To appear in Proc. UNIGRA 2001-Workshop, ENTCS, March 2001

Specifying and Implementing Visual Process Modeling Languages with DiaGen ⋆

Mark Minas [a] Berthold Hoffmann [b]

[a] *Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg, Martensstr. 3, D-91058 Erlangen, Germany*

[b] *Technologiezentrum Informatik, Universität Bremen, Postfach 330 440, D-28334 Bremen, Germany*

Abstract

This paper describes how a diagram language can be specified, based on graphs, graph grammars, and transformation rules, and how the diagram editor generator DiaGen generates a diagram editor from such a specification. DiaGen can be applied to practically every visual language, and to visual process modeling languages in particular. This is demonstrated with an editor and animator for statecharts.

Key words: visual language, diagram editor, diagram animation, graph grammar, graph transformation

Visual languages are more and more popular for modeling software in general, like Uml[1], and for process modeling in particular: *Statecharts*, for instance, are well-accepted for modeling reactive systems [2,3] and have become part of Uml. *Petri nets*, another visual notation for process modeling, lend themselves to proving lifeness or deadlock-freedom by model checking. However, it is still far from easy to implement tools for visual languages from scratch, and statecharts and Petri nets are no exception with this respect.

It has already been shown that both statecharts and Petri nets can be modelled by graph transformation systems [4,5]. Here we show that graph transformation systems can be useful for more than just conceptual meta modeling: Graphs, graph grammars, and graph transformations may be used to define visual (process modeling) languages, and this forms the basis for generating tools that implement such languages. This is demonstrated by specifying and implementing a statechart editing and animation tool with the diagram editor generator DiaGen.

⋆ Support by the Esprit Working Group AppliGraph is gratefully acknowledged.
Email addresses: minas@informatik.uni-erlangen.de (Mark Minas), hof@tzi.de (Berthold Hoffmann).

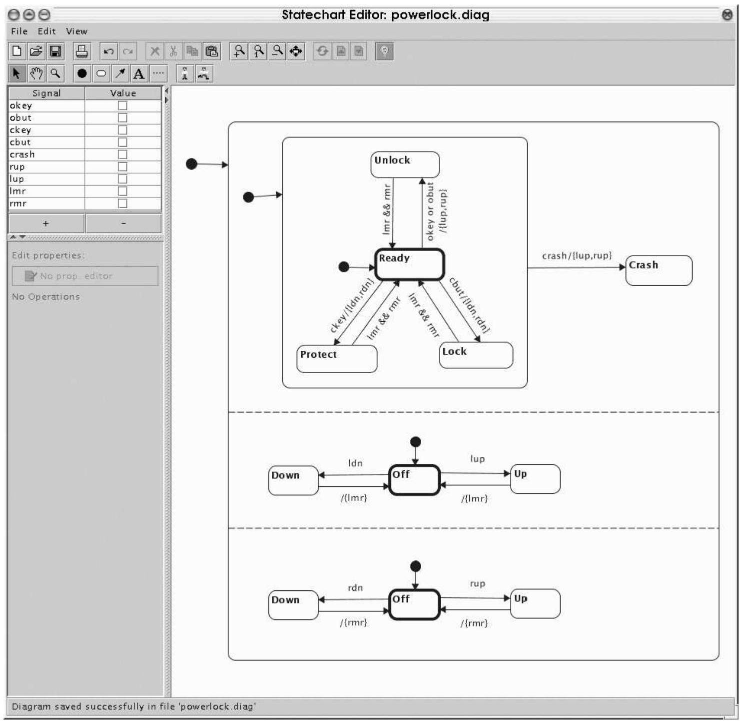

Fig. 1. Snapshot of the statechart editor and animator

Fig. 1 shows a snapshot of this tool. The window consists of a canvas on which a statechart is drawn, a control panel for displaying the status of the chart, and menu entries for operations. [1]

Fig. 1 and the simpler statechart in Fig. 2 show the main features of statecharts [2]: Each statechart is a hierarchical finite state machine where each state (*or-state*) may contain a statechart of its own. A state which does not contain another statechart is called *basic*. So-called *and-states* consist of several compartments which are separated by dashed lines. Each of these compartments contains a statechart, and they are all active simultaneously if the and-state is active. *Initial states* are drawn as black bullets. Transitions between states have annotations e/f, indicating that they will consume the *input events* e, and signal the *output events* f. [2] Transitions may cross hierarchy borders, as Fig. 2 shows.

The rest of this paper is organized as follows: The next section briefly introduces the diagram editor generator DiaGen. Section 2 surveys hypergraphs and hypergraph transformation systems as they are used by DiaGen, and Section 3 summarizes the specification of the statechart editing tool, and the major steps of diagram analysis. Section 4 describes how the editor has been extended by an animation that models the semantics of statecharts. Section 5

[1] An online version of this statechart editing and animation tool can be tried at `http://www2.cs.fau.de/DiaGen/statecharts/`.

[2] Conditional transitions of the form $e[c]/f$ have not yet been implemented.

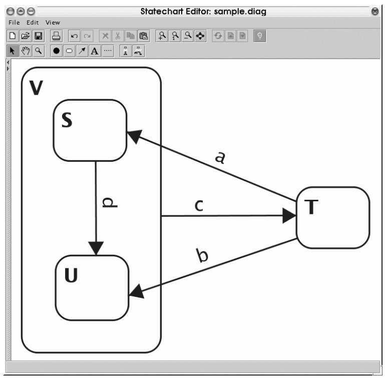

Fig. 2. A sample statechart with cross-hierarchy transitions.

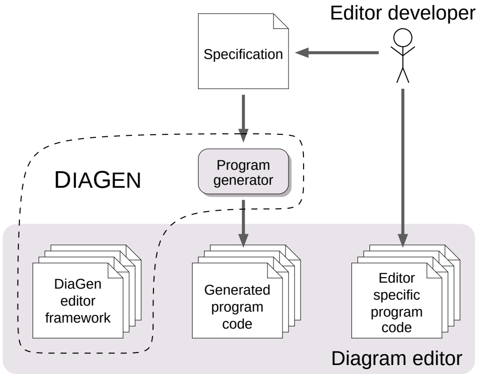

Fig. 3. Generating diagram editors with DiaGen.

reports on related work, and section 6 concludes the paper.

1 DiaGen

DiaGen provides an environment for the rapid development of diagram editors. This section outlines this environment, and how it is used for creating a diagram editor that is tailored to a specific diagram language. DiaGen can be used to create editors for a wide variety of diagram languages, e.g., finite automata, control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts, visual expression diagrams, sequential function charts, and ladder diagrams [6–9]. Actually we are not aware of a diagram language that cannot be specified so that it can be processed with DiaGen.

DiaGen is completely implemented in Java and consists of an *editor framework* and a *program generator*.[3] Fig. 3 illustrates the structure of DiaGen, and how it is used for developing diagram editors. The DiaGen editor framework, as a collection of Java classes, provides the generic functionality needed for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer supplies a textual *specification* for the syntax and semantics of a diagram language. Additional program code which is written "manually" can be supplied too. This may be necessary for the visual representation of special diagram components on the screen, and for processing objects of the problem domain, e.g., for semantic processing when the editor is used as a component in another software system. The specification is then translated into Java classes by the program generator.

The generated classes, together with the editor framework and the manually written code, implement an editor for the specified diagram language. This editor can be used as a stand-alone program, but also as a software component

[3] DiaGen is free software, available from `http://www2.cs.fau.de/DiaGen`.

since the editor framework and the generated program code conform with the *JavaBeans* standard, the software component model for Java.

Diagram editors which have been developed using DiaGen (which are called "DiaGen editors" in the following) provide the following features:

- DiaGen editors always support *free-hand editing* so that the editor user can arbitrarily create, delete, and modify diagram components (states, transitions, and annotation text for statecharts), as with an off-the-shelf drawing tool. After each editing operation, the editor analyzes the "drawing" according to the syntax of the diagram language, and informs the user about syntax errors.
- Well-formed diagrams can be translated into a semantic representation, e.g., a statechart could be translated into program code which implements the statechart specification. This process is driven by the syntactic analysis and makes use of the editor specific program code appearing in Fig. 3. However, this feature is not used in our example.
- The developer of a DiaGen editor may also specify compound operations for *syntax-directed editing.* Each of these operations is geared to modify the meaning of the diagram (e.g. for statecharts, a state could be deleted, with all its incoming and outgoing transitions).
- *Automatic layout* is another optional feature of DiaGen editors. It is obligatory when syntax-directed editing operations are specified. The automatic layout mechanism adjusts the layout of a diagram (i.e., position, size etc. of its components) after any modification. Automatic layout also assists free-hand editing: After each layout modification by the user, the layout mechanism updates the diagram so that its structure remains unchanged. DiaGen offers constraints for specifying the layout mechanism in a declarative way [9], and a programming interface for plugging in other layout mechanisms.

The following sections briefly survey the main concepts of DiaGen, and explain the statecharts editor which has been generated with DiaGen.

2 Hypergraphs and Grammars

DiaGen editors use hypergraphs as internal diagram models and hypergraph grammars as a means for syntax specification. This section briefly surveys these concepts.

Each *graph* consists of a set of labeled nodes and a set of labeled edges. Each edge visits two nodes which need not be different. *Hypergraphs* are a generalization of directed graphs: They consist of a set of labeled nodes and a set of labeled hyperedges. Each *hyperedge* has a fixed number of labeled tentacles which is determined by the hyperedge's label. Tentacles connect the hyperedge with nodes visited by the hyperedge. A regular directed graph is a hypergraph where each hyperedge has two tentacles with labels *source* and *target*. Nodes will be represented by black dots, directed edges by arrows, and hyperedges by boxes containing the hyperedge label. Thin lines are used to represent tentacles connecting the hyperedge with visited nodes. Tentacle labels are omitted in figures if this does not cause confusion.

Hypergraph grammars are similar to string grammars. Each hypergraph grammar consists of two sets of *terminal* and *nonterminal* hyperedge labels and a *starting hypergraph* which contains nonterminally labeled hyperedges only. Syntax is described by a set of *productions* of the form $L ::= R$ with L (left-hand side, LHS) and R (right-hand side, RHS) being hypergraphs. $L ::= R_1 | \cdots | R_n$ is used as an abbreviation for n productions $L ::= R_1, \ldots, L ::= R_n$. A production $L ::= R$ is applied to a (host) hypergraph H by finding L as a subgraph of H and replacing this match by R obtaining hypergraph H'. We say, H' is derived from H (written $H \to H'$) in one step. The grammar's language is then defined by the set of terminally labeled hypergraphs which can be derived from the starting hypergraph in a finite number of steps.

There are different types of hypergraph grammars which impose restrictions on a production's LHS and RHS as well as the allowed sequence of derivation steps. *Context-free* hypergraph grammars are the simplest ones: each LHS has to consist of a single nonterminally labeled hyperedge together with the appropriate number of nodes. Application of such a production removes the LHS hyperedge and replaces it by the RHS. Matching node labels of LHS and RHS determine how the RHS has to fit in after removing the LHS hyperedge. All but the last two productions of Fig. 7 are context-free. Context-free hypergraph grammars *with embeddings* are more expressive than context-free ones. They additionally allow *embedding productions* which consist of the same LHS and RHS, but with an additional ("embedded") hyperedge on the RHS, i.e., this hyperedge is embedded into the context provided by the LHS when applying such a production (the last two productions of Fig. 7). Parsing algorithms and a more detailed description of both grammar types can be found in [7,10].

DiaGen uses hypergraphs as diagram representations and hypergraph grammars for specifying syntactically correct diagrams. The following section describes how these concepts are used by the statecharts editor which has been generated with DiaGen. That section shows also how grammar productions may be annotated by additional *application conditions* that restrict their application during the parsing process (like the path expressions p_1 and p_2 in Fig. 7). Whereas "pure" context-free hypergraph grammars cannot describe the syntax of every diagram language [10], context-free hypergraph grammars with embeddings and application conditions have proven to be a suitable means for diagram syntax specification.

3 Statechart Editing

The statechart modeling tool mainly consists of a free hand diagram editor that translates drawings into a hypergraph model, creates its syntactic structure and thus checks its syntactic correctness with respect to the statechart syntax. As a result of this process, the editor has to provide visual feedback to the editor user if the drawing contains errors. The editor performs this task in a sequence of four steps after each editing operation: scanning, reduction, parsing, and attribute evaluation. These steps are illustrated for the statechart in Fig. 2.

Scanning step: Diagram components (e.g., states, transitions, text which is used as transition annotation, and dashed lines which divide and-states into simultaneously active compartments) have *attachment areas*, i.e., the parts of the components that are allowed to connect to other components (e.g., start and end of a transition). The most general and yet simple formal description of such a component is a hyperedge which connects to the nodes which represent the *attachment areas* of the diagram components. These nodes and hyperedges first make up an unconnected hypergraph. The *scanner* connects nodes by additional edges if the corresponding attachment areas are related in a specified way, which is described in the specification. The result of this scanning step is the *spatial relationship hypergraph* (SRHG) of the diagram. Fig. 4 shows the SRHG of the simple statechart shown in Fig. 2. Nodes are represented by black dots, hyperedges either by gray arrows (relationships between attachment areas) or by rectangles (diagram components) that are connected to

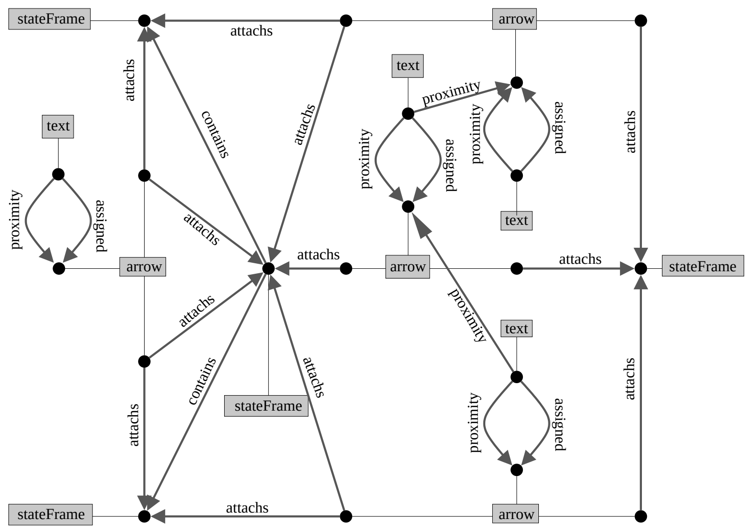

Fig. 4. SRHG of the statechart of Fig. 2.

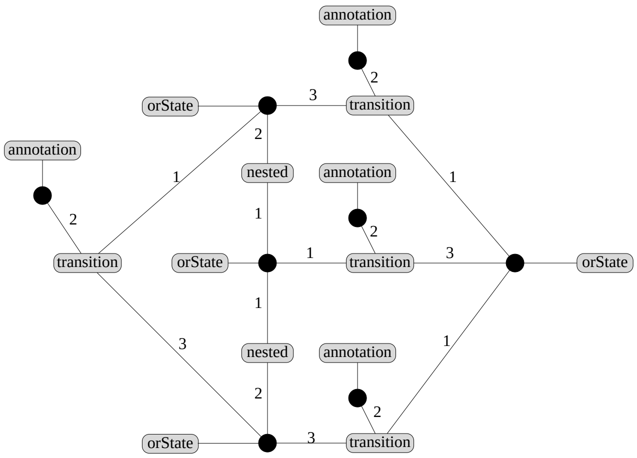

Fig. 5. HGM of the statechart of Fig. 2.

their nodes ("attachment areas") by thin lines. Unary hyperedges `stateFrame` represent or-states, and-states, or basic states, whereas `initState` represents initial states. Transitions are represented by `arrow` hyperedges which visit three nodes: two of them are the ends of the arrow, the third represents the attachment area which is related to annotating text (hyperedge `text`). Relation `proximity` indicates that a text is positioned near an arrow. Fig. 4 shows that `proximity` does not unambiguously assign text to arrows. DiaGen offers special support for dealing with this ambiguity: Assignment is represented by additional hyperedges of type `assigned` which are not generated in the scanning step, but during the following reduction step as discussed later. Finally, `contains` and `attachs` relation edges represent states which contain other states or arrows which attach to states.

Reduction step: SRHGs tend to be quite large even for small diagrams (see Fig. 4). In order to allow for efficient parsing, a reduced *hypergraph model* (HGM) is created from the SRHG first. The reducer is specified by some transformations that identify those sub-hypergraphs of the SRHG which carry the information of the diagram and build the HGM accordingly. This step is similar to the lexical analysis step of traditional compilers. Fig. 5 shows the HGM for the statechart of Fig. 2. Please note the similarity to the original diagram.

Fig. 6 shows four of the total number of 13 reduction rules for statecharts. The two rules on the left show that `stateFrame` hyperedges are "reduced" to `or-state` edges if they do not contain a dashed line that would indicate an and-state. The crossed-out sub-hypergraph represents a negative application context, i.e., a context which must not occur if the corresponding transformation rules shall be applied. Analogously, a `nested` edge is produced if an or-state directly contains another state.

The two rules on the right are somewhat special since they create `assigned`

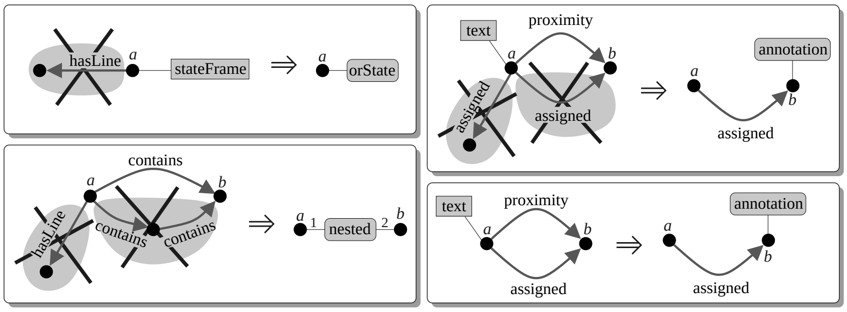

Fig. 6. Four of the 13 reduction rules of the statechart editor.

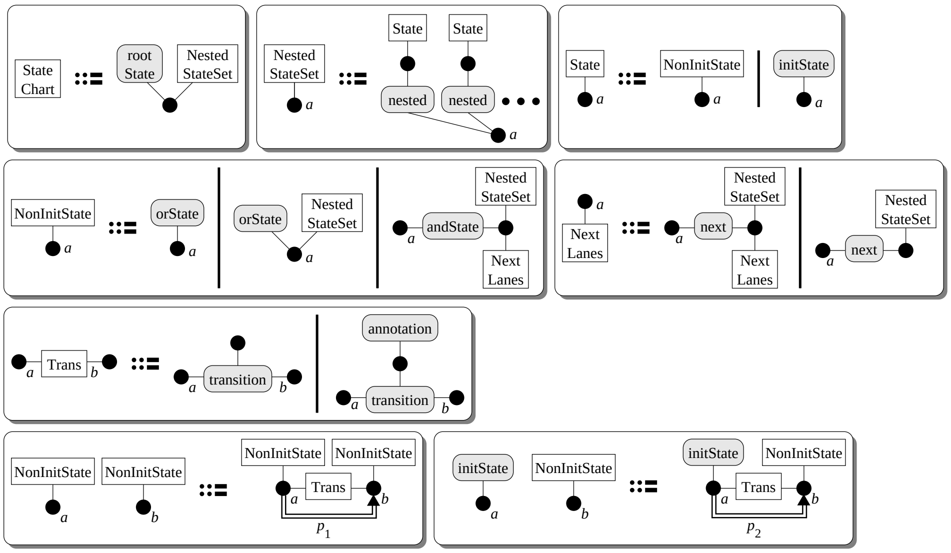

Fig. 7. Hypergraph grammar for statecharts resp. their hypergraph models. Terminal hyperedges are represented by shaded oval boxes whereas nonterminal ones are shown as rectangles. Path expressions p_1 and p_2 are explained in the text.

hyperedges which do not belong to the HGM, but to the SRHG again: They act as a memory for remembering which text has been uniquely assigned to which arrow. Text is assigned to an arrow if either the arrow is the only one which is positioned near the text (top rule) or if there was already a unique assignment in the previous analysis phase, indicated by the already existing `assigned` edge (lower rule).

Parsing step: The syntax of the hypergraph models of the diagram language—and thus the syntax of the language—is defined by a hypergraph grammar. Fig. 7 shows a context-free hypergraph grammar with embeddings[4] for state-

[4] Please note that the production with the `NestedStateSet` hyperedge on its LHS allows an arbitrary number of nested states, represented by `nested`/`State` hyperedges. This more readable notation can be regarded as a shorthand for recursive productions. However, this is actually a special kind of production which allows for more efficient parsing, but a discussion on this topic is beyond the scope of this paper.

charts. The starting hypergraph of the grammar consists of a `StateChart` hyperedge which does not visit any node. Please note the implicit representation of and-states which have not been mentioned in the paragraph on the reduction step: Each of the parallel compartments of an and-state is represented by a `NestedStateSet` hyperedge, and all these hyperedges are linked by `next` hyperedges. It is the task of the reduction step to create this HGM structure from the SRHG. But this is beyond the scope of the paper. Moreover, note the special arrows which are labelled with p_1 resp. p_2 which are actually path expressions similar as in PROGRES [11]. They are necessary for preventing transitions from connecting two states which must not be connected in statecharts. p_1 requires that the first state does not contain the other and and vice versa, whereas p_2 requires that both states are direct sub-states of the same super-state.

Similar to compilers for (textual) programming languages, a hypergraph parser which is built-in into each DIAGEN editor is used for creating the syntactic structure of the HGM of the diagram, i.e., for finding a derivation sequence from the starting hypergraph to the HGM. The parser is capable of identifying syntax errors which are then visualized to the editor user. [5]

Attribute evaluation step: The final step of the translation process creates the semantic representation of the diagram by some kind of syntax-directed translation based on a attribute grammar as it is also used in compilers for (textual) programming languages [12]: terminal and nonterminal hyperedges are augmented by *attributes*, and hypergraph grammar productions by evaluation rules.

No real attribute evaluation has been specified for the statecharts editor. However, which has not been discussed in this paper, attribute evaluation is used for providing structural information on the statechart being edited to the automatic layout algorithm which takes care of beautifying the statechart diagram [13].

4 Statechart Animation

DIAGEN has mainly been designed to generate editors, i.e. tools that handle the *syntax* of a diagram language. The attribute evaluation mechanism of DIAGEN can then be used to produce an intermediate representation for a diagram that can be read by another tool that models the *semantics* of the diagram language. However, DIAGEN's powerful instructions for generating

[5] Currently, well-formed diagram parts are highlighted. Missing highlighting therefore indicates erroneous diagram parts.

compound syntax-oriented editing commands can be used to implement simple semantic properties of a diagram language as well.

The tool shown in Fig. 1 does not only allow to edit statecharts, but animates their behaviour too. The *status* of a statechart is given by its *active states* (highlighted by thick borders), together with the external and internal events that occurred (which are shown on the control panel left to the drawing canvas). Two operations trigger the animation: *Start* (the icon with the standing man) initializes the status of a chart by activating its top-level initial states, and signalling some external events; *Step* (the icon with the walking man) performs all transitions from active states whose input events have occurred, consumes these events, deactivates their source states, activates their target states, and signals their output events.

The operations have been programmed as compound "editing" transformations that apply basic graph transformation rules, controlled by path-expressions. For simple semantic operations this is quite convenient. For more complex semantic operations, e.g. for detection of anomalies in statecharts, or optimization, this would be a tedious exercise.

5 Related Work

DiaGen is related to other approaches for specifying and generating graphical editors. These approaches can be classified according to their supported editing modes: As mentioned in section 1, diagram editors may support *free-hand* editing or *syntax-directed* (or *structured*) editing. Free-hand editing allows to create and modify diagrams unrestrictedly, but these diagrams may contain errors; syntax-directed editing provides a set of editing operations which transform correct diagrams into other correct diagrams. However, the user is restricted to these operations and the way of editing as it is defined by these operations. Most tools for creating *free-hand* editors analyze diagrams directly and avoid to create an internal model like a graph. Typical examples are Vlcc and Penguins. The first utilizes *Positional Grammars* and an LALR(1)-like parser [14] for specifying resp. checking diagram syntax, the latter *Constraint Multiset Grammars* and a Prolog-like parser [15]. Graph transformation systems, however, are a popular formalism for creating *syntax-directed* editors. Two of the more recent tools are GenGEd which allows to visually specify editing operations by graph transformation rules [16] and VisPro which uses special graph grammars for syntax specification [17]. There are many other approaches for specifying visual languages and creating editors for them (for a survey see [18]), but DiaGen is the only tool which supports free-hand editing as well as syntax-directed editing in the same editor although combining both modes combines their benefits, too. The only similar approach which is based on connected graph grammars has not been realized [19,20].

Recently, there has been some other work which integrates approaches for specifying and generating visual editors with animation concepts. Bardohl et al. have proposed an idea of using graph transformation rules of the GenGEd tool not only as specifications of editing operations, but also of animation steps [21]. This approach is of course quite similar to the one described in this paper. However, GenGEd editors are syntax-directed whereas DiaGen editors are free-hand as well as syntax-directed.

6 Conclusions

In this paper we have demonstrated, by an editor and animator of statecharts, that DiaGen may generate tools for modeling syntax and semantics of visual process modeling languages, from abstract specifications based on hypergraphs, hypergraph grammars, and hypergraph transformation. See [22] for a full description of DiaGen's concepts and implementation, and [23] for an editor for another visual process management language (signal-interpreted Petri nets) that has been generated with DiaGen.

Two further aims of our work lie close at hand:

- So far, DiaGen accepts only textual editor specifications as input. Obviously, diagrammatic specifications similar to those used in Fig. 6 and 7 would be easier to use. It should come to no surprise that we intend to use DiaGen itself for constructing an editor for such a visual notation.
- We are now designing DiaPlan, a *diagram programming language* that is visual, rule-based, and object-oriented. DiaPlan shall allow to specify more complex animations for diagrams that can then be smoothly integrated into diagram tools. (See [24,25] for details.) The language has still to be defined precisely, and then a compiler into instructions of DiaGen's transformation engine shall be implemented.

References

[1] UML documentation, Rational Software Corporation, [`http://www.rational.com/uml`].

[2] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.

[3] D. Harel, A. Naamad, The Statemate semantics of statecharts, ACM Transactions on Software Engineering and Methodology 5 (4) (1996) 293–333.

[4] H.-J. Schneider, Describing systems of processes by means of high-level replacement, in: Ehrig et al. [31], Ch. 7, pp. 402–450.

[5] H. Ehrig, M. Gajewsky, F. Parisi-Presicce, High-level replacement systems applied to algebraic specifications and Petri nets, in: Ehrig et al. [31], Ch. 6, pp. 341–399.

[6] M. Minas, G. Viehstaedt, DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams, in: VL'95 [26], pp. 203–210.

[7] M. Minas, Diagram editing with hypergraph parser support, in: Proc. 1997 IEEE Symp. on Visual Languages, Capri, Italy, IEEE Computer Society Press, 1997, pp. 230–237.

[8] M. Minas, Creating semantic representations of diagrams, in: Nagl and Schürr [27], pp. 209–224.

[9] O. Köth, M. Minas, Generating diagram editors providing free-hand editing as well as syntax-directed editing, in: Proc. International Workshop on Graph Transformation (GraTra 2000), Berlin, 2000.

[10] R. Bardohl, M. Minas, A. Schürr, G. Taentzer, Application of graph transformation to visual languages, in: Ehrig et al. [28], pp. 105–180.

[11] A. Schürr, A. Winter, A. Zündorf, Progres: Language and environment, in: Ehrig et al. [28], Ch. 13, pp. 487–550.

[12] A. V. Aho, R. Sethi, J. D. Ullman, Compilers – Principles, Techniques, and Tools, Addison-Wesley, 1986.

[13] S. S. Chok, K. Marriott, T. Paton, Constraint-based diagram beautification, in: Proc. 1999 IEEE Symp. on Visual Languages, Tokyo, Japan, IEEE Computer Society Press, 1999.

[14] G. Costagliola, A. De Lucia, S. Orefice, G. Tortora, A parsing methodology for the implementation of visual systems, IEEE Transactions on Software Engineering 23 (12) (1997) 777–799.

[15] S. S. Chok, K. Marriott, Automatic construction of user interfaces from constraint multiset grammars, in: VL'95 [26], pp. 242–249.

[16] R. Bardohl, GenGEd: A generic graphical editor for visual languages based on algebraic graph grammars, in: VL'98 [30], pp. 48–55.

[17] D.-Q. Zhang, K. Zhang, VisPro: A visual language generation toolset, in: VL'98 [30], pp. 195–201.

[18] K. Marriott, B. Meyer, K. B. Wittenburg, A survey of visual language specification and recognition, in: Marriott and Meyer [29], Ch. 1, pp. 5–85.

[19] M. Andries, G. Engels, J. Rekers, How to represent a visual specification, in: Marriott and Meyer [29], Ch. 8, pp. 245–260.

[20] J. Rekers, A. Schürr, A graph based framework for the implementation of visual environments, in: Proc. 1996 IEEE Symp. on Visual Languages, Boulder, Colorado, IEEE Computer Society Press, 1996, pp. 148–155.

[21] R. Bardohl, C. Ermel, L. Ribeiro, Towards visual specification and animation of petri net based models, in: Proc. International Workshop on Graph Transformation (GraTra 2000), Berlin, 2000.

[22] M. Minas, Specifying and generating diagram editors, Habilitationsschrift, Universität Erlangen, Germany, [In German] (To appear 2001).

[23] G. Frey, M. Minas, Editing, visualizing, and implementing signal interpreted petri nets, in: Proc. 7. Workshop Algorithmen und Werkzeuge für Petrinetze (AWPN'2000), no. TR 2/2000 in Fachberichte Informatik, Universität Koblenz-Landau, 2000, pp. 57–62.

[24] B. Hoffmann, From graph transformation to rule-based programming with diagrams, in: Nagl and Schürr [27], pp. 165–180.

[25] B. Hoffmann, M. Minas, A generic model for diagram syntax and semantics, in: J. D. P. Polim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, J. B. Wells (Eds.), ICALP Workshops 2000, no. 8 in Proceedings in Informatics, Carleton Scientific, Waterloo, Ontario, Canada, 2000, pp. 443–450.

[26] Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany, IEEE Computer Society Press, 1995.

[27] M. Nagl, A. Schürr (Eds.), Int. Workshop on Applications of Graph Transformations with Industrial Relevance (Agtive'99), Selected Papers, Vol. 1779 of Lecture Notes in Computer Science, Springer, 2000.

[28] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages and Tools, World Scientific, Singapore, 1999.

[29] K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, New York, 1998.

[30] Proc. 1998 IEEE Symp. on Visual Languages, Halifax, Canada, IEEE Computer Society Press, 1998.

[31] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Concurrency, Parallelism, and Distribution, World Scientific, Singapore, 1999.