Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006)

Object Oriented and Rule-based Design of Visual Languages using Tiger

Claudia Ermel, Karsten Ehrig, Gabriele Taentzer, and Eduard Weiss

12 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Object Oriented and Rule-based Design of Visual Languages using Tiger

Claudia Ermel[1], **Karsten Ehrig**[2], **Gabriele Taentzer**[3], **and Eduard Weiss**[1]

[1] Institut für Softwaretechnik und Theoretische Informatik,
Technische Universität Berlin, Germany
lieske@cs.tu-berlin.de, weiss@cs.tu-berlin.de

[2] Department of Computer Science,
University of Leicester, United Kingdom
karsten@mcs.le.ac.uk

[3] Fachbereich Mathematik und Informatik,
Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de

Abstract: In this paper we present the state-of-the-art of the TIGER environment for the generation of visual editor plug-ins in Eclipse, with the focus on its *Designer* component, a visual environment for object oriented and rule-based design of visual languages. Based on an alphabet for finite automata we show how a visual language can be designed by defining the abstract and concrete syntax of the visual language and syntax directed editing operations in the generated editor plug-in. For the graphical layout we use the Graphical Editing Framework (GEF) of ECLIPSE which offers an efficient and standardized way for graphical layouting.

Keywords: visual languages, editor generation, visual editor, graph transformation, Eclipse

1 Introduction

Domain specific modeling languages are of growing importance for software and system development. Meta tools are needed to support the rapid development of domain-specific tool environments. The basic component of such environments is a domain-specific visual editor. A visual language (VL) definition based on a meta model in combination with syntax rules defining syntax-directed editor commands is used in TIGER *(Transformation-based Generation of Environments)* to generate a corresponding visual editor. On the one hand, a visual language definition captures the visual symbols, links and relations of the domain specific modeling language (the alphabet); on the other hand, a syntax graph grammar defines precisely which editor operations are allowed and restrict the visual sentences of the VL to correct diagrams.

TIGER combines the advantages of precise VL specification techniques using graph transformation concepts with sophisticated graphical editor development features offered by the Eclipse Graphical Editing Framework (GEF) [GEF06]. Using graph transformation at the abstract syntax level, an editor command is modeled in a rule-based way by just specifying the pre- and

post-conditions of each command. The application of such syntax rules to the underlying syntax graph of a diagram is performed by the graph transformation engine AGG [Tae04]. TIGER extends AGG by a concrete visual syntax definition for flexible means for visual model representation. From the definition of the VL, the TIGER *Generator* generates Java 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. Thus, graphical layout constraints are defined and solved with efficient Java methods without using complex constraint solving algorithms like in GENGED [BEW03] or DIAGEN [Min02], and the generated editors appear in a timely fashion, conforming to the ECLIPSE standard for graphical tool environments.

Note that graph transformation-based editors, in contrast to related meta-model-based editors like GMF [GMF05], AToM^3 [dLVA04] or Pounamou [NLG05], do not only offer basic editor commands, the so-called *CRUD* operations (Create, Read, Update, Delete), but they can also offer complex editing commands which insert or manipulate larger model parts consisting of a number of elements. With complex editing commands, model optimizations, such as model refactoring, as well as model simulation can be performed.

TIGER [EEHT05, Tig05] is the successor project of GENGED [BEW03, Bar02], with the objective to make extensive use of today's modern functionalities for visual model-driven development and integration offered by the Eclipse platform and its plug-in mechanism. Hence, both the TIGER *Designer* component for visual VL definition and the TIGER-generated visual editors are ECLIPSE plug-ins, based on the common paradigm for visual creation, management and navigation of resources. The features of domain-specific editors generated by GENGED, DIAGEN and ATOM^3 (e.g. for layouting diagrams, undo/redo, zooming, etc.) partly differ heavily from modern standards. Moreover, 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.

In this paper we focus on the TIGER *Designer* for visual editing of visual language specifications as part of the TIGER environment [Tig05]. The second main component of TIGER is the TIGER *Generator* for generating rule-based editor plug-ins in ECLIPSE. The generator has already been presented in [EEHT05].

The paper is organized as follows: Section 2 reviews the basic concepts for visual language specification based on graph transformation, and introduces the TIGER perspective for VL design in ECLIPSE. Section 3 goes into the details and describes how a VL is specified using the TIGER *Designer*, by defining on the one hand the abstract and concrete syntax of the VL alphabet, using the visual abstract syntax editor and layout view, and on the other hand, by defining the editing operations using the visual rule editor. Section 4 shows how a visual editor is generated as ECLIPSE plug-in from the VL specification. In Section 5, we discuss ongoing and future work concerning the TIGER environment.

2 Visual Language Design based on Graph Transformation

Nowadays two main approaches to VL definition can be distinguished: grammar-based approaches or meta-modeling. Using graph grammars, multi-dimensional representations are described by graphs and allows not only a visual notation of the concrete syntax, but also a visual-

ization of the abstract syntax. While the concrete syntax contains the concrete layout of a visual notation, the abstract syntax shows the underlying structure, i.e. it provides a condense representation to be used for further processing. Similarly to textual language definition, grammar rules define the language, but for visual languages, graph rules are used to manipulate the graph representation of a language element.

2.1 VL Design Concepts based on Graph Transformation

For the application of graph transformation techniques to VL design, *typed attributed graph transformation systems* [EEPT06] have proven to be an adequate formalism. Roughly spoken a typed attributed graph transformation rule $p = (L \rightarrow R)$ consists of a pair of typed attributed graphs L and R (its left-hand and right-hand sides) and a mapping from L to R. Symbols and links appearing in L are matched with the elements of the current editor diagram and deleted or preserved according to their mapping to R. New symbols and links are created if they appear in R only. A *direct graph transformation* written $G \stackrel{p,o}{\Longrightarrow} H$, means that diagram G is transformed into diagram H by applying rule p at the occurrence o of L in G.

A VL is modeled basically by an *alphabet*, an attributed type graph which captures the definition of the underlying symbols and relations which are available. Sentences or diagrams of the VL are given by attributed graphs typed over the type graph. The abstract alphabet is extended by defining the concrete layout of diagrams. At the concrete syntax level, the VL alphabet defines the figures and their properties which are used to visualize the underlying abstract symbols.

Usually, the set of visual diagrams (sentences) over an alphabet should be further restricted to the meaningful ones. By defining this restriction via graph rules, the constructive way is followed (as opposed to the declarative MOF approach [MOF05], where OCL constraints are used). The application of abstract syntax graph rules builds up abstract syntax graphs of valid diagrams. Together with a suitable start graph, the set of syntax rules forms the syntax graph grammar which defines the models belonging to a VL in a well-defined and constructive way.

2.2 The TIGER Perspective for VL Design in ECLIPSE

The main difference between the TIGER *Designer* and related stand-alone environments for graph transformation-based VL design such as GENGED, DIAGEN and ATOM^3, is the use of the ECLIPSE platform. TIGER makes extensive use of the standard elements provided by the ECLIPSE workbench paradigm, such as perspectives, editors and views. The TIGER perspective comprises a designated group of views and editors in the ECLIPSE workbench window (the modeling desktop). A *view* is a visual component, typically used to navigate a hierarchy of information, open an editor, or display properties for elements in the active editor. An *editor* is also a visual component, typically used to edit or browse a resource. Views and editors can be active or inactive. The active component is the target for common operations such as cut, copy or paste. The TIGER perspective can be configured and customized in a flexible way (as usual for Eclipse perspectives). The user determines for instance which components are shown and how they are ordered on the desktop. Figure 1 shows an example of views and editors arranged in the Tiger perspective: the tree view (1 in Figure 1) shows the hierarchical structure of a VL alphabet. A visual editor (2 in Figure 1) is used to define the layout for a symbol type, and a

properties view (3 in Figure 1) allows to change values for graphical layout properties of the ellipse figure selected in the visual editor.

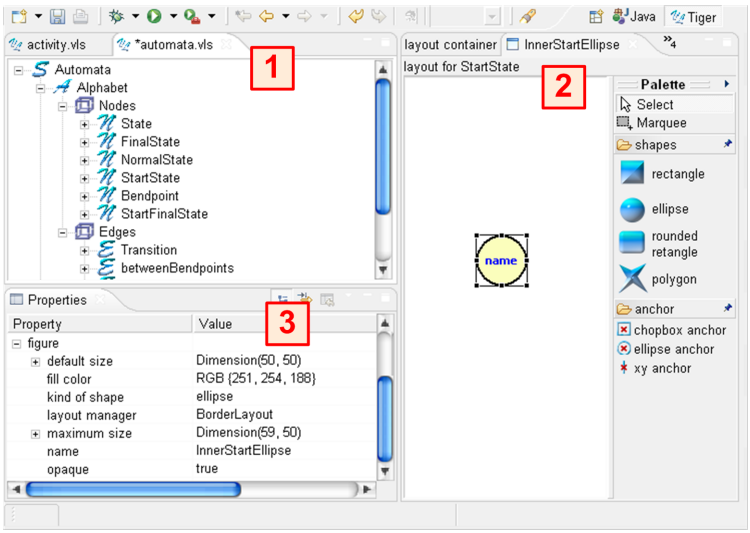

Figure 1: The TIGER Perspective in ECLIPSE

In the following, we describe how the diverse views and editors of the TIGER *Designer* are used to define a VL specification consisting of a VL alphabet and a VL syntax grammar.

3 Designing Visual Languages using Tiger

As discussed in the previous section, a VL specification (*VLSpec*) consists of an *Alphabet* containing the available symbols and links of the VL and their layout, a *RuleSet* containing syntax rules which define possible editing operations to construct diagrams, and a *StartGraph*, defining the initial diagram the syntax rules are applied to. In alphabets, rules and diagrams we distinguish the abstract syntax (the internal representation of diagrams as graphs without layout information) from the concrete syntax (describing additionally the layout properties and constraints).

3.1 The VL Alphabet

A VL alphabet consists of *SymbolTypes* and *LinkTypes*. In our approach, graph-like languages consist of *node symbol types* (e.g. states in automata) and *edge symbol types* (e.g. transitions in automata). Edge symbol types are connected to node symbol types by *LinkTypes*. Symbol types may be attributed by an ordered list of *AttributeTypes* e.g. to model the state names in automata. Classes *AttributeType*, *SymbolType* and *LinkType* have directly corresponding node and edge types in AGG forming the abstract syntax representation. Figure 2 shows package *abstractsyntax*, where the abstract syntax of alphabets is defined. This abstract alphabet syntax definition corresponds roughly to the *M*3 level of the MOF meta-model hierarchy, where also the syntax of meta-models (specifically UML meta-models) is defined.

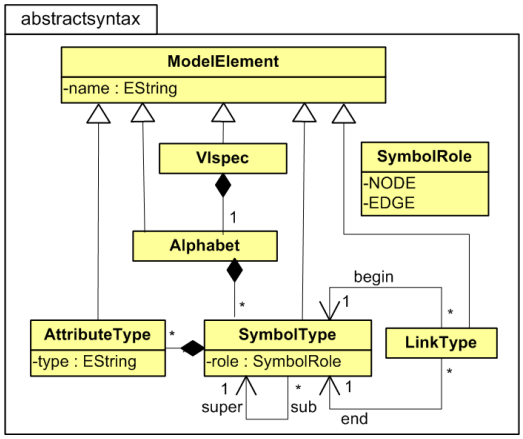

Figure 2: VL Specification: Abstract Syntax of an Alphabet

The graphical layout (the concrete syntax) is given by additional classes extending the class diagram shown in Figure 2. The TIGER *Designer* stores the concrete layout information for symbols and links in the designer model based on the meta model shown in Figure 3 which is closely connected to the layout relations of Eclipse GEF [GEF06].

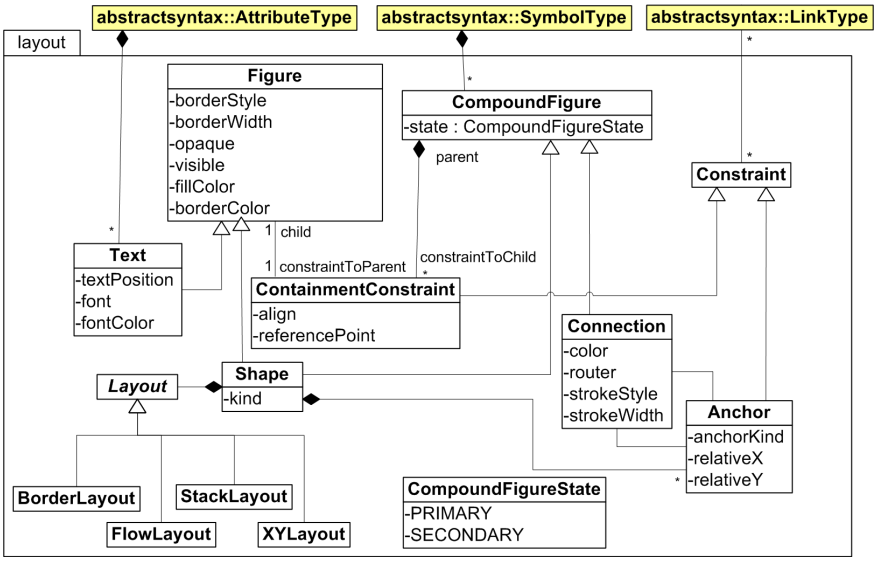

Figure 3: VL Specification: Concrete Syntax

Symbol types (for both nodes and edges) are layouted as *CompoundFigures* which may contain *Figures* again. The figure layout is defined by *Shapes* which *kind* could be a *Rectangle*, *Ellipse*, *Polyline*, ... For the automata VL, we have e.g. two *Shapes* of kind *Ellipse* nested into each other in for layouting a final state, and we have a *Text* figure (the arc inscription) as child figure of a *Connection* figure for layouting a transition. The figure arrangement is done by GEF

standard layout styles such as *StackLayout*, *BorderLayout*, *XYLayout*, and *FlowLayout*. *Connection Anchors* describe the relation between *Shapes* and *Connections*. GEF layout *Constraints* are handling the layout positions inside of *Figures* via *ContainmentConstraints*, for example a *Text* Figure is located inside an *Ellipse* figure.

Figure 4 shows the TIGER Perspective for designing a visual language for finite automata.

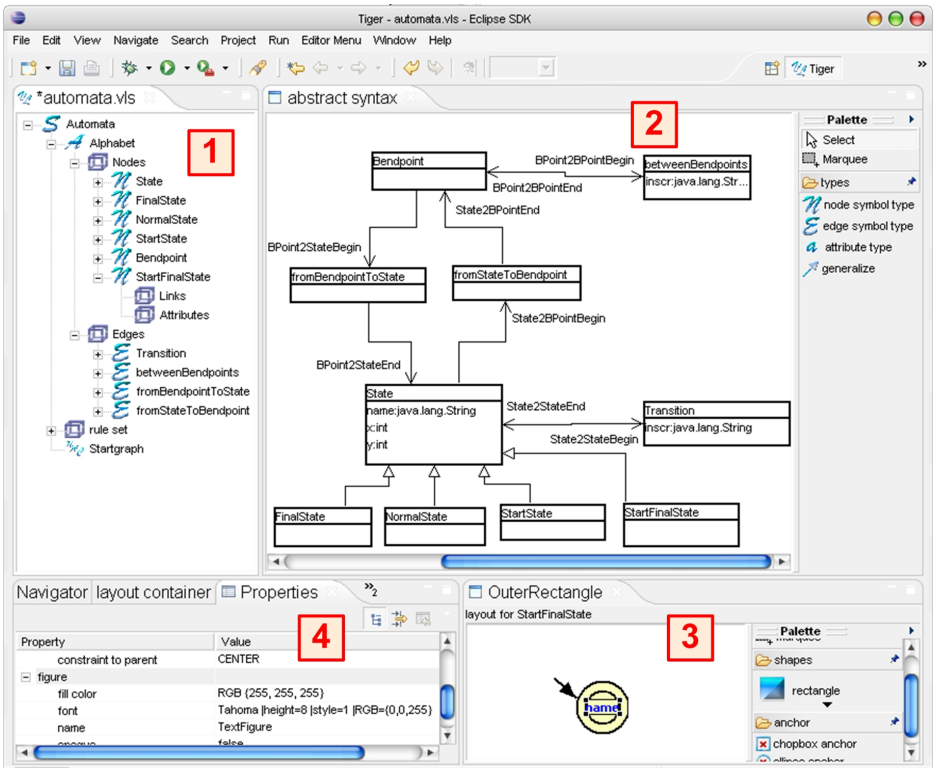

Figure 4: VL Design of the Automata VL with the TIGER *Designer*

While the tree view on the left-hand side (1 in Figure 4) shows the symbol types of the automata alphabet, the syntax rules and the start graph of the automata syntax grammar, the abstract syntax can be defined in the abstract syntax panel to the right (2 in Figure 4), where the abstract syntax of the automata alphabet is shown. The concrete layout of a symbol type is defined via a graphical editor shown at the bottom of the right-hand side (3 in Figure 4). The layout for the *StartFinalState* symbol (a start state which is a final state as well) is given by an invisible rectangle, containing the start marker (a polygon), and an outer and inner ellipse selected from the *shapes* menu of the editor *Palette*. Moreover, an attribute *name* is represented by a text figure, connected via an *anchor* to the inner ellipse figure. The *Properties View* on the left side of the bottom (4 in Figure 4) shows the layout properties like text width and style of the figure selected in the editor, here the text figure of the *StartFinalState* symbol. In the same way *NormalState*, *StartState*, *FinalState* and *State* are defined where the last one represents the other states via an inheritance relation defined in the abstract syntax.

Edges are created in a quite similar way. For a *Transition* two *Links* are defined in the tree

view for connecting the *Transition* with a *State*. The concrete layout is given by *Transition Connection* defining a line *connection* with a closed arrow *decoration* from the editor *Palette*. For better orientation, the begin and end points of a connection are visualized by small block arrows (see Figure 5). An inscription attribute *inscr* is located close to the end point of the transition line via a layout constraint. The *Properties View* shows the layout properties of the *TransitionConnection* with black color, solid line style, and normal width.

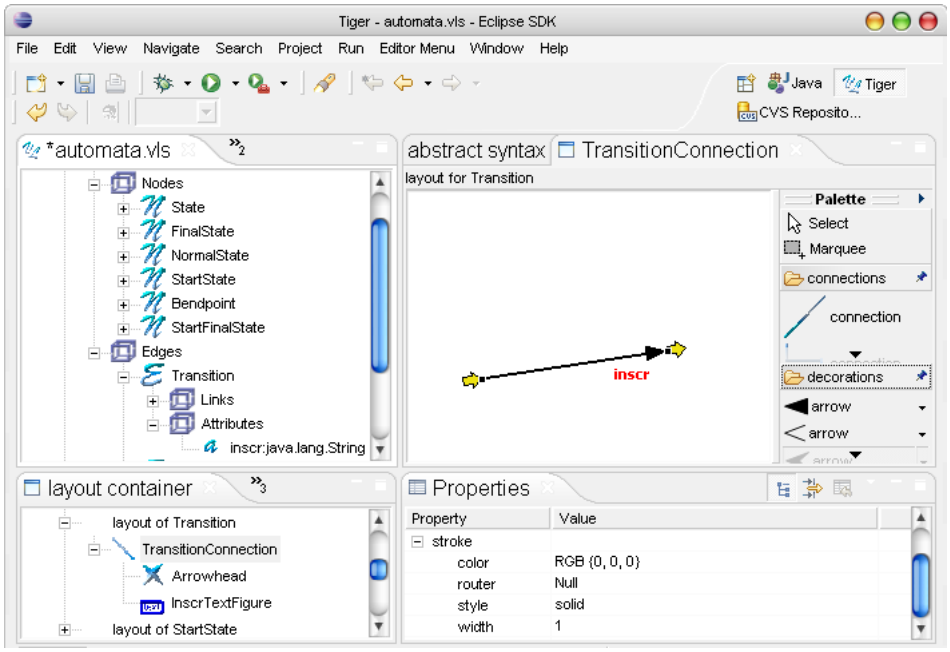

Figure 5: Creating a *Transition* connection in the TIGER *Designer*

3.2 The VL Syntax Grammar

Language constraints restricting the set of valid diagrams over an alphabet are modeled by restricting the set of editing commands, i.e. graph transformation-based editors are usually syntax-directed. An editor command is modeled as a graph rule (typed over the language's alphabet) being applied to the abstract syntax graph of the current diagram. The graph transformation approach to language definition is a constructive one, since syntax rules are used to build up all language instances from an initial state (the start graph). The start graph together with the set of syntax rules and the underlying VL alphabet, are called *VL syntax grammar* because it defines the complete syntax of the visual language.

Figure 6 shows package *rules*, where the start graph and the syntax rules are defined. The left- and right-hand sides of a rule are graphs. Additionally, a rule may contain a set of negative application conditions (NACs), which model situations in which the rule is not applicable. Moreover, a rule may have a set of input parameters defined by the user when the rule is applied, and variables for performing attribute computations. For each rule, the rule morphisms from the

left-hand side *L* to the right-hand side *R* and from *L* to the NACs are given by sets of mappings for symbols and links.

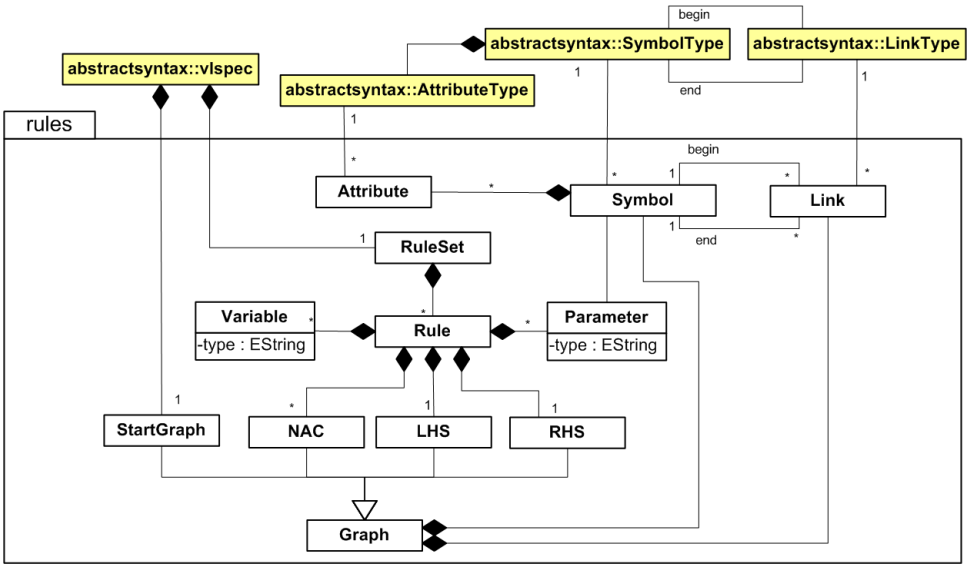

Figure 6: VL Specification: Syntax Grammar

In the TIGER rule editor in Figure 7, the editor operations for syntax directed editing of automata are defined. The rule *addTransition* inserts a *Transition* between two arbitrary *States* represented by solid rectangles. In fact, the abstract node *State* preserves the user from defining different rules for each possible pair of concrete state figures, for example to connect a *StartState* with a *NormalState*. The left-hand side of the rule defines the *States* to be selected by the user as input parameters *in=0* and *in=1* of the rule. After rule application, a *Transition* with the inscription *transinscr* is inserted between the previous *States*, where the mapping between the left- and right-hand side is indicated with *m=0* and *m=1*. The NAC *uniqueTrans* ensures that no *Transition* in the same direction exists before the rule application. Instead, we allow inscriptions consisting of more than one character for one transition. The transition name *transinscr* is listed as input parameter in the view *parameters in rule* of type *java.lang.String*. For such attribute parameters, a dialog window pops up when performing the corresponding editor operation in the generated environment, asking the user to specify the transition inscription.

In the *Properties View* the *kind* attribute specifies the rule behavior:

- The name of a *create* operation will appear as entry in the editor palette of the generated editor for inserting a new symbol or a larger structure consisting of several symbols (a sub-diagram) in the editor panel.
- A *delete* operation appears as an entry in the context menu of a symbol for deletion of the symbol or an associated sub-diagram.
- A *move* operation is associated with a symbol to change the layout position in the editor view by mouse dragging.

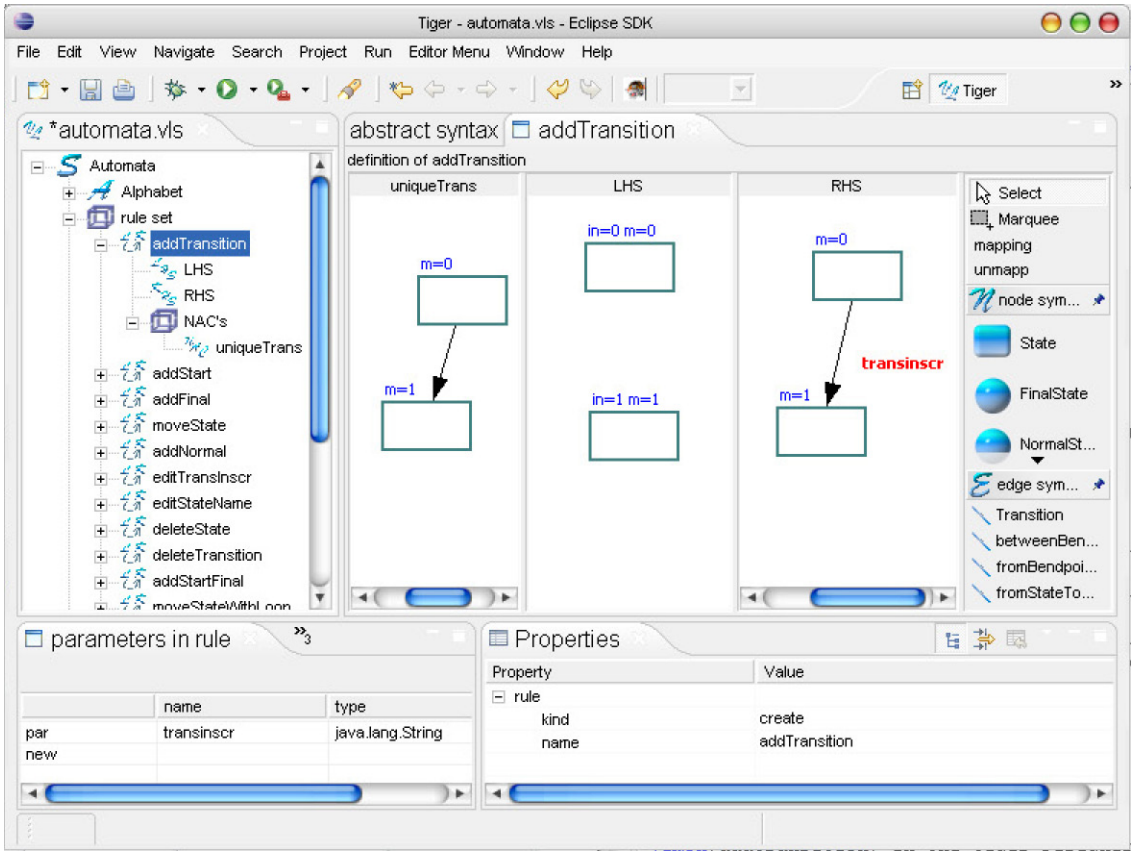

Figure 7: Editing of the syntax rule *addTransition* in the Tiger Designer

- An *edit* operation appears as another context menu entry which allows to change the properties of the associated symbol.

4 Generation of Eclipse Editor Plug-ins

After the specification of a visual language has been completed, the TIGER *Generator* can be invoked for generating the Java code of the envisaged editor plug-in. The *Tiger Generator* uses Java Emitter Templates (JET) as part of the Eclipse Modeling Framework (EMF) [EMF06] for code generation. In code templates, place holders are filled with values given from the visual language specification. The generated Java code may be executed directly in the Eclipse *Runtime-Workbench*. Figure 8 shows the generated editor plug-in for automata. In this editor, an automaton is shown generating the language $L = \{w \in \{0,1\}^* \mid w \text{ is ending with 010 or 101}\}$.

The editor palette shows icons for the GEF standard features *select* (select a single symbol) and *marquee* (select a set of symbols). VL-specific creation operations are grouped into categories *Symbols* (for creating symbols), *Connections* (for creating connections between two symbols) and *Patterns* (for creating patterns consisting of more than one symbol). After a creation oper-

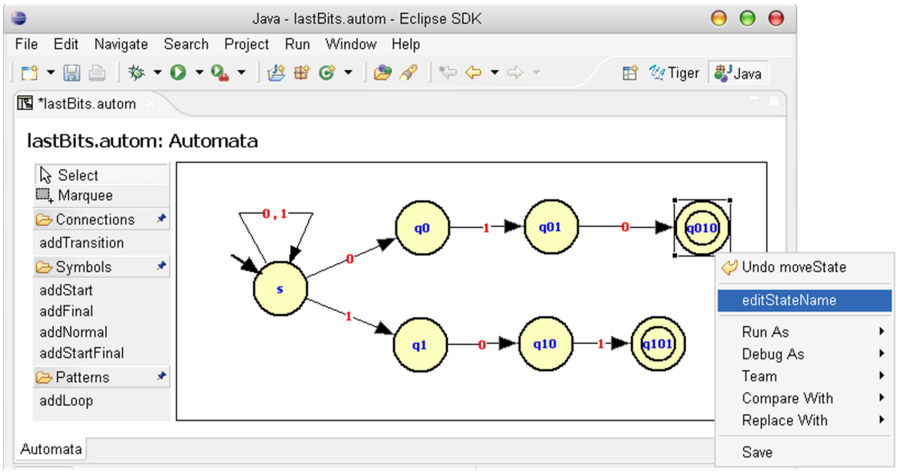

Figure 8: Generated Automata Editor Plug-in in ECLIPSE

ation (e.g. addTransition) has been selected in the palette, the required match symbols must be selected in the editor panel (the source and target state for the transition have to be clicked on). If an input parameter is defined for the syntax rule, a dialog pops up and asks for an attribute value (e.g. the transition inscription has to be given). Now, the underlying creation rule is applied, i.e. the transition is inserted between the two states. Note that *addLoop* is a creation pattern, because internally, a loop consists of three connections and two bendpoints (see the loop at the start state in Figure 8). Thus, the bendpoints can be moved by the user to readjust the loop. Move rules are applied simply when a symbol (or a set of symbols marked by Marquee) is dragged by the mouse. A move rule may also be defined for a symbol pattern. For example, in the automata VL, we defined a move rule moving a state node together with a loop. Deletion rules and editing rules appear in the context menu which is evoked by the right mouse button after a symbol has been selected. Figure 8 shows the context menu for final state $q010$, where it is possible to evoke the operation *editStateName*.

5 Conclusion

In this paper we have described the state-of-the-art of the TIGER environment (http://tfs.cs.tu-berlin.de/tigerprj) with focus on the *Designer* for specifying visual languages in ECLIPSE.

In the development of TIGER, our aim has been to bring together graph transformation-based editor generation with the Eclipse technology based on GEF which has resulted in the generation of syntax-directed GEF-editors with graphs as underlying structures. Practical experience with TIGER so far includes the VL design and visual editor plug-in generation for activity diagrams, Petri nets, automata and sequence diagrams. The TIGER *Designer* proved to be a flexible and intuitive tool for VL design. Following the pure graph transformation-based approach to visual language definition, all editor commands are defined via graph rules. Since the definition of simple editor commands might be tedious work, rules might be partly generated from the type graph as done in GenGED [BEW03]. In this way, the editor definition could be simplified, but

the result would still be a syntax-directed editor. Since both the meta-model-based approach (generating visual free-hand editors) and the graph-transformation-based approach (generating syntax-directed editors), have their advantages and disadvantages (see [Tae06]), we propose as future work to combine both approaches. This means that starting with a meta model only, a simple editor would be generated offering the basic editor operations for each symbol. A syntax check can be added by defining well-formed-ness rules or graph constraints (comparable to an OCL checker in addition to a meta model). For the generation of complex editor commands, an additional specification is needed using syntax rules.

In order to further customize the generated editors, work is in progress to replace the underlying AGG graph transformation engine by a transformation engine based on Eclipse EMF. In this way, genereated editors can be based on already existing domain models. Currently, if the generated editors have to be further adapted to specific needs, the Java code may be extended by hand. So far, changing the generated code is not specifically supported by TIGER, so the user must take into account that a regeneration by TIGER might overwrite hand-written code changes.

As a further improvement at the concrete syntax level we plan to extend TIGER to allow the nesting of figures belonging to different symbol types. With this extension, a TIGER user would be able to specify not only graph-like visual languages, but also more complex ones, like e.g. hierarchical Statecharts.

Furthermore, the TIGER environment has recently been extended by a *model transformation* graph grammar which defines the model transformation between models of either two different VL specifications (exogenous), or between models belonging to the same VL specification (endogenous). An exogenous model transformation between two generated editor plug-ins in TIGER is described in [EEEP06], where activity diagrams are transformed into Petri nets. Considering our automata example, we might define an endogenous model transformation based on the automata VL specification to transform non-deterministic automata into deterministic ones in our generated automata editor plug-in. An example for a related model transformation environment in ECLIPSE based on graph transformations is VIATRA2,which is part of the ECLIPSE Generative Modeling Tools [GMT06]. Work is in progress to support the definition of model transformations directly in the TIGER *Designer*, using the concrete layout of the visual modeling languages.

These extensions can be considered as a starting point for the generation of comprehensive domain-specific visual modeling environments.

Bibliography

[Bar02] R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming (SCP)*, 44(2):181–203, 2002.

[BEW03] R. Bardohl, C. Ermel, and I. Weinhold. GenGED - A Visual Definition Tool for Visual Modeling Environments. In J. Pfaltz and M. Nagl, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, 2003.

[dLVA04] J. de Lara, H. Vangheluwe, and M. Alfonseca. Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Software and System Modeling: Special*

Section on Graph Transformations and Visual Modeling Techniques, 3(3):194–209, 2004.

[EEEP06] H. Ehrig, K. Ehrig, C. Ermel, and J. Padberg. Construction and Correctness Analysis of a Model Transformation from Activity Diagrams to Petri Nets. In I. Troch and F. Breitenecker, editors, *Proc. Intern. IMCAS Symposium on Mathematical Modelling (MathMod)*. ARGESIM-Reports, 2006.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Generation of Visual Editors as Eclipse Plug-ins. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, Long Beach, California, USA, 2005.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theor. Comp. Science. Springer, 2006.

[EMF06] Eclipse Consortium. *Eclipse Modeling Framework (EMF) – Version 2.2.0*, 2006. http://www.eclipse.org/emf.

[GEF06] Eclipse Consortium. *Eclipse Graphical Editing Framework (GEF) – Version 3.2*, 2006. http://www.eclipse.org/gef.

[GMF05] Eclipse Consortium. *Eclipse Graphical Modeling Framework (GMF) – Version 2.0*, 2005. http://www.eclipse.org/gmf.

[GMT06] *Eclipse Generative Modeling Tools (GMT) http://www.eclipse.org/gmt*, 2006.

[Min02] M. Minas. Specifying Graph-like Diagrams with Diagen. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[MOF05] Object Management Group. *Meta-Object Facility (MOF), Version 1.4*, 2005. http://www.omg.org/technology/documents/formal/mof.htm.

[NLG05] J. Hosking N. Liu and J. Grundy. A Visual Language and Environment for Specifying Design Tool Event Handling. In M. Erwig and A. Schürr, editors, *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, IEEE Computer Society, 2005.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In J. Pfaltz, M. Nagl, and B. Boehlen, editors, *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446 – 456. Springer, 2004.

[Tae06] G. Taentzer. Towards Generating Domain-Specific Model Editors with Complex Editing Commands. In *Proc. Workshop Eclipse Technology eXchange(eTX)*, 2006.

[Tig05] Tiger Project Team, Technical University of Berlin. *Tiger: Generating Visual Environments in Eclipse*, 2005. http://www.tfs.cs.tu-berlin.de/tigerprj.