5th MATHMOD VIENNA

Proceedings

Volume 1: Abstract Volume
Volume 2: Full Papers CD

5th Vienna Symposium on Mathematical Modelling

February 8-10, 2006
Vienna University of Technology,
Austria

ARGESIM Report no. 30

ARGESIM REPORT

ISBN 3-901608-30-3

ARGESIM REPORT

ARGESIM Report no. 30

I. Troch, F. Breitenecker (Eds).

Proceedings
5th MATHMOD Vienna

Volume 1: Abstract Volume
Volume 2: Full Papers CD

5th Vienna Symposium on Mathematical Modelling

February 8-10, 2006
Vienna University of Technology, Austria

ARGESIM - Verlag, Vienna, 2006
ISBN 3-901608-30-3

ARGESIM Reports

Published by **ARGESIM** and **ASIM**, Arbeitsgemeinschaft Simulation,
Fachausschuss GI im Bereich ITTN – Informationstechnik und Technische Nutzung der Informatik

Series Editor:
Felix Breitenecker (ARGESIM / ASIM)
Div. Mathematical Modelling and Simulation, Vienna University of Technology
Wiedner Hauptstrasse 8 - 10, 1040 Vienna, Austria
Tel: +43-1-58801-10115, Fax: +43-1-58801-10199
Email: Felix.Breitenecker@tuwien.ac.at

ARGESIM Report no. 30

Titel: Proceedings 5th MATHMOD Vienna –
5th Vienna Symposium on Mathematical Modelling
Volume 1: Abstract Volume
Volume 2: Full Papers CD

Editors: Inge Troch, Felix Breitenecker
Div. Mathematical Modelling and Simulation,
Vienna University of Technology
Wiedner Hauptstrasse 8 - 10, 1040 Vienna, Austria
Email: Inge.Troch@tuwien.ac.at

ISBN 3-901608-30-3

ARGE Simualtion News (ARGESIM)
c/o F. Breitenecker, Div. Mathematical Modelling and Simulation, Vienna Univ. of Technology
Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria
Tel.: +43-1-58801-10115, Fax: +43-1-58801-42098
Email: info@argesim.org; WWW: http://www.argesim.org

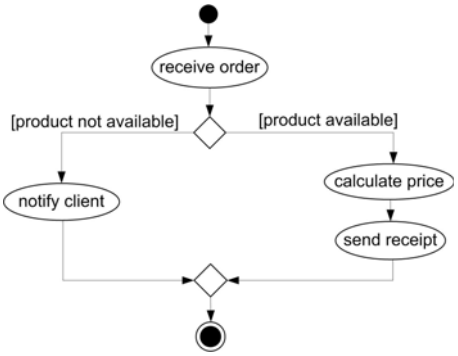
MODELING VISUAL LANGUAGES BASED ON GRAPH TRANSFORMATION CONCEPTS AND TOOLS

Claudia Ermel, Hartmut Ehrig, Karsten Ehrig
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
email:{lieske,ehrig,karstene}@cs.tu-berlin.de

Visual languages (VLs) and visual environments are used in an increasing frequency for software and system development. Prominent examples are the Unified Modeling Language (UML) for software modeling, Petri nets for the design of concurrent system behavior, and a variety of domain-specific diagrammatic notations for various purposes. Although visual languages are used wide-spread, a standard formalism for VL definition such as the Extended Backus-Naur-form (EBNF) for textual languages, is still missing. 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 visualization of the abstract syntax. While the concrete syntax contains the concrete layout of a visual notation, the abstract syntax abstracts from the layout and provides a condense representation to be used for further processing. Similarly to the EBNF, rules define the language grammar, but this time, graph rules are used to manipulate the graph representation of a language element.

For the application of graph transformation techniques to VL modeling, *typed attributed graph transformation systems* have proven to be an adequate formalism. Roughly spoken a typed attributed graph transformation rule $p = (L \to R)$ consists of a pair of typed attributed graphs L and R (its left-hand and right-hand sides). A *direct graph transformation* written $G \overset{p,o}{\Longrightarrow} H$, means that the graph G is transformed into the graph H by applying rule p at the occurrence o of the left-hand side of p in G.

A VL is modeled basically by an *attributed type graph* which captures 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 attributed graphs typed over the type graph. All concepts and constructions in this paper are illustrated by modeling the VL of activity diagrams, which are used to describe the control flow on activities. A sample activity diagram is depicted to the right.



The abstract alphabet of the sample VL defines two symbol types, *activity* nodes and *next* relations which connect activities. Activities can be of different kinds, i.e. *simple* activities (inscribed by a name), *start* and *end* nodes as well as *decisions*.

The abstract alphabet is extended by defining the concrete layout of activity diagrams. At the concrete syntax level, the VL alphabet defines that an activity is either visualized by an ellipse or by a polygon, depending on the activity kind.

Usually, the set of visual sentences (instances) over an alphabet should be further restricted to the meaningful ones (the valid visual models of the VL). By defining this restriction via graph rules, the constructive way is followed (as opposed to the declarative MOF approach where OCL constraints are used). The application of syntax graph rules builds up abstract syntax graphs of vaild models. 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.

Two tool environments have been developed at TU Berlin to support visual language modeling: the graph transformation engine AGG realizes attributed graph transformation at the abstract syntax level. The visual editor generator Tiger relies on AGG and on the graphical editor framework GEF of Eclipse, and generates a syntax-directed graphical editor from a VL alphabet and a syntax graph grammar. 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 and the generated editor code may easily be extended by further functionalities.

MODELING VISUAL LANGUAGES BASED ON GRAPH TRANSFORMATION CONCEPTS AND TOOLS

Claudia Ermel, Hartmut Ehrig, Karsten Ehrig
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
email:{lieske,ehrig,karstene}@cs.tu-berlin.de

Abstract. Visual languages and visual environments are increasingly important for software development. In this paper, we focus on the syntax definition of visual languages and visual models based on graph transformation. In analogy to textual language definition, graph grammars are used to define the structure of visual notations as well as their construction. Two tool environments are presented which have been developed at TU Berlin to support visual language modeling: The graph transformation engine AGG realizes attributed graph transformation at the abstract syntax level. The visual editor generator TIGER relies on AGG and on the graphical editor framework GEF of Eclipse, and generates a syntax-directed graphical editor from a visual language model given as a typed attributed graph transformation system.

1. Introduction

Although visual languages are used wide-spread, a standard formalism for visual language definition such as the Extended Backus-Naur-form (EBNF) for textual languages, is still missing. Nowadays two main approaches to visual language definition can be distinguished: grammar-based approaches or meta-modeling. Using graph grammars, multi-dimensional representations are described by graphs. This allows not only a visual notation of the concrete syntax, but also a visualization of the abstract syntax. While the concrete syntax contains the concrete layout of a visual notation, the abstract syntax abstracts from the layout and provides a condense representation to be used for further processing. Similar to the EBNF, rules define the language grammar, but this time, graph rules are used to manipulate the graph representation of a language element. Meta-modeling is also graph-based, but uses constraints instead of a grammar to define the visual language. While visual language definition by graph grammars can borrow a number of concepts from classical textual language definition, this is not true for meta-modeling.

In this paper, we present the graph grammar-approach to visual language definition. The concrete as well as the abstract syntax of visual notations are described by typed attributed graphs. The type information given in the type graph, captures the definition of the underlying visual alphabet, i.e. the symbols and relations which are available. A visual language is defined by a pair of typed attributed graph grammars, one for the concrete and one for the abstract syntax, both taking their type graph into account. For tool support, the attributed graph transformation development environment AGG [20] and the TIGER environment [13, 12] for generating visual editor plug-ins in ECLIPSE [5] have been developed at TU Berlin.

The paper is structured as follows: Section 2 provides a short introduction into graph transformation. In Section 3, we introduce the concepts for the definition of visual languages based on graph transformations. In Section 4, tool support is discussed, presenting the AGG tool for dealing with visual languages at the abstract syntax level and the TIGER tool, extending the visual definition of visual languages to the concrete syntax level. Finally, we summarize the main points of our approach in Section 5.

2. 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

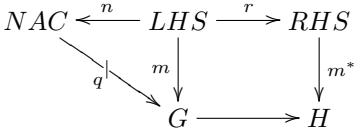
grammars can be used on the one hand to generate graph languages similar to Chomsky grammars in formal language theory. On the other hand, graphs can be used to model the states of all kinds of systems and graph transformation to model state changes. Meanwhile, graph transformation has been investigated as a fundamental concept for programming, specification, concurrency, distribution, visual modeling and model transformation [8, 9, 10].

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 (partial) graph morphism $r : LHS \rightarrow RHS$. 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 (VL) 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.

Intuitively, the application of rule p to graph G via a match m from LHS to 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.

Definition 2.1 (*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 r, match m, and comatch m^* is shown in the diagram to the right. The rule r may be extended by a set of *negative application conditions* $n : LHS \rightarrow NAC$ (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 $G_0 \overset{*}{\Rightarrow} G_n$. △

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

△

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 activity diagrams (see Fig. 4) which is explained in detail in Section 3. Graph objects which are preserved by the rule occur in both LHS and RHS (indicated by equal numbers for the same objects).

3. Visual Language Definition

Two main approaches to visual language definition can be distinguished: the declarative way, called *meta-modeling* and the constructive way, based on *(graph) grammars*.

Meta-modeling in the MOF Approach. UML is defined by the Meta Object Facilities (MOF) approach [18] which uses classes and associations to define symbols and relations of a VL. Within the MOF approach, each UML metamodel is structured in four sections: (1) Class diagrams, (2) explanations of the class diagram features, (3) well-formedness rules formulated in the Object Constraint Language OCL [17], and (4) informal description of the semantics of the features as natural-language comments.

VL Definition using Graph Grammars. While constraint-based formalisms such as MOF provide a declarative approach to VL definition, grammars are more constructive, i.e. closer to the implementation. In [16], for example, textual grammar as well as graph grammar approaches are considered for VL definition.

Using graph transformation, a *type graph* defines the visual alphabet, i.e. the symbols and symbol relations of a visual language. Layout information is integrated in the type graph by special layout-related nodes or edges connected to symbol nodes, and by constraints on the relations of visual representations. The layout-related nodes or edges include information about the symbol's shape (any kind of graphical figure or line), and the constraints establish certain visual relations (like "The shape for this symbol type is always drawn inside the shape for another symbol type." or "The shape for this symbol type has always a minimal size of ...").

A type graph together with a *syntax graph grammar* can directly be used as high-level visual specification mechanism for VLs [2]. The syntax grammar restricts the allowed visual sentences over the alphabet to the meaningful ones. Syntax grammar rules define language generating syntax operations. A syntax operation is modeled as a typed graph rule (typed over the VL type graph) being applied to the concrete syntax graph of the current diagram. Thus, only syntactical changes are allowed which are described by a syntax rule and which result again in a valid VL diagram. A syntax operation (i.e. the application of a syntax rule) results in a corresponding change of the internal abstract syntax graph of the diagram and its layout. The induced graph language determines the corresponding VL. Visual language parsers can be immediately deduced from a syntax graph grammar. Furthermore, abstract syntax graphs are also the starting point for model simulation, model transformation and model analysis by graph transformation [3, 21, 15].

All concepts and constructions in this paper are illustrated by activity diagrams, the visual language we use as running example. Activity diagrams are used to describe the control flow on activities. A concrete activity diagram is depicted in Fig. 1.

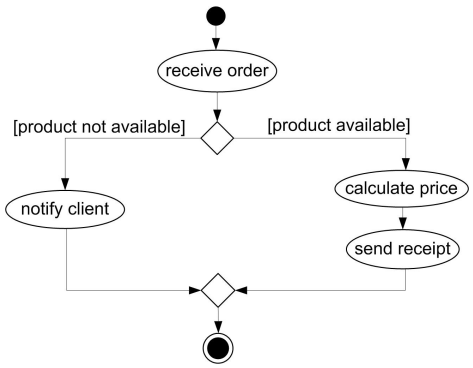


Figure 1: Sample Activity Diagram (Graphical Notation)

3.1 Definition of a Visual Alphabet

The underlying structure of a diagram is naturally described by an *abstract syntax graph (ASG)*. The extension of the ASG to a graph which also considers the concrete layout, i.e. the kind of figures, lines, and their relations, establishes the *concrete syntax graph (CSG)*. The CSG covers all aspects of diagram representation.

The abstract syntax graph contains symbols and links. Symbols may be attributed by additional data. The concrete layout is described by visuals which may be any kind of figures and lines, and by

layout constraints to establish certain visual relations. Additional attributes are needed to define the properties of visual representations.

Symbols and links of a specific visual language as well as their specific visual representations are defined in the abstract type graph T_A and the concrete type graph T_C, where T_A is included in T_C.

Definition 3.3 (*Visual Alphabet*)
A *visual alphabet* $Alph = (TG_C, TG_A)$ of a visual language consists of two type graphs $TG_C, TG_A \in \mathbf{Graphs_{TG}}$ where TG_C represents the concrete syntax of the visual language and TG_A represents the abstract syntax. △

Example 3.4 (*Visual Alphabet for Activity Diagrams*)
The visual 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 or merge nodes. Simple activities are usually described by some text. Moreover, next-relations may have inscriptions which are used to formulate conditions. This abstract syntax part of the visual alphabet described by type graph T_A, is depicted by all types without filling and their adjacent edge types in Fig. 2. All node types drawn as white rectangles are symbols, while all edge types are relations. Symbol attributes are denoted inside the rectangles.

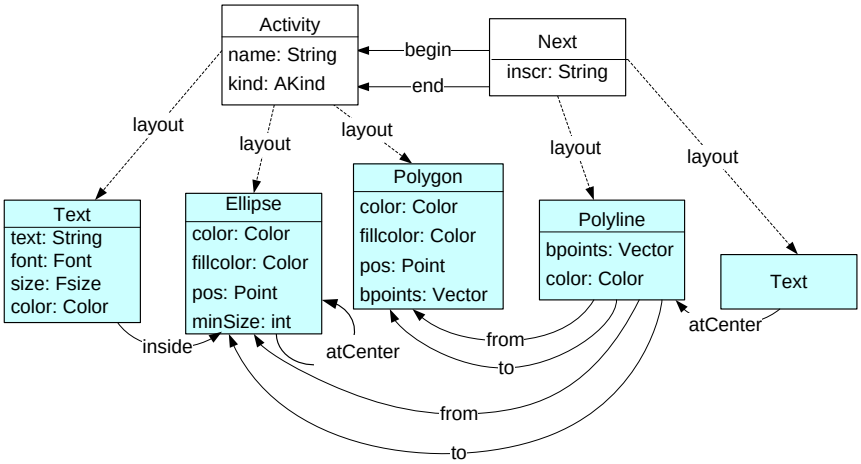


Figure 2: Visual Alphabet for Activity Diagrams

The abstract alphabet is extended by defining the concrete layout of activity diagrams. Activities are either represented by ellipses or by polygons, depending on their kinds. End activities are depicted by two ellipses, a black one inside another without fill color. A simple activity usually has a textual description which is put inside its ellipse. A next relation is shown by a poly line which is attached to two activity figures. A possible inscription is described by text and positioned at the line center. The figures and lines are categorized as visuals while their spatial relations are described by constraints. Each visual has the obvious properties such as font, font size, color, fill color, etc. as attributes. The whole visual alphabet described by a type graph T_C, is depicted in Fig. 2. △

A visual sentence over a visual alphabet (T_C, T_A) is given by a pair of graphs (S_C, S_A). The abstract syntax graph (ASG) S_A shows the abstract syntax structure of this sentence. It is typed over type graph T_A. Correspondingly, the concrete representation S_C of a visual sentence, i.e. the spatial relations graph (SRG) and its connection to the ASG, is typed over type graph T_C.

Definition 3.5 (*Visual Sentence over Visual Alphabet*)
Let $Alph = (TG_C, TG_A)$ be a visual alphabet. A *visual sentence*, also called model, over alphabet *Alph* is defined by the concrete and abstract graphs (S_C, S_A) with S_C typed over TG_C, and S_A typed over TG_A.

According to the typing, we have $S_A \subseteq S_C$. The restriction of S_C to the types in TG_A yields the abstract syntax graph S_A, i.e. formally the diagram to the right is a pullback in the category of graphs [8].

$$\begin{array}{ccc} S_A & \hookrightarrow & S_C \\ \downarrow & (PB) & \downarrow \\ TG_A & \hookrightarrow & TG_C \end{array}$$

△

Example 3.6 (*Activity Diagram as Visual Sentence over the Activity Diagram Alphabet*)

In Fig. 3 the abstract syntax of a larger part of the activity diagram in Fig. 1 is shown. Only the start activity is omitted for space limitations. Its typing over T_A in Fig. 2 is shown explicitly by type names inside of all nodes and at all edges. Moreover, a smaller section of the ASG, namely the activities *receive order, calculate price* and the decision and edges between them, is equipped with concrete layout showing a part of the CSG which is explicitly typed over T_C in Fig. 2.

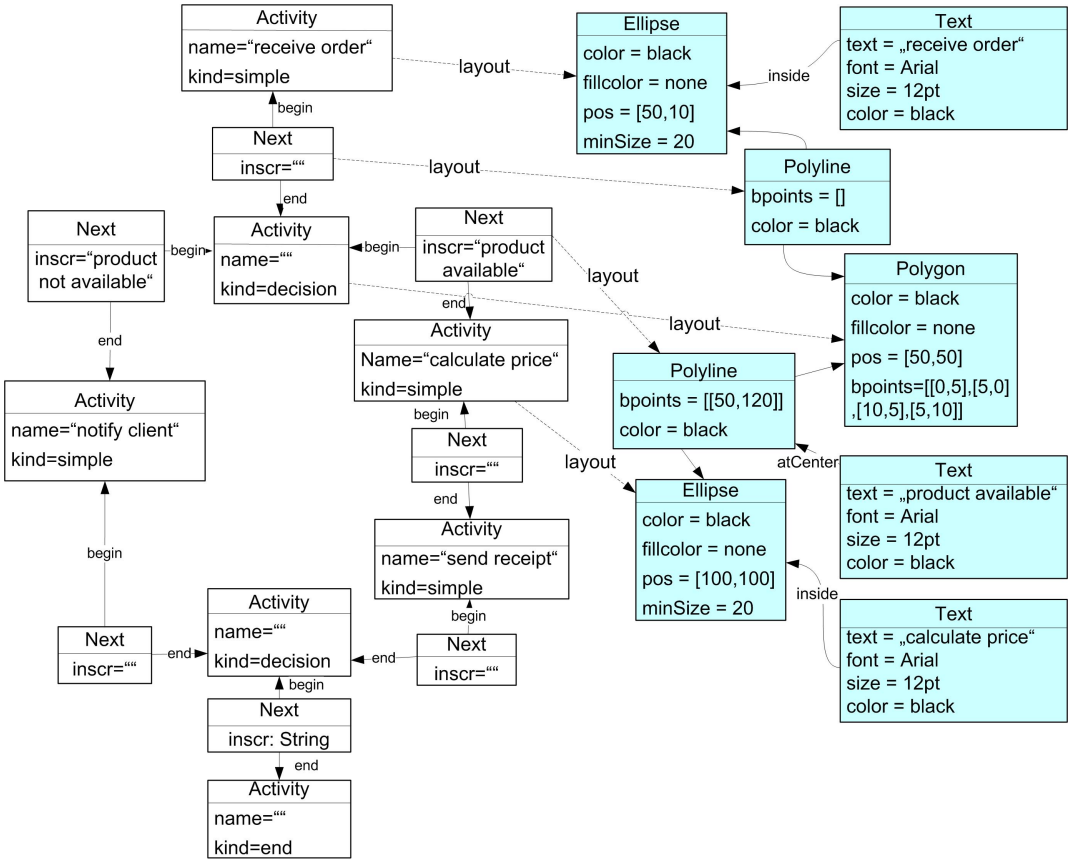


Figure 3: Parts of the Abstract and Concrete Syntax Graph of the sample Activity Diagram (Graph Layout)

△

3.2 Visual Language over a Visual Alphabet and a Syntax Grammar

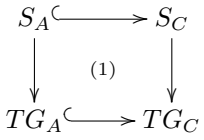
Defining a VL by a graph grammar, the application of graph rules builds up syntax graphs of visual sentences. This is a constructive approach to generate the language VL in contrast to the declarative MOF approach.

In general, a *VL syntax specification* is a pair of graph grammars (GG_C, GG_A) where $GG_C = (TG_C, S_C, P_C)$ is the VL syntax grammar and contains the complete syntax description of the VL.

Definition 3.7 (*VL Syntax Grammar*)
A *VL syntax grammar* $GG = (GG_C, GG_A)$ consists of a concrete syntax grammar and an abstract syntax grammar GG_A. The *concrete syntax grammar* $GG_C = (TG_C, S_C, P_C)$ contains the concrete

type graph TG_C of the visual alphabet, a start graph S_C, and a set of concrete syntax rules P_C. All graphs in GG_C, i.e. the start graph S_C and all rule graphs, are typed over TG_C. The *abstract syntax grammar* $GG_A = (TG_A, S_A, P_A)$, contains the abstract type graph TG_A of the visual alphabet, a start graph S_A and a set of abstract syntax rules P_A.

The abstract start graph S_A is constructed by restricting S_C to the types in TG_A, i.e. square (1) in the diagram to the right is a pullback in the category of graphs. Analogously, for each concrete rule $p_C = (L_C \to R_C) \in P_C$, we define the abstract rule $p_A = (L_A \to R_A) \in P_A$ by restricting the rule graphs of the concrete rule p_C to the types in TG_A.



△

Example 3.8 (*VL Syntax Grammar for Activity Diagrams*)
The syntax rules for activity diagrams define 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 abstract syntax grammar (see Fig. 4) 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.

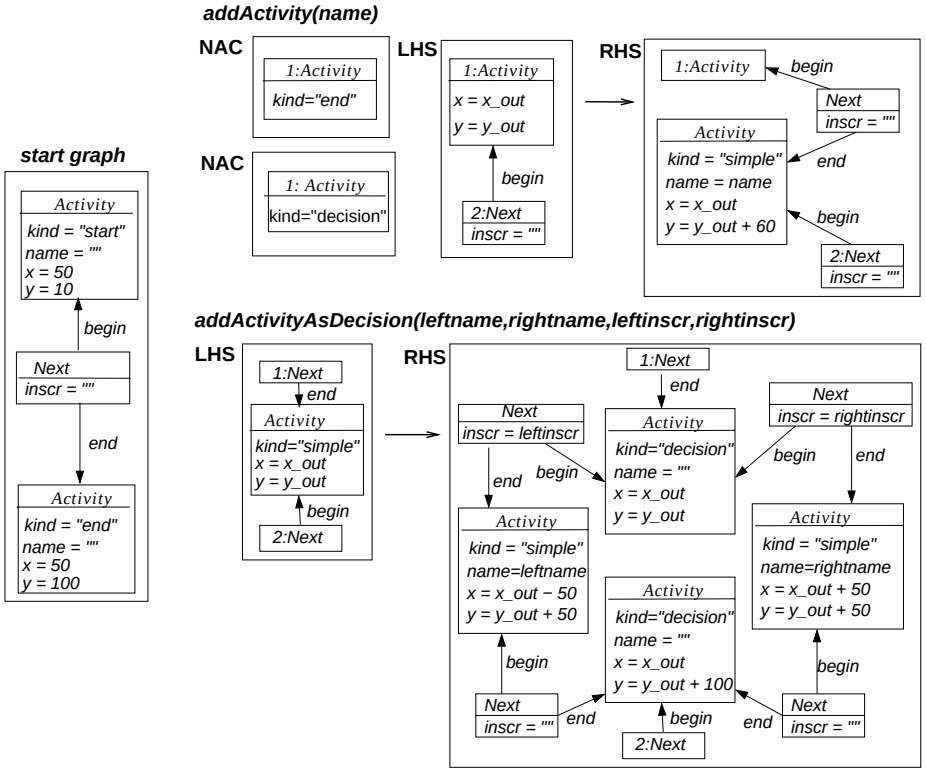


Figure 4: Syntax Grammar for Activity Diagrams

The syntax grammar contains two main rules for symbol creation: 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.

The activity diagram in Fig. 1 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. 4) exist for deleting and moving activities in order to obtain a well-layouted diagram. The deletion rules roughly correspond to the inverted creation rules from Fig. 4. △

All graphs generated by the rules of the syntax grammar GG define its visual language VL.

Definition 3.9 (*Visual Language over VL Syntax Grammar GG*)
The *visual language* $VL = (VL_C, VL_A)$ over a VL alphabet $Alph = (TG_C, TG_A)$ and a VL syntax grammar $GG = (GG_C, GG_A)$ consists of a *concrete visual language* VL_C, the elements of which are typed over TG_C and constructed by the rules of the concrete syntax grammar GG_C: $VL_C = \{G_C | S_C \stackrel{*}{\Longrightarrow} G_C\}$, and an *abstract visual language* VL_A, the elements of which are typed over TG_A and constructed by the rules of the abstract syntax grammar GG_A: $VL_A = \{G_A | S_A \stackrel{*}{\Longrightarrow} G_A\}$. △

4. Tool Support

In this section we discuss tool support provided by the AGG tool at the abstract syntax level and by the TIGER tool, extending the visual definition of visual languages to the concrete syntax level.

4.1 Abstract Syntax Modeling and Analysis by AGG

AGG is a general development environment for typed attributed graph transformation systems which follows the interpretative approach [1]. Its special power comes from a very flexible attribution concept. AGG graphs are allowed to be attributed by any kind of Java objects. Graph transformations can be equipped with arbitrary computations on these Java objects described by a Java expression. The AGG environment consists of a graphical user interface comprising several visual editors, an interpreter, and a set of validation tools. The interpreter allows the stepwise transformation of graphs as well as rule applications as long as possible. AGG supports several kinds of validations which comprise graph parsing, consistency checking of graphs and conflict detection in concurrent transformations by critical pair analysis of graph rules. Applications of AGG include graph and rule-based modeling of software, validation of system properties by assigning a graph transformation based semantics to some system model, and graph transformation based evolution of software.

On the one hand, AGG comes with its own visual development environment including graphical editors for graphs and graph transformation rules, on the other hand AGG offers an interface for the use of the graph transformation machine to external tools which have their own graphical user interface (such as TIGER). Right now, AGG version 1.3.0 is integrated in TIGER as transformation engine for the syntax-directed editing of visual diagrams in the generated ECLIPSE editors.

Critical pair analysis [8] is used to support the check of confluence of a graph grammar, i.e. uniqueness of the resulting graph, independent of the order of rule applications.

4.2 Concrete Syntax Modeling using TIGER

The tool environment TIGER [13, 12] (Transformation-based generation of modeling environments) supports the generation of visual environments, based on the one hand on recent MDA development tools integrated in the development environment ECLIPSE [5], and on the other hand on typed attributed graph transformation to support syntax-directed editing.

ECLIPSE offers rich support for graphical editor development in form of a number of plug-ins (e.g. EMF [4], Draw2D and the Graphical Editor Framework GEF [6]). TIGER combines the advantages of precise VL specification techniques (offered by AGG [20]) with sophisticated graphical editor and layout features (offered by GEF). Graph transformation is used at the abstract syntax level. TIGER extends the AGG engine by a concrete visual syntax definition. From the definition of the visual language, the TIGER *generator* generates Java [19] 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 and the generated editor code may easily be extended by further functionalities. Fig. 5 shows an overview of the TIGER software architecture.

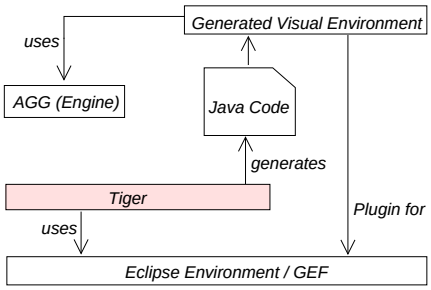


Figure 5: TIGER Architecture Overview

In TIGER, a VL alphabet has to contain not only the definition of the VL's abstract syntax, but also a specification of the intended layout which controls the generation of the visual editor.

Fig. 6 shows (a part of) the TIGER meta type graph, where all VL alphabet type graphs have to be typed over this meta typegraph. At the abstract syntax level (the upper part of Fig. 6), each VL alphabet consists of *NodeSymbolTypes* (e.g. activities in activity diagrams), *EdgeSymbolTypes* (e.g. next-relations in activity diagrams) and *LinkTypes* (the connection of *EdgeSymbolTypes* to *NodeSymbolTypes*). Moreover, *NodeSymbolTypes* may be attributed by *AttributeTypes* (e.g. names of simple activities).

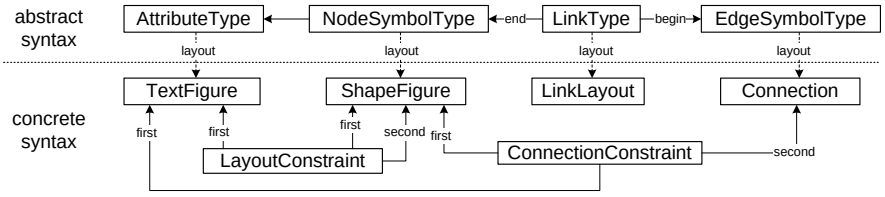


Figure 6: Meta Type Graph for VL Alphabets in TIGER

At the concrete syntax level (the lower part of Fig. 6), the graphical layout for a node symbol of a certain *NodeSymbolType* is given by a *ShapeFigure*. The shape of *NodeSymbols* can be a simple form, e.g. a rectangle, circle, ellipse or a closed polygon. Shape figure properties such as stroke and fill colors are given by additional attributes (not shown in Fig. 6). The standard layout for a textual attribute of type *AttributeType* is given by a *TextFigure* (with attributes *font*, *fontColor*, ..). The graphical relations between *TextFigures* and *ShapeFigures* are expressed by *LayoutConstraints*, such as *below(TextFigure, ShapeFigure)*. *Figures* can be connected by *Connections* (i.e. lines or polylines) which represent the concrete graphical layout for the *EdgeSymbolTypes*. The graphical representation of a link (e.g. a colored arrow head) is modeled by a *LinkLayout* object. Graphical relations between a *Figure* and a *Connection* can be modeled as *ConnectionConstraints*, such as *atCenter(TextFigure, Connection)*.

Based on a VL alphabet, TIGER uses the default GEF graph layouter to compute the layout of the symbols and links in the generated editor. Editing operations modeled by abstract syntax rules are performed in TIGER-generated editor plug-ins by AGG operating on the abstract syntax of the VL diagrams. The concrete layout is computed after each operation on the basis of the generated GEF layouting features.

At the current development stage, TIGER generates a GEF-based diagram editor from a VL specification which consists of a VL alphabet and a syntax grammar. For example, the visual editor for activity diagrams generated by TIGER from the alphabet in Fig. 2 and the syntax grammar in Fig. 4, is shown in Fig. 7.

Moreover, model transformation from a model over a source VL to a model over a target VL is also possible using TIGER. For this purpose, a model transformation graph grammar at the abstract syntax level has to be provided in AGG, where the rules are typed over a union of the source and target VL

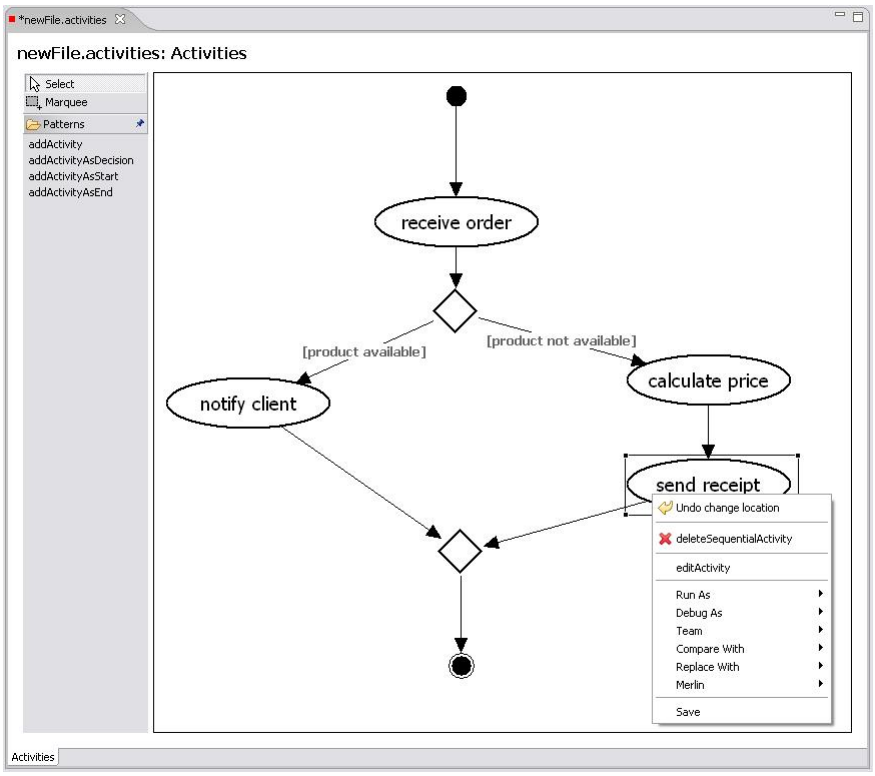


Figure 7: TIGER-generated Editor Plug-In for Activity Diagrams

type graphs and specify the operations which transform elements of the source language to elements of the target language. An example for a model transformation from activity diagrams to Petri nets is described in another contribution to this conference [7].

Future work is planned to extend the TIGER environment to support also simulation and animation of visual behavior models based on graph transformation. Up to now, we allow graph-like languages only, such as Petri nets, automata or class diagrams.

5. Conclusion

In this paper we have described how graph transformations can be used for modeling visual languages, where the abstract and concrete syntax of visual languages is based on typed attributed graphs. Furthermore two tool environments have been introduced: the AGG graph transformation environment based on the abstract syntax and the TIGER environment for generation of visual editor plug-ins in ECLIPSE based on the concrete syntax.

In contrast to the meta-modeling approach, for the graph transformation approach suitable analysis techniques have been developed [8]. For example critical pair analysis and strict local confluence of critical pairs ensures local confluence of the graph transformation system. This means that the resulting graph is always unique, independent of the order of rule application.

References

[1] AGG Homepage. `http://tfs.cs.tu-berlin.de/agg`.

[2] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of Graph Transformation to Visual Languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds., *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[3] 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.

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

[5] Eclipse Consortium. *Eclipse – Version 2.1.3*, 2004. `http://www.eclipse.org`.

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

[7] 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, eds., *Proc. Intern. IMCAS Symposium on Mathematical Modelling (MathMod)*. ARGESIM-Reports, 2006.

[8] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006. to appear.

[9] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[10] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[11] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In F. Parisi-Presicce, P. Bottoni, and G. Engels, eds., *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*. Springer, 2004.

[12] K. Ehrig, C. Ermel, S. Hänsgen, and G. Taentzer. Towards Graph Transformation based Generation of Visual Editors using Eclipse. In M. Minas, ed., *Visual Languages and Formal Methods*, volume 127/4 of *ENTCS*, pages 127–143. Elsevier Science, 2004.

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

[14] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[15] R. Heckel, J. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains . In H.-J. Kreowski, ed., *Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002)*, pages 11 – 22, 2002.

[16] K. Marriott and B. Meyer. *Visual Language Theory*. Springer, 1998.

[17] Object Management Group. *UML 2.0 OCL Specification*, 2003. `http://www.omg.org/docs/ptc/03-10-14.pdf`.

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

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

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

[21] D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, eds., *Proc. ICGT 2002: 1st Int. Conf. on Graph Transformation*, volume 2505 of *LNCS*, pages 378–392. Springer, 2002.