Tool-Based Specification of Visual Languages and Graphic Editors

Magnus Niemann and Roswitha Bardohl

Department of Computer Science
Technische Universität Berlin
{maggi,rosi}@cs.tu-berlin.de

Abstract In this contribution we introduce GenGEd, an environment which is used to interactively specify and generate syntax-directed editors for visual languages.
In analogy to textual languages a visual language is specified by both, an alphabet and a grammar. Hence, the GenGEd environment provides an Alphabet Editor and a Grammar Editor, respectively. The grammar rules defined using the Grammar Editor specify not only language-generating rules but additionally the editing commands of the Graphic Editor for the specific visual language. The language-specific Graphic Editor then can be used in various environments to allow for syntax-directed drawing of diagrams.

1 Introduction

Visual languages are everywhere! Since often a graphical description of a problem or a model provides more readability and takes less space than a textual one, diagrams are used to visualize complex facts. To support the communication within larger communities, diagrams need some syntax and semantics to be understood by all members. In analogy to textual languages (natural but also formal ones) we can summarize syntactical and even semantical information in diagrams.

The graphical capabilities of today's computer systems permit the construction of diagrams completely with a computer, like it is done in architecture and electrical engineering. Nowadays, diagrams are additionally used in computer science for modeling and programming of complex systems. Such diagrams concern *visual languages*. It depends on the purpose of the visual language whether it is called *visual modeling language* or *visual programming language*. Visual modeling languages used for software engineering are, e.g., the Unified Modeling Language (UML) [Rat98] and statecharts [Har87]. Without a doubt, there are needs for editors to draw diagrams in those languages. Furthermore, in order to state about software quality and correctness it is important to draw *correct*[1] diagrams.

[1] Diagrams can be *correct* with respect to some formal specification.

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 456–470, 2000.

The problem with existing tools and editors for, e.g., software development (Rational Rose, Statemate, etc.) and also with so-called visual programming environments (Visual Basic, Delphi, etc.) is that the visual means are tightly integrated in the visual environment. Re-implementation is necessary whenever the concepts of a visual language or the basic visual means change. The GenGEd environment [Bar00,BNS00] on the other hand permits the easy and interactive definition of arbitrary Graphic Editors for visual languages[2].

There are two major ways to build an editor for a visual language according to some specification: The first way is to take a simple graphical editor with which some kind of diagrams based on graphical primitives (like line, rectangle, circle etc.) can be drawn. This can be either some existing vector graphics editor or a slightly modified editor adapted to the symbols we use in our language. To check for syntactical correctness of a drawn diagram we will have to do some scanning- and parsing-like operations on the diagram according to a graphical syntax given in some way. We will denote those editors as *freehand editors*. The second way is to provide the user drawing a diagram with only the operations (insert a symbol, remove a symbol) with which he or she is "forced" to draw syntactically correct diagrams. That circumvents the syntax checking but adds more restriction to the user of such an editor. These editors will be called *syntax-directed editors*.

The path we choose in the GenGEd environment is the second one, i.e., we consider syntax-directed editors. We do this because we can provide the designer of a visual language with powerful means to write grammars – in analogy to formal language grammars – with which we then generate a Graphic Editor for this language. So our environment can be used to specify existing visual languages like graph-like class diagrams occurring in the UML or box-like Nassi-Shneiderman diagrams [NS73], but also to construct completely new visual languages from scratch.

This presentation is built as follows: We start by giving an informal definition about the concepts of visual language specifications underlying the GenGEd environment in section 2. For illustrational reasons we use some features of the well-known visual language of class diagrams [Rat98]. In section 3 we introduce the GenGEd environment which is elucidated by screenshots. Some related approaches concerned in generating editors from visual language definitions will be mentioned in section 4. Concluding remarks will be made in section 5.

2 Concepts of Visual Language Specifications

When we take a closer look on two-dimensional diagrams following a certain specification there are two major parts we have to consider. On one hand we have some graphical *symbols* like classes or associations in an UML class diagram (see fig. 1 (a)). On the other hand there are spatial *relations* such that a symbol

[2] Research is partially supported by the German Research Council (DFG), and the ESPRIT Basic Research Working Group APPLIGRAPH

must be aligned with another one or, more precisely, that an association arrow must start at the border of a class symbol (see fig. 1 (b)).

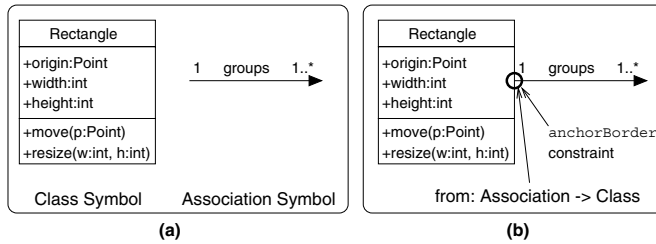

Figure1. Symbols (a) and connections (b) of class diagrams

In formal textual languages, relations between symbols like "follows" are not described explicitly in an alphabet. However, in analogy to formal languages, for a visual language we first need an *alphabet* over which sentences, namely diagrams, can be constructed. In the GenGEd approach, an alphabet keeps all the information about the graphical symbols and furthermore, the possible relations between symbols. We will denote those relations as *connections* because we consider not only spatial relations like the `anchorBorder` constraint in figure 1 (b) but additionally the corresponding underlying structural relationships `from: Association` → `Class`. These structural relationships are used to express the *logical* meaning of a diagram. E.g., according to the UML specification, it is not allowed that an association symbol is "dangling" as illustrated in figure 1 (b). The end of the association arrow has to be connected with a class symbol, too. In order to avoid such *incorrect* constellations, a second structural relationship like `to: Association` → `Class` has to be defined together with suitable spatial relations. Both connections (`from, to`) then ensure that an association symbol is always connected with class symbols.

It is not sufficient to construct diagrams from an alphabet only. And additionally, it is not sufficient to generate a specific Graphic Editor where one can arbitrarily insert and remove symbols and connections. In this case there will be diagrams with illegal syntactical constellations. Like in formal languages, we have to give some *rules* to define insertion and deletion of symbols and implicitly the corresponding connections. The rules together with a *start diagram* (which is analogous to a start sentence in formal languages) form a *grammar*. The grammar rules are used as editing operations on the start diagram[3] – so we will not only give language-generating rules, but also editing rules allowing for changing or deleting symbols and connections in a language-specific Graphic Editor.

The alphabet and the grammar of a visual language establishes a visual language specification over that diagrams can be edited. These concepts are based on the well-established theory of algebraic graph transformation that is fully described in [Bar00]. In the following we do informally explain these concepts which are implemented in the GenGEd environment.

[3] In fact this will be done in the GenGEd environment later on.

2.1 The Alphabet

An alphabet comprises a set of symbols and a set of connections. From the graphical point of view, symbols are expressed by sets of graphical *primitives*, and connections are expressed by sets of graphical *constraints*.

Primitives Graphical *primitives* are simple graphics like lines, rectangles, circles, etc. They all have a location in a two-dimensional space and some properties like width, height, color, starting point etc.

Constraints Graphical *constraints* denote spatial relations between graphics. One constraint may concern to one or more graphics. A constraint is given by a set of equations and inequalities over the graphic's properties. Some examples for constraints are given in figure 2. A *graphically correct* diagram is a diagram where all constraints are satisfied. So we have to make sure that the symbol's properties like size and location are chosen in a way that the corresponding constraints are satisfied. This can be achieved using a constraint solver.

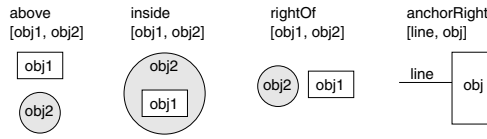

Figure2. Constraint examples

Symbols A symbol consists of a certain symbol name and a symbol graphic. This graphic is a grouping of several graphical *primitives*. The grouping is defined by graphical constraints like that of figure 2. Nevertheless, each symbol graphic consists of a box enclosing the corresponding primitives. This box can be treated like a common graphical primitive.

Connections A *connection* is defined between two symbols with respect to the symbol names and the symbol graphics. According to the symbol names it is a directed connection where one symbol name is in the source and the other one is in the target of the connection (see fig. 1 (b)). Note that due to the underlying theory [Bar00] the insertion of a directed connection on the instance level requires for a symbol that is in the source of the connection the presence of a symbol that is in the target such that the connection is correctly defined. We will call this property "structural correctness". In addition to a directed connection, according to the structural relationship a connection is defined by a set of constraints between the involved symbol graphics.

Data Attributes Some symbols need attributes we cannot handle in a purely graphical way. Such attributes are e.g., class names or association cardinalities as shown in figure 1. Data attributes can be strings, integers, lists or any other

well-defined[4] datatype. Every data type has a set of operations (like `append: StringList String → StringList`) which can be used to code attribute changes (renaming, increasing counters and the like) into our rules. This goes beyond the capabilities of graphic editors which allow only the use and the simple renaming of strings. Nevertheless, each datatype must have a certain layout that depends on the specific datatype. For strings, e.g., the layout information includes text size and text font.

2.2 The Grammar

With the means to describe the graphical structure of diagrams we will now add the concepts used to construct syntactically correct diagrams. This is made possible by the grammar that is based on the language-specific alphabet. The grammar is defined by a start diagram and a set of rules. The rules are not restricted to be context-free; they are context-sensitive and may be enhanced with some application conditions. Moreover, the rules define the editing commands of the aimed Graphic Editor that is generated from the visual language specification consisting of an alphabet and a grammar. These rules can be applied to a given start diagram.

Start diagram A *start diagram* comprises symbols and connections that are uniquely instantiated from the alphabet. Due to the alphabet, each symbol consists of a certain symbol name and a symbol graphic, probably some symbol constraints. Each connection is defined by a directed connection between the involved symbol names and some constraints between the corresponding symbol graphics.

Rules and Rule Application A visual language *rule* consists mainly of a left hand side (LHS) and a right hand side (RHS). A mapping of symbols from the LHS to the RHS (see upper part of fig. 3) indicates that the mapped symbols are preserved when the rule is applied to a given diagram. The connections are mapped implicitly.

Applying a rule to a given diagram, the symbols of the LHS have to be mapped to the symbols of the diagram we want to transform. This mapping is called *match*. The connections are mapped implicitly if there are some. This implicit mapping is called "match completion".

The RHS of a rule contains another diagram comprising all the elements which persist through the transformation (namely those which are mapped from the left to the right) and all elements which are added through the transformation. The elements (symbols and connections) which appear in the LHS but not in the RHS will be deleted from the diagram where the rule is applied to.

Figure 3 illustrates the application of the rule that allows for the insertion of an association symbol. The two class symbols of its LHS are mapped to the

[4] Well-defined in GenGEd means that the attribute can be handled like a "black box" which can be drawn somewhere in the diagram.

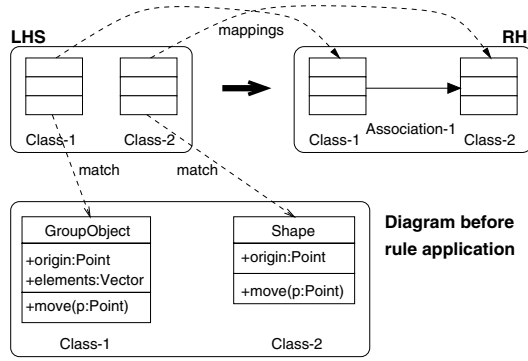

Figure3. Example rule `InsertAssociation` and its application to a diagram

class symbols of the diagram. Due to the mappings, the transformation process inserts the association symbol between these two class symbols. Note that it is also conceivable to map both class symbols of the rule's LHS to one class symbol of the diagram.

Negative Application Conditions Rules like that of figure 3 are sometimes not sufficient to describe the complete syntax of a visual language and the editing commands of the aimed Graphic Editor. Therefore, rules can be enhanced with *negative application conditions* (NACs) expressing that some constellations must not occur in the diagram where the rule is applied to. So one of our rules consists of a LHS, a RHS, a mapping from the LHS to the RHS and a (possibly empty) set of NACs, each one with a mapping from the LHS into the NAC diagram. The mappings into the NACs deliver the connection for the conditions.

In order to illustrate NACs let us have a look to figure 4 showing the rule for deleting a class symbol. This rule is enhanced with two NACs stating that the class symbol that is to be deleted is not connected with an association symbol, neither by the structural `from` connection nor by the `to` connection. For the application of a rule with NACs, we have to check whether one of the NACs can be satisfied after matching the LHS' elements to a diagram. If this is the case, the application will not take place.

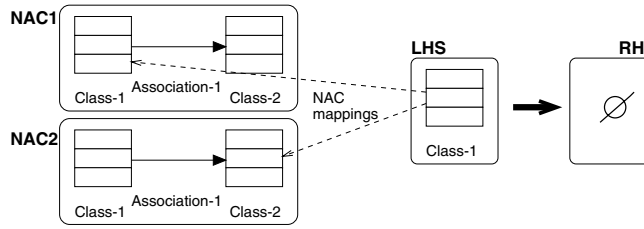

Figure4. Rule with negative application conditions

Assuming the rule of figure 4 without any NACs. We have to mention that the application of such a rule would lead to the deletion of all adherent association symbols due to the `from` and the `to` connections defined for the alphabet. This behavior is probably not desired which is the reason for defining the NACS.

Data Attributes and Rule Parameters Until now we have presented rules without data attributes for symbols, so in the examples there have been no association cardinalities nor class names. *Data attributes* may appear in the LHS, RHS and in the NACs, namely as variables, constants or complex expressions that are defined for the corresponding datatypes. In contrast to the rules, a diagram where rules can be applied to, comprises constants only.

Rule parameters allow for the external definition of data attributes. A rule parameter consists usually of a variable and a datatype. The rule is applied with user-defined values for the rule parameters. Then, the variables take the role of matched variables, i.e., they are matched with a constant. One example is given by figure 5 showing the rule for the insertion of a class symbol. The rule comprises a rule parameter with the variable *cn* of type *String*. This variable occurs in the NAC as well as in the RHS of the rule. Applying this rule, the user is asked to define a class name for the variable that is substituted by the name. Hence, the NAC states that the class symbol with this name must not be existent in the diagram where the rule is applied to.

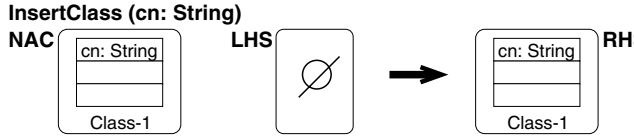

Figure5. Rule for inserting a class name

An example for complex expressions is given by figure 6. Each class symbol comprises a method list (modeled by the datatype *StringList*) that is connected to the lower rectangle of a class symbol. The rule that allows for adding a method to the method list of a class symbol is given by figure 6. It contains a variable for the user-defined method in its rule parameter. In its LHS the variable for the method list is illustrated together with the datatype. This variable together with the rule parameter variable are part of the available operation *add* denoted in the RHS of the rule. The operation is executed when the rule is to be applied to a given diagram. Therefore, the user is forced to define a concrete method as described above.

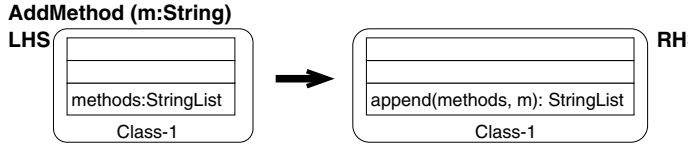

Figure6. Rule for adding methods to the method list

Until now we have suggested the most important concepts for visual language specifications and editing of diagrams. These concepts are implemented in the GenGEd environment that is explained in the following section.

3 The GenGEd Environment

The GenGEd environment comprises two major components: the *Alphabet Editor* and the *Grammar Editor* (see fig. 7), each editor corresponds to the respective part of the visual language, namely the alphabet and the grammar. To assure the graphically correct drawing of diagrams both editors use the constraint solver ParCon [Gri96]. The transformation of diagrams via rule application is done by the graph transformation system Agg [TER99].

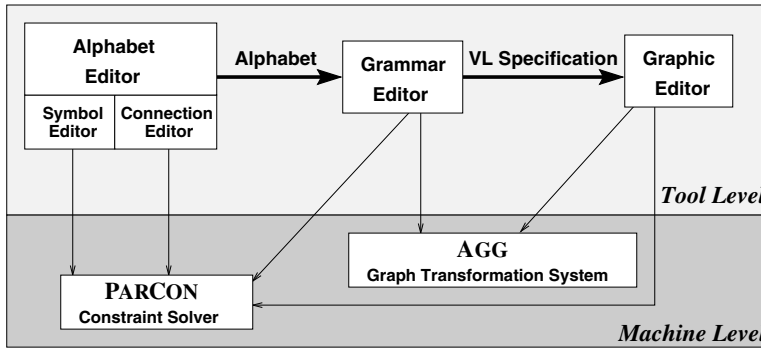

Figure7. Overview about the GenGEd environment

Simply speaking, the specification of a Graphic Editor for a visual language using GenGEd works like this:

1. We define the symbols and connections of a specific visual language using the Alphabet Editor.
2. The final alphabet is taken as an input to the Grammar Editor which then generates simple insertion/deletion rules. Those rules are to be used (in the notion of editor commands) to construct more complex visual language rules and to define a start diagram for the visual language.
3. The final visual language specification, consisting of the alphabet, the visual language rules and the start diagram, are then fed into a parameterized Graphic Editor. The user-defined editing commands of this editor ("insert", "delete" etc.) are built from the grammar rules.

The GenGEd environment is implemented in Java, also is the Agg system. Because the ParCon constraint solver – implemented in Objective-C – is only available for Linux and Solaris, GenGEd runs only on these two platforms. A prototype is available for download at

http://cs.tu-berlin.de/~genged.

3.1 The Alphabet Editor

The *Alphabet Editor* is a bundling of two minor editors – the *Symbol Editor* (see fig. 9) and the *Connection Editor* (see fig. 10). The Alphabet and Connection Editors feature the usual GUI elements like a menu and toolbar to add symbols, data attributes, connections and constraints and a statusbar to display various useful information. The appearance of the editors is the same: on the left side there is a structure display of the objects we work on showing primitives and constraints. On the right there is a graphical display of the selected symbol or connection. This graphical display is already constraint-based, so each added constraint will be checked for solvability and will be visualized immediately after creation. Both editors use a further subcomponent, the *Constraint Editor*.

Constraint Editor The constraint editor as shown in figure 8 is available in both, the Symbol Editor as well as the Connection Editor. In both editors, constraints can be defined on arbitrary primitives.

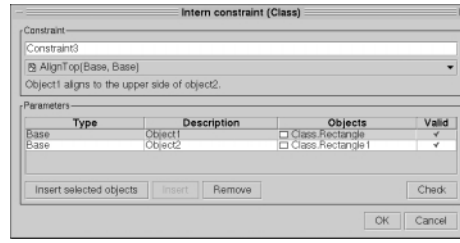

Figure8. The Constraint Editor

The constraint solver ParCon that is used for constraint solving permits only the definition of very low level constraints. We have enhanced these constraints by a *high-level constraint language* (HLCL). This language features extensibility, types over the graphical primitives and built-in definitions for user dialogs. Here is an example for an `above`-Constraint taken from the extendable HLC database:

```
constraint Above(Base a, Base b) {
  Dialog(a, "English", "Object1");
  Dialog(a, "Deutsch", "Objekt1");
  Dialog(b, "English", "Object2");
  Dialog(b, "Deutsch", "Objekt2");
  Description("English", "Object1 lies above object2.");
  Description("Deutsch", "Objekt1 liegt \"{u}ber Objekt2.");
  a.lt.y < b.lt.y - a.h; }
```

The graphical attributes of the primitives being part of the constraint (width, height, x/y-location, etc.) can be accessed using a path notation. The HLCL is easily editable and extendable by editing a simple text file.

All constraints we define in one of the GenGEd Alphabet Editors are immediately applied to the symbol(s). When the user scales or moves single objects in the display all constraints are automatically solved, which leads possibly to a new arrangement of the whole graphic in the display. Constraints which are unsolvable will be marked for overworking.

Symbol Editor The Symbol Editor is shown in figure 9. It works similar to well-known vector graphic editors except that the grouping of symbols is handled as described in section 2 – using constraints to connect the primitives in a graphic. Implemented primitives available are lines, polylines, bezier curves, rectangles, ellipses, images (GIF/JPEG), text, invisible boxes (which can serve as placeholders) and connection points which can be used to define complex connections in the Connection Editor. The primitives' properties like color, line width or text properties can be edited.

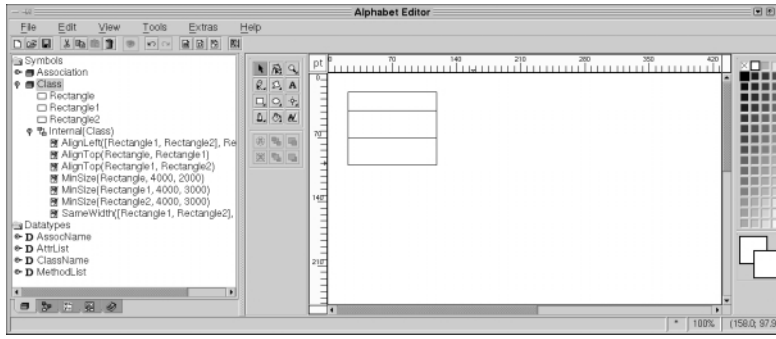

Figure9. The Alphabet Editor with activated Symbol Editor

Data attributes appear as independent graphical objects. From the constraint view they are just "boxes with something in it". Each datatype is implemented by a unique Java class. Similar to a JavaBean, the datatype class has to provide methods for drawing the attributes and for changing the properties (like text font, text size or, for a list of strings, the arrangement of text elements) either interactively (using an editing dialog) or by calling a changing method. Other methods can be used to build complex Java expressions which will be evaluated during rule application (see section 3.2).

Currently implemented are the classes `StringDT`, `StringListDT`, `IntegerDT` and `FloatDT`. Using the given interface for datatype classes and the existing implementations as templates, the designer of a visual language may add own datatype classes.

Connection Editor Concerning the constraint definitions, the Connection Editor as illustrated in figure 10 works in just the same way as the Symbol Editor.

In contrast to the Symbol Editor, we can select any two symbols as source and target of the connection that is to be defined. Again the Constraint Editor is used to define constraints between the primitives of the involved symbol graphics. Note that the connection according to the structural relationship connects two symbols (namely the symbol names). In contrast to the concepts, in the current implementation the connection constraints can be defined between the primitives only, and not between the boxes enclosing the symbol graphics.

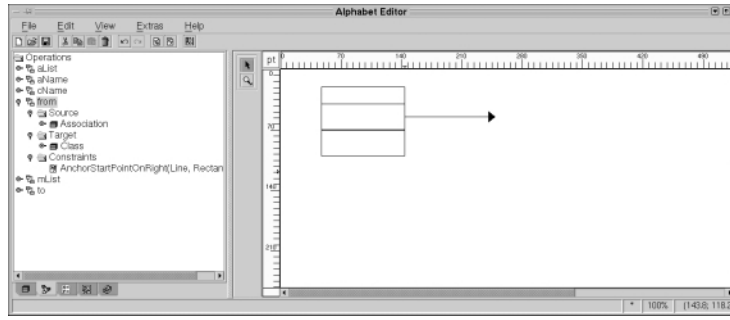

Figure10. The Alphabet Editor with activated Connection Editor

3.2 The Grammar Editor

The behavior of the Grammar Editor (see Fig. 11) is slightly different from the Alphabet Editor because we use instances of the already defined symbols and connections to generate simple editing rules. When we start the specification of a new visual language grammar, we are first asked to give an alphabet.

Simple Rules and Structural Correctness From the alphabet, some simple rules are generated that are the editing commands of the grammar editor. These rules reflect the structure of the alphabet in the sense that they are "structurally correct". This property has been described in section 2 when we talked about connections. This means that on the one hand every single rule diagram is structurally correct and that on the other hand every rule does a "'structurally correct"' transformation, i.e. correct diagrams are transformed into correct diagrams. An example for a generated rule `InsertAssociation` is given in figure 3. Note that some of these automatically generated rules may be already a rule of the intended grammar.

Data Attributes and Rule Parameters For the data attribute part of the visual language specification we have means to define rule parameters as well as to define and change the expressions for the transformation of datatypes. Because of the data attributes belonging to Java classes, the attribute expressions depicted in section 2 are in fact Java expressions which are evaluated to get an

object of the corresponding data attribute class. For example, the expression on the right hand side in figure 6 will look like this: `methods.append(m)`. Applying this rule to a given diagram, the user is first asked to match the class symbol of the rule's LHS to one class symbol in the diagram. Then, the editor window for the data attribute `StringDT` (the rule parameter) will pop up. When the user has given a value for the parameter, the expression on the RHS will be evaluated during transformation. In this example, the user-defined method is added to the method list of the class symbol.

The Graphical User Interface The Grammar Editor is shown in figure 11. On the left hand side there is a structure view of the grammar which contains all the names of the automatically generated rules, the start diagram and the rules we build using the Grammar Editor. The names of NACs which may occur in a rule are written below the rule names. On the right hand side we have two parts: The upper part shows the LHS and RHS of a rule which will be used for transformation. The NACs (selected in the structure view) can be displayed, too. The lower part is the *work display*: Here we built the LHS and RHS (or LHS and a NAC respectively) of a new rule, add mappings between the two rule sides and edit the rule parameters.

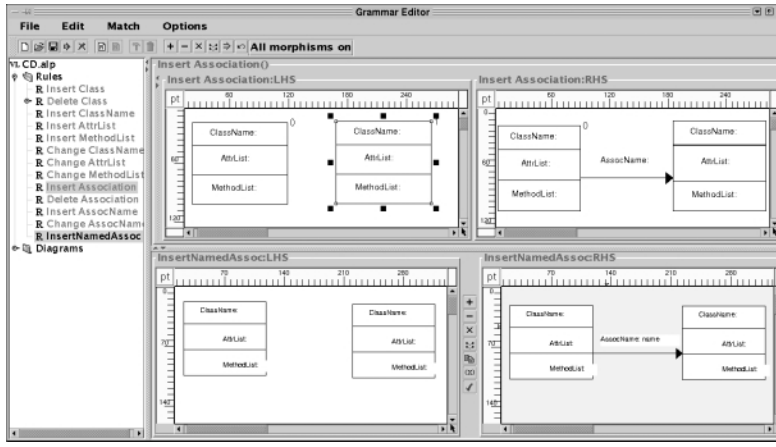

Figure11. The Grammar Editor

There are two toolbars: The main toolbar (below the menu) is used to add and delete rules and NACs and to provide save/load and other main functionality. Furthermore, the main toolbar is used to define a match from the rule which is to be applied onto one of the work diagrams and to trigger the transformation. The smaller toolbar (in the work display) provides functionality to add/remove mappings in the work rule and to edit the rule parameters.

Because the graphical displays which are used in the Grammar Editor are derived from those in the Alphabet Editor, they provide constraint solving, moving and scaling of symbols and also single graphical primitives in the same fashion.

3.3 Generating a Graphic Editor

The final step is to export a set of rules and a start diagram into a visual language grammar[5]. Then, the Graphic Editor that is a parameterized editor takes this grammar and uses the grammar rules to provide the language-specific editing commands. The Graphic Editor for our simple class diagram language is is illustrated in figure 12.

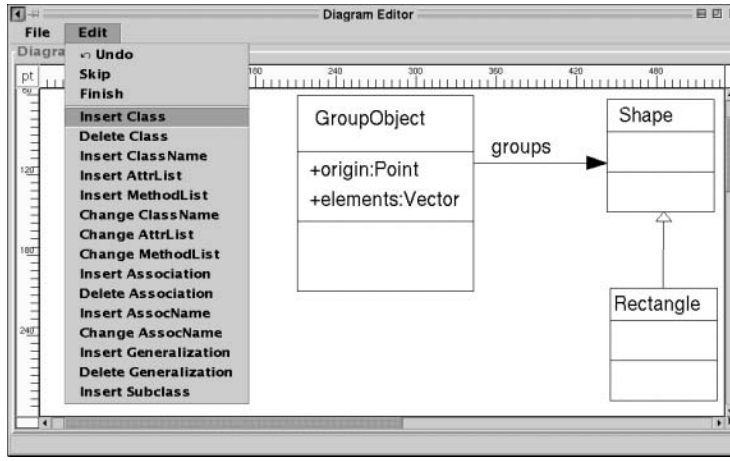

Figure12. A Graphic Editor for simple class diagrams

In the current implementation, the Graphic Editor allows for syntax-directed editing only. Nevertheless, each edited diagram comprises two levels of description. These are the *logical* structure of a diagram and its layout. The logical structure can be used for further extensions as, e.g., for code generation.

4 Related Work

Many different tools have been proposed supporting visual programming. The reader is referred to [Shu88,Cha90,BGL95] giving a broad overview. However, most existing tools are developed for specific application purposes. Moreover, the tools allow for visual programming and not for modeling languages like GenGEd. This means that the visual means are tightly integrated with the corresponding programming environment. In contrast to such environments, GenGEd is a generic framework based on a user-defined visual alphabet and a grammar.

The purpose of GenGEd is the visual definition of visual languages. From the definition a language-specific Graphic Editor is generated. Some other

[5] The alphabet is added automatically, so in fact we export a visual language *specification*.

approaches with the same purpose are to mention. These are e.g., PROGRES [SWZ99], DIAGEN [MV95], and VLCC [CODL95]. We have to mention that a comparison between GENGED, PROGRESand DIAGEN according to the underlying theory as well as the tools is given in [BTMS99]. A brief summary is given below.

PROGRES is a graph-based framework allowing for programming with rules and for the generation of prototypes. The PROGRES environment offers assistance for creating, analyzing, compiling, and debugging graph transformation specifications. Outside the PROGRES environment such a specification can be executed in a stand-alone prototype. In PROGRES, visual languages can be specified using the visual means of graphs. It is not possible to specify a visual language under consideration of the appearances of symbols as it can be done using GENGED.

DIAGEN is a diagram editor generator. The input of the generator is a textual specification of a visual language. In general, visual statements are difficult to describe textually because of their graphical structure. This is due to the fact that multi-dimensional representations have to be coded into one-dimensional strings. Moreover, the DIAGEN approach is concerned with a restricted kind of context-sensitive grammars. Hence, the class diagram language, e.g., cannot be specified using the DIAGEN approach [BTMS99].

G.Costagliola et al. introduced the VLCC-environment [CODL95] supporting the visual definition of visual languages, too. 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 positional grammar rules. In contrast, we use grammars which are not restricted to be context-free.

5 Summary

In this contribution we informally introduced visual languages as they can be defined using the GENGED environment. Similar to textual languages, a visual language is defined by an alphabet and a grammar. Correspondingly, the GENGED environment comprises an Alphabet Editor and a Grammar Editor. These editors as well as a generated Graphic Editor are described. An in-depth description of the design and the implementation of the environment can be found in [Sch99] and [Nie99], the underlying theory is depicted in [Bar00].

Despite the prototypical character of the environment, there are several case studies. These are restricted kinds of visual languages like statecharts, class diagrams, sequence diagrams and Nassi-Shneiderman diagrams. Some more are yet to come. Thereby, we investigate how to integrate several visual languages similar to the UML in order to generate not only a Graphic Editor but a visual environment. Future work is concerned with a major overhaul of the Grammar Editor, simplifying the rule definition. Another idea is to change the generated Graphic Editor into a JavaBean, thus providing other tools (e.g., the AGG system) with a generic graphic display and making the underlying structure of a diagram accessible. We are also planning to allow for more freedom in diagram

editing, combining syntax-directed and freehand editing. The latter one is concerned with parsing.

References

Bar00. R. Bardohl. *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. PhD thesis, Technische Universität Berlin, Berlin, 2000. 457, 458, 459, 469

BGL95. Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications Co., Greenwich, 1995. 468

BNS00. R. Bardohl, M. Niemann, and M. Schwarze. GenGEd – A Development Environment for Visual Languages. In *Application of Graph Transformations with Industrial Relevance*, LNCS. Springer, 2000. 457

BTMS99. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In *[Roz99]*. 1999. 469

Cha90. Shi-Kuo Chang, editor. *Principles of Visual Programming Systems*. International Editions. Prentice Hall, Englewood Cliffs, NJ, 1990. 468

CODL95. G. Costagliola, S. Orefice, and A. De Lucia. Automatic Generation of Visual Programming Environments. *IEEE Computer*, 28(3):56–66, March 1995. 469

Gri96. P. Griebel. *ParCon - Paralleles Lösen von grafischen Constraints*. PhD thesis, Paderborn University, February 1996. 463

Har87. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. 456

MV95. M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. IEEE Symposium on Visual Languages*, pages 203–210, 1995. 469

Nie99. M. Niemann. Konzeption und Implementierung eines generischen Grammatikeditors für visuelle Sprachen. Master's thesis, Technische Universität Berlin, 1999. 469

NS73. I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), 1973. 457

Rat98. Rational Software Corporation. UML – Unified Modeling Language. Technical report, Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951, 1998. http://www.rational.com/uml. 456, 457

Roz99. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools*. World Scientific Publishing, Singapore, 1999. 470

Sch99. M. Schwarze. Konzeption und Implementierung eines generischen Alphabeteditors für visuelle Sprachen. Master's thesis, Technische Universität Berlin, 1999. 469

Shu88. N.C. Shu, editor. *Visual Programming*. Van Nostrand Reinhold, New York, 1988. 468

SWZ99. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES Approach: Language and Tool Environment. In *[Roz99]*. 1999. 469

TER99. G. Taentzer, C. Ermel, and M. Rudolf. The AGG Approach: Language and Tool Environment. In *[Roz99]*. 1999. 463