VLFM 2004 Preliminary Version

Towards Graph Transformation Based Generation of Visual Editors using Eclipse

Karsten Ehrig[1], Claudia Ermel[2], Stefan Hänsgen[3], Gabriele Taentzer[4]

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
Germany

Abstract

This work discusses the state-of-the-art of visual editor generation based on graph transformation concepts on one hand, and using the Eclipse technology which includes the Graphical Editor Framework (GEF), on the other hand. Due to existing shortcomings in both approaches, we present a combined approach for a tool environment that allows to generate a GEF-based editor from a formal, graph-transformation based visual language specification.

Key words: editor generation, visual editor, Eclipse, graph transformation, visual languages

1 Introduction

Visual language techniques play an important role in software system development. Often application-specific visual notations are used for which a tool environment consisting of visual editors, simulators, etc. is needed. A lot of work has been done to develop concepts and tool support for generating the desired tool environments. They rely on meta-modeling concepts, grammar-based approaches, or some kind of logics. In the following, we concentrate on generators based on graph transformation like DiaGen [16], AToM^3 [12] 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 support for graphical editor development based on visual language models in form of a number of

[1] Email: karstene@cs.tu-berlin.de
[2] Email: lieske@cs.tu-berlin.de
[3] Email: haensgen@cs.tu-berlin.de
[4] Email: gabi@cs.tu-berlin.de

plug-ins (e.g. EMF [5], Draw2D and GEF [4]). The aim of this paper is to bring together graph transformation-based tool generation with the Eclipse technology.

Following a graph-transformation based approach to visual language (VL) definition, a graph grammar is specified which describes the visual alphabet by a type graph and the language syntax by graph rules. Additional attributes store the concrete layout of all language elements. Thus, graph grammars can precisely define the syntax of a VL. From this VL definition visual editors are generated in e.g. DiaGen and GenGED. The generated editors cover the basic functionalities of visual editors, but often more sophisticated features are not captured and difficult to be added by the customer.

Visual editor development can be based on the Eclipse technology. It contains the Eclipse Modeling Framework (EMF) for generating mainly the underlying models of visual editors. From an EMF class diagram, EMF generates a set of Java classes for manipulating the model and a basic, tree based editor for model instances. The generated classes provide basic support for *creating/deleting* model elements and persistency operations like *loading and saving*. For a complete VL description the generated model has to be extended by additional syntax checks implementing certain constraints e.g. by the *Object Constraint Language (OCL)* [17]. Moreover, the visual editor has to be hand-coded on the basis of GEF, no high-level description of visual representations is offered to support a complete editor generation.

In this paper, we present the first development steps of a new tool environment, called Tiger (Transformation-based generation of modeling environments). It combines the advantages of formal VL specification techniques (as offered by the graph transformation engine AGG [20]) with sophisticated graphical editor development features (as offered by the Eclipse Graphical Editor Framework GEF). Using the AGG engine makes direct use of graph transformation concepts following the double-pushout approach to typed, attributed graph transformation [8]. Graph transformation is used on the abstract syntax level. Tiger is extending the AGG engine by a concrete visual syntax definition for flexible means for visual representation of models. From the definition of the visual language the Tiger *generator* will generate Java [19] source code which can be easily extended by various kinds of functionalities. The generated Java code will implement a standard visual editor using the Graphical Editor Framework (GEF). The resulting visual editor is offered to the user in form of a plug-in for Eclipse. Fig. 1 shows an overview of the Tiger software architecture.

This paper is organized as follows: In Section 2, we review and compare the basic concepts of visual editor development, on the one hand based on graph transformation, and, on the other hand based on Eclipse/GEF. Section 3 describes the aims and the architecture of our envisaged Tiger framework for visual language specification and GEF-based editor generation. In Section 4, we present the results of the first development step of the Tiger framework.

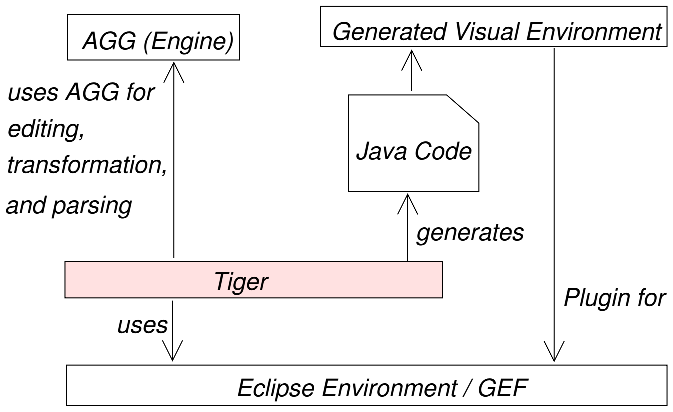
AGG (Engine)
Generated Visual Environment
uses AGG for editing, transformation, and parsing
Java Code
generates
Tiger
Plugin for
uses
Eclipse Environment / GEF

Fig. 1. Architecture Overview

In this step, we realized the generation of editors for graph-like diagrams on the basis of a VL specification and an AGG syntax grammar. The VL of Petri nets is the example to demonstrate in Section 5 the use of the Tiger VL structures and the look-and-feel of the generated editors. The paper concludes with an outlook on ideas for future development steps of Tiger.

2 Graphical Editor Development: State of the Art

In this chapter we review the state of the art of model-based graphical editor development. We compare editor generation concepts using models for visual languages based on graph transformation to editor development concepts using the graphical editor framework GEF.

2.1 Graph-Transformation based Editor Generation

Graph-transformation based editor generators have the benefit of providing a solid, formal VL specification compared to other metamodel-based approaches like EMF. 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. Language constraints restricting the set of valid VL diagrams are modeled by restricting the editing operations which are allowed in the generated editor. Using a meta-model like EMF, many language constraints can only be expressed by adding e.g. OCL constraints or natural-language comments to the metamodel class diagram. Hence, the meta-modeling approach is better suited for graph-like diagrams of low complexity. Using graph transformation, only editing operations are allowed which result in a valid VL diagram. An editing operation in the generated editor is combined to 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 using negative application conditions 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 deletion of a complete hierarchy level in a statechart in one step. When a diagram has been edited, other graph grammars (like simulation grammars) can be applied to perform model simulation or model analysis, based on the same graph transformtion engine.

Visual editor and environment generators like DiaGen [16], AToM^3 [12] and GenGED [1] generate their own VL specific editors from VL specifications based on graph-transformation. They create their own editor features for layouting diagrams, undo/redo, zooming, etc. 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. AToM^3 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 by positioning the diagram elements by hand on the panel.

All of the generated environments are not meant to be integrated into other existing tool environments. As standalone applications they do not always offer the standard look-and-feel of common editor features like e.g. zooming or undo/redo.

2.2 Eclipse Graphical Editor Framework (GEF)

Eclipse [3] is an open platform for tool integration managed by an open community written in Java [19]. Eclipse is open source, i.e. the source code is freely available under a worldwide *Public License*. The *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 basic application functionalities.

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. Sample GEF-based editors are available for state diagrams, activity diagrams, class diagrams, GUI builders, process flow editors, etc. Basic editor functionalities like cut, copy and paste, undo and redo are offered by GEF to be included in a graphical editor. The development of graphical editors based on GEF makes use of these basic editor functionalities.

GEF-based editors require at least a minimal Eclipse environment such as the *Eclipse Runtime-Workbench*. The GEF-based editor is linked by action handlers to the Eclipse environment. GEF assumes a meta-model of the visual language the editor is to be built for. This meta-model (called *model* in GEF) is a distinct package offering all model-based operations like creation and deletion of symbols, connection of symbols by links as well as persistency operations. The values defined by the model are the only data that is persisted and restored for each model instance (diagram). Hence, the model has to include layout information (e.g. symbol positions) for each of its elements.

GEF provides viewers that can be used anywhere in the Eclipse workbench (e.g. graphical or tree-based viewers). GEF editors are based on the *Model-View-Controller* architecture (Fig. 2). The controllers (usually there is one so-called *EditPart* for each symbol type) build the bridge between the views and the model. These EditParts are responsible for the communication between the model and the corresponding views: a *Command*, e.g. a DeleteCommand, leads to a change in the model and to a notification from the model to the corresponding EditPart which initiales an appropriate update of the view.

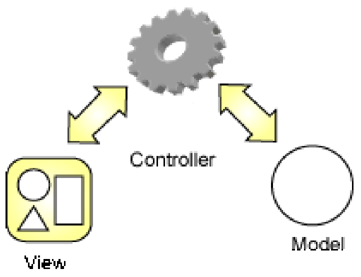

Fig. 2. Model-View-Controller

The disadvantage of using GEF is that the underlying model is not defined completely (it is only given in terms of a meta-model when EMF is used, or as a hand-implemented Java package). Therefore it may be the case that the editor allows the editing of diagrams which are not valid in the VL.

Hence, our approach is to combine GEF features and formal, graph-transformation based VL specification in a new editor generator as described in the following sections.

3 The Tiger Environment: Aims and Architecture

With Tiger, we envisage a fruitful combination of the features for graphical editor development offered by Eclipse and GEF and the power of graph transformation tools for defining the syntax and semantics of visual modeling languages. 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 (e.g. syntax grammars, simulation grammars, consistency checking grammars, model transformation grammars, etc.).

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. The formal basis of Tiger differs from that of GenGED. Instead of transforming attributed graph structures as done in GenGED, we transform typed attributed graphs now. Both kinds of graphs are equally powerful [7], 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 component *designer* which allows the visual definition of the VL specification itself 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 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*.

In Fig. 3, the basic components of the Tiger software architecture are shown. The basic AGG data structures are in the package *AGG-Engine*.

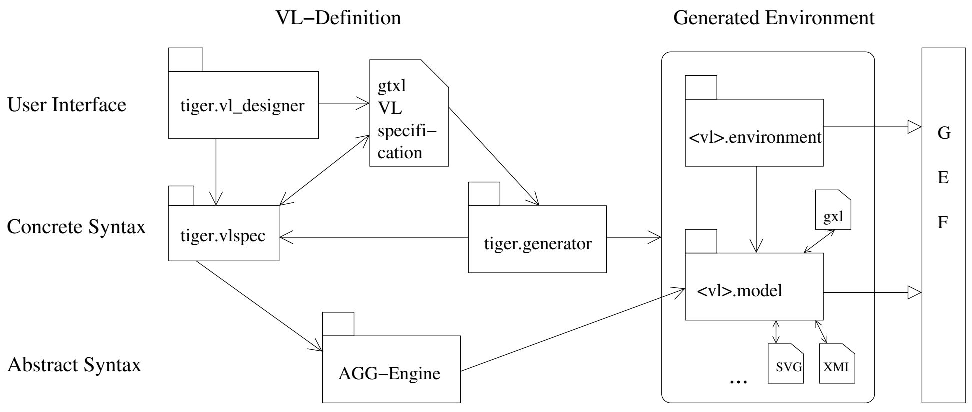

Fig. 3. Software Architecture

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.vl_designer* component building the VL specification using the data structures defined in the *tiger.vlspec* package.

The *tiger.generator* generates the diagram and grammar editors, as well as the simulation, animation, analysis and model transformation components belonging to the *<vl>.environment* where *<vl>* should be replaced with the concrete editor name. Moreover, the *tiger.generator* generates the visual model from the VL specification in the package *<vl>.model*.

The basic VL specification package *tiger.vlspec* allows a VL specification to be saved in the *Graph Transformation Exchange Language (GTXL)* format [21,11]. This is an XML based exchange format for graph transformation systems which is based on the exchange format for graphs GXL [22] and reflects different graph transformation system structures and different graph transformation techniques in order to allow tool cooperation in the graph transformation community.

4 The First Development Step

Since the Tiger project is ongoing work in an early stage, we here present the current state of the development. As the aim of our software design we have almost completed the first development step in order to have a feedback for designing the next developing steps. Therefore, we have imposed the following design decisions on the first development step:

- We generate diagram editors (instead of complete modeling environments).
- We use the VL alphabet plus syntax grammar as VL specification; thus we generate graph-transformation based editors. Later we add additional grammars to the VL specification for e.g. simulating, parsing.
- We allow graph-like languages (e.g. Petri nets, class diagrams) [15] only.
- We use *Eclipse JET* [6] for source code generation as part of the *Eclipse Modeling Framework (EMF)* [5]; thus we use *Eclipse* to generate our new *Eclipse Editor Plug-in*.

Fig. 4 shows the package *tiger.vlspec* which is implemented in the first development step. A VL specification (*VLSpec*) consists of a VL alphabet and a VL syntax grammar (*AGG-Engine.GraGra*). A VL alphabet consists of *SymbolTypes* and *LinkTypes*. In our approach, graph-like languages consist of *NodeSymbolTypes* (e.g. places and transitions for Petri nets) and *EdgeSymbolTypes* (e.g. arcs for Petri nets) with the restriction that *EdgeSymbolTypes* may be connected to *NodeSymbolTypes* only. This restriction is embedded in the constructor of class *LinkType* at the first development step only. The classes *AttributeType*, *SymbolType* and *LinkType* have direct correspondences to the *AGG-Engine* and hence to the abstract syntax representation.

The graphical layout for a *NodeSymbol* of a certain *NodeSymbolType* is given by the class *ShapeFigure*. The shape of *NodeSymbols* can be rectangle, circle, ellipse and closed polygon for graph-like languages. The stroke and fill

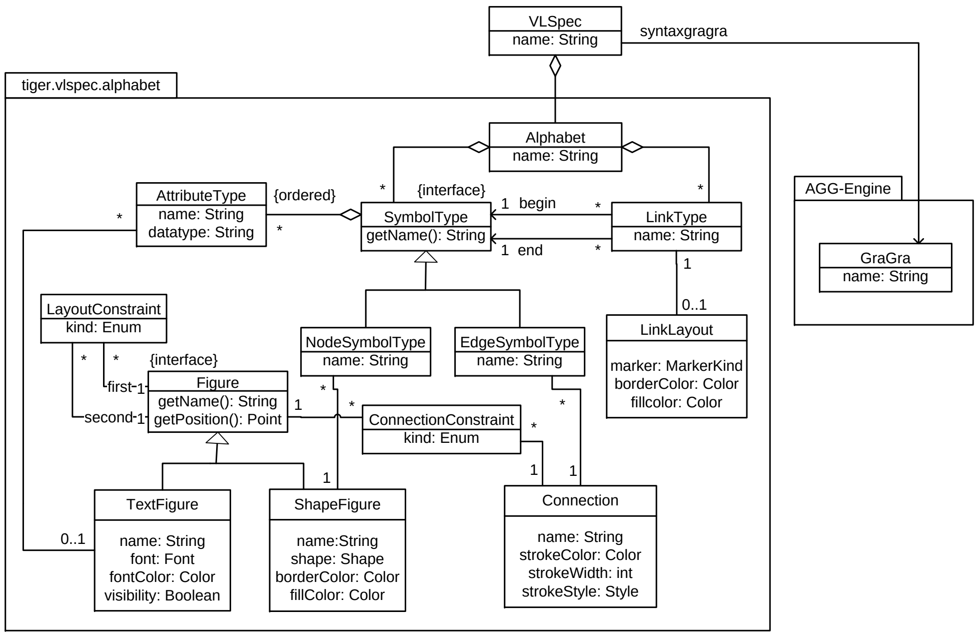

Fig. 4. Package tiger.vlspec

colors are given by the attributes *borderColor* and *fillColor*. *SymbolTypes* may be attributed by an ordered list of *AttributeTypes* e.g. to model the place and transition names in Petri nets. 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* 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 the attributes *strokeColor* (color of the connection), *strokeWidth* (connection width), and *strokeStyle* (e.g. dashed or solid connection). The graphical representation of a link (e.g. a colored arrow) is modeled by the attributes *marker*, *borderColor*, and *fillColor* of the class *LinkLayout*.

Relations between two *Figures* can be modeled as *LayoutConstraints* (e.g. *below(TextFigure, ShapeFigure)*), and relations between a *Figure* and a *Connection* can be modeled as *ConnectionConstraints* (e.g. *nearCenter(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 in the first development step.

For the editor generation the JET compiler reads the visual alphabet specification and the generator template files which define the code generation skeleton with code placeholders. JET replaces the placeholder with the specific code from the visual alphabet specification. This generation process leads to a new generated *Eclipse Editor Plug-in Project* which could be directly executed in the *Eclipse Runtime-Environment*.

In the new *Editor Plug-in* the generator creates a *SymbolFigure* and an *EditPart* for each symbol type in the visual alphabet. The *SymbolFigures* contain the editor code for the graphical layout of the symbols and its attributes. The *EditParts* represent the controller framework between the editor and the underlying model which is directly represented by an *AGG Instance Graph*. Therefore model changes lead to an update of the editor view via the corresponding *EditParts*.

For user interaction the generator creates icons for the defined symbols in the editor palette. Now the user is allowed to draw a symbol in the editor panel and to define the symbol attribute values in a *Properties Dialog*. Invoking an edit operation leads to an execution of the corresponding syntax rule in the *AGG-Engine*. The transformed diagram is directly displayed in the editor panel by the editor controller framework.

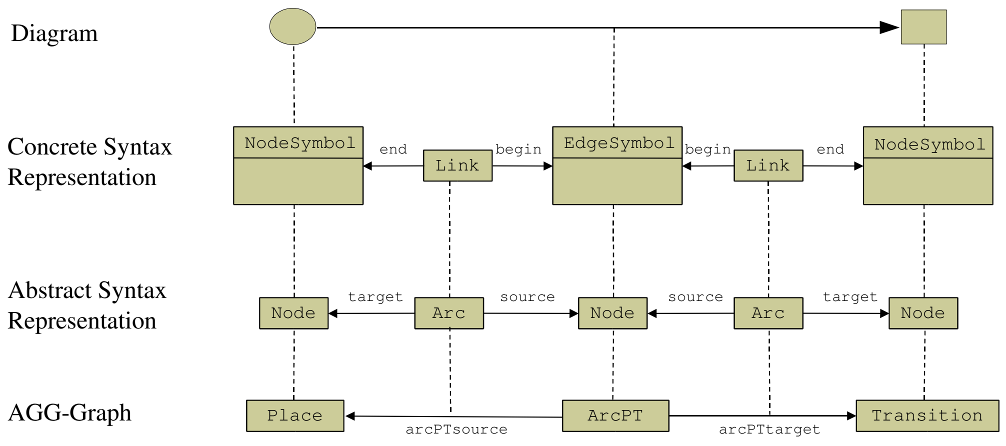

Fig. 5. Representation of a Diagram

Fig. 5 shows the representation of a simple diagram (a Petri net) in Tiger as concrete and abstract syntax. In the diagram, a *place* is connected to a *transition* by an *arc*. *Place* and *transition* are represented by *NodeSymbols* and *arc* is represented by an *EdgeSymbol* in the concrete syntax. *NodeSymbols* and *EdgeSymbols* are connected by *Links*. In the abstract syntax, *NodeSymbols* and *EdgeSymbols* are both represented by *Nodes*, and *Links* are represented by *Arcs* in between in order to have a one-to-one correspondance to the AGG data structures (see *AGG-Graph* in Fig. 5).

5 Example: Petri Nets

As example to present the concepts in more detail, we use Place/Transition nets (P/T nets for short) [18]. Places are ellipses in our visual editor, and transitions are rectangles. The marking of a place is represented as natural number inside the place ellipse. Places and transitions have names. A place name is shown below the place ellipse and a transition name inside the transition rectangle. For simplicity, arc weights uniformly correspond to the token number “1”, hence arc inscriptions are omitted.

5.1 The Petri Net Alphabet

A sample alphabet for the VL of Petri nets is presented in Fig. 6 and conforms to the general structure of VL alphabets as given in Fig. 4. We use the NodeSymbolTypes Place and Transition for the Petri net nodes, the EdgeSymbolTypes ArcPT for Petri net arcs from a place to a transition and ArcTP for arcs from a transition to a place, and the LinkTypes arcPTsource, arcPTtarget, arcTPsource and arcTPtarget for linking the edge symbols to the node symbols. AttributeTypes (textual attributes) include the names of places and transitions, their positions and the token number on a place. Layout information (depicted in the bottom of Fig. 6) is given by ShapeFigures, Connections and TextFigures linked to the corresponding NodeSymbolTypes, EdgeSymbolTypes and AttributeTypes, respectively.

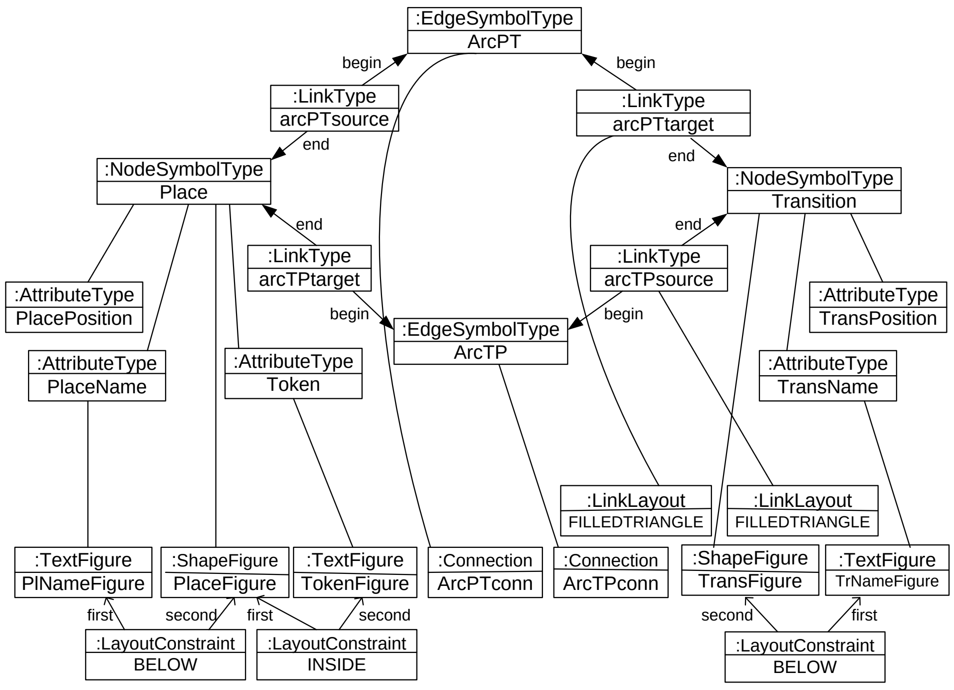

Fig. 6. Alphabet for Place/Transition Nets

For the NodeSymbolType Place the ShapeFigure PlaceFigure defines the shape to be an ellipse. We do not add information about the shapeColor and fillColor attributes of the ShapeFigure here because we use the default values, e.g. shapeColor=black and fillColor=none.

The Place attribute PlName contains the place name of type *String*, an

AttributeType. String attributes are by default layouted as TextFigure, making use again of default definitions for the layout of the text, e.g. font=("Arial", Font.ITALIC, 12), fontColor=black. The TextFigure PlNameFigure for the place name is connected to the ShapeFigure of the Place NodeSymbolType by the LayoutConstraint BELOW which means that the place name text is positioned below the ellipse shape of the place.

The Place attribute Token is represented by the AttributeType Token, whose layout is again a TextFigure. The LayoutConstraint INSIDE defines that the TokenFigure is always drawn inside the PlaceFigure.

The last Place attribute, the PlacePosition is again an AttributeType, namely the x and y coordinates (type *Point*) of the place figure. The position of a SymbolType figure is the only layout information which is given as Attribute-Type because the position contains information that is necessary to store and load a diagram (a concrete P/T net) in the generated editor. All the other layout attributes like ShapeFigure or LayoutConstraint serve for the generation of the visual editor features. For example, the class generated for the *CreatePlace* command implements the *Ellipse* figure class and thus incorporates the shape information. The LayoutConstraint 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.

Fig. 7 shows the abstract syntax of the instance diagram in Fig. 9 (a P/T-Net) over the alphabet shown in Fig. 6. For the instance diagram we use the more compact *AGG* notation and represents *Links* by *Arcs* and *Attributes* of the alphabet symbols as *AGG Attributes* of the corresponding *AGG Nodes*.

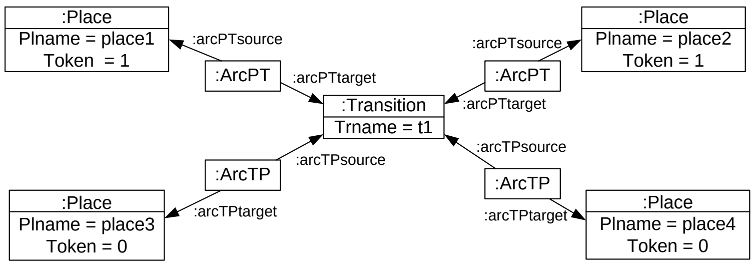

Fig. 7. AGG Instance Diagram over the P/T-Net Alphabet

5.2 The Petri Net Syntax Grammar

The VL syntax grammar contains only language generating rules. We get the abstract syntax grammar if we restrict the start graph and all rules according to the VL type graph. Fig. 8 shows the abstract syntax rules which define our Petri net VL. The start graph of the Petri net VL syntax grammar is empty. For each symbol of the Petri net VL there exists one generating rule. Negative application conditions (NACs) ensure in the first two rules that place names and transition names are unique. In the last two rules the NACs require 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.

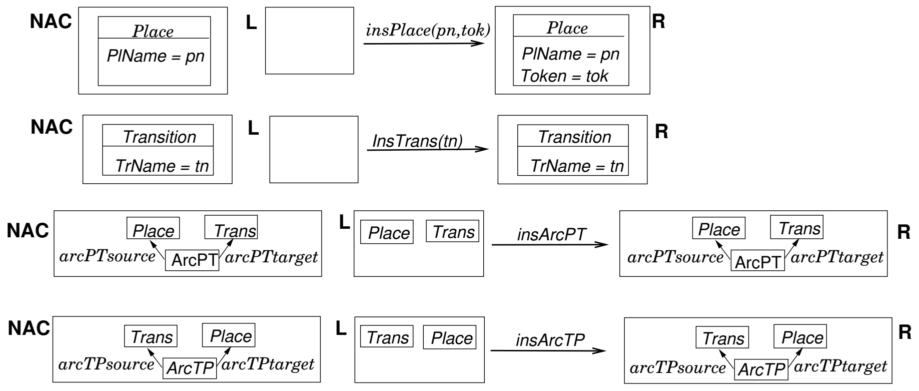

Fig. 8. Syntax grammar for the Petri net VL.

5.3 The Generated Petri Net Editor

Fig. 9 shows the generated GEF based Petri net editor which relies on the Petri net VL model.

The Petri net editor is divided into two panels. On the left hand side is the editor palette and on the right hand side the editor frame. The editor palette has the two default items *Select* for marking an editor symbol in the frame and *Marquee* for selecting a set of symbols with a dashed rectangle in the frame. Below the default items the palette contains icons for the generated VL-specific language elements (the symbol types). The editor frame contains a sample Petri net with one transition (*t1*), two pre-places (*place1, place2*), and two post-places (*place3, place4*). There is a token marking “1” inside *place1* and *place2*. The symbol properties (e.g. names, token markings) can be changed in a *Properties Dialog*. This plug-in also provides *undo/redo* functionality, *zooming* and *loading/saving*.

Internally, the editing operations are now realized by applying editing rules to the current diagram in the editor panel. All rules are applied to the abstract syntax of a diagram only, hence, the transformation can be executed using the AGG transformation engine. For example the syntax grammar rules depicted in Fig. 8 can be applied to edit the P/T-Net shown in Fig. 7. The layout of the resulting diagram is computed according to the layout information as provided by the VL model resp. as incorporated in the generated VL editor.

The layout constraints (e.g. *above, below, inside, right, left* for figures and *atSource, atCenter, atTarget* for connections) are translated by the editor generator to static GEF *constraints* which are data attached to each figure that gives additional guidance to the GEF layout manager. In the case of the place layout constraint *below(name text, place shape)* the generator treats the name text as *child* of the place shape figure. The position of the child is computed relative to the position of its parent figure by defining a surrounding rectangle

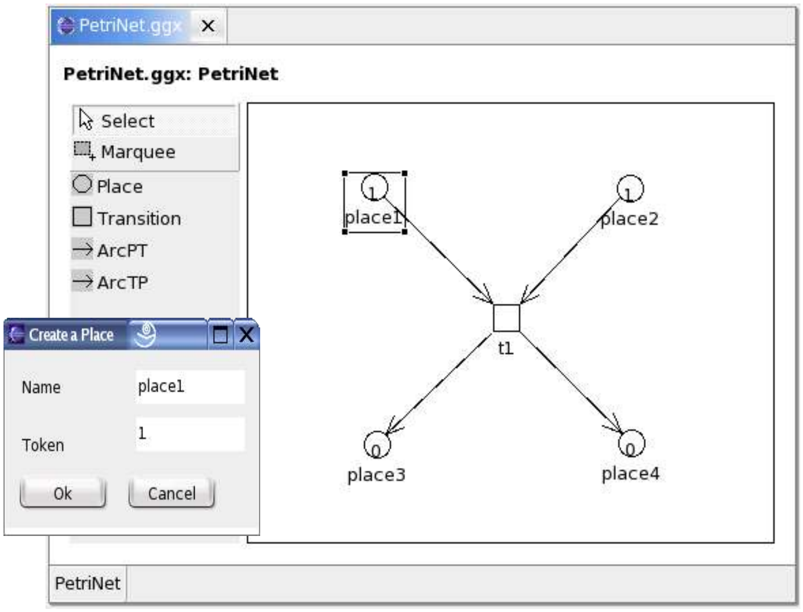

Fig. 9. Generated Petri Net Editor Example with Properties Dialog

(the *constraint*) around both parent and child figure (see the selected area for *place1* in Fig. 9). Another typical example for the use of GEF constraints for layout computation is the addition of a label to a connection where the label should appear near the center of the connection line (layout constraint *atCenter(text figure, connection)*).

6 Conclusion and Future Work

In this paper, we describe first ideas and the software architecture for a tool generating visual modeling environments from formal visual language specifications based on graph transformations and ECLIPSE-GEF. The envisaged tool environment TIGER combines the advantages of formal VL specifications using graph transformation (as offered by AGG) and of sophisticated graphical editor development features (as offered by ECLIPSE-GEF). The generated modeling environments themselves are ECLIPSE plug-ins and hence can be integrated in the ECLIPSE framework. The current state of this ongoing work (the first development step) is restricted to generating graphical editors for graph-like languages where the generation is graph-transformation based. This means, the VL specification so far consists of an alphabet (a type graph plus layout attributes) and a syntax grammar. In addition to the purely language generating syntax grammar a complete VL editing grammar could be defined. The editing grammar contains additional rules to define necessary and convenient editing operations. These additional rules concern e.g. the deletion of symbols or the change of attribute values. Moreover, editing rules can be defined by the editor designer to specify complex editor operations concerning more than one symbol whereas in the generated editor at the first development step only insertion / deletion of basic symbols were allowed. For example, a complex editing rule for statecharts can specify the deletion of a complete hierarchy level in a statechart [10] in one step. Note that the extension of the VL syntax grammar to the VL editing grammar must not lead to an extension of the defined visual language.

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 like in the first development step (emulated by simple editing rules without NACs), but add a parse button in the toolbar which evokes the parsing of the current diagram, internally realized by applying parse rules. For our Petri net example, the *parsing rules* are the inverted rules of the VL syntax grammar shown in Fig. 8. The application of the parsing rules tries to reduce the abstract syntax graph of the diagram edited so far to the empty stop graph (see e.g. [2]). If this is possible, the diagram is valid, otherwise an error message informs the user that the diagram is invalid. The advantage of the free-hand editing approach is that the editing of intermediate invalid diagrams is tolerated by the editor. Similar to parsing, a diagram could also be checked according to additional model constraints, as done in e.g. AToM^3 where a class diagram (the meta-model) is combined with constraints in e.g. *OCL* [17] which can be checked at any time during the editing process.

A sample editor for Petri nets is presented, together with the corresponding Petri net alphabet the editor is generated from. Since the development of Tiger is at a very first stage, it is beyond the scope of this paper to compare it with none graph transformation-based generators for visual modeling environments (such as MetaCASE [14], and GME [13], etc.).

Near future work (the second development step) extends the VL specification to include additional transformation rules (e.g. parsing rules, simulation rules) to allow more specific means for model manipulation in the generated environment. Further development steps aim at allowing more general kinds of diagrams instead of graph-like languages only. To allow a user friendly definition of the VL specification a *VL-Designer* component should 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. More general kinds of diagrams then can be used to realize even more sophisticated components of the generated modeling environment, such as animation of model behavior in different views [9] or model transformation.

References

[1] Bardohl, R., *GenGED – Visual Definition of Visual Languages based on Algebraic Graph Transformation*, Technical University of 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 2.1.3*, 2004, available at http://www.eclipse.org.

[4] Eclipse Consortium, *Eclipse Graphical Editing Framework (GEF) – Version 2.1.3*, 2004, available at http://www.eclipse.org/gef.

[5] Eclipse Consortium, *Eclipse Modeling Framework (EMF) – Version 1.1.1*, 2003, available at http://www.eclipse.org/emf.

[6] Eclipse Consortium, *Java Emitter Templates (JET)*, Eclipse Modeling Framework – Version 1.1.1, 2003, available at http://www.eclipse.org/emf.

[7] Ehrig, H., *Attributed Graphs and Typing: Relationship between Different Representations*, Technical University of Berlin, 2003.

[8] 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., 2004.

[9] 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. 5, Springer, 2004.

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

[11] Lambers, L., *A new Version of GTXL: An Exchange Format for Graph Transformation Systems*, International Workshop on Graph-Based Tools (GraBaTs), Italy, 2004.

[12] de Lara, J., Vangheluwe, H., 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In Proc. FASE'02, Springer LNCS 2306, pp. 174 - 188. See also the AToM^3 home page, http://atom3.cs.mcgill.ca

[13] Ledeczi, A. and Maroti, M. and Bakay, A. and Karsai, G. et al., *The Generic Modeling Environment.* In Proc. WISP'01, published by IEEE, Budapest, Hungary, 2001.

[14] MetaCase Consulting, *Domain Specific Modeling: 10 Times Faster Than UML*, Whitepaper available at http://www.metacase.com/papers/index.html.

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

[16] 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.

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

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

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

[20] 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.

[21] Taentzer, G., *XML-based Exchange Formats for Graphs and Graph Transformation Systems*, http://www.tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html, EU Working Group APPLIGRAPH on Application of Graph Transformation, 2002.

[22] 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.