Specifying Visual Languages with GenGED

Roswitha Bardohl, Karsten Ehrig, Claudia Ermel,
Anilda Qemali, Ingo Weinhold

Technische Universität Berlin
{rosi,karstene,lieske,aqemali,bonefish}@cs.tu-berlin.de

Abstract. This contribution gives an overview about the current concepts of GenGED, an environment for the visual definition of visual languages (VLs). From the visual definition, a VL specification is obtained that serves as a configuration of a VL-specific environment, i.e., the configuration is dependent on the parameters available in a VL specification: GenGED allows for the visual specification of syntax-directed editing, parsing, and simulation as well. In addition to these features, we show how to define an animation view for a certain VL model.
All GenGED features are based on the formal concepts of algebraic graph transformation and graphical constraint solving. Hence, we have a well-defined theory which serves as a basis for proper extensions of GenGED.
Keywords: visual languages, visual specification, editing, simulation, animation.

1 Introduction

The use of visual modeling and specification techniques today is indispensable in software system specification and development, so are corresponding visual environments. As the development of specific visual environments is expensive, generators for visual environments have gained importance, especially in the field of rapid prototyping. Most existing generators like DiaGen [10] rely on a textual specification of a visual language instead of a visual one. However, because of the at least two-dimensional character of visual representations the description by one-dimensional texts is not always adequate.

In this contribution we briefly present the current state of GenGED, developed at the TU Berlin, for visually specifying visual languages (VLs) and corresponding environments [1]. We start with a review on the underlying structure, namely alphabet and grammars based on graph transformation and graphical constraint solving. These structures form the basis for the specification of syntax and behavior of visual models (diagrams over a specific VL, such as statecharts, automata or Petri nets). The resulting specification is the basis for the configuration of a visual environment for (syntax-directed or free-hand) editing and simulation. The different components of GenGED are only loosely coupled to allow the user as much flexibility as possible. Thus, it is not necessary to define a parse grammar if the user wants to have a syntax-directed editor.

Syntax and parse grammars are needed for editing whereas simulation grammars describe the dynamic aspects of the specified system. The simulation of the system's behavior is defined by the means of the VL, e.g. the different states of the system are still given by diagrams over the VL, such as an active state of a statechart or an automaton, or a Petri net with an initial marking. Yet, in order to have an intuitive understanding of a model, it is even better to have an animation view which shows the dynamic behavior directly in the application domain. We sketch our ideas concerning an extension of GenGED towards allowing domain specific animation based on the formal simulation grammar.

The editing, parsing and simulation features are now implemented in the GenGED tool environment, whereas the extension concerning animation is still work in progress. All GenGED concepts are illustrated by the specification of a VL for automata comprising features for simulation and animation.

The paper is organized as follows: In Sect. 2 we briefly review the GenGED concepts which are illustrated by the specification of automata. The automata specification is extended in Sect. 3, where we discuss the specification of syntax-directed editing, parsing and simulation. In Sect. 4 we sketch our ideas for defining animation views of visual models.

2 Review of GenGED Concepts

GenGED is based on the well-defined concepts of algebraic graph transformation. Diagrams are represented by attributed graph structures covering the abstract syntax (the logical language elements) and the concrete syntax (the layout). A graph structure is given by disjoint sets, called *vertices* and unary operations from a source vertex to a target vertex, also called *edges*. Each vertex is typed over a *type graph* (the VL alphabet defining the vocabulary of a VL) such that the operations are structure preserving. The concrete syntax extends the abstract syntax by graphics defining the layout for each symbol type given by the abstract syntax. A graphical constraint satisfaction problem (CSP) over positions and sizes of the graphics defines the spatial relations between different symbols by restricting the scope of constraint variables. The CSP has to be solved by an adequate variable binding in each instance diagram over the alphabet. Thus, the CSP defines layout conditions for diagrams of a VL.

Using syntax-directed editing available in a VL-specific editor, a diagram is edited by applying the graph grammar rules to a given start diagram. The start diagram and the rules are part of the corresponding VL syntax grammar. In the following we illustrate the concepts *VL alphabet* and *VL grammar* by the specification of a VL for automata, our running example.

2.1 VL Alphabets

A VL alphabet establishes a type system for *symbols* (vertices) and *links* (edges) of a specific VL, i.e. it defines the vocabulary of a VL. The VL alphabet is represented by an algebraic graph structure signature and a constraint satisfaction problem [1].

A conceivable alphabet for automata is illustrated in Fig. 1. In the upper part of the figure, the abstract syntax of the alphabet is shown, namely the symbols State, Trans (short for *Transition*), and a Start (resp. Final) marking for a state. Both a State and a Trans symbol are enhanced with data attributes, namely a state name (short SN) and a transition label (short TL). We also introduce already the symbol Active which is used for the simulation (cf. Sect. 3.3). The links are indicated as arcs in the abstract syntax part of Fig. 1.

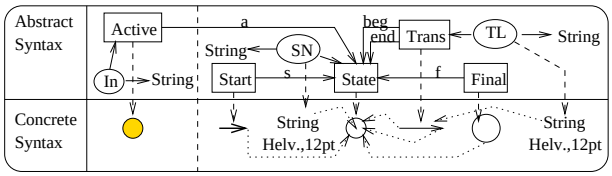

Fig. 1. VL alphabet for automata.

In addition to logical vertex attributes, symbol graphics are represented as a further kind of attributes. In Fig. 1, e.g., the graphic for the symbol type State is given by a circle. Thus, each State symbol (instance of the State type) in a diagram is represented by a circle. In general, the position and size of all instances occurring in a diagram depend on graphical constraints (illustrated by dotted arrows in Fig. 1). The condition that each start and end point of a transition arc must touch the boundary of a state circle is one example for a layout condition defined by constraints in our alphabet for automata.

Fig. 2 depicts the abstract and the concrete syntax of an instance over the automata alphabet modeling a process of the well-known producer/consumer system.

The states of the automaton represent the states of the system: A producer can be idle or busy and deliver a product to a buffer. A consumer can order a product, remove it from the buffer and consume it. The automaton verifies strings of the form $(pdro)^n pdrc$ where each character corresponds to a possible state transition (p: produce, d: deliver, r: remove, o: order, c: consume).

2.2 VL Grammars

Given a VL alphabet, a VL grammar over the VL alphabet consists of a start diagram and a set of rules. Usually, a rule consists of a rule name, optionally a set of parameters, and two graphs, namely a left-hand side (LHS or L) and a right-hand side (RHS or R), which are combined via a rule morphism (a graph structure morphism on the abstract syntax level). Moreover, a rule may be extended by negative application conditions (NACs) and attribute conditions that are boolean expressions over variables and parameters of a rule.

Graph transformation defines a rule-based manipulation of graphs. In GenGED we follow the Single-Pushout (SPO) approach to graph grammars [11] as well as

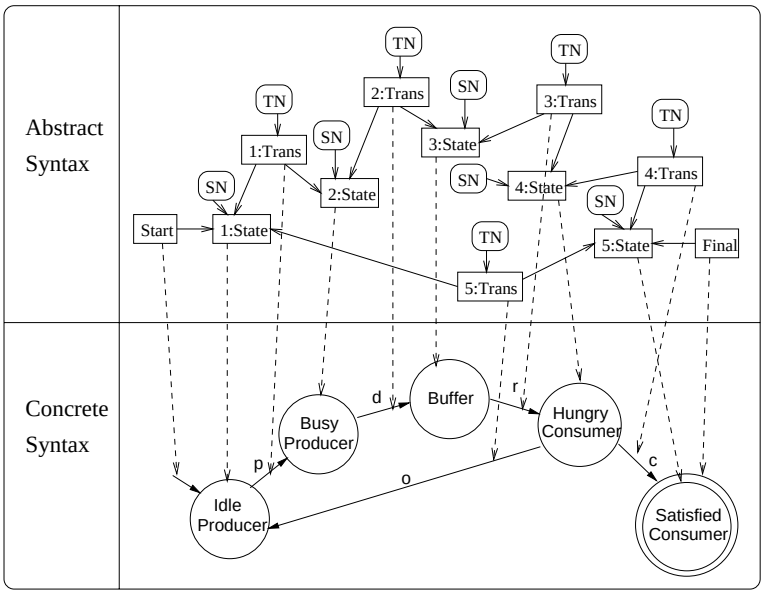

Fig. 2. Automaton modeling the Producer/Consumer system

we support the *dangling condition* well-known from the Double-Pushout (DPO) approach [6] [1]. The application of a rule r to a graph G (*derivation*) requires a mapping (total graph structure morphism) from the abstract syntax level of the rule's LHS to the abstract syntax level of this graph G. Due to the derivation result, the corresponding graphical attributes and constraints are instantiated from the alphabet. The positions and sizes of the graphical objects are calculated by a constraint solver (cf. [9]).

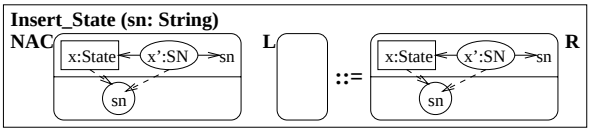

Fig. 3. Syntax-directed editing rule supporting the insertion of a state symbol.

Fig. 3 illustrates a syntax rule for the insertion of a State symbol. This rule contains a rule parameter, namely a state name indicated by the variable sn of type String. The left-hand side L of this rule is empty, i.e., nothing is required for

[1] In contrast to the SPO approach where all dangling edges are deleted implicitly, the dangling condition of the DPO approach forbids the rule application if the transformed graph H contains dangling edges.

applying the rule. By the right-hand side R a state symbol is generated together with a state name. The NAC states that the state names have to be unique in a diagram.

3 VL Specification

In GenGED algebraic graph transformation and graphical constraint solving techniques are combined to support the definition of VL specifications which configure a VL-specific visual environment. Each VL specification consists of a VL alphabet and some kinds of grammars, respectively specifications; cf. Fig. 4.

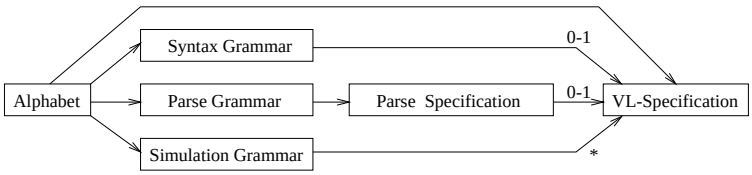

Fig. 4. Workflow for the visual definition of a VL specification.

If only a syntax grammar is in the focus of a VL specification, it should be defined in a way such that it expresses the correct syntax of the VL. Moreover, it should not cover only language-generating rules but additionally language-manipulating rules for comprehensive syntax-directed editing. Unfortunately, such a rule set can be very detailed and large, and an end user [2] may be irritated because of many rules doing more or less the same. Therefore, free-hand rather than syntax-directed editing is conceivable in a VL-specific visual environment. In this case a parse grammar should be available. In order to define complex VLs (like Statecharts in [2]) we propose the combination of a simple syntax grammar together with a parse grammar. A parse grammar may be extended by the definition of a layering function and a critical pair analysis in order to optimize the parsing process. The parse grammar together with these extensions result in a parse specification. Similar to the syntax definition via syntax grammar and parse grammar, the simulation is defined by a simulation grammar.

3.1 Syntax Grammar

Each rule of the syntax grammar provides an editing command. The grammar's start diagram serves as a template for new diagrams to be created. Usually, the editing process begins with an empty start graph. The syntax rules presented in

[2] We distinguish two kinds of users, namely users defining a VL (*language designer*), and those who use a VL specific environment (*end user*).

Fig. 5 are language generating; the rules for modification and deletion of elements work analogously. Fig. 5 illustrates three of four rules of the syntax grammar for the automata example – the first one, Insert_State(), was already given in Fig. 3. Insert_Transition() inserts a labeled transition between two states, Mark_Start() and Mark_Final() make a state the start respectively a final state by attaching the corresponding symbol. The Mark_Final() rule contains an NAC to avoid attaching the Final symbol twice. A similar NAC could be added to Mark_Start() as well, but since our parsing grammar covers this case, we omit it here.

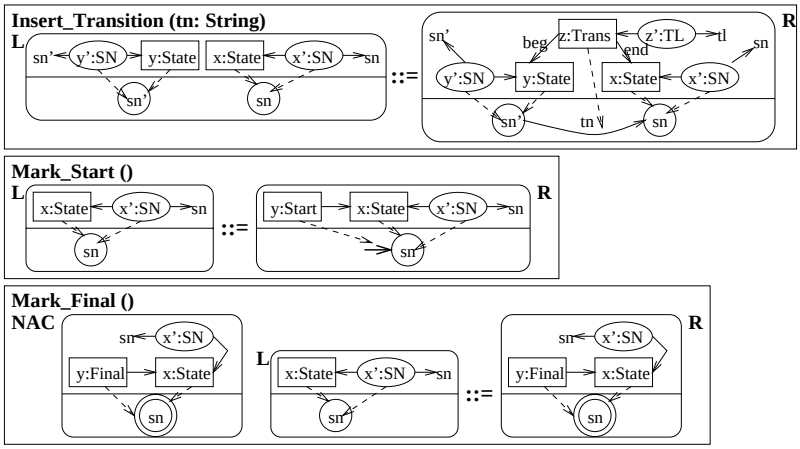

Fig. 5. Syntax grammar for automata.

3.2 Parse Grammar

Using the rules of the parse grammar the parser tries to reduce a given diagram to the grammar's stop diagram. If such a derivation exists then the diagram is accepted, otherwise rejected. An optional layering function assigns an integer number (a layer) to each rule. The parsing algorithm considers only those derivations consisting of rule applications of ascending order. Another optional feature, the critical pair analysis, is used to optimize the parsing process. Usually each possible order of applications of rules (of the same layer) has to be considered. Therefore for each pair of rules it is analyzed whether or not their matches might interfere with each other. In the latter case their application order does not play a role and thus only one single (arbitrary) order needs to be checked.

The parse grammar for the automata example (see Fig. 6) does not require layering. It is quite simple, since the only condition to be checked is whether the given automaton has exactly one start state. The rules Remove_Transition(), Unmark_Final() and Remove_State() resemble the respective inverted syntax rules (not requiring any NACs, if the dangling condition is used). They reduce the

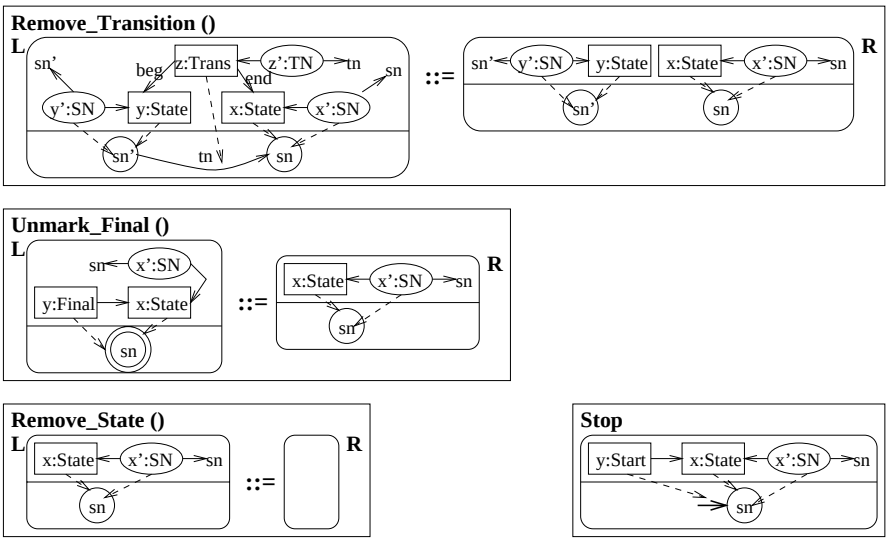

Fig. 6. Parse grammar for automata.

diagram by removing the Trans, Final and State symbols. The only symbols that cannot be removed are State symbols with attached Start symbols. Thus what should remain reducing a correct diagram is the start state, just as given by the grammar's stop diagram.

3.3 Simulation Grammar

In order to visualize which state is the active one, the automata alphabet (Fig. 1) contains an Active symbol, a colored circle. We want to simulate how an automaton reacts to a given input string. Therefore in each step the Active mark should move to the succeeding state. The Active symbol has an In attribute which contains the remainder of the input string still to be processed. Since we do not exclude nondeterministic automata, more than one transition might be triggered at a time. Fig. 7 illustrates the simulation grammar for automata.

The simulation grammar contains only two rules, Init() and Trigger_Transition(). The first one adds an Active symbol to the start state of a diagram. The rule parameter in, the input string, is stored as In attribute of the Active symbol. The second rule moves the Active symbol from the source to the target state of a transition. The attribute condition ensures that the transition label is indeed a prefix of the remaining input string. After the rule application the prefix has been removed.

Given the string to be processed, the simulation grammar calculates a state at which the automaton may terminate, if Init() is applied exactly once and then Trigger_Transition() is applied as long as possible.

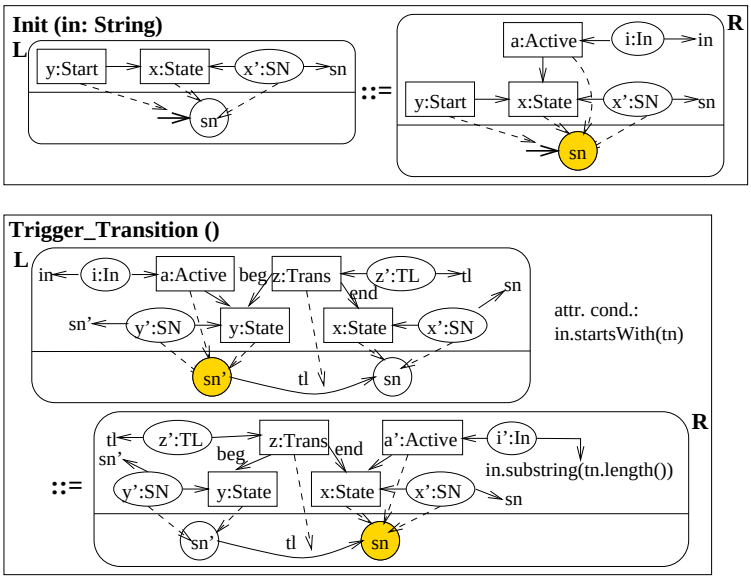

Fig. 7. Simulation grammar for automata.

The concepts of VL specification presented so far are the basis to generate a VL environment supporting editing and simulating specific diagrams (e.g. automata). In general, simulation grammars capture the behavior of formal visual models whose VLs allow the description of dynamic state transitions (like e.g. Petri nets, Statecharts or automata). Each simulation step (the application of a simulation rule) models a state transition.

However, the simulation process is still visualized by sequences of formal VL diagrams, i.e. specific automata or Statecharts are shown. In order to support an intuitive understanding of system behavior, especially for non-experts in the specific formal model, it is desirable to have a layout of the model in the application domain.

4 Defining Animation Views for Visual Models

In order to support an intuitive understanding of system behavior, especially for non-experts in the specific formal model, it is desirable to have a layout of the model in the application domain. In the GenGED approach it is possible to define a relationship between the formal system model and a corresponding layout of the model as icons from the application domain. Such an *animation view* directly shows the states and dynamic changes of the system.

Fig. 8 a) shows a certain state of the producer/consumer automaton from Fig. 2. The Buffer is highlighted as active here. For the animation view we choose the application domain of a kitchen. Producing is visualized as baking and consuming

as eating cakes[3]. Fig. 8 b) shows a snapshot of the active system state in the animation view where the consumer has removed the product from the buffer.

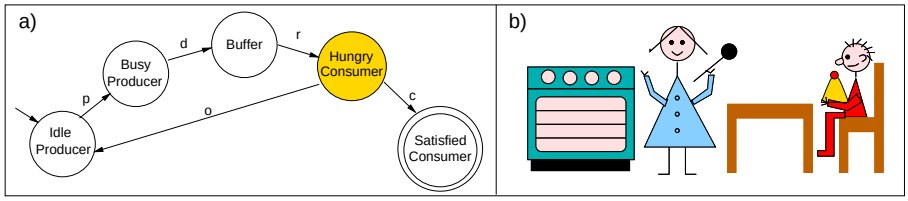

Fig. 8. Automaton and Animation View Snapshot of a state of the Producer/Consumer System

Within the GenGED framework, we suggest a generic approach how to visualize the animation of a system based on a VL specification, a VL model (VL diagrams for the different states of a system), and a VL simulation grammar. The transformation from the layout of the formal model to the layout of the animation view is called *view transformation*. Naturally this view transformation is formalized as a visual grammar based on the VL alphabet which is extended by the new graphics and constraints needed for the domain-specific layout. The simulation rules are transformed into animation rules for the animation view defining the state transitions in the new animation layout. We enforce compatibility between animation and simulation rules by applying the view transformation rules to the LHS and the RHS of each simulation rule instance.

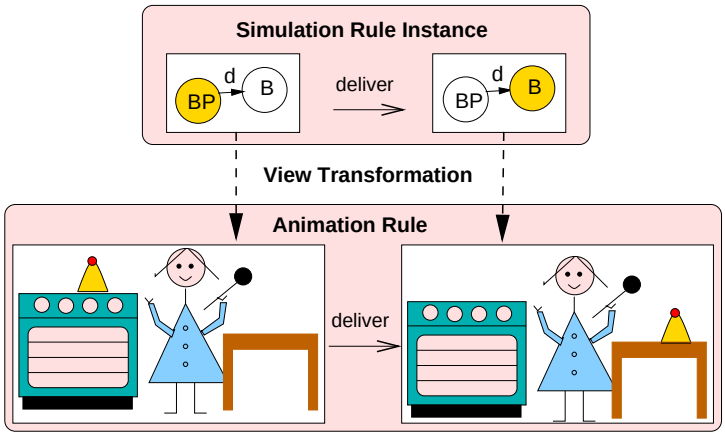

Fig. 9. View Transformation from Simulation to Animation View

[3] This is not meant to be discriminating: Also men bake cakes!

The simulation rule in the upper part of Fig. 9 models the state transition d (for deliver) from the state Busy Producer to the state Buffer by highlighting the current state. The animation rule in the lower part of Fig. 9 shows the same state transition in the application domain oriented layout. The dashed arrows from the simulation rule to the animation rule indicate the formal view transformation between both views.

The implementation of these concepts in the GenGED environment is work in progress and sketched in [7].

5 Conclusion

In [1] GenGED is proposed for the visual definition of visual languages (VLs) and graphical editors supporting syntax-directing editing. Meanwhile, GenGED has been extended in different fields as presented in this contribution. Not only syntax-directed editing may be defined visually but a parse specification and a simulation grammar as well. These specifications based on algebraic graph transformation allow comprehensive editing and analysis as well as they support the visualization of behavioral aspects of VL models.

Apart from the animation concepts, all the proposed concepts are implemented in the GenGED environment (see http://tfs.cs.tu-berlin.de/genged). The development of the animation approach is also joint work in the area of applying graph transformation techniques to Petri nets [5]. More details can be found in [3,7] where different types of Petri nets (i.e. Elementary nets, Place/Transition nets and Algebraic High-Level nets) have been specified as VLs in GenGED .

Future directions concern the development of animation modules for different views of system behavior [4]. The specification of model evolution [8,12] based on two different VLs modeling two layers of abstraction (architecture and components) is an example for the integration of views in one specification. Both VLs are coupled via distinguished (abstract) vertices. This may serve as basis for integrating several VLs in a way that it is possible to handle different kinds of views which are standard practice in the software specification process.

Acknowledgements The research is partially supported by the German Research Council (DFG), and the projects APPLIGRAPH (ESPRIT Basic Research WG), GRAPHIT (CNPq and DLR), and the joint research project "DFG-Forschergruppe Petri Net Technology". Many thanks also to our anonymous referees for valuable comments.

References

1. R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.
2. R. Bardohl and C. Ermel. Visual Specification and Parsing of a Statechart Variant using GenGEd. In *Proc. Symposium on Visual Languages and Formal Methods (VLFM'01)*, Stresa, Italy, September 5–7 2001.

3. R. Bardohl, C. Ermel and H. Ehrig. Generic Description of Syntax, Behavior and Animation of Visual Models using GenGED. Techn. Report No. 2001/19, ISSN 1436-9915, TU Berlin, 2001.
4. R. Bardohl, C. Ermel, and L. Ribeiro. A Modular Approch to Animation of Simulation Models. In *Proc. 14^{th} Brazilian Symposium on Software Engineering*, Joao Pessoa, Brazil, October 2000.
5. B. Braatz, K. Ehrig, K. Hoffmann, J. Padberg, and M. Urbášek. Application of Graph Transformation Techniques to the Area of Petri Nets. In H.-J. Kreowski, editor, *Proc. AGT 2002: APPLIGRAPH Workshop on Applied Graph Transformation*, 2002. To appear.
6. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.
7. C. Ermel, R. Bardohl, and H. Ehrig. Specification and Implementation of Animation Views for Petri Nets. In Weber et al. [13], pages 75–92.
8. C. Ermel, R. Bardohl, and J. Padberg. Visual Design of Software Architecture and Evolution based on Graph Transformation. In *Int. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01), ENTCS Vol. 44, No. 4, 2001.*
9. P. Griebel. *Paralleles Lösen von grafischen Constraints.* PhD thesis, University of Paderborn, Germany, February 1996.
10. O. Köth and M. Minas. Generating Diagram Editors Providing Free-Hand Editing as well as Syntax-Directed Editing. In H. Ehrig and G. Taentzer, editors, *Proc. GRATRA'2000 - Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 32–39. TU Berlin, March 25-27 2000.
11. M. Löwe, M. Korff, and A. Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.
12. J. Padberg, C. Ermel, and R. Bardohl. Rule-Based and Visual Model Evolution using GenGEd. In *Proc. Satellite Workshops of 27th Int. Coll. on Automata, Languages, and Programming (ICALP'2000)*, pages 467–475, Geneva, Switzerland, 2000. Carleton Scientific, Canada.
13. H. Weber, H. Ehrig, and W. Reisig, editors. *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, Sept. 2001. Research Group »Petri Net Technology«, Fraunhofer Gesellschaft ISST.