ELSEVIER Science of Computer Programming 44 (2002) 181–203

Science of Computer Programming

www.elsevier.com/locate/scico

A visual environment for visual languages

Roswitha Bardohl

Technical University, Berlin, Germany

Abstract

The visual environment GenGEd supports the visual definition of visual languages (VLs). Each VL is defined by an alphabet and a grammar. From a specific VL-definition, a VL-specification is generated which is the input of a graphical editor allowing for syntax-directed editing of diagrams over the specified VL. GenGEd as well as each VL is based on the well-defined concepts of algebraic graph transformation and graphical constraint solving. The underlying formalism is hidden from the user, but it is essential for a formal presentation and manipulation of graphical structures. In this contribution, the GenGEd concepts and environment are briefly proposed and illustrated by the definition of a simple kind of the well-known statechart language. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Visual languages; Visual environment; Graphical editors; Statecharts

1. Introduction

Visual languages (VLs) are used within lots of application areas: teaching children and adults, programming for non-programmers, adaptation 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 visual modeling and specification languages can be found, for example, by the unified modeling language (UML) [30], automata and Petri nets.

All the tasks (programming, modeling, etc.) are mostly supported by visual environments but it depends on the purpose of the environment which functionality it offers [34]. Basically, visual environments comprise graphical editor(s) for diagram editing. Such diagrams correspond to a specific VL where the visual means of expressions are tightly integrated in the editors [8].

In general, the development of VL-specific graphical editors is time-intensive and thus, expensive. Whenever the visual means of a VL will be changed (as it is done by

E-mail address: rosi@cs.tu-berlin.de (R. Bardohl).

PII: S0167-6423(02)00038-2

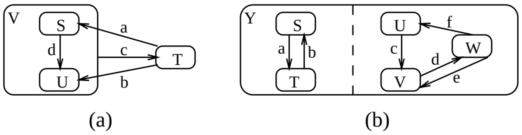


Fig. 1. Hierarchical composition (a) and parallel composition (b) of statecharts.

the UML, a successor to the modeling languages found in the Booch, OOSE/Jacobson, OMT and other methods [36]) or adapted to a certain application domain, a partial re-implementation will be necessary which is also time-intensive and expensive. This has been the main motivation for the development of the GenGEd approach which is already implemented [2]. The GenGEd environment supports the visual definition of VLs and generates VL-specifications. A VL-specification is the input of a graphical editor allowing for syntax-directed editing of diagrams over the language specified.

Similar to formal textual languages, a VL is specified by a (visual) alphabet and a (visual) grammar. In general, visual statements are difficult to describe textually because of their graphical structure. On the other hand, visual descriptions are mostly not sufficient to define all necessities [32]. We show that GenGEd supports the convenient definition of VLs which is flexible enough concerning the graphical layout. Moreover, in order to support the definition of a broad variety of VLs, the grammars are not restricted to be context-free.

Each VL is based on the well-defined concepts of algebraic graph transformations for the logical level of a VL, called *abstract syntax*, and graphical constraint solving for the layout, called *concrete syntax*. Both syntactical levels together establish the *visual syntax*. An alphabet is represented by a type graph and a constraint-satisfaction problem, and a grammar by a graph grammar together with constraint-satisfaction problems for each alphabet instance (diagram) occurring in a grammar. All diagrams are represented by attributed graphs typed over the alphabet.

The GenGEd concepts and the environment are introduced in this contribution, and illustrated by the definition of a simple kind of the well-known statechart language as it has been originally introduced by Harel [22]: statecharts are automata which are extended by concepts for hierarchical and parallel (state) composition. A hierarchical composition expresses the composition of several states (*sub-states*) to one state (*super-state*). A parallel composition models the combination of several sub-states to one super-state by conjunction. Two examples concerning these extensions are shown in Fig. 1. Diagram (a) presents a hierarchical composition of a state called V. Diagram (b) shows a parallel composition of a state called Y.

This article is structured as follows: we start with an informal review of the basic notions in Section 2. In Section 3 we give a survey on the GenGEd concepts and the environment. The concepts are applied to our statechart language in Section 4. In Section 5 we discuss some related work, and in Section 6 we draw conclusions and sketch some extensions.

2. Graph transformation and constraint-satisfaction problems: basic concepts

Graphs play an important role in many areas of computer science. They are especially helpful in analysis and design of software applications, like database systems or distributed systems. Prominent representatives for graph-like notations are entity relationship diagrams, message sequence charts, Petri nets, automata and all kinds of UML diagrams. Like graphs, also the constraint-satisfaction approach has been used in a variety of situations. For example, constraints are used for the interactive description of geometrical figures in [35], so as in [7] where additionally simulation is supported, or for building user interfaces [29].

In GenGEd, attributed graphs are used as internal representation model of diagrams like the two statecharts in Fig. 1, and graph transformation as well as graphical constraint solving is used for the transformation of diagrams.

2.1. Graph transformation

First, we will recall the concept of a typed graph [11,24]: usually a graph is given by two disjoint sets (graph objects), namely *nodes* (in the following visualized as rectangles) and directed *arcs* (visualized as arrows) from a source node to a target node. Every graph object is typed over a type graph. Moreover, graph objects may be labeled by attributes that are used to store data together with the graph objects [25]. An attribute will be denoted by a node (visualized as rounded rectangle), and an attribute arc connecting the attribute node with its attribute type (a set) in the case of type graphs. In the instance graphs this attribute arc will connect an attribute node with the current value of that attribute.

Note that we allow as attribute types abstract data types, that is, we consider not only sets of types, but also operations on these types according to the algebraic specification formalism introduced in [18]. In particular, the use of abstract data types allows us to use variables and terms as attributes (by choosing a term algebra as attribute algebra). Moreover, in [2] an alphabet is represented by an algebraic graph structure signature (which is an algebraic signature with unary operation symbols only), and a grammar is represented by an algebraic graph structure grammar. This kind of formalism is more general than typed graphs and offers more flexibility [24], however, the concepts of typed graphs are better known. Hence, we present the concepts of typed graphs, especially for those readers who are not familiar with the algebraic specification formalism. As a result, a type graph poses some requirements on the instances. These requirements are explained in the following example.

Fig. 2(a) illustrates an attributed graph that is typed over the type graph shown in (b). The graph in (a) represents two named states (attributed nodes 1:State and 2:State), both states are related (by arcs called in) to a statechart (node 1:Statechart). The state names are represented by the attribute nodes 1:SN, 2:SN (SN stands for *State Name*) and the attribution arcs SN-attr (short for *state name attribute*) holding the current values S and U, respectively. The arcs called stn connect the state names with the corresponding states.

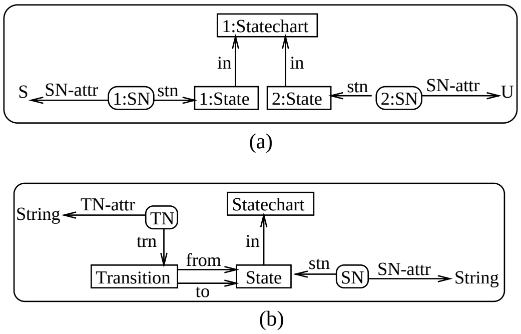


Fig. 2. An attributed graph (a) typed over the type graph (b).

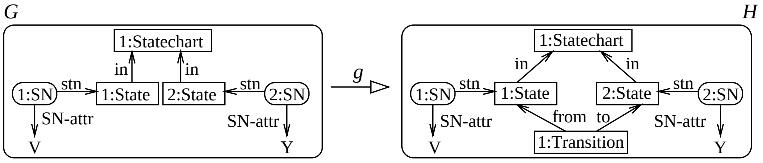


Fig. 3. A graph morphism between attributed graphs.

The type graph shown in Fig. 2(b) specifies a small part of the statechart language, i.e., the type graph poses the following requirements on the possible instances: A Transition node can be inserted if at least one State node is available in the instance graph such that the arcs from and to are defined. This is due to the fact that in instance graphs the graph part must be *total* whereas the attribute part may be *partial*. Furthermore, State nodes must be always connected to a Statechart node by an in arc. In no instance graph a Transition node can be connected to a Statechart node because such a situation is not present in the type graph. Note that Transition nodes must be attributed by the attribute node TN (short for *Transition Name*), similar to the State node attribution described above, because in instance graphs the attribution is required to be total.

A relationship between two graphs G and H can be expressed by a graph morphism that maps the nodes and arcs of graph G to nodes and arcs of graph H, respectively. The graph objects in G are called *origins* and in H *images*. The mappings have to be *type compatible* (nodes and arcs are mapped to nodes and arcs of the same type) and *structure compatible* (the source/target node of an arc is mapped to the source/target node of the arc's image). The attribute values also have to coincide. A graph morphism g between graphs G and H is denoted by $g:G\rightarrow H$ (or simply $G\xrightarrow{g}H$). Note that sometimes not all graph objects are mapped; we call such morphisms *partial* whereas morphisms that map all objects in the origin are called *total*. Fig. 3 shows a (total) graph morphism between attributed graphs by equally named graph objects.

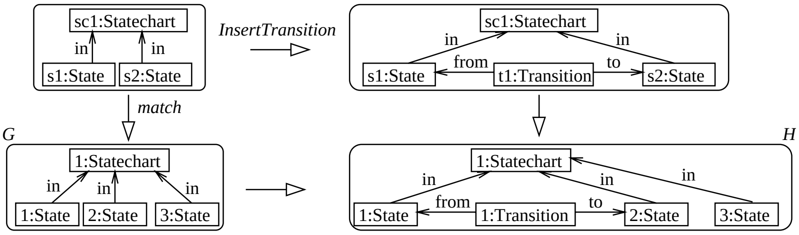


Fig. 4. Application of a rule.

Graph transformation defines a rule-based manipulation of graphs.[1] Graph rules can be used to capture the dynamic aspects of systems. The resulting notion of a graph grammar (consisting of a start graph and a set of graph rules) generalizes Chomsky grammars from strings to graphs. The start graph represents the initial state of the system, whereas the set of rules describes the possibles state changes that can occur in the system. A rule comprises two graphs: the left-hand side L and the right-hand side R, and a graph morphism $r: L \rightarrow R$ between graphs L and R. Graph objects in L which do not have an image via r in R are deleted; graph objects in R without origin in L are created, and graph objects in L which are mapped to R by r are preserved by the rule.

The application of a rule to a graph G (*derivation*) requires a mapping from the rule's left-hand side L to this graph G. This mapping, called *match*, is a total graph morphism $m: L \rightarrow G$. A match marks the graph objects in the working graph that participate in the rule application, namely, the graph objects in the image of m. The rule application itself consists of three steps. First, the graph objects marked in the rule for deletion are deleted. Thereafter, the new graph objects are appended to the graph. As a last step, all dangling arcs are deleted from the graph. The graph transformation results in a transformed graph H. Fig. 4 shows the application of the rule *InsertTransition* modeling the insertion of a transition between two states where the arcs from and to are inserted, too. We have indicated the rule morphism and the chosen match by equally named graph objects.

Rules often use variables and terms as attributes. Using attributed graphs, the attribute values or variables of the rule's left-hand side have to match as well. An attribute variable is bound to an attribute value in the mapped graph object by the match. In the transformed graph, the attribute values are evaluated depending on the rule's right-hand side, and result in a constant value.

Variables or constant values of certain attribute types may extend rule names used as rule morphisms. For example, the rule shown in Fig. 4 may model not only the insertion of a transition but additionally the insertion of a transition name of type String. The term *InsertTransition* (*tn* : *String*) with variable tn depicts the corresponding rule. In the rule's right-hand side a corresponding attribute and arc are inserted connecting the

[1] Here we follow the Algebraic Single-Pushout approach to graph grammars [17,25]. For an overview of the main approaches see [31].

transition node with its attribute. When a rule with rule parameter is applied to (a working) graph G, the rule parameter is replaced by a concrete (user-defined) value.

We have seen that the left-hand side of a rule states the necessary conditions the current graph must fulfill so that the rule can be applied. Sometimes, the need arises to express that something *must not* be in a working graph for a rule to be applicable. We cannot express such a *negative application condition* (*NAC*) with the means we already know. Therefore, we introduce another graph N to the rule holding the negative conditions [21,23] just like the left-hand side graph L contains the positive ones. A rule's left-hand side can have a set of NACs that are graph morphisms from the rule's left-hand side to the NAC graph ($L \xrightarrow{l} N$). Hence, the NAC graph contains graph objects of L and additionally the forbidden graph pattern. The match m satisfies a NAC if there is no graph morphism from the NAC graph N to the working graph G extending m, i.e. there is no graph morphism $n : N \rightarrow G$ such that $L \xrightarrow{l} N \xrightarrow{n} G = m$. Informally speaking, within an NAC you specify exactly that fraction of a matching situation that you do *not* want to find.

Attribute conditions are logical expressions for data attributes. Such conditions may occur in the left-hand sides of rules as well as in NACs if the attribute type offers corresponding operations. Attribute conditions must be satisfied by a match, too.

2.2. Constraint-satisfaction problems

A constraint is a declarative description of a boolean operation between one or more *constraint variables* [14]. Each constraint variable has a certain domain; a value of this domain can be assigned to the constraint variable. A *constraint network* results from the combination of constraints which share constraint variables. A constraint network can be acyclic or cyclic, as well as over- or under-specified. In the literature one can find several possibilities for the treatment of such cases. However, we do not focus on these multiple constraint solving algorithms but expect that we always get one solution.

A *solution* of one constraint is given by concrete values for the corresponding constraint variables. A solution satisfying all constraints in a constraint network is called *constraint satisfaction*. For a given constraint network, a *constraint-satisfaction problem* (CSP) is to find a value for each constraint variable of the network such that all constraints are satisfied.

A CSP is given, for example, by constraint variables line.length indicating the length of a line, rectangle.width indicating the width of a rectangle, and constraints line.length $= 5$, rectangle.width $<$ line.length. The domain of the constraint variables is given by natural numbers. We get four solutions, i.e., possible values for rectangle.width, these are the values 1–4.

In graphical applications it is important that the solution of a CSP satisfies the expectations of a user. In [20] this requirement is treated in by the so-called *Least Astonishment Principle*:[2] for a given CSP, local constraint variables are modified only (according to that diagram part the user has changed), and furthermore, it is tried to

[2] Here we use the constraint solving algorithms and the corresponding constraint solver proposed in [20].

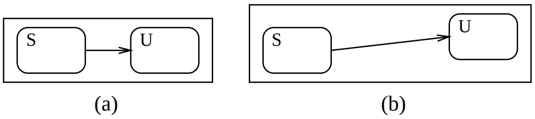


Fig. 5. *Least Astonishment Principle*: original situation (a) and situation after moving state U (b).

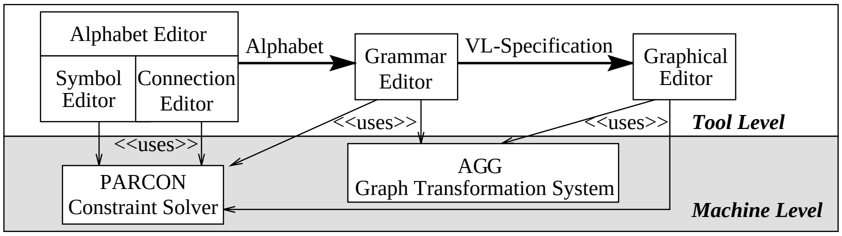


Fig. 6. Overview about the GENGED environment.

change few variables only. Fig. 5 illustrates the behavior according to this principle: moving the state called U effects the dragging of the transition arrow; the position of state S remains unchanged.

In GENGED, graphical constraints concern positions and sizes of graphics. The constraints can be defined by equations but also by in-equations. Such in-equations offer flexible means to fulfill common layout requirements.

3. Visual languages defined using GENGED

The definition of visual languages (VLs) using GENGED is completely based on graph transformation and graphical constraint solving as briefly introduced in the previous section (cf. [2] for more details). The power of graph transformation allows us to define context-sensitive VLs. A VL is defined by a *visual alphabet* and a *visual grammar* yielding a *VL-specification*. Given a VL-specification, a VL is the set of all VL-diagrams that can be derived by applying grammar rules to the start diagram. Note that we distinguish the *abstract syntax* (the logical part) and the *concrete syntax* (the layout) for VL-specifications and diagrams.

According to the constituents of a VL-specification, the GENGED environment as sketched in Fig. 6 comprises two major components: the *Alphabet Editor* and the *Grammar Editor* for the visual definition of VLs. From the VL-definition using these editors, a VL-specification is generated which is the input of the *Graphical Editor* for syntax-directed diagram drawing. This means that the language-specific editing commands of the Graphical Editor are given by the grammar rules of the visual grammar. Hence, not only a VL is specified but the VL-specific Graphical Editor also. Note that we distinguish two kinds of users, namely users defining a VL (*language-designer*), and those who use a Graphical Editor.

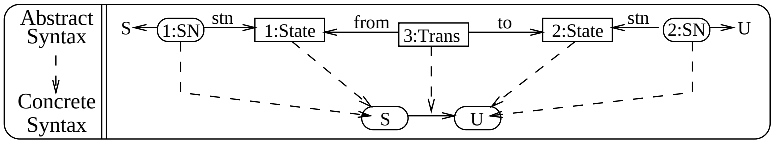


Fig. 7. A diagram typed over the alphabet shown in Fig. 8.

To assure the graphically correct drawing, all GenGEd editors use the constraint solver ParCon [20]. The transformation of diagrams via rule application in the Grammar Editor and the Graphical Editor is done by the graph transformation system Agg [19]. The GenGEd environment is implemented in Java, so is the Agg system. The ParCon constraint solver—implemented in C—is available for Linux and Solaris, thus GenGEd runs on these two platforms.

In the following we have a closer look to the GenGEd approach: we start with a brief introduction into the kind of diagram representation. Thereafter we discuss the concepts of visual alphabets and visual grammars as well as rule application and show how these concepts are realized by the GenGEd environment.

3.1. Diagram representation

In general, a diagram consists of a set of *symbol graphics* that are spatially related. We offer graphical constraints for these spatial relationships, called *link constraints*. Symbol graphics and link constraints concern the layout of diagrams, called *concrete syntax*. The logical part of diagrams is called *abstract syntax*. Both together establish the *visual syntax* which is represented by attributed graphs. Fig. 7 illustrates a diagram whose abstract syntax is typed over the alphabet shown in Fig. 2(b).

Note that we distinguish different kinds of attributes, these are attributes for structural information and for layout purposes: the abstract syntax level of diagrams is represented by attributed graphs as introduced in Section 2.1. Each node represents a *symbol* and each arc represents a *link*. The visual syntax of diagrams extends the abstract syntax by the concrete syntax: all symbol nodes of the abstract syntax level are attributed by symbol graphics (visualized by dashed arrows in Fig. 7), and for each link arc between two symbol nodes there is at least one link constraint defined between the corresponding symbol graphics. All these constraints must be satisfied when speaking about *diagrams*.

3.1.1. Symbol graphics and link constraints

Symbol graphics occurring in diagrams can be bitmap graphics, primitive graphics like rectangles, lines, etc., or graphics comprising several primitive graphics which are grouped by graphical constraints, called *symbol constraints*. Moreover, a graphic can be defined to be invisible, then it is called *placeholder*.[3] The graphics belonging to

[3] Placeholders denote either logical symbols which do not have a layout or they denote (help) symbols used to connect some other symbols visually.

one symbol graphic are enclosed by a non-visible box. The positions (the upper left points) of enclosed graphics are set relatively to the position of the outer box; this can be hierarchically continued. We assume as drawing area the next layer where the box is placed. The so-called *box concept* supports similar treatment of graphics independently whether it is a symbol graphic, a drawing area or a diagram part.

As introduced in Section 2.2, graphical constraints are defined on constraint variables having a certain domain. In GENGED, the constraint variables concern position and sizes of graphics, i.e., the domains are given by [Real,Real] indicating a tuple of real values, or Real. The latter one is used, e.g., for constraint variables defining the width or height of a graphic.

In [2] it is shown how constraint variables are obtained from access operations available on graphics. For example, the upper left point of a graphic designates the graphic's position which is obtained by the operation .nw: Graphic → [Real,Real] indicating the *north-west* point. In dependence of the abstract syntax of symbols like 1:State or 1:Transition and corresponding concrete graphics like rectangle or arrow, the access operations define the unique constraint variables necessary to obtain diagram-specific constraint-satisfaction problems (CSPS). For example, the variable arrow_1:Transition.beg indicates the begin point of the transition arrow which denotes the primitive graphic of a symbol called 1:Transition.

A diagram-specific CSP is derived from the CSP of the corresponding user-defined alphabet in accordance to the graph objects (symbol nodes and link arcs) occurring in the abstract syntax of the diagram. Usually, such a situation arises during transformation of diagrams, i.e., when a grammar rule is applied to a given diagram.

3.2. *Visual alphabets*

A *visual alphabet* establishes a type system for *symbols* and *links*, i.e., it defines the vocabulary of a VL. The abstract syntax of an alphabet is represented by exactly such a type graph as introduced in Section 2.1. Note that in an alphabet, the symbol and link types have to be unique as well as the link arcs have to be acyclic. The concrete syntax is given by the abstract data type Graphic (according to [18]) used as the graphical types of node attributes, and a CSP for the distinguished (user-defined) symbol graphics and constraints. The CSP is obtained as follows: from each symbol graphic and access operations available on graphics (cf. Section 3.1), unique constraint variables are derived which form the basis for constraint definitions. Constraints *must* be defined for each link (arc), however, constraints *may* be defined for grouping primitive graphics used to build up one symbol graphic.

The visual syntax of an alphabet is a combination of the abstract and the concrete syntax by attributing the symbol nodes with the data type Graphic, illustrated by dashed arrows in Fig. 8 showing some symbols and links of the statechart vocabulary. Lexical symbols indicate the major symbols, like the symbols State and Trans (transition). Symbols like SN (*state name*) and TN (*transition name*) of type String denote the attributes of lexical symbols. As for symbols, we distinguish *attribution links*, each connecting a lexical symbol with its attribute symbol, and *connection links* between lexical symbols.

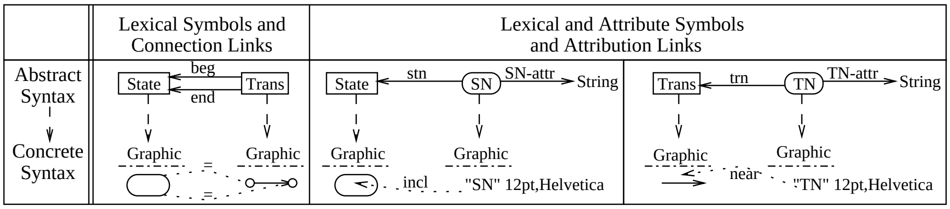


Fig. 8. Part of an alphabet for statecharts.

In Fig. 8, the (user-defined) symbol graphics are drawn below the data types Graphic. From these symbol graphics unique constraint variables are obtained, like rectangle_State.nw, arrow_Transition.beg, etc., forming the basis for graphical constraints. Symbol constraints are defined on the layout of the attribute symbol SN of type String (TN of type String, respectively). These constraints are given by text size (12 pt) and text font (Helvetica), whereas the term “SN” (“TN”) is used as a textual placeholder (indicating an attribute symbol in the Grammar Editor). The four link constraints (according to the arcs beg, end, stn, trn) are illustrated by dotted lines or—in order to clarify the order of variables in a tuple over which the constraints are defined—by arrows. The two constraints for the connection links state that each transition arrow begins (ends) at the border of a state rectangle. The constraint incl forces state names to be surrounded by state rectangles, and the constraint near expresses that transition names should be placed near by transition arrows.

In an alphabet there is a sub-assignment for some constraint variables with values taken from their domains. These sub-assignments concern the sizes and positions of primitive graphics occurring in each symbol graphic given by a non-visible box. The sizes of primitive graphics like that of the rounded rectangle used for the State symbol in Fig. 8 are user-defined. The positions of primitive graphics are set relatively to the position of the box. In the case of the State symbol, for example, the position of the rectangle is set to [0,0] which is taken as the value (sub-assignment) of the corresponding constraint variable rectangle_State.nw indicating the position. The box size is calculated from its sub-graphics. So we also have a sub-assignment for constraint variables indicating the box sizes. All these variables and values defining a Csp of an alphabet are instantiated when building up diagrams as they occur already in grammars.

The concepts mentioned so far are implemented as *Alphabet Editor* which is a bundle of two sub-editors—the *Symbol Editor* and the *Connection Editor*. A snapshot of the Symbol Editor is shown in Fig. 9. For each symbol the user gives a unique symbol name (like Transition) due to the abstract syntax, and a symbol graphic (like an arrow) and possibly some symbol constraints due to the concrete syntax. For the definition of the symbol graphic, the Symbol Editor works similar to well-known vector editors except that the grouping of symbols is handled as described in Section 3—using constraints to connect the primitives in a symbol graphic. Primitives available are lines, poly-lines, bezier curves, rectangles, ellipses, images (GIF/JPEG), text, non-visible rectangles (graphical placeholders) and connection points needed for defining

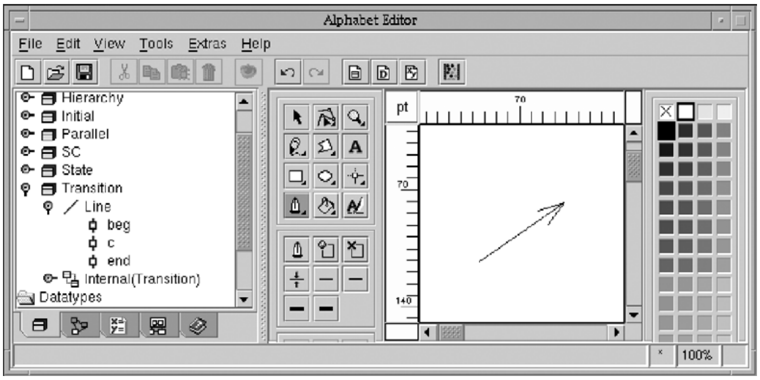


Fig. 9. The Alphabet Editor with activated Symbol Editor.

link constraints in the Connection Editor. The primitives' properties like color, line width or text properties can be edited, too.

Attribute symbols appear as independent graphical objects in the Symbol Editor. From the constraint view they are just "boxes with something in it". Each data type is implemented by a unique Java class. Similar to a Java Bean, the data type 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. Currently implemented are the classes `StringDT`, `StringListDT`, `IntegerDT` and `FloatDT`. Using the given interface for data type classes and the existing implementations as templates, the designer of a VL may add own data type classes.

A snapshot of the Connection Editor supporting the definition of links between symbols is shown in Fig. 10. In order to define a link, the user can select any two symbols as source and target of the link due to the abstract syntax. A constraint dialog supports the definition of link constraints due to the concrete syntax. Note that in the current implementation, the link constraints can be defined on the primitives only and not on the boxes denoting the symbol graphics. Such a box is just selected in the snapshot of the Connection Editor; it is visible by the anchor points. The box belongs to the symbol `SC` (denoting a statechart) whose layout is given by a graphical placeholder.

The Alphabet Editor generates a visual alphabet which is the basis to define the visual grammar of a specific VL. Before we introduce the Grammar Editor, we first present the underlying concepts.

3.3. Visual grammars and rule application

A *visual grammar* is represented by a graph grammar: it consists of a *start diagram* and a finite set of *rules*. The start diagram, both sides of a rule and the NACs are

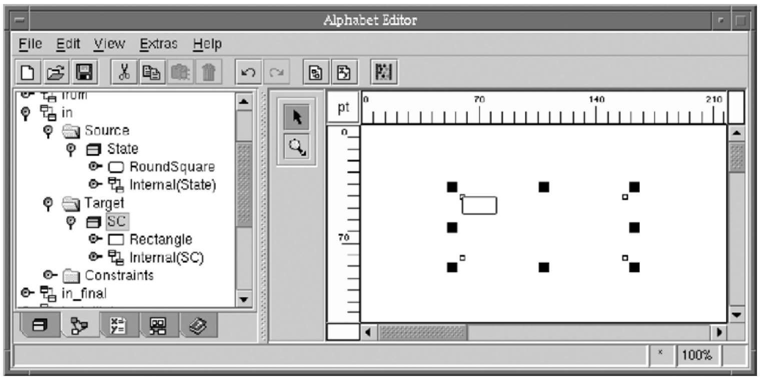


Fig. 10. The Alphabet Editor with activated Connection Editor.

diagrams typed over a specific alphabet, as well as such diagrams which can be derived by applying grammar rules. The latter ones are called *VL-diagrams*.

Grammar rules will be used as the editing commands of the intended VL-specific Graphical Editor. These rules concern *generating rules* and *updating rules* as they are needed for syntax-directed editing. Generating rules support the insertion of symbols and links, whereas updating rules support their deletion and exchange. Beside pure editing rules, it is also possible that grammar rules express comprehensive language features as needed in the grammar for statecharts. For example, it is necessary to prevent transitions between states which must not be connected like states that are in different sub-states of the same parallel state. Such conditions can be checked according to the power of graph transformation.

The application of a rule to a diagram G is obtained via a match morphism on the abstract syntax as explained in Section 2.1. The derivation of the abstract syntax of diagram G yields the abstract syntax of diagram H which has to be extended by its concrete syntax afterwards. The concrete syntax of H is obtained after deriving the diagram-specific CSP from the CSP of the alphabet, and a solution. The diagram-specific CSP is obtained as follows: for each graph object (abstract syntax of symbols and links) occurring in the diagram H, the corresponding constraint variables are uniquely derived from the CSP of the alphabet as well as the (symbol and link) constraints. For example, in the alphabet we have a constraint variable called rectangle_State.nw describing the north-west corner of a rectangle for the node type State. An instance of this node type may be denoted by 1:State such that the constraint variable rectangle_1:State.nw is obtained which is typed over rectangle_State.nw. In the same way we obtain the constraints. For the resulting CSP we require a solution. The solution provides the values (sizes and positions) necessary to obtain the symbol graphics, i.e., the (graphical) node attributes.

Fig. 11 illustrates the application of a rule supporting the insertion of a transition symbol between two state symbols. Note that this rule allows transitions between arbitrary states, i.e., statecharts with hierarchical or parallel states are not regarded now.

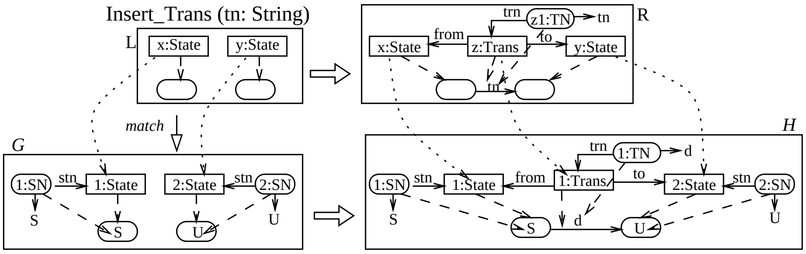


Fig. 11. Application of rule Insert_Trans (d) to diagram G yields diagram H.

In the rule's LHS the existence of two state symbols is required. These symbols are preserved by the rule indicated by same names. In the rule's RHS the symbol z:Trans is inserted and linked to the two state symbols. The rule parameter tn: String indicating a variable of type String denotes an attribute symbol which is additionally inserted in the rule's RHS and linked to the transition symbol. The rule is applied to diagram G; the dotted arrows show the mappings of the match. According to the rule and match morphisms and a solution for the diagram-Csp we obtain diagram H.

The definition of a visual grammar is supported by the Grammar Editor available in the GenGEd environment: first of all, the Grammar Editor gets an alphabet as input. Then, so-called *alphabet rules* are generated from the alphabet that define the editing commands of the Grammar Editor. Note that the set of alphabet rules comprises rules for the insertion and deletion of symbols. These rules reflect already the structure of the alphabet in the sense that each diagram occurring in a rule is represented by a typed graph, and the requirement for defined link arcs is satisfied. In the snapshot of the Grammar Editor shown in Fig. 12 one can see the alphabet rule for the insertion of a transition name in the upper part.

The lower part of the Grammar Editor denotes the *working areas*: here we build the start diagram, and the LHS and RHS (or LHS and NACs, respectively) of a VL-rule, add mappings between the two rule sides and edit the rule parameters. Since the data attributes belong to Java classes, the attribute expressions are in fact Java expressions which are evaluated to get an object of the corresponding data attribute class. Applying a rule with rule parameter to a diagram in one of the working areas, the user is first asked to define the match morphism, i.e., to map the symbols of the rule's LHS to type-consistent symbols in the diagram. Then, the user has to give a value (or a variable) for the parameter such that the expressions in the RHS can be evaluated during transformation.

The final step is to export the set of VL-rules and the start diagram into a visual language grammar.[4] Then, the Graphical Editor takes this grammar and uses the grammar rules to provide the language-specific editing commands. Note that in the current implementation the Graphical Editor works similar to the Grammar Editor. Hence, we omit a snapshot.

[4] The alphabet is added automatically, so in fact we export a VL-specification.

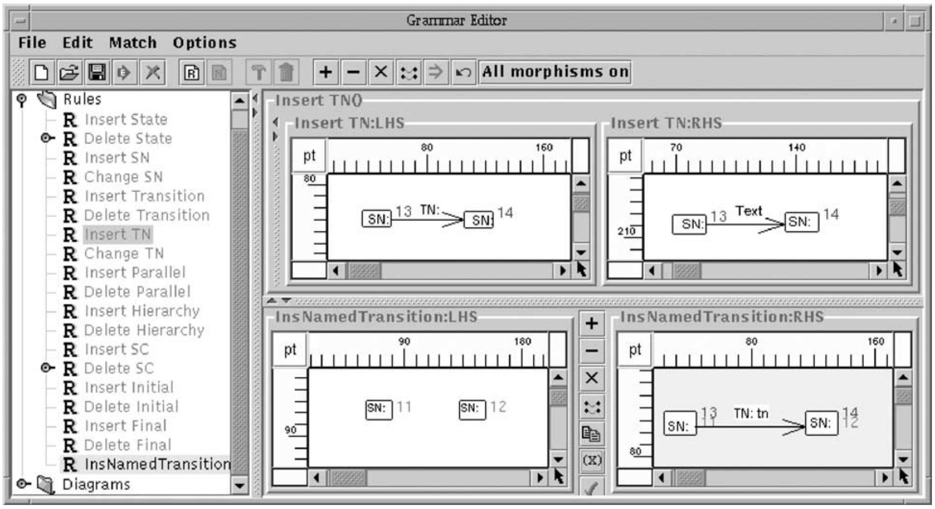


Fig. 12. The Grammar Editor.

So far, we have introduced the main concepts of GenGEd and the implementation realizing these concepts, we are now able to specify a specific visual language, namely statecharts.

4. Defining statecharts using the GenGEd concepts

Fig. 1 shows two statecharts which are diagrams of the statechart language we are now going to specify. First, we give the alphabet and then the grammar. Thereafter, we show how the statechart in Fig. 1(a) can be obtained by applying grammar rules. Note that we omit some visual means of expressions like the possibility to mark a state as an initial state or a final state, as well as we omit history connectors for the reason of space limits. All these symbols are usually available in the original statechart language well-known from [22,30] and can be captured by the GenGEd concepts as well.

4.1. An alphabet for statecharts

We directly start with the presentation of the symbols used for our statechart language; those are shown in Fig. 13. As before, the upper part denotes the abstract syntax of symbols and the lower part the concrete syntax, i.e., the layout. As described in the previous section, the symbol graphic for symbol State is defined by a rounded rectangle, and that for symbol Trans (transition) by an arrow. Usually, the layout of a parallel state is given by a rounded rectangle with a dashed line inside. In order to obtain such a layout, we define the layout of symbol Parallel by a non-visible

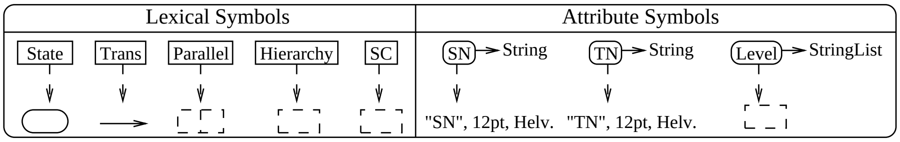


Fig. 13. Symbols of the statechart language.

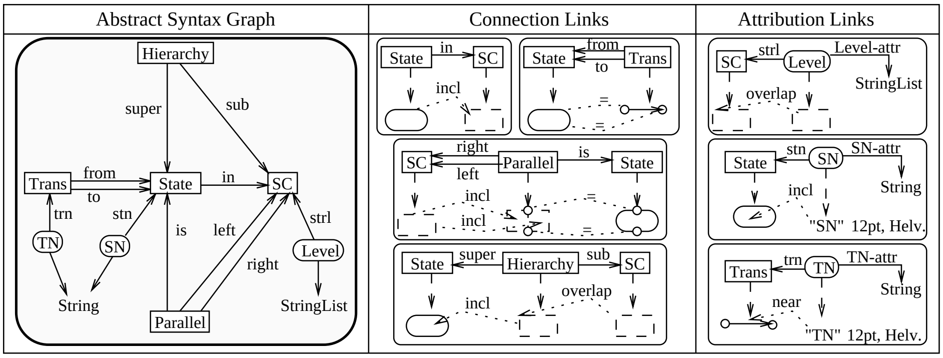


Fig. 14. Links between the statechart symbols.

rectangle, i.e., by a graphical placeholder; we add the dashed line.[5] A reason for this kind of modeling is given by the fact that we do not distinguish terminal and nonterminal symbols usually occurring in an alphabet; we consider terminal symbols only. The layouts of symbols Hierarchy and SC (statechart) are defined by placeholders, too. The attribute symbol Level of type StringList will be used as an internal attribute (linked to the symbol SC) in order to hold the state names on higher levels of hierarchy. This attribute symbol will be used to express the insertion of allowed transitions by a rule. As required, the symbol names are unique in this alphabet.

Fig. 14 illustrates how the symbols are linked. The type graph on the left illustrates the abstract syntax of the alphabet. It is obvious that the link arcs are acyclic as required. The other graphs give a survey on symbols linked with respect to the visual syntax. Some of them are already introduced by Fig. 8, the others will be discussed during the presentation of the rules.

The symbols and links introduced so far are a design decision, i.e. other modeling/ specification possibilities are conceivable. For the development of the alphabet, we started with the symbols well-known from statecharts. We proceeded by considering also the allowed statechart expressions (definable by rules). Then we extended the set of symbols by those with placeholders as layout and defined the corresponding links. Have a look, for example, to the connection link in: State → SC and the attribution link strl: Level → SC. On the instance level these links ensure that each state symbol

[5] In order to keep the example small, parallel states are modeled to hold two sub-statecharts only.

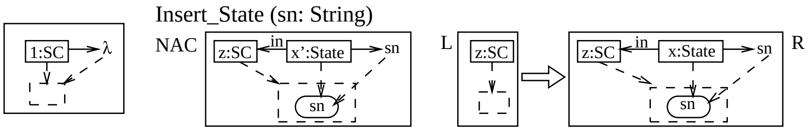
is uniquely associated with a statechart at a certain level. This kind of expression will be used for modeling the composition of statecharts by rules.

4.2. A grammar for statecharts

A grammar is given by a start diagram, and a set of rules defining the editing commands of a Graphical Editor. Here we concentrate on the language-generating rules, i.e., we do not present a fully specification of a Graphical Editor for statecharts. The start diagram of our grammar is shown in Fig. 15(a). It consists of a statechart symbol which is linked to a level list initialized by λ (the empty list). Note that in this start diagram as well as in the following rules we omit the attribute nodes on the abstract syntax level for illustrational reasons; the attribute values are directly appended to the nodes representing lexical symbols.

The rule that allows us to insert a state symbol together with a state name is shown in Fig. 15(b). In its left-hand side (LHS), the existence of a statechart symbol is required which is preserved by the rule. A state and state name symbol is inserted in the rule's right-hand side (RHS) where the state symbol is linked to the statechart symbol according to the alphabet. The negative application condition (NAC) on the left makes sure that the state name is not already in the statechart the rule is applied to.

The insertion of a parallel state is supported by the rule shown in Fig. 16. In its LHS, the existence of a state symbol that is linked to a statechart is required. Both symbols are preserved by the rule. Two further statechart symbols (z2,z3:SC) are inserted in the rule's RHS as well as a parallel symbol whose layout is defined by a non-visible rectangle with the dashed line vertically arranged in the middle. According to the links defined in the alphabet, one of the newly created statechart symbols is placed left to the dashed line, the other one is placed right. Each level list of the statechart symbols z2:SC and z3:SC inherits the level list l of the statechart symbol z1:SC which includes the



(a) (b)

Fig. 15. Start diagram (a) and a rule for inserting a state symbol (b).

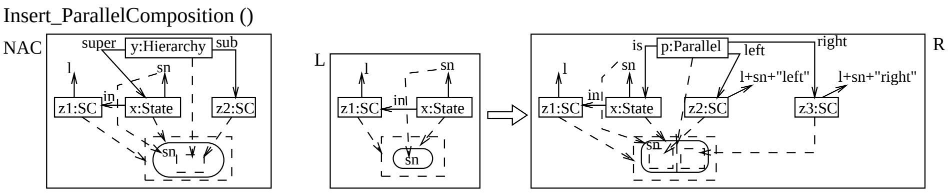


Fig. 16. Rule for inserting a parallel composition.

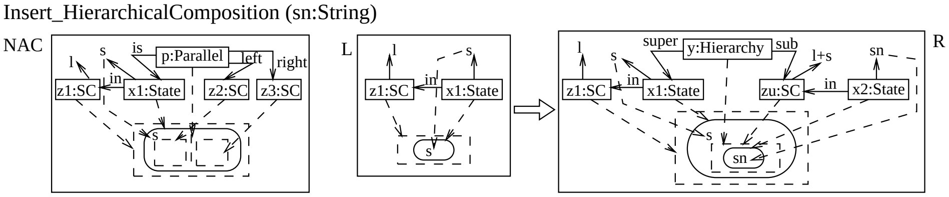


Fig. 17. Rule for inserting a hierarchical composition.

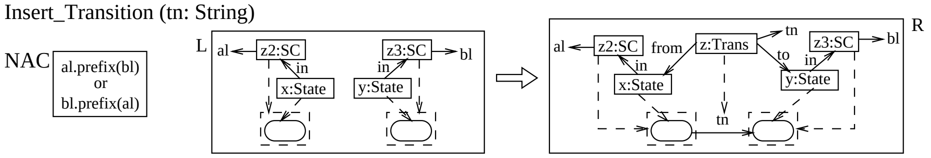


Fig. 18. Rule for inserting a transition between states on different levels.

parallel state x:State. The name *sn* of this state symbol together with markings "left" and "right" are added to the corresponding level lists of the two statechart symbols z2,z3:SC. Note that these markings prohibit the insertion of transition symbols (and corresponding links) between state symbols which will be linked to the left and right statecharts. The NAC requires that the state symbol which gets a parallel state by the rule must not be a super-state involved in a hierarchical composition.

The rule for inserting a hierarchical composition together with a state and state name symbol is shown in Fig. 17. Again, a state symbol that is linked to a statechart symbol is required by the rule's LHS. These symbols are preserved by the rule. In the rule's RHS, a hierarchy symbol is inserted and linked according to the alphabet. The link super denotes the preserved state symbol x1:State to be the super-state; the rectangle of the super-state includes the placeholder of the hierarchy symbol. Link sub indicates the sub-statechart of the hierarchy (zu:SC); the placeholders of the hierarchy and the sub-statechart symbol overlap. The sub-statechart contains a new state symbol x2:State whose name is given by the rule parameter. The state name s:SN belonging to state x1:State which is going to be the super-state by the rule is added to the level list of the sub-statechart zu:SC. As for the parallel state described above, the level list of the statechart allows us to prevent the insertion of transitions between states that are in different statecharts, however, which belong to one parallel state. The NAC connected to this rule makes sure that the intended super-state x1:State is not already a parallel state.

Transitions between states can be inserted by the rule shown in Fig. 18. Like the rules for the parallel and hierarchical composition (cf. Figs. 16 and 17), the level lists of the sub-statecharts in the rule's LHS hold the names of the states on the next higher level. These lists are indicated by the variables al and bl. The statecharts and states occurring in the LHS are preserved by the rule. A transition symbol is inserted in the rule's RHS and the transition name, given by the rule parameter, is linked to this

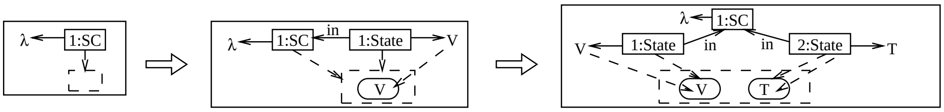


Fig. 19. Two derivation steps starting at the start diagram.

transition symbol. Applying this rule, the attribute condition in the NAC (given by the operation prefix defined over the data type StringList) checks whether the level lists al and bl contain the same name of a parallel state. Exactly this is forbidden when the rule is going to be applied because otherwise the transition would connect two states, each in different statecharts that are sub-statecharts of one parallel state.[6]

Up to now we have defined a small alphabet and the corresponding language-generating rules only. These rules are not sufficient for syntax-directed diagram editing, i.e., some updating rules supporting, for example, the deletion of symbols are also needed.

Usually, the movement of a symbol must not be defined by a grammar rule if the abstract syntax is not taken into account. However, it is obvious that we need several rules for the movement of state symbols, especially if they are involved in a hierarchical or parallel composition. Such rules must express all the editing situations which may occur during statechart editing. The difficulty arises to explain a user of the Graphical Editor the meaning of different rules supporting the movement of one state symbol. A mechanism for the controlled application of repeating rules as available in the Progres language and tool [33] would help. Moreover, the programmed graph transformation approach of Progres provides means for defining path expressions as they are needed to check whether the insertion of a transition is allowed. We used the attribute symbol Level of type StringList with the prefix method for this purpose but the possibility to use path expressions seems to be more convenient. As a result, we will improve GenGEd by such features in the future.

4.3. Sample derivation of a statechart

Now we will give a sample derivation of the statechart shown in Fig. 1(a) using the grammar presented above. We begin with the grammar's start diagram and the insertion of two state symbols by applying the rule in Fig. 15(b) twice. Therefore, we map symbol z1:SC of the rule's LHS to symbol 1:SC in the start diagram and give the state names V and T, respectively. The derivation result is shown in Fig. 19.

We now insert a hierarchy symbol, i.e., we apply the rule in Fig. 17. For the rule application we map z1:SC to 1:SC and x1:State to 1:State, i.e. the symbol 1:State with state name V becomes the super-state; we give the state name S according to the rule parameter. Note that by this rule not only a state symbol is generated but additionally a statechart symbol and a link between these two symbols. This is due to the fact that the hierarchy symbol (the parallel symbol as well) is used as a help symbol: it

[6] In the UML [30] this situation is called *a more esoteric case.*

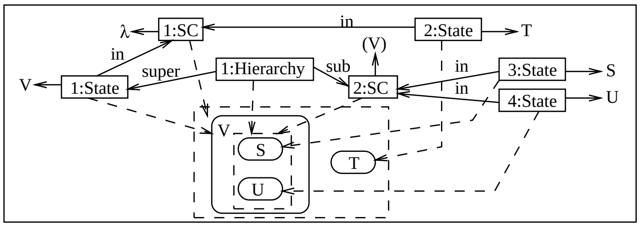


Fig. 20. Insertion of a hierarchical composition and a state.

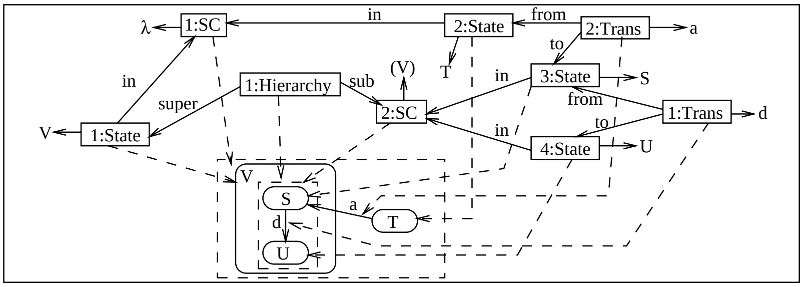


Fig. 21. Insertion of two transitions.

denotes the sub-statecharts. A further reason is given by the dependencies imposed to an alphabet: the links must be acyclic.

After the insertion of a hierarchy symbol (together with a statechart and a state symbol), we insert a further state symbol by applying the rule in Fig. 15(b) with state name U; we map z:SC to the just inserted symbol 2:SC. The result is presented by Fig. 20.

So far, we have inserted all the states occurring in the diagram shown in Fig. 1(a), we now insert a transition symbol by applying the rule shown in Fig. 18. Therefore, we map z2:SC to 2:SC, x:State to 3:State, z3:SC to 2:SC, y:State to 4:State and give the transition name d. Then, we apply the rule again: we map z3:SC to 2:SC, x:State to 2:State, z3:SC to 2:SC, y:State to 3:State and give the transition name a. These steps yield the diagram shown in Fig. 21.

We have to apply the rule in Fig. 18 twice in order to obtain the diagram shown in Fig. 1(a). Although these last steps are omitted, it is obvious that the layout of the statechart generated using the grammar corresponds to the diagram in Fig. 1(a), and moreover, it is obvious that the presented visual grammar is sufficient for the generation of statecharts.

During the presentations in Sections 3 and 4, we have pointed out that in the GenGEd approach there is a clear distinction between the abstract and concrete syntax. The abstract syntax is essential for a formal presentation and manipulation of diagrams, but this formalism can be hidden from both kinds of users. Furthermore, we have demonstrated that the GenGEd approach is sufficient to define a VL (mainly[7]) visually such that also common user requests concerning the concrete syntax are satisfied. So we may have a look to similar approaches in the following.

5. Related work

In the literature one can find many approaches for specifying VLs and creating editors for them [27]. This is the reason that a lot of classification systems exist [32,9] resulting in a difficulty to make a decision for *the best* approach. Such a decision depends on the purpose of the approaches, e.g., whether a visual or textual definition of a VL is in the fore, or which kind of editing mode (freehand or syntax-directed editing) is supported in a Graphical Editor, or—if available—which kind of internal representation model is used.

Most tools for creating freehand editors analyze diagrams directly and avoid to create an internal representation model like a graph [28]. No internal model is taken into account, for example, in VisPro [37], Penguins [10], and Vlcc [12]. 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 special graph grammars and a graph parser are taken into account. However, in VisPro the set of VLs is restricted to diagrammatic VLs, i.e., symbols can be connected by lines and arrows only.

Possibly, freehand editing is desired in a Graphical Editor because a user can create and modify diagrams unrestrictedly; but these diagrams may contain errors.[8] In contrast, 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 now captured in the DiaGen approach [28]. Additionally, an internal representation model is taken into account as it is done by Kogge [15], and by Progres [33].

Progres [33] uses graph grammars for syntax specification and supports syntax-directed editing in a Graphical Editor. The abstract syntax of a VL can be specified using means of graphs, but it is not possible to use the VL vocabulary as it is provided by GenGEd. In Kogge, several formalisms are integrated for syntax specification: *extended entity-relationship* (EER) *descriptions* are provided, A further *Graph Specification Language* allows for the description of language properties that cannot be defined only by an EER. The editing commands of a syntax-directed Graphical Editor are described by statecharts, however, actions are specified by Modula-like programs. In contrast to Kogge, GenGEd provides one formalism for syntax specification. In DiaGen, hypergraphs are taken as internal model, and hypergraph grammars are used

[7] In GENGED, the attribute symbols and the constraints are textually defined.

[8] The discussion whether error detection is sufficient without having an internal model is out of the scope of this article.

for the specification of VLs. DiaGen makes a clear distinction between the abstract and the concrete syntax, and moreover, DiaGen captured the ideas mentioned in [1] and provides both editing modes. However, the specification of a VL is done textually.

6. Conclusion

In this article we proposed GenGEd for the visual definition of VLs, and the generation of graphical editors for VLs specified. So we distinguish VL-specifications and VL-diagrams which can be derived by applying grammar rules to a given start diagram. Furthermore, we distinguish two syntactical levels, namely, the abstract syntax (the logical meaning) and the concrete syntax (the layout) for all constituents.

GenGEd is completely based on graph transformation and graphical constraint solving. The power of graph transformation allows us to define context-sensitive VLs and moreover, we are able to define advanced syntactical checks like unique state names in a statechart. The definition of the statechart language shows how to use GenGEd and that it provides many possibilities. GenGEd is already used for the definition of several VLs like class diagrams and sequence charts (well-known from the UML) as well as Nassi-Shneiderman diagrams. Moreover, in [3] we specified a kind of a Petri-net language and proposed how to define behavior and step-wise animation for a certain (domain-specific) application. It is conceivable that the ideas concerning behavior and animation can be applied to the statechart specification, too.

Up to now, the GenGEd environment supports syntax-directed editing in both, the Grammar Editor as well as in the Graphical Editor. In the Grammar Editor alphabet rules are automatically generated from the alphabet the language-designer has defined. These alphabet rules are the language-specific editing commands for the visual definition of a visual grammar. From the VL-definition (alphabet and grammar), GenGEd generates a VL-specification which is the input of the Graphical Editor where the grammar rules provide the language-specific editing commands. In this way, GenGEd offers one formalism for both, the definition of VLs and the VL-specific Graphical Editor.

In general, syntax-directed editing is not always desired but freehand editing or a combination of both kinds of editing modes as it is supported by DiaGen. We just extended GenGEd, namely we allow for freehand editing in Graphical Editors, where we make use of the Agg parsing features [5]. A further direction concerns the coupling of tools, namely, graph transformation tools with different purposes. The coupling can be realized via XML, the extensible exchange format such that, for example, other tools can use GenGEd for the visual specification of VLs, whereas from the GenGEd point of view other tools can be used, for example, for code generation or verification purposes. A direct coupling via interfaces is already proposed in [4].

Acknowledgements

The efforts concerning GenGEd are sponsored by the German Research Council, the TMR network GETGRATS and the ESPRIT Basic Research WG APPLIGRAPH.

Special thanks to Hartmut Ehrig, Gabi Taentzer and Claudia Ermel for many discussions about the GenGEd approach and the specification of statecharts. The author is also grateful to the anonymous reviewers for fruitful comments on an earlier version of this contribution.

References

[1] M. Andries, G. Engels, J. Rekers, How to represent a visual program? in: K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, Berlin, 1998, pp. 245–260.

[2] R. Bardohl, Visual definition of visual languages based on algebraic graph transformation, Ph.D. Thesis, TU Berlin, 1999, Verlag Dr. Kovac, 2000.

[3] R. Bardohl, H. Ehrig, C. Ermel, Generic description, behaviour and animation of visual modeling languages, Proc. Integrated Design and Process Technology (IDPT'00), 2000.

[4] R. Bardohl, C. Ermel, L. Ribeiro, A modular approch to animation of simulation models, Proc. Brazilian Symp. on Software Engineering (SBES'00), 2000, pp. 498–520.

[5] R. Bardohl, T. Schultzke, G. Taentzer, Visual language parsing in GENGED, Proc. Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'01), ENTCS 50(3), (2001).

[6] R. Bardohl, G. Taentzer, M. Minas, A. Schürr, Application of graph transformation to visual languages, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation. vol. 2: Applications, Languages and Tools, World Scientific, Singapore, 1999, pp. 103–180.

[7] A.H. Borning, The programming language aspects of thinglab, A constraint oriented simulation laboratory, ACM Trans. on Programming Languages and Systems 3 (4) (1981) 353–387.

[8] M. Burnett, A. Goldberg, G. Lewis (Eds.), Visual Object-Oriented Programming: Concepts and Environments, Manning Publications Co., Greenwich, 1995.

[9] M. Burnett, Visual language research bibliography. URL: http://www.cs.orst.edu/burnett/vpl.html.

[10] S.S. Chok, K. Marriott, Automatic construction of user interfaces from constraint multiset grammars, Proc. IEEE Symp. on Visual Languages, 1995, pp. 242–249.

[11] A. Corradini, U. Montanari, F. Rossi, Graph processes, Special Issue of Fund. Inform. 26 (3,4) (1996) 241–266.

[12] G. Costagliola, A.D. Lucia, S. Orefice, G. Totora, Positional grammars: a formalism for LR-like parsing of visual languages, in: K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, Berlin, 1998, pp. 171–192.

[13] G. Costagliola, G. Polese, Extended positional grammars, Proc. IEEE Symp. on Visual Languages, 2000, pp. 171–192.

[14] R. Dechter, P. van Beek, Local and global relational consistency, Theoret. Comput. Sci. 173 (1997) 283–308.

[15] J. Ebert, A. Franzke, A declarative approach to graph based modeling, Proc. Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, vol. 903, Springer, Berlin, 1995, pp. 38–50.

[16] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, World Scientific, Singapore, 1999.

[17] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini, Algebraic approaches to graph transformation II: single pushout approach and comparison with double pushout approach, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, vol. 1, World Scientific, Singapore, 1997, pp. 247–312.

[18] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics, EACTS Monographs on Theoretical Computer Science, vol. 6, Springer, Berlin, 1985.

[19] C. Ermel, M. Rudolf, G. Taentzer, The AGG-approach: language and tool environment, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by

Graph Transformation, vol. 2: Applications, Languages and Tools, World Scientific, Singapore, 1999, pp. 551–603.

[20] P. Griebel, ParCon—Paralleles Lösen von grafischen Constraints, Ph.D. Thesis, Paderborn University, February 1996.

[21] A. Habel, R. Heckel, G. Taentzer, Graph grammars with negative application conditions, Special Issue of Fund. Inform. 26 (3,4) (1996) 287–313.

[22] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Programming 8 (1987) 231–274.

[23] M. Koch, Integration of graph transformation and temporal logic for the specification of distributed systems, Ph.D. Thesis, TU Berlin, 1999.

[24] M. Korff, Generalized graph structure grammars with applications to concurrent object-oriented systems, Ph.D. Thesis, TU Berlin, 1996.

[25] M. Löwe, M. Korff, A. Wagner, An algebraic framework for the transformation of attributed graphs, in: M.R. Sleep, M.J. Plasmeijer, M.C. van Eekelen (Eds.), Term Graph Rewriting: Theory and Practice, Wiley, New York, 1993, pp. 185–199 (chapter 14).

[26] K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, Berlin, 1998.

[27] K. Marriott, B. Meyer, K. Wittenburg, A survey of visual language specification and recognition, in: K. Marriott, B. Meyer (Eds.), Visual Language Theory, Springer, Berlin, 1998, pp. 5–86.

[28] M. Minas, B. Hoffmann, Specifying and implementing visual process modeling languages with DIAGEN, Proc. Uniform Approaches to Graphical Process Specification Techniques (Unigra'01), Genova, Italy, 2000.

[29] B. Myers, Garnet: comprehensive support for graphical, highly interactive user interfaces Computer 23 (11) (1990) 71–85.

[30] Rational software corporation. UML—Unified Modeling Language, version 1.3, Technical Report, URL: http://www.rational.com, 1999.

[31] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformaitons, vol. 1, Foundations, World Scientific, Singapore, 1997.

[32] J. Schiffer, Visuelle programmierung, Addison-Wesley, Reading, MA, 1998.

[33] A. Schürr, A.J. Winter, A. Zündorf, The PROGRES approach: language and tool environment, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, World Scientific, Singapore, 1999, pp. 487–550.

[34] N.C. Shu, Visual programming languages: a perspective and a dimensional analysis, in: S.-K. Chang, T. Ichikawa, P.A. Ligomenides (Eds.), Visual Languages, Plenum Press, New York, 1986.

[35] Ivan E. Sutherland, SketchPad: a man–machine graphical communications system, Proc. IFIPS Joint Computer Conference, 1963, pp. 329–345.

[36] The object managment group. URL: http://www.omg.org.

[37] D.-Q. Zhang, K. Zhang, VisPro: a visual language generation toolset, Proc. IEEE Symp. on Visual Languages, 1998, pp. 195–202.