (here *text* `t` and *line* `1`). In the current implementation

[2] This can be achieved by dragging *text* `t` to *line* `1` and releasing the mouse button there.

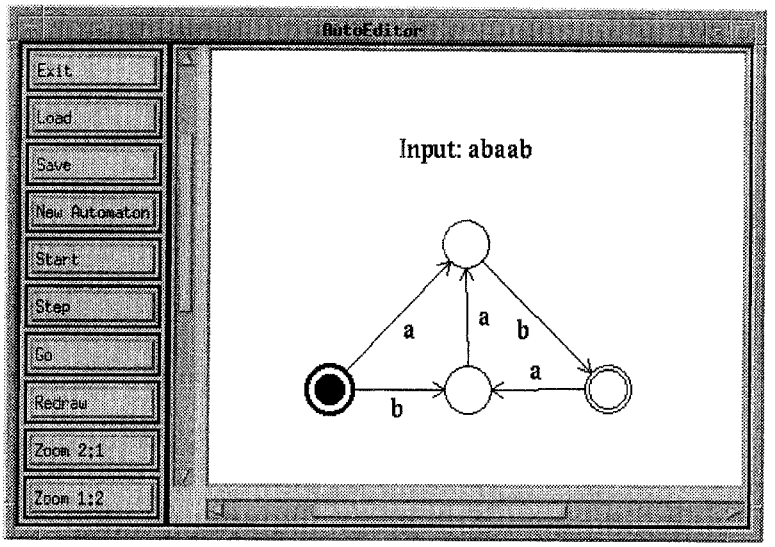

Fig. 8. *Simple automaton after pressing button* `Start`. *The initial state is marked by a thick border line, the final state by a double border line.*

the programmer of a diagram editor, i.e., the user of our generator, has to make sure that transformations, which are part of an editor specification, contain all relevant rules. If the programmer fails to provide a complete set of rules, this only means that for certain contexts a transformation doesn't cause any change. An analysis during parsing the specification would be helpful to prevent such pitfalls.

Transformation rules as illustrated in Fig. 7 are sufficient for tree structured diagrams like NSDs. For other diagram classes, e.g., finite state machines, a context-free hypergraph grammar (see Fig. 5) is too weak. In such cases context-sensitive productions resp. transformations are needed. This is possible in *DiaGen*, but not detailed here since the principle is about the same as for the transformations described above.

In diagram editors generated with *DiaGen* transformations can also be used to specify execution of a diagram, as in our sample editor for finite state machines. A user first constructs an automaton with its states and transitions. Some states can be marked as final states, one as the initial state. Fig. 8 shows the situation after constructing an automaton and pressing button `Start`. The user was asked to provide an input string for the automaton, and the current state was set to the initial state. If there is a suitable transition for the currently first character in the input string, each press on button `Step` causes a transition

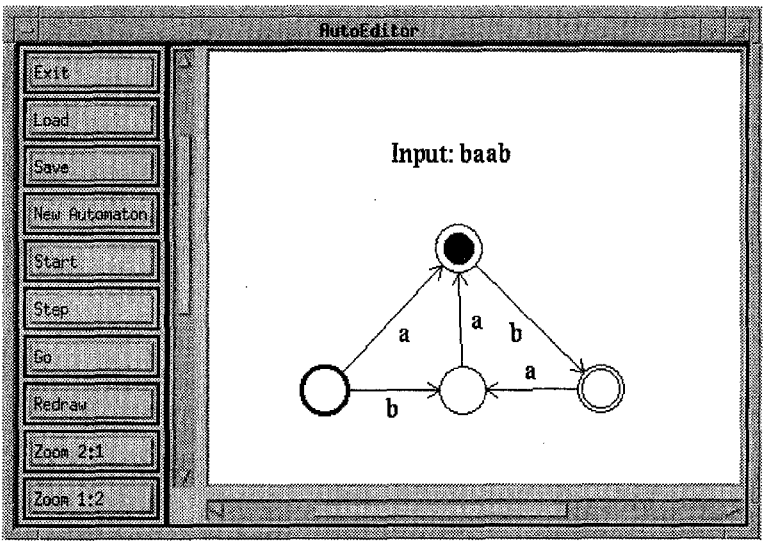

Fig. 9. *Situation arising from Fig. 8 after pressing button* `Step` *once.*

and removes the first character from the input. Pressing `Step` once yields Fig. 9. Button `Go` continues execution until either the input is finished, or a transition for the current input character is impossible.

4 Conclusions

Sophisticated user interfaces should permit the representation and easy manipulation of complex structures by a diagram editor. *DiaGen* has been presented as a tool for generating diagram editors from a specification. A diagram class is specified by a formal model based on hypergraph grammars. This kind of grammar is useful for describing multidimensional relationships between diagram elements. Furthermore, layout of diagrams is defined on a high level by constraints. Editing can be done in a very convenient way by direct manipulation. Complex editing operations and execution of a diagram can be specified in *DiaGen*. Nevertheless, a diagram edited by the user always is in a consistent state, which is a hypergraph in the formal model. This hypergraph can be used by other parts of the application.

A prototype of *DiaGen* has been implemented. Future work will be to generate more sample editors and link them to applications.

References

1. C. Berge. *Hypergraphs.* North-Holland, Amsterdam, 1989.
2. A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.
3. F. Drewes and H.-J. Kreowski. A note on hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Lecture Notes in Computer Science*, volume 532, pages 1–11. Springer, 1991.
4. H. Göttler. Graph grammars, a new paradigm for implementing visual languages. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 1989.
5. D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
6. M. Minas and G. Viehstaedt. Specification of diagram editors providing layout adjustment with minimal change. In *Proc. 1993 IEEE Symposium on Visual Languages*, pages 324–329. IEEE Computer Society Press, 1993.
7. A. Morse and G. Reynolds. Overcoming current growth limits in UI development. *Communications of the ACM*, 36(4):73–81, April 1993.
8. B.A. Myers, D.A. Giuse, R.B. Dannenberg, et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–84, November 1990.
9. B.T. Vander Zanden. Constraint grammars - a new model for specifying graphical applications. In K. Bice and C. Lewis, editors, *Proc. CHI'89*, volume 20 of *SIGCHI Bulletin*, pages 325–330, March 1989.