Electronic Notes in Theoretical Computer Science 72 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 11 pages

AGG and GENGED: Graph Transformation-Based Specification and Analysis Techniques for Visual Languages [1]

R. Bardohl, C. Ermel, I. Weinhold

Dept. of Computer Science
TU Berlin, Germany
{*rosi,lieske,bonefish*}*@cs.tu-berlin.de*

Abstract

The GENGED concepts and environment allow for the visual definition of visual languages (VLs) and to generate VL-specific visual environments for editing and simulation. The editing features capture either syntax-directed editing and/or free-hand editing. In the latter case, a user-defined diagram has to be analyzed in order to check the correctness of the diagram. In addition, behavioral diagrams can be simulated, i.e. the behavior of situations specified by diagrams can be validated. The specification and analysis of VLs by GENGED is based on algebraic graph transformation concepts realized by the AGG system. In this article we give a brief survey on AGG and GENGED.

1 Introduction

Nowadays graphs and graph grammars are used in different areas in Computer Science, as e.g. for software specification or as underlying formalism to specify visual languages (VLs). In connection with suitable control mechanisms graph grammars can describe specific situations on a very high level of abstraction. This is exploited for the specification and analysis of VLs in the AGG/GENGED approach and environment. Moreover, GENGED (short for *Generation of Graphical Environments for Design*) allows for the visual definition of VLs and environments, respectively. This includes the definition of editing operations available in a VL-specific environment (called *syntax-directed editing* or *structured editing*) as well as the visual definition of analysis features given by a parse grammar and a simulation grammar, respectively. From the visual definition, a VL specification is generated which configures a VL-specific environment.

[1] Research is partially supported by the German Research Council (DFG), and the project GRAPHIT (CNPq and DLR).

Usually, a VL covers the language's abstract syntax (the meaning) and the concrete syntax (the layout). In GENGED, the transformation of diagrams and its analysis is supported by the graph transformation engine AGG [16,32] with respect to the abstract syntax. The transformation of diagrams is necessary for syntax-directed editing as well as for parsing and simulation, i.e. for the analysis. The powerful parsing features originally provided by AGG [8,9] (short for *Attributed Graph Grammar* system) have been adopted by GENGED [7,6]. Although AGG comes up with a graphical editor for editing graph grammars, a VL can be defined by the common means of graphs only. Using GENGED, however, a VL with arbitrary concrete layout can be specified: Once defined the symbols and the links of a VL (the VL alphabet), a syntax grammar, a parse grammar and a simulation grammar may be established.
In [5] we have shown how a simple simulation grammar can be defined and used for simulating automata. However, control mechanisms as supported by transformation units introduced for GRACE [2,23] are needed for simulating more sophisticated VLs like statecharts, one is shown in Fig. 1.

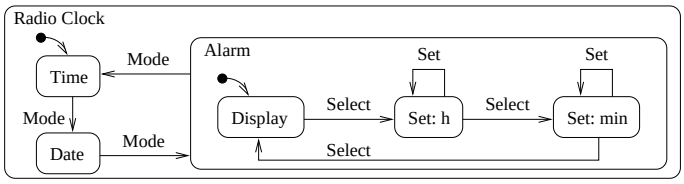

Fig. 1. A sample statechart.

In the following two sections we give a brief summary of AGG/GENGED. Some concepts are explained along the specification of hierarchical statecharts. Due to space limitations we do not present the whole specification which can be found in [10]. In Sect. 4 we discuss some related approaches and in Sect. 5 some final remarks are made.

2 AGG Graph Transformation Concepts

In AGG *typed (labeled) and attributed graphs* consisting of vertices and directed edges are the basis for graph transformation. Fig. 2 illustrates the AGG (abstract syntax) graph of the statechart modeling the behavior of a radio clock shown in Fig. 1. This graph is built up by GENGED editing rules and imported by AGG. In Fig. 2, all rectangles represent attributed vertices (symbols of a VL) and the arrows represent edges (links between VL symbols). We use the abbreviations S for State, H for Hierarchy, and T for Transition. Each state and transition is equipped with a String attribute, namely each state has a certain state name (SN) and each transition holds an event (EV). The attribution of vertices and edges [2] by Java objects and expressions follows the ideas of attributed graph grammars introduced in [26].

A relation between two graphs G and H is expressed by a *graph morphism* which maps the vertices and edges of G to vertices and edges of H. These mappings

[2] Attributes for edges are not used by GENGED.

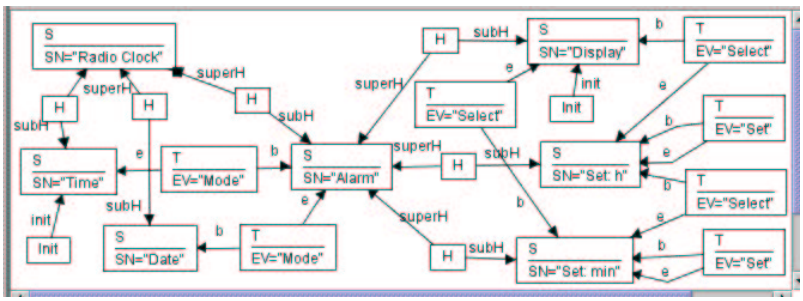

Fig. 2. Sample abstract syntax graph in AGG.

have to be type compatible and attribute values have to coincide also. Fig. 3 shows two graphs and a graph morphism which is illustrated by numbers the vertices are annotated with, e.g. 1:S in G is mapped to 1:S in H.

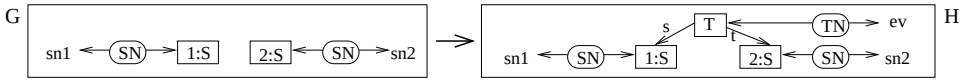

Fig. 3. Sample morphism in AGG .

Graph transformation defines a rule-based manipulation of graphs. In general, graph grammars (consisting of a start graph and a set of rules) generalize Chomsky grammars from strings to graphs. The start graph represents the initial state of a system (here a diagram), whereas the set of rules describes the possible changes. A *graph rule* comprises two graphs: A left-hand side L and a right-hand side R, a named rule morphism, and optionally a set of parameters. Moreover, a rule may contain a set of NACs specifying exactly those fractions of matching situations that *must not* exist for a rule to be applicable. Fig. 4 illustrates a graph rule supporting the insertion of a state together with a state name (the parameter). The NAC requires that the state with the given state name is not already in the graph the rule is applied to.

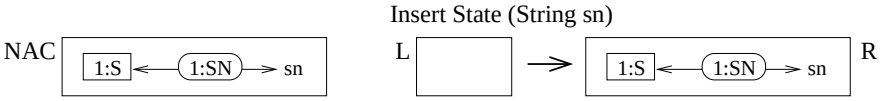

Fig. 4. Sample graph rule with NAC.

The application of a rule to a graph G requires a morphism (also called *match*) from the rule's left-hand side L to this graph G. A match marks the vertices in the working graph that participate in the rule application, namely the vertices in the image of the match. In AGG two kinds of transformation concepts are realized, namely the *Single-Pushout* (SPO) and the *Double-Pushout* (DPO) approach. Applying a rule in the SPO approach, all dangling edges are deleted implicitly, whereas in the DPO approach the application of a rule is forbidden if the resulting graph would have dangling edges (cf. [15] for more details).

The *parsing algorithm* proposed in [8] (which is based on *Contextual Layered Graph Grammars* (CLGG) and critical pair analysis) is already implemented in AGG. This component is called AGG *graph parser*: Assigning rules as well as vertex and edge types to layers such that the layering condition in [8] is satisfied,

the layer-wise application of rules to a given terminal graph always terminates. Roughly speaking, the layering condition is fulfilled if each rule deletes at least one vertex or edge coming from a lower level (*deletion layer*) and creates graph objects of a higher level (*creation layer*). In AGG the definition of layers is supported by a respective dialog; the user is informed about the satisfaction of the layering condition.

Critical pair analysis [31,27] is used to make parsing by graph transformation more efficient: Decisions between conflicting rule applications are delayed as far as possible. This means to apply non-conflicting rules first and to reduce the graph as much as possible. Afterwards, rule application conflicts are handled by creating decision points for the backtracking part of the parsing algorithm. For critical pair analysis of CLGG rules [8], a layer-wise analysis is sufficient, since a rule of an upper layer is not applied as long as rules of lower layers are still applicable.

All the AGG features mentioned so far are used by GENGED which is explained in the following section.

3 VL Specification and Validation in GENGED

The GENGED environment implements concepts for the visual definition of VLs. Based on a visual alphabet where types of symbols and links occurring in a VL are defined by a language designer, several kinds of visual grammars may be given, namely a syntax grammar for syntax-directed editing, a parse grammar and a simulation grammar. Accordingly, the GENGED environment comprises an *alphabet editor* and a *grammar editor*, respectively. The alphabet is the input of the grammar editor, where so-called *alphabet rules* are generated defining the editing commands of this editor. Once an alphabet and possibly some grammars have been defined, a *specification editor* supports the combination of the constituents. If a parse grammar is given, it can be extended by the layering function made available by the AGG graph parser. Also critical pairs may be computed. Moreover, in order to have a controlled execution of diagram behavior, simulation steps can be defined on top of a simulation grammar. The AGG system is used for the transformation of the abstract syntax of diagrams in the grammar editor as well as in a VL-specific environment. The graphical constraints defined with respect to the concrete syntax (the layout) are solved by the constraint solver PARCON [19]. Fig. 5 illustrates the GENGED components, the data flow and the use relations.

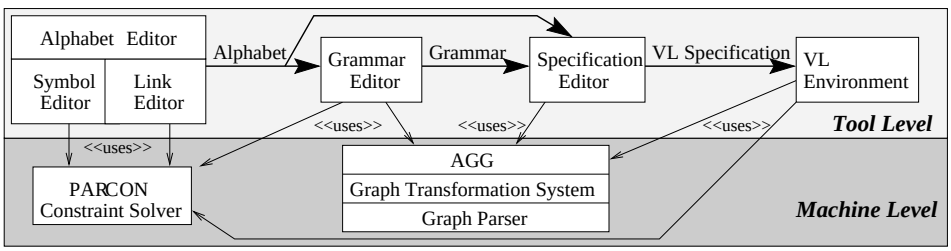

Fig. 5. Components of the GENGED environment.

In contrast to AGG where typed (labeled) graphs are in the fore, in GENGED type graphs [3] are realized that are extended by constraint satisfaction problems according to the layout. These extended type graphs describe the visual alphabet: Symbols like states (S) or data attributes like a state name (SN) of type String represent the (abstract syntax) vertices. These vertices are enhanced by graphical attributes denoting the layout. The definition of visual symbols is supported by the symbol editor; the link editor allows the definition of the links. Each link is represented by a directed edge between two symbols according to the abstract syntax. Graphical constraints specify how the symbol layouts are to be connected according to the concrete syntax.

Fig. 6 illustrates the visual alphabet for our statechart language; for the abstract syntax we use the same notation as in Figs. 3 and 4. According to the concrete syntax, the vertices are enhanced by graphical objects (dashed arrows) and graphical – link – constraints are indicated by dotted lines/arrows. The graphics denoted by PH describe placeholder symbols which are non-visible rectangles in the VL-specific environment. Here, the hierarchy symbol (H) is used in order to allow for nested statecharts. We omit the modeling of parallel states and we do not consider final state markings due to space limitations. Instead we have already introduced the symbols needed for the simulation grammar enclosed by a shadowed ellipse. An active state is denoted by the A symbol and visualized by a rounded rectangle with red color; it is bound to a common state graphic by an overlapping constraint. The other two symbols PA and PD are logical helper symbols. They are needed by the simulation grammar to follow the initial markings and to trigger transitions.

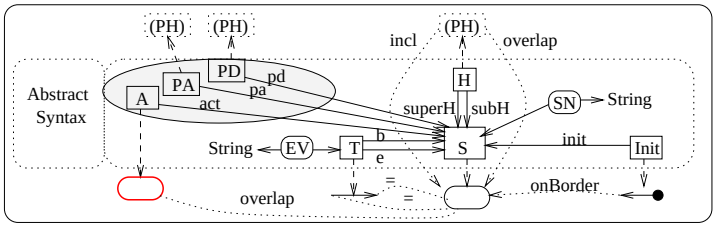

Fig. 6. A statechart alphabet.

The alphabet is the basis to define the distinguished kinds of grammars using the grammar editor. Fig. 7 shows some language-generating editing rules that occur in a syntax grammar. Such rules are defined visually in the GENGED grammar editor by applying alphabet rules generated automatically from the visual alphabet. Using these language-generating rules, abstract syntax graphs as that one shown in Fig. 2 are built up. They are extended by layout information (graphical objects) defining the graphical attributes of each abstract vertex. This means, based on the start graph given by a single state symbol, it is possible to insert sub-states by applying rule InsertSubState and to mark a state as an initial state by rule MarkInit. Each two state symbols may be connected by applying the rule InsertTransition.

[3] For some discussions concerning differences between typed graphs and graphs typed over a type graph the reader is referred to [20].

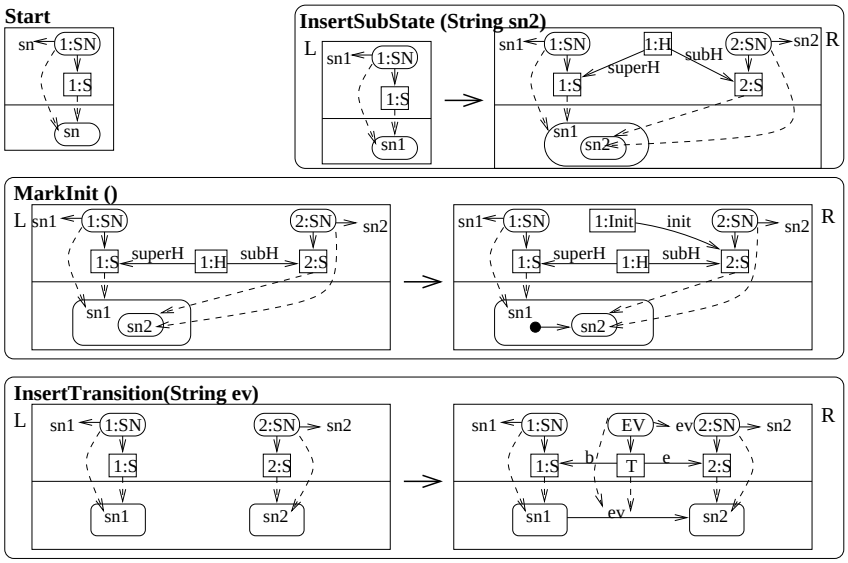

Fig. 7. Some language-generating editing operations.

It is obvious that the initial marking of states can be performed arbitrarily by applying rule MarkInit, i.e. the editing does not conform to the common statechart semantics. In order to allow for syntax checking of diagrams, a parse grammar has to be defined; it is given in [10]. Also the insertion of transition symbols is not restricted by the corresponding rule. The reader is referred to [6] in order to have a closer look at the parse grammar recognizing statecharts with transition symbols that are not allowed. Important is the fact that also the parse grammar is defined visually as well as the syntax and the simulation grammar. However, the definition of the layering function a parse grammar can be extended with, is driven by a respective dialog similar to that one of the AGG graph parser. I.e., in the area of VL analysis, there is no difference between graph parsing in AGG and syntax checking in GENGED.

Being our only means of algorithmic manipulation of diagrams, rule application forms the basis for the simulation concepts. Thus a simulation specification, a container for all simulation related data, consists of a simulation grammar and optionally of a set of simulation steps, which specify how the rules of the grammar have to be applied. A simulation step describes an atomar step to be executed during the simulation process. It is parameterized and undividable, so that after binding values to the parameters and applying the step to a given diagram the resulting diagram is generated without any intermediate results. The core of a simulation step is a simulation expression, a kind of program executed when the step is being applied. Given a diagram and an assignment for the variables used in a simulation expression, its evaluation results in a new diagram and a boolean value. Keeping that in mind the inductive definition of the syntax and semantics of simulation expressions is straight forward.

The basic simulation expression is the **rule**() expression. It has two parameters, a rule of the simulation grammar and an assignment for the rule's parameters. The latter binds the parameters to either concrete values or to variables, precisely,

parameters of the simulation step the simulation expression belongs to. When evaluating a **rule**() expression its rule is applied to the given diagram. If the application succeeds, the evaluation yields the derived diagram and the value *true*, otherwise the unchanged diagram and *false*.

Two simulation expressions a and b can logically be compound to $(a$ **and** $b)$ or $(a$ **or** $b)$. In both cases a is evaluated first and, using the resulting diagram, b afterwards. Thus the evaluation of the complete expression yields the same diagram for **and** and **or**, but the boolean value is the result of a logical *and* respectively *or* of the values yielded by the evaluation of a and b. The $(a$ **andc** $b)$ and $(a$ **orc** $b)$ variants feature a conditional evaluation, that is b is evaluated, only if a yields *true* or *false* respectively.

Finally control flow expressions are realized. Additionally given a simulation expression c, **if** a **then** b **else** c **fi** and **while** a **do** b **done** represent those. Evaluating an **if then else fi** expression, a is evaluated first and if it yields *true*, b is evaluated afterwards, c otherwise. For a **while do done** expression, a is evaluated and, if yielding *true*, b as well and thereafter the whole expression another time. On *false* the evaluation terminates with the diagram the evaluation of a yielded and the boolean result value is *true*, if the body of the loop has been executed at least once, *false* otherwise.

As already mentioned, in GENGED the definition of simulation steps and expressions is based on a given simulation grammar, hence it is supported by the specification editor (cf. Fig. 5). As before, the presentation and discussion of the whole simulation specification for our statechart language is out of the scope of this paper. The reader is referred to [10] for a comprehensive survey.

4 Related Work

In the literature one can find many concepts and tools for the specification and generation of VL-specific environments (cf. [28,11,22]). This fact makes a comparison very difficult. For example, most of the tools do not allow for a visual specification like GENGED; they expect a textual specification for VLs. Another criterion is given by the kind of editing mode (free-hand or syntax-directed) supported in the VL-specific environment, and – if available – the kind of internal representation model.

Closely related to AGG/GENGED are non-commercial graph-based approaches and tools that allow for specifying and analysis of general-purpose VLs. This means that purpose-specific tools like Fujaba [17] (From UML to Java And Back Again) are out of the scope for a comparison.

Most tools for creating free-hand editors analyze diagrams directly and avoid to create an internal representation model like a graph. No internal model is taken into account, for example, in VisPro [36], Penguins [30], and Vlcc [12,33]. Vlcc employs positional grammars and an LALR(1)-like parser. Moreover, in [13] extended positional grammars are introduced such that this approach is no longer restricted to context-free grammars. In Penguins constraint multiset grammars and

a Prolog-like parser are used, whereas in VisPro reserved graph grammars and a graph parser are taken into account [35]. However, in VisPro the set of VLs is restricted to diagrammatic VLs, i.e., symbols can be connected by lines and arrows only.

VLCC [12,33] (extended positional grammars) and DIAGEN [24,14] (hypergraph grammars) use restricted context-sensitive rules to parse VLs. It is mentioned in several publications that all VLs can be captured by the corresponding approaches. However, critical pair analysis as it is supported by AGG/GENGED is not yet regarded. Nevertheless, after parsing, in editors generated by DIAGEN non-correct diagram parts are marked by a specific color, whereas in the GENGED generated editor the user is only informed about errors in the diagram.

Possibly, free-hand editing is desired because a user can create and modify diagrams unrestrictedly; but these diagrams may contain errors. In contrast, pure syntax-directed editing provides a set of editing commands which transform correct diagrams into other correct diagrams; but the user is restricted to these commands. In [1] an integration of both kinds of editing modes is proposed, but it is not implemented yet. The idea of combining both editing modes is captured by DIAGEN [14] as well by GENGED. Additionally, an internal representation model (a graph) is taken into account by DIAGEN as it is done, e.g., by GENGED and by Kogge [25]. However, Kogge allows for syntax-directed editing only.

A further important criterion is given by the layout handling and hence the performance of constraint solving techniques implemented by the corresponding tools. Up to now no efficient constraint solving handling is regarded by GENGED which reduces the performance in the VL-specific environment. Especially, if graph-like languages like statecharts are considered, for the placement of each arrow so-called or-constraints are needed in order to find a correct begin and end point. For each or-constraint the whole constraint satisfaction problem is duplicated and solved. This is better realized by DIAGEN where several layout algorithms are implemented for distinguished kinds of VLs under consideration. However, also DIAGEN has some performance problems concerning constraint solving if large diagrams are drawn. Moreover, DIAGEN supports textual specifications only whereas in GENGED the visual specification of VLs is in the fore.

Concerning the concrete syntax, i.e. the layout, a promising approach is maybe given by VL-Eli [34], which offers a library of predefined *visual patterns* (List, Table, Form, Line, etc.) that can be combined for defining the layout of VLs. The calculation of the layout is then dependent on some abstract syntax attributes (attribute grammars are the underlying formalism). However, in its current state VLs must be specified textually, as well as only syntax-directed editing is supported in a generated editor.

5 Conclusion

In this paper we have given an overview on the specification and validation techniques for VLs offered by the tools AGG [3] and GENGED [18] (both are im-

plemented in Java). GENGED is developed especially for the visual definition of VLs including a visual alphabet and distinguished kinds of grammars, namely a syntax grammar, a parse grammar and a simulation grammar. Based on a given parse grammar, graph parsing and critical pair analysis which is made available by AGG can be used for syntax checking of VLs in GENGED. First experiences of parsing class diagrams [7] and statecharts [7,6,10] have been made. We are going to consider the parsing of further VLs in the future. Additionally, the implemented parsing algorithm has to be compared with related approaches concerning efficiency.

The simulation concepts briefly proposed in this paper are already realized in GENGED. We use graph rewriting rules of the same type as the syntax rules described in this paper to specify the operational semantics of a visual behavior model. Active states are modeled by marking them by a specific active symbol. In this sense, the application of simulation rules simulates the statechart behavior. Moreover, simulation expressions support the controlled execution.

Visual behavior models like statecharts or Petri nets usually are better accepted by practitioners if their behavior is shown in an application-specific layout. On the one hand the behavior of a statechart can be modeled as the change of markings. On the other hand, defining a new layout for states and transitions, the behavior shall be visualized in the application-specific layout which hides the underlying structure. This approach towards animation of visual behavior models has been described for Petri nets in [4], and we plan also to apply it to statecharts. This means we are going to implement the corresponding animation concepts as well as we have to improve our constraint handling concept.

AGG is going to be extended by the graph constraints proposed in [21] to ensure consistency of conditional graph grammars: Graph constraints can express conditions (or even graph parts) as the existence or uniqueness of certain vertices and edges. Such conditions can be transformed into post-conditions that have to be satisfied by the result of each transformation step. Since in GENGED the transformation of the abstract syntax is done by AGG, also this feature can be exploited by GENGED in the future.

Acknowledgment. We would like to thank the anonymous reviewers for valuable comments on a previous version of this paper.

References

[1] M. Andries, G. Engels, and J. Rekers. How to represent a Visual Program? In [28], pages 245–260.

[2] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.

[3] AGG Homepage, http://tfs.cs.tu-berlin.de/agg

[4] R. Bardohl, H. Ehrig, and C. Ermel. Generic Description, Behaviour and Animation of Visual Modeling Languages. In *Proc. Integrated Design and Process Technology (IDPT 2000)*, Dallas (Texas), USA, June 2000.

[5] R. Bardohl, K. Ehrig, C. Ermel, A. Qemali, and I. Weinhold. GENGED – Specifying Visual Environments based on Visual Languages. In H.-J. Kreowski, editor, *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 71–82, 2002.

[6] R. Bardohl and C. Ermel. Visual Specification and Parsing of a Statechart Variant using GENGED. In [29].

[7] R. Bardohl, T. Schultzke, and G. Taentzer. Visual Language Parsing in GENGED. *Proc. of Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'01), Satellite Workshop of ICALP'01*, ENTCS, Vol. 50, 2001.

[8] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as Technical Report SI-2000-06, University of Rom.

[9] P. Bottoni and G. Taentzer. Parsing And Semantics of a Statechart Variant by Contextual Layered Graph Transformation. In [29].

[10] R. Bardohl and I. Weinhold. Specification and Analysis Techniques for Visual Languages in GENGED. Technical Report 2002/13, TU Berlin, 2002. ISSN 1436-9915.

[11] M. Burnett. Visual Language Research Bibliography, http://www.cs.orst.edu/ burnett/vpl.html

[12] G. Costagliola, A.D. Lucia, S. Orefice, and G. Totora. Positional Grammars: A Formalism for LR-Like Parsing of Visual Languages. In *[28]*, pages 171–192. 1998.

[13] G. Costagliola and G. Polese. Extended Positional Grammars. In *Proc. IEEE Symposium on Visual Languages*, pages 171–192, 2000.

[14] DIAGEN Homepage, http://www2.informatik.uni-erlangen.de/DiaGen/

[15] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and 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*, pages 247–312. World Scientific, 1997.

[16] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999.

[17] Fujaba Homepage, http://www.upb.de/cs/fujaba/

[18] GENGED Homepage, http://tfs.cs.tu-berlin.de/genged

[19] P. Griebel. *Paralleles Lösen von grafischen Constraints*. PhD thesis, University of Paderborn, Germany, February 1996.

[20] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and Vertical Structuring of Typed Graph Transformation Systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996. Also as Technical Report 96-22, TU Berlin, 1996.

[21] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, ENTCS, Vol. 2, 1995.

[22] B. Ibrahim. Visual Languages and Visual Programming, http://cui.unige.ch/eao/www/Visual/

[23] H.-J. Kreowski, G. Busatto, and S. Kuske. GRACE as a unifying approach to graph-transformation-based specification. In *Proc. of Int. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01), Satellite Event of ETAPS 2001*, ENTCS, Vol. 44, No. 4, 2001.

[24] 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.

[25] Kogge Homepage, http://www.uni-koblenz.de/ist/kogge.en.html

[26] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. Wiley & Sons Ltd, 1993.

[27] M. Löwe and J. Müller. Critical Pair Analysis in Single-Pushout Graph Rewriting. In G. Valiente Feruglio and F. Rosello Llompart, editors, *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. Technical Report B-19, Universitat de les Illes Balears, 1995.

[28] K. Marriott and B. Meyer, editors. *Visual Language Theory*. Springer, 1998.

[29] M. Minas and A. Schürr, editors. *Statechart Modeling Contest at IEEE Symposium on Visual Languages and Formal Methods (VLFM'01)*, Stresa, Italy, 2001. http://www2.informatik.uni-erlangen.de/VLFM01/Statecharts/.

[30] Penguins Homepage, http://www.csse.monash.edu.au/ css/projects/penguins/

[31] D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M.R Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 201–214. Wiley & Sons Ltd, 1993.

[32] G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *LNCS 1779*, pages 481–490, Springer, 2000.

[33] VLCC Homepage, http:www.dmi.unisa.it/people/costagliola/www/home/ricerca/vlcc/vlcc.htm

[34] VL-Eli Homepage, http://www.upb.de/fachbereich/AG/agkastens/forschung/vl-eli/

[35] D.-Q. Zhang and K. Zhang. Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs. In *Proc. IEEE Symp. on Visual Languages*, Capri, Italy, pages 288–295, 1997.

[36] D.-Q. Zhang, and K. Zhang. VisPro: A Visual Language Generation Toolset. In *Proc. IEEE Symposium on Visual Languages*, pages 195–202, 1998.