GenGEd
A Generic Graphical Editor for Visual Languages based on Algebraic Graph Grammars*

Roswitha Bardohl
Department of Computer Science
Technical University Berlin, Germany
rosi@cs.tu-berlin.de

Abstract

GenGEd *is a generic graphical editor supporting the graphical definition of visual languages. Given an alphabet and rules of a specific visual language* GenGEd *generates a syntax-directed graphical editor for this language.* GenGEd *as well as each visual language defined using* GenGEd *is based on algebraic graph grammars. A sentence is given by a graphical structure consisting of a logical (abstract syntax) and a visual level (concrete syntax). Both levels are connected by layout operations. Visual language rules are defined by graph grammar rules. The underlying logical structure, however, is hidden from the user, but it is essential for a formal presentation and manipulation of graphical structures on both levels. The manipulations are performed by a graph transformation machine working on the logical level, whereas a graphical constraint solver manages the layout the user works with.*
Keywords: graphical definition of VLs; generation of syntax-directed graphical editors; visual specification of VLs by algebraic graph grammars.

1. Introduction

Visual languages (VLs) are used in many application areas: teaching children and adults, programming for non-programmers, adaption of standard software to individual requirements, development of graphical user interfaces, etc. VLs are also used for software development, especially for the analysis and design of software systems. Well-known examples of modeling and specification languages are UML - the Unified Modeling Language ([4]), automata, Petri nets and so on. All these tasks can and should be supported by graphical tools. Common to all graphical tools is the fact, that they offer a VL instead of a textual one allowing the manipulation of visual sentences.

In general, visual statements are difficult to describe textually because of their graphical structure. In the opposite, visual descriptions are mostly not sufficient enough to define all necessities ([17]). Many different formalisms have been proposed for the definition of VLs (cf. [12] for a survey over the common approaches.) In contrast to common approaches including algebraic techniques ([21]) as well as to the graph grammar approaches presented by Rekers, Schürr ([14, 18, 1, 15]) and Minas, Viehstaedt ([22, 13]), our approach uses two different techniques for defining the alphabet on the one side and the language grammar on the other side. We use algebraic specification techniques to define graphical symbols, interrelations and layout constraints in an axiomatic way. This seems to meet the definition of the very basic issues of a visual language in a natural way. The proper language, i.e. its grammatical structure, is described by graph transformation which is again a very natural formalism for this purpose (cf. [3]). In this paper we present GenGEd, an environment supporting the convenient definition of VLs. GenGEd is flexible enough concerning the graphical layout.

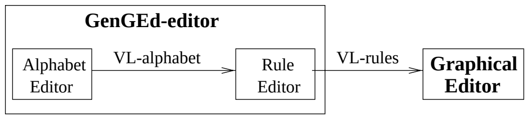

Figure 1. The GenGEd-Environment

The GENGED environment offers tool support for the visual definition of VLs on the one hand and the generation of syntax-directed graphical editors for a defined VL on the other hand (cf. Figure 1). In our approach a VL constists of a VL-alphabet and a grammar, i.e. VL-rules. The alphabet can be defined using an alphabet editor.

*Research partially supported by the German Research Council (DFG), Grant Eh 65/8-1

A rule editor allows the definition of VL-rules. These VL-rules are used as the edit commands of a generated syntax-directed graphical editor in order to manipulate correctly sentences of the defined VL (cf. [3]).

To develop a graphical editor of a specific VL this language must be analyzed first. It is important to work out the visual means of the VL and their interrelations. The visual means are to be determined and mapped onto graph items (nodes, edges, attributes). The interrelations can be defined using graphical constraints. The user has to decide whether the constraint correspond to a source or target operation of an edge, or to the attribution of a node or an edge. This kind of VL-alphabet description establishes a type definition consisting of two representations: the abstract syntax which is connected with layout informations (the concrete syntax) the user works with. VL-rules can be defined under consideration of the alphabet description. These VL-rules are not restricted to be context-free, they may have additional context in their left-hand-side. In this sense, we regard sentences of a defined VL to be type consistent graphical structures consisting of graphical symbols and interrelations between them. Note, with "graphical" we mean both, the graph structure together with the connected visual structure.

Many different tools just as formalisms have been proposed supporting the definition of VLs. In contrast to common approaches GENGED allows the explicit definition of graphical symbols and interrelations for a specific VL. Rekers and Schürr base their language definition on a fixed number of atomic types containing allowed picture objects. Minas and Viehstaedt deals with a textual specification of VLs. G. Costagliola et.all introduced the vlcc-environment supporting the visual definition of VLs ([7, 6]). A symbol editor can be used to define terminal and non-terminal symbols. The defined symbols are then available within a production editor allowing the definition of context-free grammar rules. In contrast, we use algebraic graph grammars which are not restricted to be context-free. Furthermore, we present automatically generated alphabet rules, which can be used for editing the VL-rules. That means, we keep in one formalism given by algebraic graph grammars. Nevertheless, vlcc supports the generation of a graphical environment including a graphical editor and a compiler for the defined VL.

The role of algebraic graph transformation. The aim of GENGED is to support the convenient specification of VLs, and to generate graphical editors. Theoretical support, e.g. allowing analysis of the specified languages is as important as tool support. This theory is based on well developed concepts and tools for algebraic graph transformations ([8, 5, 11]) and can be used for further analysis of the language like consistency checks, layout constraints, etc. ([10, 27]). A more comprehensive discussion on the definition of VLs by algebraic specification techniques and graph grammars in combination with GENGED, e.g. Petri nets, Statecharts, Sequence Diagrams, etc., is presented in ([2]). To our point of view a visual specification is a sentence of the GENGED language consisting of all visual specifications which can be derived by the GENGED-rules. The manipulation of GENGED-sentences, i.e. the visual specification of visual languages, is supported by the integrated graph transformation machine AGG ([11, 16]) together with the graphical constraint solver PARCON ([9]). AGG is responsible for the logical structure of a visual specification, i.e. the graph structure, and PARCON manages the visual structure the user works with.

The paper is organized as follows: Section 2 treats our running example which is a simple VL for class diagrams. Their visual means and rules are analyzed. This is the basis for the usage of GENGED presented in section 3. Especially the editors supporting the VL-definition of our small example are regarded. In section 4 we sum up the results presented within this paper and finish it by some remarks on future extensions.

2. Analysis of a Visual Language

Before using GENGED for the generation of a graphical editor of a specific VL, it is important to analyze this language first. Especially the graphical expressions of the visual means, their interrelations and the intended VL-rules have to be regarded. In this section we present our small example-VL given by simple class diagrams (cf. [4]). This simple VL consists of class diagrams, classes, associations between classes, and association classes. One sentence over the VL is illustrated in Figure 2. The typed elements are denoted by a unique number followed by the corresponding type.

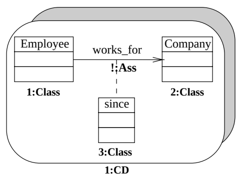

Figure 2. Example of a class diagram

In the following we briefly analyze the simple VL of class diagrams under consideration of the VL-alphabet and the VL-rules. We start with the VL-alphabet. Note, the logical items we use are underlined. They represent nodes, edges or attributes as it is used to build up a type definition, i.e. an alphabet graph. On top of the alphabet discussion the intented VL-rules are presented.

The visual means for a class diagram (CD) is given by two overlapping rounded rectangles. Each class is included in this visual means. A class consists of three parts: one for the class name (CN), one for the attribute list (AL) and one for the method list (ML). These parts are represented by connected rectangles, that include the class name, respectively the attribute and method list. The visual means for an association (Ass) is given by an arrow starting and ending at a class. This arrow should be at least as long as its parameter. The parameter of an association is given by the association name (AN) positioned above the arrow. An association can be attributed by an association class (Class) whose layout is the same as for a class. The connection between an association and an association class is given by a dashed line (AssLine).

After determination of the graphical expressions of the visual means and their interrelations the VL-rules can be introduced. Figure 3 shows a rule representing the insertion of a class diagram without any preconditions in the left-hand-side of the rule nor rule parameters.

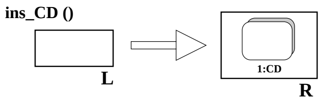

Figure 3. VL-rule: insertion of a class diagram

Figure 4 illustrates the VL-rule that presents the insertion of a class. The precondition is given by the left-hand-side of this rule denoting the existence of a class diagram a class can be inserted in. The rule parameter (string_cn: String_CN) indicates the insertion of a class name. This parameter is used when the rule is applied, i.e. the user is asked to insert a class name.

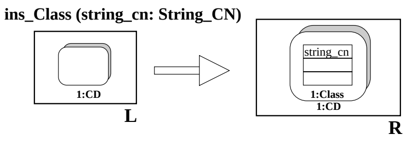

Figure 4. VL-rule: insertion of a class

A further VL-rule, given by Figure 5, puts an association between two existing classes (left-hand-side of the rule). As before, applying this rule the user has to insert an association name.

Figure 6 illustrates the VL-rule describing the insertion of an association class. As for the VL-Rule of Figure 4 the rule parameter (string_cn: String_CN) indicates the insertion of a class name when this rule is applied.

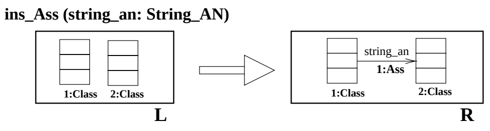

Figure 5. VL-rule: insertion of an association

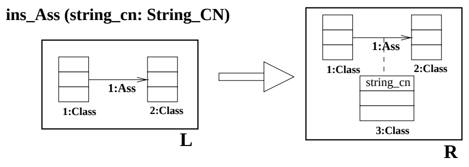

Figure 6. VL-rule: insertion of an association class

In the following section we present the GENGED-environment and show how to define our simple VL using the GENGED-editor.

3. GENGED-Environment

The GENGED environment comprises the GENGED-editor and the corresponding generated graphical editor (cf. Figure 1). The GENGED-editor allows the visual definition of a VL-alphabet and VL-rules. These VL-rules are the edit commands of the generated graphical editor supporting syntax-directed the manipulation of sentences over the VL defined.

The GENGED-editor is divided into an alphabet editor allowing the definition of the graphical symbols and their interrelations, and a rule editor allowing the definition of VL-rules. Based on the alphabet definition some alphabet rules are generated automatically, which are used as the edit commands of the syntax-directed rule editor.

In the following these editors are explained for our running example presented in section 2. We start with the alphabet editor in section 3.1. Then, the automatically generated rules are presented in section 3.2. The rule editor is discussed in section 3.3.

3.1. The Alphabet Editor

The alphabet editor supports the definition of the visual alphabet. This task is supported by three sequently toggled editors illustrated in Figure 7: the GraphicalObjectEditor

(GOE) allows the drawing of graphical objects, i.e. the visual means, the Typi(fying)Editor (TE) supports the connection of the graphical objects with graph items (nodes, edges or attributes) yielding graphical symbols. Thereafter, the Con(necting)Editor (CE) can be used to define the interrelations between the graphical symbols. These interrelations concern to source and target of graphical symbols representing edges, and the attribution of graphical symbols by certain data types. Figure 7 illustrates the components of the alphabet editor using UML (cf. [4]): packages are represented by rectangles, interfaces by circles. The usage of an interface is indicated by a dashed arrow. The solid line indicates the implementation of the interface.

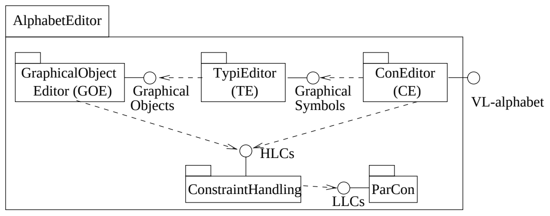

Figure 7. Components of the alphabet editor

The GraphicalObjectEditor (GOE) supports the drawing and naming of graphical objects. Each graphical object consists either of a primitive graphical object as given by a rectangle, circle, arrow, etc. or of several primitive graphical objects which are connected using graphical constraints. The possibility to connect graphical objects by constraints is the main difference between our GOE and other graphical editors as xfig or idraw.

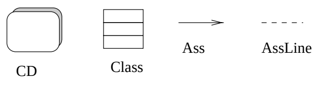

Figure 8. Graphical objects

Figure 8 illustrates the graphical objects used for our example (cf. section 2): the left one shows the visual means for a class diagram, named CD. The two rounded rectangles are connected using the Overlapping-constraint together with certain connection points. The second graphical object denotes a class. It consists of three rectangles which are connected at certain connection points using the EqualPosition-constraint. The binary SameWidth-constraint is used to force the same width for each rectangle. The third graphical object is a primitive one (an arrow), used for the association. The graphical object in the right is additionally a primitive one. This dashed line is used to connect an association with an association class.

For each graphical object that is defined by using the GOE the user has to insert a name for it as presented in Figure 8. The named graphical object is used by the TypiEditor for further process.

The TypiEditor (TE). The graphical objects defined using the GOE are available within the sequently toggled TypiEditor. Their image is put onto buttons with its name below the image. They can be selected and determined to represent a node or an edge. For each determination the user is asked to insert a name. This name is used to define the graph type for the corresponding graphical symbol. The user defined names must be unique.

In addition to the graphical objects defined within the GOE place holders (PH) are available. Such a place holder is only visible within the GENGED-editor. It can be used for the determination of such graph items the user of a generated graphic editor don't like to see but which are necessary to define interrelations using the ConEditor (connecting editor). As before, by selecting the place holder the user has to insert a unique name (for the graph type).

A nice feature is the possibility to determine data types. This comprises the class name, the attribute list and the method list of a class, and the association name of an association. Such data types are implemented components. Within the current implementation the data types String and ListOfString are available. The visual means for String is given by Text and that for ListOfString by List. As for each graphical object, respectively place holder, the user has to insert a unique name for each data type used.

According to our example, Figure 9 illustrates the graphical symbols resulting after using the TE: graphical objects are connected (dashed two-directioned arrows) with graph items, where the node types are represented by rectangles and the edge types by ellipses. The names inside both, the rectangles and the ellipses, denote the names of the corresponding graph types. The four symbols in the left are already discussed for Figure 8; now they are mapped onto node resp. edge types. The four symbols in the right denote the data types. Each symbol is an instantiation of the corresponding implemented data type and gets a unique name: String_CN for the class name, LOS_AL for the attribute list, LOS_ML for the method list of a class, and String_AN for the association name. The three symbols in the lower part of Figure 9 are place holders (PH) we need to build up one alphabet graph, i.e. nodes have to be connected by edges. More concretely, nodes have to be connected by edge operations given by source and target.

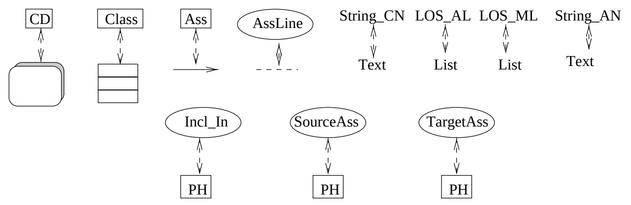

Figure 9. Graphical symbols

The ConEditor (CE). The graphical symbols defined using the TypiEditor are available within the sequently toggled CE. Their image is put onto buttons with their graph type name below. They can be selected and used for the interrelation definition. The interrelation definition is supported by this editor under intensively usage of graphical constraints. Two kind of interrelations are supported: the attribution and the link mode. The first one allows the attribution of nodes and edges by data types. The link mode supports the edge operations, i.e. source and target definition of an edge. The user works with the corresponding graphics and constraints. Certain connection points are related to each graphical object and help to execute these tasks.

Starting with the attribution mode the user has to determine one graphical symbol representing a node or an edge, and one or more of the data type symbols. They can be connected by graphical constraints while the underlying logical operation is built up implicitely. E.g. the symbol for a class consists of three rectangles, each one should include a data type symbol. This task is supported by the binary Include-constraint forcing the user to select two graphical symbols, in this case this would be one of the rectangle and one data type symbol.

The link mode supports the source and target definition for edges. As within the attribution mode the user has to determine one graphical symbol representing an edge, and one or two node symbols. E.g. the edge named Incl_In should connect the symbols for Class and CD (cf. Figure 9). Several constraint definitions are conceivable, we use the following: the place holder representing Incl_In is connected with the three Class-rectangles by the Overlapping-constraint (at the upper left corner). This connection is defined according to the source of the edge. The target of the edge is defined using the Include-constraint corresponding to the symbols for CD and Incl_In.

After using the CE the visual alphabet is defined yielding an alphabet graph as illustrated by Figure 10. According to the syntax discussion in section 1 the figure illustrates the two levels: the upper part corresponds to the logical level (the abstract syntax), and the lower part corresponds to the visual level (the concrete syntax). Both levels are connected by suitable layout operations (dashed two-directioned arrows). The attribution as well as the source and target operations (solid arrows in the upper part) correspond to graphical constraints not solved nor illustrated in the lower part of Figure 10.

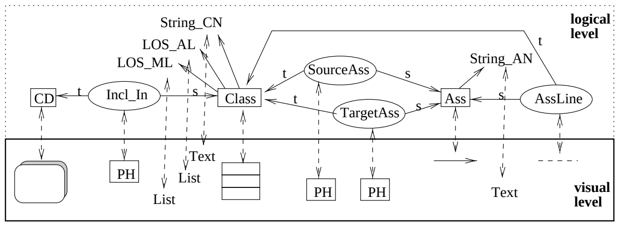

Figure 10. Alphabet graph (without constraints)

The ConstraintHandling component offers a high-level constraint language. Each high-level constraint (HLC) is translated into corresponding low-level constraints (LLCs) of the constraint solver ParCon. The predefined HLCs are collected within a library which can be seen as a super set of the used ParCon constraint language (cf. [9]). These predefined HLCs may not be sufficient enough to support all necessities for the definition of graphical symbols and their connections. Therefore, a user can extend the library by own constraints.

The graphical constraint solver ParCon ([9]) is used as a server. It is implemented in Objective C. All other components discussed in this article are implemented in Java[1].

3.2. Generated Rules

The alphabet graph (also called type graph) presented in Figure 10 is the basis for the automatically generation of alphabet rules. These alphabet rules are used as the edit commands of the rule editor. Some alphabet rule may express already a VL-rule. Nevertheless, for most VL-alphabets not all generated rules can be used as the intended VL-rules.

The alphabet rules are built up componentwise, i.e. for each node a rule is generated describing the insertion of an instance of this node. The left-hand-side of such a “node”-rule is empty, i.e. no precondition exists (cf. Figure 3). This is unlike the “edge”-rules, where at least one node instance is required in the left-hand-side. Figure 11 shows one “edge”-rule, i.e. the rule describing the insertion of

[1] “Java” is a trademark of Sun Microsystems, Inc.

the edge called Incl_In. Additionally, Figure 11 presents both levels: the logical level according to the abstract syntax and the visual level according to the concrete syntax the user works with.

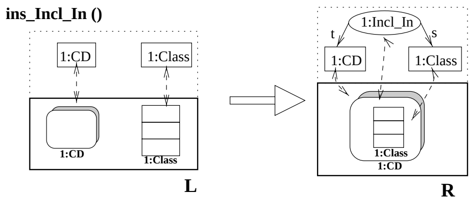

Figure 11. Generated rule: insertion of an edge

As for nodes and edges, data type rules are generated automatically according to each attribution definition using the CE. This means, for each attribution a rule is generated describing the insertion of the corresponding data type. Such a rule is illustrated by Figure 12.

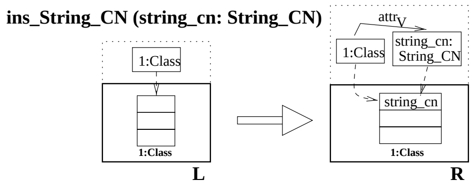

Figure 12. Generated rule: insertion of a String_CN for the class name

The name of an alphabet rule is in addition generated automatically. The rule names are derived by the user defined names, given for the graph types using the TE. Thus, for each graphical symbol an alphabet rule is generated where the name consists of the graph type name prefixed with the phrase ins_. However, insertion rules are generated as well as rules describing the deletion.

3.3. The Rule Editor

The rule editor supports the definition of such VL-rules as illustrated in Figures 4, 5 and 6 of section 2. The edit commands are presented by alphabet rules automatically generated from the defined alphabet. We discuss in this section how VL-rules can be built up using alphabet rules according to Figure 13, where the components of the rule editor are presented.

The RuleViewer supports the visualization of one rule (see upper part of Figure 14) and the selection of particular elements (MatchObject) of the rule's left-hand-side which is necessary for rule application. Figure 14 illustrates one match morphism (structure preserving mapping) consisting of two mappings between the rule's left-hand-side and the working graph, one for 1:CD and one for 1:Class. Applying a rule means informally, that the identified objects of the working graph are removed and substituted by the rule's right-hand-side. The other parts of the working graph (2:Class and 3:Class) are not affected.

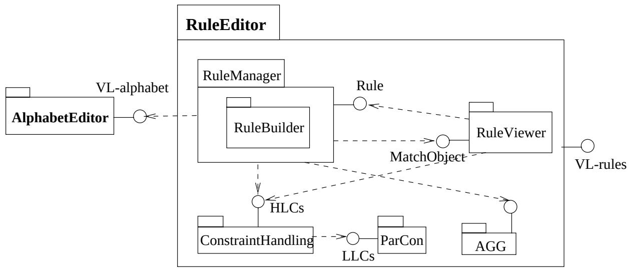

Figure 13. Components of the rule editor

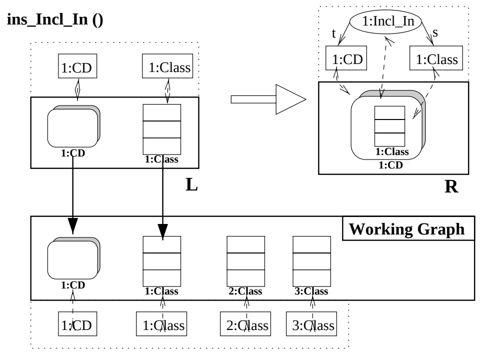

Figure 14. Match morphism for rule application

The RuleManager is the main component of the rule editor. It is the component responsible for all rules, i.e. for the generated alphabet rules and for the VL-rules. Furthermore, this component supports on the logical level the application of rules by graph transformation (cf. [11, 16]), as well as it supports on the visual level graphical constraint solving (cf. [9]).

The RuleBuilder consists of two working graphs, one for the left-hand-side of a VL-rule and one for the right-hand-side. Both sides can be edited. If a VL-rule is complete the components of both sides must be related by a homomorphism (structure preserving mapping) (cf. [3]).

For illustration we construct the VL-rule presented in Figure 4. Note, in contrast to the rule application mentioned above (cf. **RuleViewer**) now we have two working graphs, one for the left-hand-side and one for the right-hand-side of the VL-rule. Starting the construction of a VL-rule, the left-hand-side as well as the right-hand-side of the VL-rule is empty. We apply the alphabet rule ins_CD of Figure 3 to the VL-rule's empty left-hand-side. The left-hand-side of the VL-rule is substituted by the right-hand-side of the alphabet rule. The left-hand-side of the VL-rule is then copied into its right-hand-side. The above mentioned rule homomorphism is defined implicitely. This strategy results in an intermediate state illustrated by Figure 15.

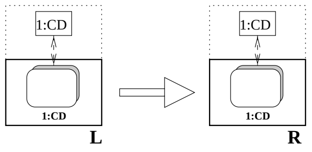

Figure 15. Intermediate state of a VL-rule

As mentioned in section 3.2 for each node an insertion rule is generated automatically, i.e. one rule allows the insertion of a class. This rule will be applied now onto the right-hand-side of the (intermediate) VL-rule presented in Figure 15. After rule application the right-hand-side looks like the left-hand-side of the alphabet rule illustrated in Figure 11. Now we are able to apply this alphabet rule (Figure 11) onto the right-hand-side of the VL-rule because the preconditions (existence of a class diagram and a class) is fulfulled. After rule application the right-hand-side of the VL-rule looks like the right-hand-side of the alphabet rule illustrated in Figure 11. The next step is to apply the rule ins_String_CN (cf. Figure 12) yielding the desired VL-rule (Figure 4). Note, the rule application will done by graph transformation working on the logical level. On the contrary, the constraint solver has to solve the corresponding graphical constraints for the visual level. To store the constructed VL-rule the user is asked to insert a unique rule name. This name is put into the list of all rule names presented in the graphical user interface of the rule editor.

The AGG**-system** ([11, 16]) supports the graph transformation for the logical level of each rule. According to our rule manager it allows to define a complete graph grammar. The rules correspond on the logical level to the rules the AGG-system works with.

The ConstraintHandling component offers a high-level constraint language as already mentioned in section 3.1.

The graphical constraint solver PARCON works as a server as already mentioned in section 3.1.

3.4. Generated Graphical Editor

A syntax-directed graphical editor is generated from the VL-rules, and supports the editing of visual sentences by application of these rules. The rule application of a GENGED-generated graphical editor is performed by the graph transformation machine AGG. AGG is working on the formal based logical structure. The constraint server PARCON solves the graphical constraints. In contrast to the rule editor a graphical editor offers only one working area. This is the area for drawing the visual sentences, i.e. the area the VL-rules can be applied on. Apart from that, a graphical editor has the same behaviour as the rule editor.

VL-rules can be applied by mapping the graphical symbols of the VL-rule's left-hand-side (visualized by the RuleViewer) onto type consistent graphical symbols of the working area. Then, these objects are replaced by the corresponding objects of the VL-rule's right-hand-side. Automatic layout adjustment of the adjacent graphical symbols is forced if one graphical symbol occuring in the working area is moved. A dialogue window asks the user to insert the corresponding value, when applying VL-rules with rule parameters (see e.g. the rule in Figure 4) according to the implemented data types.

4. Summary and Conclusions

In this paper we introduced the GENGED environment supporting the visual definition of VLs. This definition is twofold: it consists of an alphabet and a rule definition. The user-defined VL-rules are used as the edit commands of a GENGED-generated graphical editor. In this framework VLs are represented by algebraic graph grammars and graph transformations. The manipulation of sentences is done by the integrated graph transformation system AGG ([11, 16]) together with the graphical constraint solver PARCON ([9]). Moreover, the theory of algebraic graph transformation offers analysis techniques, for example for consistency checks. Consistency conditions concerning the existence and non-existence of graph parts can be proven for all graphs within a graph grammar produced language ([10, 27]). The idea is to transform a condition into equivalent application conditions for rules in the sense that given a consistent graph a condition satisfying graph transformation with these rules yields a consistent graph again.

GENGED can be used to define several kinds of VLs because it allows any kind of connections. Furthermore, GENGED offers a hybrid language allowing the user to annotate graphical symbols with string or list sentences. These are the attributes of nodes or edges.

A first prototype of GENGED is realized within a student's project at the Technical University of Berlin. It is directly implemented using Java 1.1., so is the graph transformation system AGG. The graphical constraint solver PARCON is used as a server. It is implemented in Objective C. The architecture of GENGED is component-based allowing the independent (re)implementation of their components. The following extensions of the GENGED environment are planned:

- The specification of application conditions,
- the specification of user interactions,
- the specification to connect several generated editors,
- an animation component, useful for diagrams modelling dynamical aspects as firing in Petri nets or automatas.

References

[1] M. Andries, G. Engels, and J. Rekers. How to represent a Visual Program ? In *[19]*, 1996.

[2] R. Bardohl. *A Generic Graphical Editor for Visual Languages*. PhD thesis, Technical University, Berlin, 1998. In preparation.

[3] R. Bardohl and G. Taentzer. Defining Visual Languages by Algebraic Specification Techniques and Graph Grammars. In *[20]*, pages 27–42, 1997.

[4] R. S. Corporation. UML – Unified Modeling Language. Technical Report, Rational Software Corporation, http://www.rational.com, 1998.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars, Volume 1: Foundations*. World Scientific, 1997.

[6] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A Framework of Syntactic Models for the Implementation of Visual Languages. In *[26]*, 1997.

[7] G. Costagliola, S. Orefice, and A. De Lucia. Automatic Generation of Visual Programming Environments. *IEEE Computer*, 28(3):56–66, March 1995.

[8] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *1st Graph Grammar Workshop, Lecture Notes in Computer Science 73*, pages 1–69. Springer, 1979.

[9] P. Griebel. *Parcon - Paralleles Lösen von grafischen Constraints*. PhD thesis, Paderborn University, February 1996.

[10] R. Heckel and A. Wagner. Ensuring Consistency of Conditional Graph Grammars – A Constructive Approach. *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation", Electronic Notes of TCS*, 2, 1995. http://www.elsevier.nl/locate/entcs/volume2.html.

[11] M. Löwe and M. Beyer. AGG — An Implementation of Algebraic Graph Rewriting. In *Proc. Fifth Int. Conf. Rewriting Techniques and Applications, '93, LNCS 690*, pages 451–456. Springer, 1993.

[12] K. Marriott and B. Meyer. Towards a Hierarchy of Visual Languages. In *[19]*, 1996.

[13] M. Minas and G. Viehstaedt. DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. In *[24]*, 1995.

[14] J. Rekers. On the use of Graph Grammars for defining the Syntax of Graphical Languages. Technical Report tr94-11, Leiden University, Dep. of Computer Science, 1994.

[15] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *[25]*, 1996.

[16] M. Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Diplomarbeit, FB 13, Technische Universität Berlin, 1998.

[17] J. Schiffer. *Visuelle Programmiersprachen*. Addison-Wesley, 1998.

[18] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. LNCS 903, 1994.

[19] *Proc. of the AVI'96 Workshop Theory of Visual Languages*, Gubbio, Italy, May 30. 1996.

[20] *Proc. Workshop on Theory of Visual Languages*, Capri, Italy, 27 September 1997.

[21] S. Üsküdarlı. Generating Visual Editors for Formally Specified Languages. In *[23]*, pages 278–285, 1994.

[22] G. Viehstaedt and M. Minas. Interaction in Really Graphical User Interfaces. In *[23]*, pages 270–277, 1994.

[23] *IEEE Proc. Symp. on Visual Languages*, St. Louis, Missouri, October, 4-7 1994.

[24] *IEEE Proc. Symp. on Visual Languages*, Darmstadt, Germany, September, 5-9 1995.

[25] *IEEE Proc. Symp. on Visual Languages*, Boulder, Colorado, September 1996.

[26] *IEEE Proc. Symp. on Visual Languages*, Capri, Italy, September 1997.

[27] A. Wagner. *A Formal Object Specification Technique Using Rule-Based Transformation of Partial Algebras*. PhD thesis, TU Berlin, 1997.