Generation of Visual Editors as Eclipse Plug-Ins[*][†]

Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer,
Technische Universität Berlin, Germany
Email: {karstene,lieske,haensgen,gabi}@cs.tu-berlin.de

ABSTRACT

Visual Languages (VLs) play an important role in software system development. Especially when looking at well-defined domains, a broad variety of domain specific visual languages are used for the development of new applications. These languages are typically developed specifically for a certain domain in a way that domain concepts occur as primitives in the language alphabet. Visual modeling environments are needed to support rapid development of domain-specific solutions.

In this contribution we present a general approach for defining visual languages and for generating language-specific tool environments. The visual language definition is again given in a visual manner and precise enough to completely generate the visual environment. The underlying technology is Eclipse with its plug-in capabilities on the one hand, and formal graph transformation techniques on the other hand. More precisely, we present an Eclipse plug-in generating Java code for visual modeling plug-ins which can be directly executed in the Eclipse Runtime-Workbench.

1. INTRODUCTION

In software system development, often application-specific visual notations are used for which a tool environment consisting of visual editors, simulators, etc. is needed. Existing approaches for generating the desired tool environments rely on meta-modeling concepts, grammars, or some kind of logics. In the following, we concentrate on generators based on graph transformation like DiaGen [18], AToM³ [15] and GenGED [1], which allow the precise description of visual modeling languages and the generation of visual environments. Furthermore, we consider the development environment Eclipse [3] which offers rich support for graphical edi-

tor development on a model-view-controller basis in form of a number of plug-ins (e.g. EMF [6], Draw2D and the Graphical Editor Framework GEF [4]). A visual editor based on GEF has to be hand-coded, as no high-level description of visual representations is offered to support a complete editor generation. The aim of this paper is to bring together graph transformation-based tool generation with the Eclipse technology.

Graph-transformation based editor generators have the benefit of providing a solid, precise visual language (VL) specification. The static part of such a formal VL specification, i.e. the VL alphabet, is given by a type graph (the abstract syntax of the VL) plus the necessary layout specification. Using graph transformation, an editor operation is modeled in a rule-based way by just specifying the pre- and post-conditions of such an operation. The application of such syntax rules to the syntax graph of a diagram is performed by the graph transformation engine AGG [24], developed also at the Technical University of Berlin. AGG makes direct use of graph transformation concepts following the double-pushout approach to typed, attributed graph transformation [11].

In this paper, we present a new tool environment, called Tiger [25] (Transformation-based generation of modeling environments), the first ideas of which have been presented in [10]. Tiger combines the advantages of precise VL specification techniques (offered by the graph transformation engine AGG) with sophisticated graphical editor development features (offered by GEF). Graph transformation is used on the abstract syntax level. Tiger extends the AGG engine by a concrete visual syntax definition for flexible means for visual model representation. From the definition of the visual language, the Tiger *generator* generates Java [23] source code. The generated Java code implements an Eclipse visual editor plug-in based on GEF which makes use of a variety of GEF's predefined editor functionalities. Hence, the generated editor plug-in appears in a timely fashion. Moreover, the generated editor code may easily be extended by further functionalities.

The paper is organized as follows: In Section 2, we review and compare the basic concepts of visual editor development, being based on graph transformation on the one hand, and on Eclipse/GEF on the other hand. Section 3 describes the aims and the architecture of the Tiger framework for visual language specification and GEF-based editor generation. In Section 4 we give an introduction to graph transformation and in Section 5, we discuss VL specification by Tiger. In Section 6, we describe the generation of editors

for graph-like diagrams on the basis of a VL specification. Petri nets and activity diagrams are chosen as sample VLs to demonstrate the use of the TIGER VL structures and to present the look-and-feel of the generated editors. The paper concludes with an outlook on future development steps of TIGER.

2. GRAPHICAL EDITOR GENERATION: STATE OF THE ART

In this chapter, we review the state of the art of model-based graphical editor generation. We compare editor generation concepts using models for visual languages based on graph transformation to editor generation concepts based on models defined in the Eclipse modeling framework EMF.

2.1 Graph-Transformation based Editor Generation

The static part of a graph transformation-based VL specification, i.e. the VL alphabet, is given by a type graph (the abstract syntax of the VL) plus the necessary layout specification. An editing operation in the generated editor is combined with a corresponding change of the internal abstract syntax graph of the diagram. An editor operation is modeled as a graph rule (typed over the VL type graph) being applied to the abstract syntax graph of the current diagram. The graph grammar defining these editor operations is called *VL syntax grammar* because it defines (together with the VL alphabet) the complete syntax of the VL. Syntax rules are a well-defined and constructive way to express which diagrams belong to a VL. Editing based on a syntax grammar is called *syntax-directed editing* and allows to edit syntactically correct diagrams only. Besides, syntax rules can specify complex editing operations like the insertion of a complete *if-then-else* construct in activity diagrams in one step (see Section 7.2.2, where a syntax grammar for activity diagrams is discussed). When a diagram has been edited, other graph rules (like simulation rules) can be applied to perform model simulation or model analysis, based on the same diagram.

Visual environment generators like DIAGEN [18], ATOM³ [15] and GENGED [1] generate their own VL specific editors from VL specifications based on graph transformation. The created editor features (e.g. for layouting diagrams, undo/redo, zooming, etc.) differ heavily. In GENGED, layouting is based purely on graphical constraint solving, a flexible and elegant way to model layout constraints. Unfortunately, in some cases this leads to performance problems as the computation of large constraint satisfaction problems can be quite complex. Therefore, GENGED (as well as DIAGEN) allows the editor designer to write VL-specific layout algorithms in Java which replace the constraint solver when the VL becomes more complex. This solution requires some knowledge in Java programming and about the internal language model from the editor designer. ATOM³ offers a standard layout algorithm in its generated editors which is adequate only for simple graph-like diagrams. For more complex VLs, the editor user has to take care of an adequate diagram layout.

All of the generated environments are not meant to be integrated into other existing tool environments. As stand alone applications they do not always offer the standard look-and-feel of common editor features.

2.2 Eclipse-based Editor Generation

ECLIPSE [3] is an open platform for tool integration managed by an open community, written in Java [23]. Its *plug-in technology* allows flexible program development and integration. Extensive plug-in construction toolkits and examples allow the easy development of own application plug-ins supporting specific application functionalities.

The *Eclipse Modeling Framework (EMF)* [6] allows to generate code from meta-models, called *models* in EMF, defined as class diagrams using appropriate CASE plug-ins. Using EMF for visual language specification, the class diagram describes mainly the abstract syntax, (i.e. the symbol and link types used in the diagrams) but does not contain information about their concrete layout, such as shapes and lines and their poperties. The generated model code thus consists of the basic classes allowing to handle the internal model of the editor. Furthermore, EMF allows to generate a primitive, tree-based editor which can directly be executed in the Eclipse Runtime-Workbench. In this editor, a "diagram" can be edited by defining instances for the symbols and values for their properties, in order to test the underlying generated model code, but is not layouted visually.

The *Graphical Editor Framework (GEF)* [4] plug-in is part of the ECLIPSE project and allows the designer to develop graphical editors for models of a specific application domain. GEF-based editors require at least a minimal Eclipse environment such as the *Eclipse Runtime-Workbench*. A GEF-based editor is linked by action handlers to the Eclipse environment. Basic and advanced editor functionalities are offered by GEF to be included in a graphical editor.

Unfortunately, EMF does not support the generation of graphical editors based on GEF. Therefore, the model-specific editor features must be coded by hand, e.g. by defining figures for the concrete layout of diagrams in the graphical editor, and commands to be used in the editor, thereby strictly obeying the *Model-View-Controller* architecture of GEF applications. The *model* in GEF is a distinct package offering all model-manipulating operations (and may be generated by EMF). The values defined by the model are the only data that are saved and restored for each model instance (diagram). Hence, the model has to include also the part of the layout information (e.g. symbol positions) which is specific to the diagram.

In order to bridge the gap between EMF-models and GEF-based graphical editors, the ECLIPSE *Graphical Modeling Framework (GMF)* project [5] started recently as Eclipse technology subproject and aims to provide the fundamental infrastructure and components for developing visual design and modeling surfaces in ECLIPSE. In essence, a diagram definition will be linked to a domain visual language model which serves as input to the generation of a visual editor. GMF is a generative approach allowing to add diagramming capabilities to a visual language model expressed in EMF where a visual editor is desired. In many ways, GMF is an extension to the capabilities of EMF.

The disadvantage of the Eclipse approach to visual editor generation based on EMF/GEF and the GMF project, is that the underlying meta-model (i.e. the EMF model) mainly defines the visual language alphabet. Therefore it may be the case that an editor based on this model allows the editing of diagrams which are not valid in the VL. Additional language constraints can be expressed by adding e.g. OCL constraints [20] to the EMF model. A resulting edi-

tor can only offer simple editing operations based on such a visual language specification. For the generation of syntax-directed editor operations, the graph transformation-based approach to VL definition offers better support.

Hence, our approach is to combine GEF features and formal, graph-transformation based VL specifications instead of EMF models in the visual editor generator TIGER.

3. THE TIGER ENVIRONMENT: AIMS AND ARCHITECTURE

The overall aim of TIGER is to allow the generation of modeling environments based on GEF and on formal graph transformation specifications defining, checking or transforming the diagrams of a specific VL.

For the graph transformation structures used for VL definition, we rely on the tool environment AGG which offers not only a graph transformation engine for typed, attributed, conditional graph rewriting but also algorithms for checking graph conditions and analysis of graph grammars (such as critical-pair analysis). These analysis techniques can be used to provide syntactic as well as semantic checks on visual languages. Despite of a similar approach, the formal basis of TIGER differs from that of GenGED [1]. Instead of transforming attributed graph structures as done in GenGED, we transform typed attributed graphs now. Both kinds of graphs are equally powerful [11], but typed attributed graphs offer a simpler and more compact approach to visual language definition.

For the generation of modeling environment components we rely on the GEF framework. Visual editors based on model definitions, will play an important role within nearly all TIGER components: On the one hand, we will have the *designer* component which allows the visual definition of VL specifications themselves from which the modeling environment is generated, and, on the other hand, we will have the generated modeling components, namely the *editor* component for editing a model, the *simulator/animator* component for simulating/animating a model's behavior, the *analysis* component for performing model analysis and visualizing the analysis results, the *model transformation* component [9] for realizing model conversions from one modeling language to another (e.g. from function block diagrams to Petri nets in order to perform Petri net based analyis of the model). All these components need visual editors or at least viewers for showing parts of the model or intermediate model states. In the following we summarize all the different components which may be generated from one VL specification by the notion $<vl>.environment$, where $<vl>$ should be replaced with the concrete visual language name.

In Fig. 1, the basic components of the TIGER software architecture are shown. The basic AGG data structures are in the package *AGG*. The packages *tiger.generator* and *tiger.vlspec* build the core packages of the TIGER tool. The VL specification will be designed by the VL designer using the *tiger.designer* component building the VL specification using the data structures defined in the *tiger.vlspec* package.

The *tiger.generator* generates diagram editors, as well as the envisaged simulation, animation, analysis and model transformation components belonging to the $<vl>.environment$. Moreover, the *tiger.generator* generates the code for the visual runtime model from the VL specification in the package $<vl>.model$. The basic VL specification package
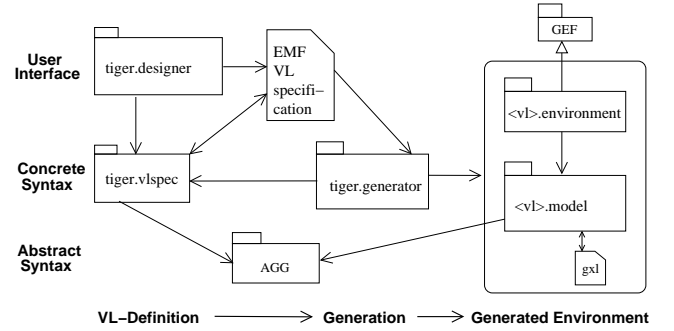

Figure 1: Software Architecture

tiger.vlspec allows to create and modify VL specifications. They are saved in an EMF VL specification model [6]. The visual runtime model $<vl>.model$ could be exported to an XML based exchange format for graphs (GXL [27]) in order to allow tool cooperation in the graph transformation community.

Since the TIGER project is ongoing work in an early stage, we here present the current state of the development. We have imposed the following design decisions on the first development step:

We generate diagram editors (instead of complete modeling environments) on the basis of a VL specification which consists of a VL alphabet and a syntax grammar; thus we generate graph-transformation based visual editors. Up to now, we allow graph-like languages only such as Petri nets or activity diagrams (see also [17]). For source code generation, we use *Eclipse JET* [7] as part of the *Eclipse Modeling Framework (EMF)* [6].

4. GRAPH TRANSFORMATION

The main idea of graph grammars and graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a graph transformation step. Graph grammars can be used on the one hand to generate graph languages by Chomsky grammars in formal language theory. On the other hand, graphs can be used to model the states of all kinds of systems which allows to use graph transformation to model state changes of these systems.

The core of a graph transformation rule $p = (LHS, RHS)$ is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective morphism $r : LHS \rightarrow RHS$ as shown in Fig. 2. Applying the rule $p = (LHS, RHS)$ means to find a match of LHS in the source graph and to replace LHS by RHS leading to the target graph of the graph transformation. Especially for the application of graph transformation techniques to visual language modeling, *typed attributed graph transformation systems* [11, 8] have proven to be an adequate formalism. A VL is modeled by a type graph capturing the definition of the underlying visual alphabet, i.e. the symbols and relations which are available. Sentences or diagrams of the VL are given by graphs typed over the type graph. In order to restrict the visual sentences to valid visual models, a syntax graph grammar is defined, consisting of a set of language-generating graph transformation rules describing editing operations which lead to the construction of valid visual models only.

Definition 4.1 (*Graph Transformation*)
Let $p = (LHS \rightarrow RHS)$ be a typed graph transformation rule and G a typed graph with a typed graph morphism $m : LHS \rightarrow G$, called match. A *graph transformation step* $G \overset{p,m}{\Longrightarrow} H$ from G to a typed graph H via rule p, match m, and comatch m^* is shown in Fig. 2. The rule p could be extended by a set of *negative application conditions (NACs)* [14, 8]. The match $m : LHS \rightarrow G$ satisfies the NAC with the injective NAC morphism $n : LHS \rightarrow NAC$, if there does not exist an injective graph morphism $q : NAC \rightarrow G$ with $q \circ n = m$. A sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of graph transformation steps is called *transformation* and denoted as $G_0 \overset{*}{\Rightarrow} G_n$. $\triangle$

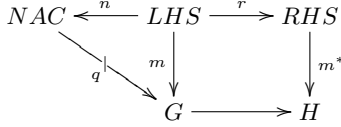$$NAC \xleftarrow{n} LHS \xrightarrow{r} RHS$$

Figure 2: Graph Transformation Step

Intuitively, the application of rule p to graph G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing condition* is satisfied, i.e. the deletion step must not leave *dangling edges*, and for two objects which are identified by the match, the rule must not preserve one of them and delete the other one.

Now we define graph grammars and languages. The language of a graph grammar consists of the graphs that can be derived from the start graph by applying the transformation rules.

Definition 4.2 (*Graph Grammar and Language*)
A *typed graph grammar* $GG = (TG, P, S)$ consists of a type graph TG, a set of typed graph transformation rules P, and a typed start graph S.
The *graph language* L of GG is defined by

$$L = \{G \mid \exists \ typed \ graph \ transformation \ S \overset{*}{\Rightarrow} G\}.$$

$\triangle$

Although we do not define the attribution concept for graphs formally in this paper (see [11, 8] for a complete definition of the theory), we use node attributes in our examples, e.g. text for the names of nodes, or integers for their positions. This allows us to perform computations on attributes in our rules and offers a powerful modeling approach. For flexible rule application, variables for attributes can be used, which are instantiated by concrete values in the rule match.

An example for a graph grammar with NACs and node attributes is the visual syntax grammar for Petri nets (see Fig. 9) which is explained in detail in Section 7. Graph objects which are preserved by the rule occur in both L and R (indicated by equal numbers for the same objects).

5. VISUAL EDITOR SPECIFICATION

Fig. 3 shows the packages *abstractsyntax* and *rules* for a VL specification (*VLSpec*) which consists of a *Alphabet*, a *RuleSet* and a *StartGraph*.

5.1 The VL Alphabet

A VL alphabet consists of *SymbolTypes* and *LinkTypes*. In our approach, graph-like languages consist of *NodeSymbolTypes* (e.g. places and transitions in Petri nets) and *EdgeSymbolTypes* (e.g. arcs in Petri nets). *EdgeSymbolTypes* are connected to *NodeSymbolTypes* by *LinkTypes*. *SymbolTypes* may be attributed by an ordered list of *AttributeTypes* e.g. to model the place and transition names in Petri nets. Classes *AttributeType*, *SymbolType* and *LinkType* have directly corresponding node and edge types in AGG forming the abstract syntax representation.

Graphical Layout

The graphical layout (the concrete syntax) is given by additional classes extending the class diagram shown in Fig. 3, as indicated in Fig. 4, where the upper part shows the main classes for the abstract syntax, and the lower part the additional classes for the concrete syntax.
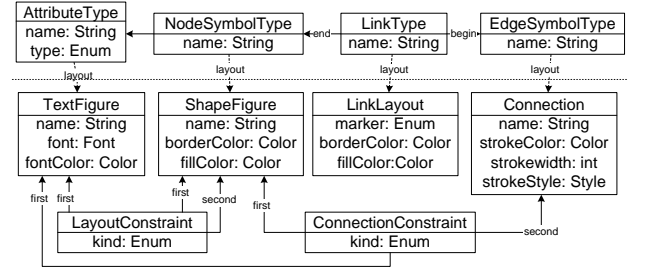

Figure 4: Concrete Syntax in VL Specifications

The graphical layout for a *NodeSymbol* of a certain *NodeSymbolType* is given by class *ShapeFigure*. The shape of *NodeSymbols* can be a simple form, e.g. a rectangle, circle, ellipse or a closed polygon. The stroke and fill colors are given by attributes *borderColor* and *fillColor*. The standard layout for a textual attribute of type *AttributeType* is given by class *TextFigure* with attributes *font* and *fontColor* where *font* is of type *java.awt.Font* which already includes style and size attributes. The graphical relations between *TextFigures* and *ShapeFigures* are expressed by *LayoutConstraints*, such as *below(TextFigure, ShapeFigure)* on interface *Figure*.

Figures can be connected by *Connections* which represent the concrete graphical layout for the *EdgeSymbolTypes*. Connections are lines or polylines. The graphical layout is given by attributes *strokeColor*, *strokeWidth*, and *strokeStyle* (e.g. dashed or solid connection). The graphical representation of a link (e.g. a colored arrow head) is modeled by attributes *marker*, *borderColor*, and *fillColor* of class *LinkLayout*. Graphical relations between a *Figure* and a *Connection* can be modeled as *ConnectionConstraints*, such as *atCenter(TextFigure, Connection)*. For graph-like languages we use the default GEF graph layouter and therefore we do not need a graphical constraint solver to compute the layout of the symbols and links.

5.2 The Syntax Rules

The *RuleSet* of a VL specification contains the rules for syntax-directed editing, defining high level diagram modification operations in a flexible way. The design for rule sets is the following: a *Rule* consists of (at least two) *Graphs*, a left hand side (*LHS*), a right hand side (*RHS*), and, optional
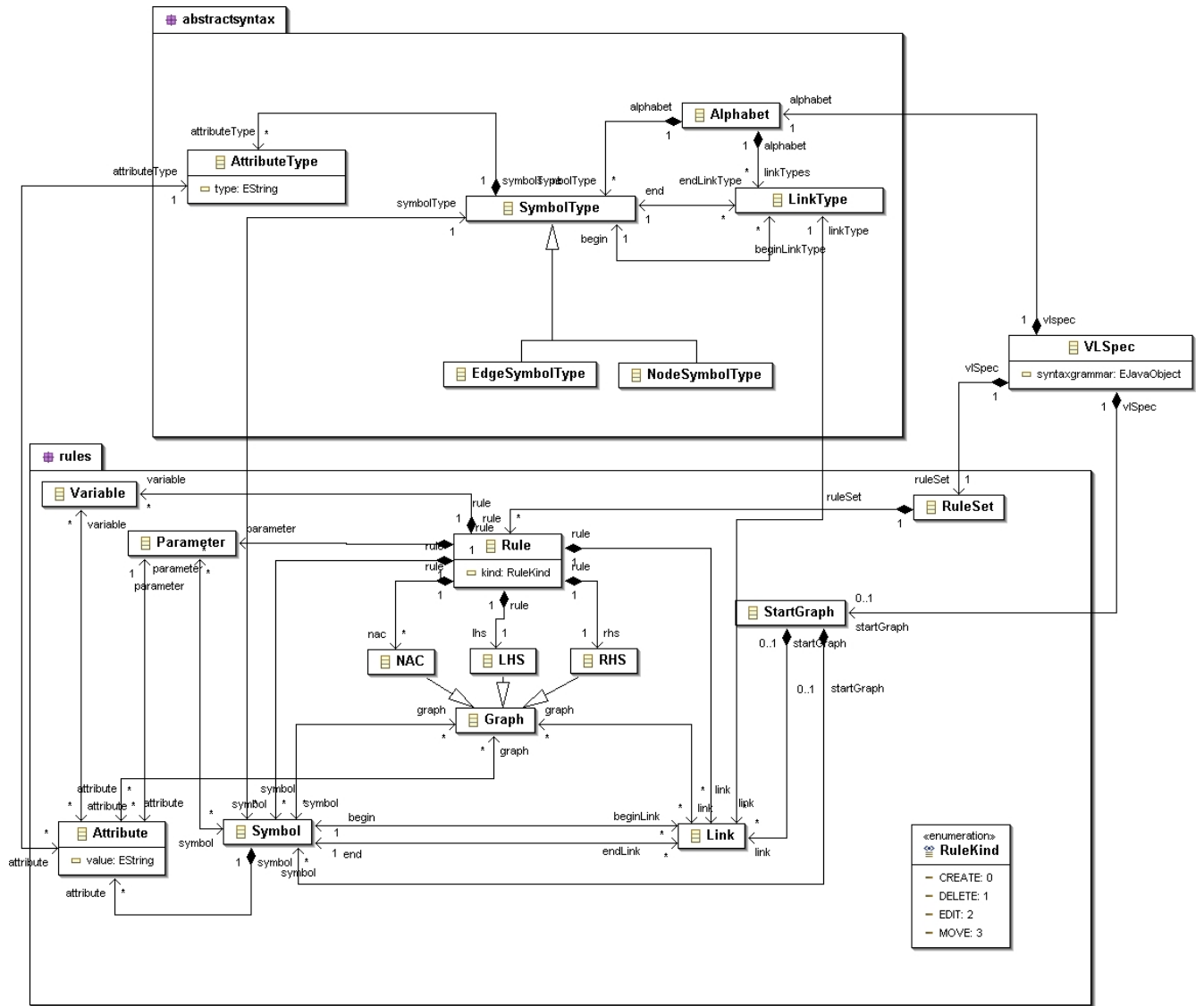
Figure 3: VL Specification

negative application conditions (*NACs*). A rule morphism between a *Symbol* or *Link* exists iff *LHS* and *RHS* resp. *NAC* contain the same *Symbol* or *Link*. In addition a *Rule* may have input *Parameters* which could be *Attributes* or *Symbols* selected by the user, and *Variables* for manipulating and comparing *Attribute* values during the rule application.

The attribute *kind* associates a *Rule* with a specific *Rule-Kind*

- *CREATE*: New *Symbols* are created.

- *DELETE*: Existing *Symbols* are deleted.

- *EDIT*: Attribute values of existing *Symbols* are changed.

- *MOVE*: *NodeSymbols* are moved in the editor to new positions.

The *RuleKind* controls the generation of user interface components (such as entries in the editor palette or in context menus).

Fig. 5 shows how the different kinds of syntax rules influence the editor generator to generate different user interface components for the application of the specific rule kinds. Hence, the expected user actions for the application of a rule depend on the rule kind and on the type and number of the symbol(s) the rule is applied to.

By default, rules of kind *CREATE* are represented by symbol icons or rule names in the editor palette. Rules for deletion or attribute changes (kinds *DELETE* or *EDIT*) are applied via a context menu after selecting the symbols for the rule match in the editor panel. *MOVE* rules are called when symbols are moved in the panel by mouse. User guidance concerning the required input parameters is given by popup dialog windows. Information of currently required user actions (e.g. the selection of specific match symbols) and error information are displayed in a status line below the editor panel.

Note that the generated editor provides a grouping function *Marquee* in the palette to select more than one symbols

Rule Kind	Rule		User Action to trigger Rule Application
	LHS	RHS	
CREATE₁	empty	one NodeSymbol	select *NodeSymbol* in *Symbol group* of the editor palette; select required match symbols in editor panel; click in editor panel to create *NodeSymbol* at click position.
CREATE₂	two NodeSymbols	one EdgeSymbol connecting the two NodeSymbols	select *EdgeSymbol* in *Connection group* of the editor palette; select required match symbols in editor panel; click on source and on target *NodeSymbols* in the panel to create an *EdgeSymbol* between them.
CREATE₃	neither case **CREATE₁**, nor case **CREATE₂**		select rule name in the editor palette; select required match symbols in editor panel.
DELETE₁	one NodeSymbol	empty	select *NodeSymbol* in the editor panel; select "Delete" operation from context menu.
DELETE₂	one EdgeSymbol connecting two NodeSymbols	the two NodeSymbols	select *EdgeSymbol* in the editor panel; select "Delete" operation from context menu.
DELETE₃	neither case **DELETE₁**, nor case **DELETE₂**		select required match symbols in editor panel; select "Delete" operation from context menu of one of the match symbols
EDIT	one Symbol	same Symbol, changed attributes	select *Symbol* in the editor panel; select edit operation from context menu (right mouse button); edit attribute values in property dialog.
MOVE	one NodeSymbol	same Symbol, changed position attributes	move *NodeSymbol* by dragging it over the editor panel; release mouse button at target position.

Figure 5: Rule Kinds and User Interface

at once. For such a group of selected symbols, *MOVE* operations can be applied in the same way as for single symbols. Internally, the corresponding *MOVE* rules for each of the selected symbols are applied in parallel.

6. THE TIGER GENERATOR

The TIGER generator (package *tiger.generator*) generates Java source code from the visual language specification and the corresponding syntax grammar (package *tiger.vlspec*).

6.1 The Generation Process

For the editor generation the JET compiler [7] reads the VL specification and the generator template files which define the code generation skeleton with code placeholders. JET replaces the placeholders with the specific code from the VL specification.

The generation process leads to a new *Eclipse Editor Plug-in Project* which can be directly executed in the *Eclipse Runtime-Environment* (see Fig. 6).
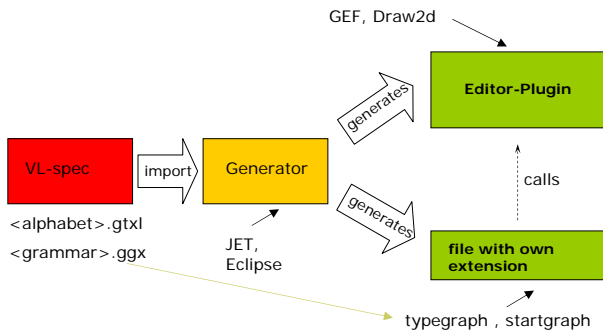

Figure 6: Generator Components Overview

Furthermore the generator creates a file with an own extension which contains the type graph and the start graph from the given syntax grammar (see VLSpec in Fig. 6). The data management classes of the generator encapsulate the *SymbolTypes* of the VL alphabet and for each *SymbolType* a controller class is generated (in GEF „EditPart"). These *EditParts* represent a controller framework linking the editor and the underlying model instance which is directly represented by an AGG *instance graph*.

The layout constraints defined in the VL alphabet (e.g. *above, below, inside, right, left* for figures and *atSource, atCenter, atTarget* for connections) are translated by the generator to static GEF *constraints* which are data attached to each figure giving additional guidance to the GEF layout manager. For example, in the case of the layout constraint *below(text, shape)* the generator treats the text as *child* of the shape figure. The position of the child is computed relative to the position of its parent figure.

6.2 The Generated Editor

A generated GEF-based editor is divided into two parts: on the left hand side is the editor palette and on the right side the editor panel (see the sample editor in Fig. 7). The editor palette has the two default items *Select* for marking an editor symbol in the panel and *Marquee* for selecting a set of symbols in the panel with a dashed rectangle. Below the default items the palette may contain further icons for the generated VL-specific language elements (which in Fig. 7 represent *SymbolTypes* of the Petri net VL). For user inter-
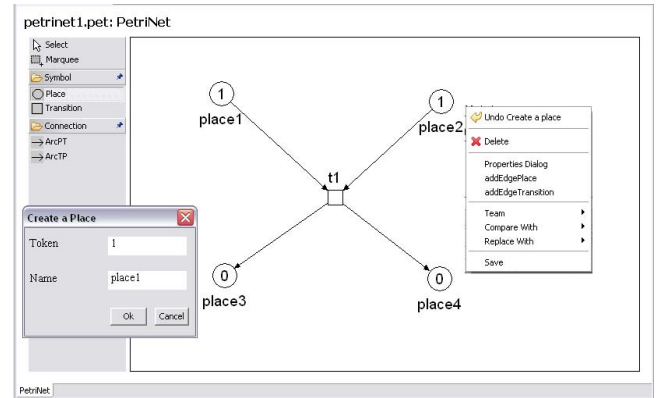

Figure 7: Generated Editor for Petri Nets

action *context menus* and *dialogs* are generated. For each *SymbolType* which has visible attributes such as „name", a dialog is created by the generator (see e.g. the "Create a Place" dialog in Fig.7). The creation of the context menu and the contents of the palette depends on the rule kind (see Fig. 5). For a simple *CREATE* rule (i.e. case $CREATE_1$ or $CREATE_2$) which creates exactly one symbol, an entry in the editor palette is generated, with the name of the *SymbolType*, either in the *Symbol* group (if the rule creates a NodeSymbol, which corresponds to case $CREATE_1$) or in the *Connection* group (if the rule creates an EdgeSymbol, which corresponds to case $CREATE_2$). For more complex *CREATE* rules (case $CREATE_3$), an entry of the rule name is generated in the editor palette. For an *EDIT* rule, an entry of the rule name in the context menu of the corresponding *SymbolType* is generated. For simple *DELETE*

rules (case $DELETE_1$ or $DELETE_2$), which delete one symbol, an entry "Delete" is generated in the context menu of the corresponding *SymbolType* (see e.g. the context menu for place2 in Fig. 7). More complex *DELETE* rules (case $DELETE_3$) lead to the generation of the rule name in the context menus of the types of all deleted symbols. A *MOVE* rule is coupled to the mouse listener.

Rules of all kinds are applied to the abstract syntax of the current diagram in the editor panel. Note that for positions, we have the exception that they are stored as abstract syntax attributes although they rather would belong to the concrete syntax. The reason for this exception is that the persistent parts of a diagram, i.e. the parts that have to be saved, in this way correspond precisely to the diagram's abstract syntax. Hence, the transformation can be executed using the AGG transformation engine. The changes of the diagram lead to an update of the editor view via the corresponding *EditParts*. If an error occurs during the rule application, an exception is thrown by AGG that is displayed as an error message in the status line of the editor (not shown in the screen dumps). The transformed diagram is directly displayed in the editor panel by the editor controller framework.

7. EXAMPLES

As examples we present two visual languages and the respective generated editors, first for place/transition Petri nets (P/T nets) [22] and second for simple UML activity diagrams [21].

7.1 The Generated Petri Net Editor

In Petri nets, places are visualized as ellipses, and transitions as rectangles. The marking of a place is represented as natural number inside the place ellipse. Places and transitions have names which are shown below the corresponding ellipse or rectangle. For simplicity, arc weights uniformly correspond to the token number "1", hence arc inscriptions are omitted here. Fig. 7 shows the generated GEF-based P/T net editor which relies on the VL specification for P/T nets given in this section.

The editor panel contains a sample Petri net with one transition (*t1*), two pre-places *place1, place2* which are marked by one token each, and two unmarked post-places *place3* and *place4*. The symbol properties such as names or token numbers can be changed in a *Properties Dialog*.

We now give the components of the P/T net VL specification in detail, namely the P/T net alphabet and the P/T net syntax grammar.

7.1.1 The VL alphabet

The alphabet for the VL of P/T nets is presented in Fig. 8 and conforms to the general structure of VL alphabets as given in Fig. 3. For the abstract syntax (the upper part of Fig. 8) we use node symbol types Place and Transition for the Petri net nodes, edge symbol types ArcPT for Petri net arcs from a place to a transition and ArcTP for arcs from a transition to a place. Link types arcPTsource, arcPTtarget, arcTPsource and arcTPtarget are used for linking the edge symbols to the node symbols. Attribute types (textual attributes) include the names of places and transitions, and the token number in a place. Additionally, the model-specific layout information in form of symbol positions is stored in attribute types. Model-independent layout information (the lower

part of Fig. 8) is given by shape figures, connections and text figures linked to the corresponding node symbol types, edge symbol types and attribute types, respectively. All these attributes describe model-independent layout. They are set to default values, e.g. shapeColor=black and fill-Color=none for shape figures and font=("Arial", Font.ITALIC, 12), fontColor=black for text figures. For our Petri net editor we use these default values, and hence do not have to define layout attribute values in the alphabet.
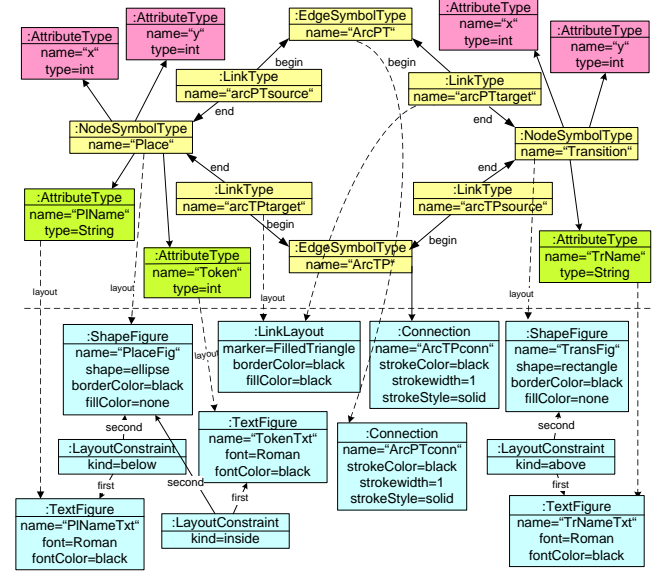

Figure 8: Alphabet for Place/Transition Nets

The position of a *SymbolType* figure is the only layout information which is given as attribute type because this information is necessary to be held persistent in the generated editor. All the other layout attributes like shape figures or layout constraints serve for the generation of the visual editor features. For example, the class generated for the *CreatePlace* command implements the *Ellipse* figure class of Draw2D and thus incorporates the shape information. Layout constraint INSIDE leads to the generation of a hierarchy of figures in GEF, where e.g. the TokenFigure is a child figure of the parent figure PlaceFigure.

7.1.2 The Syntax Grammar

The start graph of the Petri net syntax grammar is empty. The abstract syntax of four *CREATE* rules for P/T nets is given in Fig. 9. According to Fig. 5, the two rules creating places and transitions in Fig. 9 are *CREATE* rules, case $CREATE_1$. Hence, the SymbolTypes Place and Transition are entered into the *Symbol* group of the Petri net editor palette. The two rules creating arcs are *CREATE* rules, case $CREATE_2$. Hence, the SymbolTypes ArcPT and ArcTP are entered into the palette's *Connection* group. Negative application conditions (NACs) ensure that place and transition names are unique, and that no more than one arc in each direction may be inserted between a place and a transition. Note that this uniqueness of arcs cannot be expressed by multiplicity constraints as e.g. used in EMF models, as a Place may have more than one outgoing ArcPT, and a Transition may have more than one incoming ArcPT in general. Such a condition would have to be defined by extra
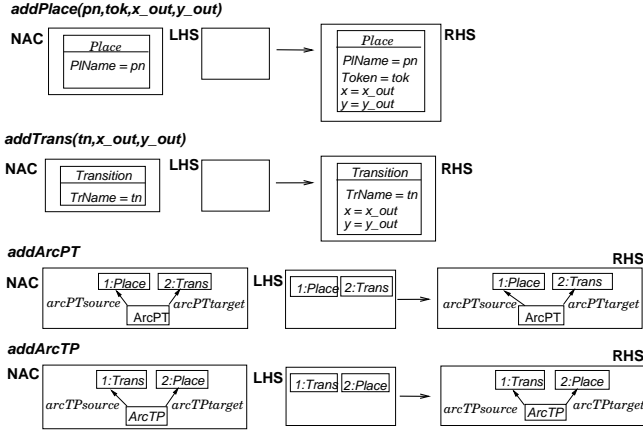
constraints in EMF.

Figure 9: Syntax grammar for P/T Nets

In addition to the syntax rules depicted in Fig. 9, *MOVE* rules and simple *DELETE* rules are generated for each SymbolType (where the *DELETE* rules correspond to inversed *CREATE* rules without NACs). The syntax rules are applied internally to edit e.g. the P/T net shown in Fig. 7. The layout of the resulting net is computed according to the layout information as provided by the P/T net alphabet resp. as incorporated as features of the generated editor.

7.2 The Generated Activity Diagram Editor

Activity diagrams are used to describe the control flow on activities. A concrete activity diagram is drawn in the editor panel of the generated editor for activity diagrams in Fig. 10 modeling the workflow of order processing in a shop.
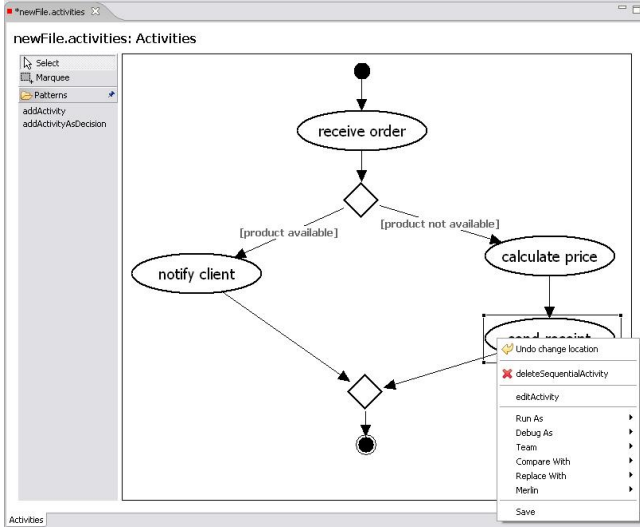
Figure 10: Generated Editor for Activity Diagrams

7.2.1 The VL alphabet

The VL alphabet for activity diagrams contains two kinds of symbols, activities and next-relations which begin and end at activities. The activities can be of different kinds,

i.e. simple activities, start and end nodes as well as decision nodes. Simple activities are usually inscribed by some text. Moreover, next-relations may have inscriptions which are used to formulate conditions. The abstract syntax part of the visual alphabet for activity diagrams is depicted in Fig. 11. The attribute type *kind* is an enumeration type consisting of the kinds $ActKind = \{start, simple, decision, end\}$.
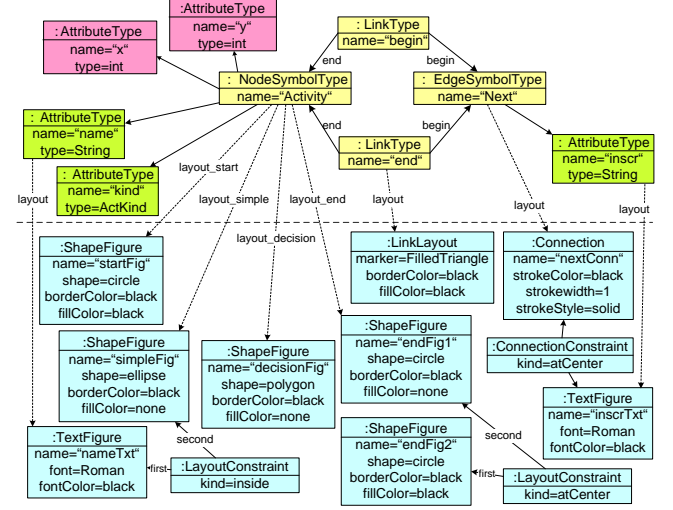
Figure 11: VL Alphabet for Activity Diagrams

The abstract syntax part of the alphabet is extended by defining the concrete layout for the components of an activity diagram. An activity is either represented as ellipse or as polygon, depending on its *ActKind*, as visualized in Fig. 12.
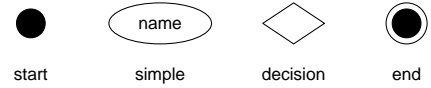
Figure 12: Concrete Layout Figures for an Activity

The generator in this example relates a NodeSymbol (i.e. an activity) to its ShapeFigure according to the value of the *ActKind* attribute. This allows a much more flexible layout definition than in the Petri net editor example, where NodeSymbols of the same NodeSymbolType always are layouted by the same ShapeFigure.

A next-relation is shown by a poly line which is attached to two activity figures. A possible inscription is positioned at the line center. Again, each figure and line has layout attributes describing properties such as font, font size, color, fill color etc.

7.2.2 The Syntax Grammar

The syntax rules for activity diagrams decide important aspects of the visual language, e.g. the number of start and end activities which are allowed in one diagram, or the question whether decision branches have to be merged again. Our variant of activity diagrams allows only one start and one end activity. This is realized in the syntax grammar (see Fig. 13) by defining an activity diagram as start graph which consists of exactly one start and one end activity, connected

by a next-relation. As none of the syntax rules adds or deletes start or end activities, their number will always be fixed to one each.
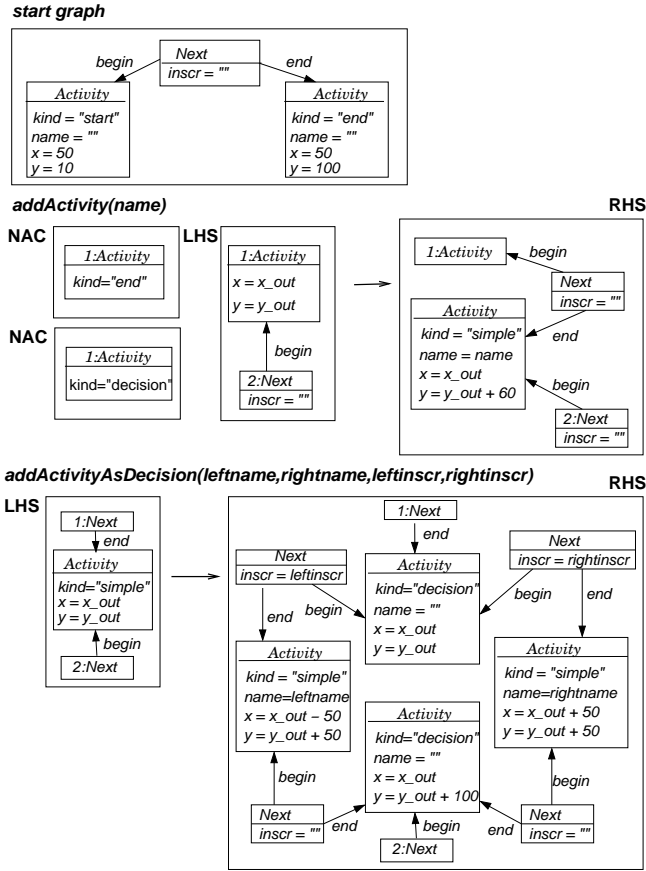
start graph

addActivity(name)

addActivityAsDecision(leftname,rightname,leftinscr,rightinscr)

Figure 13: Syntax Grammar for Activity Diagrams

The syntax grammar contains two main *CREATE* rules, both belonging to the more complex case $CREATE_3$ in Fig. 5: Rule *addActivity* inserts a simple activity after another activity (which must not be a decision or the end activity). The name of the new activity is given by input parameter *name*. Rule *addActivityAsDecision* replaces a simple activity by a decision activity with two branches. Each branch contains one simple activity. The branches are merged afterwards by another decision activity. This rule has four input parameters: two arc inscriptions *leftinscr* and *rightinscr*, and two names *leftname* and *rightname* for the simple activities in both branches. Positions of newly inserted activities are computed from the positions of already existing ones, e.g. in rule *addActivity* the new activity is inserted at a fixed distance (60 points) below the activity identified by number 1. The start graph defines the initial position of the start and end activities. All other layout properties are either constant (such as colors and minimal sizes) or relative (i.e. the connection points of next-relation lines and the size of simple activities which depends on the size of the text inside the ellipse.

As both syntax rules are complex $CREATE_3$ rules, the rule names appear in the editor palette of the generated editor in Fig. 10. Apart from the two rule names, the editor palette contains only the default actions *Select* and *Marqee*,

but no VL-specific symbol or connection group. Thus, in contrast to the Petri net editor, editing of an activity diagram is realized by first selecting a rule name in the palette, and then selecting symbols in the panel which will be part of the rule match.

The activity diagram in Fig. 10 has been edited by applying first rule *addActivity("receive order")* with the left-hand side activity matched to the start activity, then again applying rule *addActivity("simple activity")* to obtain a match for the next rule application, namely of rule *addActivityAsDecision("notify client", "calculate price", "product available", "product not available")* with the left-hand side activity is matched to the activity *"simple activity"* which is now replaced by the branch-and-merge structure. At last, rule *addActivity("send receipt")* is applied, where the left-hand side activity this time is matched to the activity *"calculate price"*.

Further syntax rules (not depicted in Fig. 13) exist for deleting and moving activities in order to obtain a well-layouted diagram. Analogously to the Petri net syntax grammar, the *DELETE* rules correspond to the inverted *CREATE* rules.

8. CONCLUSION AND FUTURE WORK

In this paper, we described the first development steps for a generator for generating visual editors from formal visual language specifications based on graph transformations and ECLIPSE-GEF. The tool environment TIGER[1] combines the advantages of formal VL specifications using graph transformation (offered by AGG) with sophisticated graphical editor features (offered by ECLIPSE-GEF). The generated editors themselves are ECLIPSE plug-ins and hence can be integrated in the ECLIPSE framework. The current state of this ongoing work is focussed on generating graphical editors for graph-like languages, where *ShapeFigures* for *NodeSymbols* are connected via poly lines for *EdgeSymbols*.

The VL specification so far consists of an alphabet (a type graph plus layout attributes) and a syntax grammar.

An alternative for syntax-directed editing based on graph transformation is *free-hand editing*. A free-hand editor would offer more general symbol editing commands (modeled by simple editing rules), but add parsing facilities for the current diagram, internally realized by applying *parsing rules*. The parsing rules for a VL are more or less the inverted rules of the VL syntax grammar. The applications of the parsing rules reduce the abstract syntax graph of the diagram edited so far that the start graph is produced (see e.g. [2]). As advantage of the free-hand editing approach the editing of intermediate invalid diagrams is tolerated by the editor. Similarly to parsing, a diagram could also be checked according to additional well-formedness constraints, as done in e.g. ATOM[3] where a class diagram (the meta-model) is combined with constraints in e.g. *OCL* [20] which can be checked at any time during the editing process.

Since generators for visual editors like MERLIN [16] and OPENARCHITECTUREWARE [26] follow the declarative MOF [19] approach to VL definitions, the generated editors do not support syntax directed editing. TIGER with its underlying graph transformation engine AGG follows a constructive approach allowing syntax-directed editing in which each editor

---

[1]The TIGER environment can be downloaded at `http://tfs.cs.tu-berlin.de/tigerprj`.

operation leads to a valid diagram of the specific VL.

Near future work (the second development step) will extend the VL specification to include additional transformation rules (e.g. parsing rules, simulation rules, model transformation rules) for model manipulation in the generated environment in addition to editing. To allow a user friendly definition of the VL specification a *VL-Designer* component is planned to be implemented soon. Here, the experiences made with GenGED [1], a generator for graphical environments providing a nice graphical user interface for editing VL specifications, will be helpful. For flexible GUI specification the user should be supported to specify a non-default editor environment in the VL specification. Further development steps aim at allowing more general kinds of diagrams instead of graph-like languages only. An example for a more sophisticated, non-graph-like visual language is the language of nested UML state diagrams [21] where NodeSymbols (boxes) are spatially related by being nested into each other. Such general kinds of diagrams are also needed to realize animation of model behavior in the layout of the application domain [12].

9. REFERENCES

[1] Bardohl, R., *GenGED – Visual Definition of Visual Languages based on Algebraic Graph Transformation*, PhD Thesis, TU Berlin, Verlag Dr. Kovac, 1999.

[2] Bardohl,R. and Ermel,C., *Visual Specification and Parsing of a Statechart Variant using GenGED*, Statechart Modeling Contest at IEEE Symposium on Visual Languages and Formal Methods (VLFM'01), Stresa, Italy, 2001. `http://www2.informatik.uni-erlangen.de/VLFM01/Statecharts/`

[3] Eclipse Consortium, *Eclipse – Version 3.0.1*, 2004, available at `http://www.eclipse.org`.

[4] Eclipse Consortium, *Eclipse Graphical Editing Framework (GEF) – Version 3.0.1*, 2004, available at `http://www.eclipse.org/gef`.

[5] Eclipse Consortium, *Eclipse Graphical Modeling Framework (GMF)*, 2005, available at `http://www.eclipse.org/gmf`.

[6] Eclipse Consortium, *Eclipse Modeling Framework (EMF) – Version 2.0.1*, 2003, available at `http://www.eclipse.org/emf`.

[7] Eclipse Consortium, *Java Emitter Templates (JET)*, Eclipse Modeling Framework – Version 2.0.1, 2003, available at `http://www.eclipse.org/emf`.

[8] Ehrig, H. and Ehrig, K. and Prange, U. and Taentzer, G., *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in TCS, Springer to appear, 2005.

[9] Ehrig, K. and Ermel, C. and Hänsgen, S., *Towards Model Transformation in Generated Eclipse Editor Plug-Ins.* Proc. International Workshop on Graph and Model Transformation (GraMoT'05). Tallinn, Estonia, 2005.

[10] Ehrig, K. and Ermel, C. and Hänsgen, S. and Taentzer, G., *Towards Graph Transformation based Generation of Visual Editors using Eclipse.* Visual Languages and Formal Methods (VLFM), 2004.

[11] Ehrig, H. and Prange, U. and Taentzer, G., *Fundamental Theory for Typed Attributed Graph Transformation.* In Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Parisi-Presicce, F. and Bottoni, P. and Engels, G., eds., Springer LNCS 3256, pp. 161–177, 2004.

[12] Ermel, C. and Bardohl, R., *Scenario Animation for Visual Behavior Models: A Generic Approach*, Journal on Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques, Vol. 3(2), Springer, pp. 164–177, 2004.

[13] Harel, D., *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, vol. 8, pp. 231-274, Elsevier Science Publ., Amsterdam, 1987.

[14] Habel, A. and Heckel, R. and Taentzer, G., *Graph Grammars with Negative Application Conditions*, Special Issue of Fundamenta Informaticae, vol. 26, no. 3,4, pp. 287–313, 1996.

[15] de Lara, J., Vangheluwe, H., 2002. *AToM³: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In Proc. FASE'02, Springer LNCS 2306, pp. 174 - 188. See also the AToM³ home page, `http://atom3.cs.mcgill.ca` .

[16] Merlin, *Merlin Generator Project*, Available at `http://sourceforge.net/projects/merlingenerator/`.

[17] Minas, M., *Specifying Graph-like Diagrams with DiaGen*, in Electronic Notes in Theoretical Computer Science, vol. 72, issue 2, published by Elsevier, 2002.

[18] Minas, M. and Viehstaedt, G., *DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams*, Proc. IEEE Symp. on Visual Languages, September, 5-9, Darmstadt, Germany, pp. 203–210, 1995.

[19] Object Management Group , *Meta-Object Facility (MOF) – Version 1.4*, 2002, Available at `http://www.omg.org/mof`.

[20] Object management group (OMG), *Object constraint language – Version 2.0*, 2002, available at `http://www.klasse.nl/ocl`.

[21] Object management group (OMG), *Unified Modeling Language (UML) – Version 2.0*, 2005, available at `http://www.uml.org`.

[22] Reisig, W., *Petri Nets*, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer-Verlag, 1985.

[23] Sun Microsystems, *Java – Version 1.5*, 2004, available at `http://java.sun.com`.

[24] Taentzer, G., *AGG: A Graph Transformation Environment for Modeling and Validation of Software*, Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03), Pfaltz, J. and Nagl, M., Charlottesville/Virgina, USA, 2003, `http://tfs.cs.tu-berlin.de/agg`.

[25] TIGER Project, 2005, available at `http://tfs.cs.tu-berlin.de/~tigerprj`.

[26] Völter, M., *OpenArchitectureWare Generator*, 2005, available at `www.openarchitectureware.org`.

[27] Winter, A., *An Overview on the GXL Graph Exchange Language*, S. Diehl (ed.) Software Visualization, International Seminar at Dagstuhl Castle, Germany, Springer LNCS 2269, pp. 324–336, 2002.