Electronic Notes in Theoretical Computer Science 50 No. 3 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume50.html 7 pages

Abstraction in Graph-Transformation Based Diagram Editors

Oliver Köth and Mark Minas [1]

Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen, Germany

Abstract

This extended abstract recalls how visual language editors benefit from graph grammars and how the editor-generator DiaGen is based on this approach. We then outline how generated editors can create abstract diagram representations for further processing, e. g. for communication with other tools or for diagram visualizations with varying, adjustable detail level which is a valuable aid when editing large diagrams. These concepts are illustrated with UML class diagrams.

1 Introduction

An important application of graph grammars and graph parsing techniques is the definition of visual languages (VLs) and the creation of VL editors [2]. The use of a parser for the analysis of diagrams allows to treat the graphical interface as a kind of extended drawing program; thus the same front-end can be used across a broad spectrum of different VLs with very little adaption; only the set of allowed VL symbols (boxes, arrows, etc.) must be specified. The diagrams are then created by drawing these symbols using "direct manipulation"; the resulting drawings are analyzed by a VL parser to check the correctness of the diagram and create an abstract representation for further processing.

In this paper we explain how these concepts are realized in the DiaGen system [6,7] and show how the grammar and the parsing process can be exploited to generate such an abstract (graph) model in a very direct and simple manner. Subsequently, we present a way of using the same abstraction concepts to enhance the usability of VL editors. This can be done by generating "abstraction views" of a diagram, which allow the user to concentrate on the diagram parts that are important for the current editing task.

[1] Email: Mark.Minas@informatik.uni-erlangen.de

The main advantage of including a parser in a VL editor is that it avoids the necessity for specifying a complete set of structure-based editing operations so that the language-specific part of the editor is kept as small as possible. This makes it possible to use an "editor generator" in combination with a class library to create editors for complex VLs from relatively short and concise syntax descriptions. DiaGen is such an editor-generator framework, which combines the parsing approach for VL editors with the transformation-oriented approach, because it also allows complex structure-based transformations on the edited diagrams (described as graph transformations on the internal graph model of the diagram [7]).

2 Modeling Diagram Syntax

For the rest of this paper, we will use a DiaGen editor for UML class diagrams as an example. Fig. 1a shows a small model created with this editor. Examples for the different graph structures that we will discuss can be found in Fig. 1c through f (The graphs cover only a part of the sample diagram.) Except for the ASG in Fig. 1e, all structures have been simplified somewhat to show the underlying concepts more clearly.

DiaGen represents diagrams internally as a labeled hypergraph consisting of "component" and "relation" (hyper) edges. A component edge corresponds to a visual symbol in the actual diagram. Every symbol has a number of "connector regions" where it can interact with other components. Those connector regions are mapped into hypergraph nodes which the respective component hyperedge attachs to. Finally, the syntax description can define arbitrary geometric relations between the connector regions (most often containment and intersection) which are mapped into binary relation edges that connect the corresponding nodes.

The resulting hypergraph model (HGM, cf. Fig. 1c) of the diagram is subjected to a two-step analysis process: first it is converted into a "reduced hypergraph model" (rHGM, cf. Fig. 1d) by a kind of graph transformation [7]. The hyperedges of the rHGM then serve as the terminal set for the parsing process. DiaGen uses context-free hypergraph grammars with some extensions (described in [2,7]) for the description of the VL syntax. A fault-tolerant parser analyzes the rHGM and creates a derivation structure, which mostly consists of (context-free) derivation trees, but it also contains "embedding" edges. These express the fact that elements that are not part of a single major entity (nonterminal) but are embedded into a context of several nonterminals. The derivation structure thus forms a directed acyclic graph (the derivation DAG or DDAG, cf. Fig. 1f), but the "nodes" of this graph (terminals and nonterminals) are themselves hyperedges which connect to nodes (from the rHGM, not shown in the figure). All the mentioned edge types can be attributed, and attribute propagation rules allow to build an abstract representation of the diagram from the properties of the symbols (e. g. textual

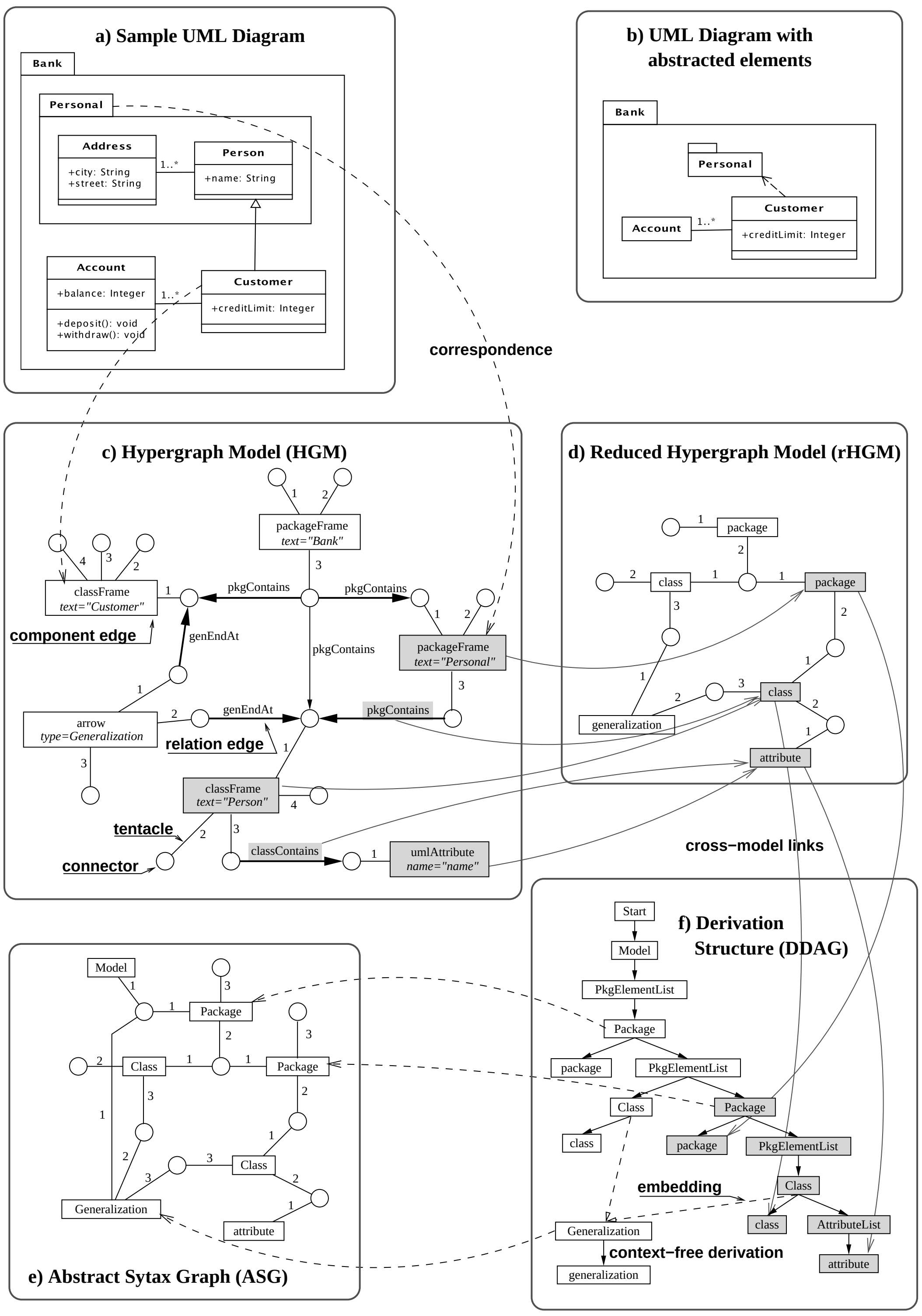

Fig. 1. A sample UML class diagram (a), its *focus and context view* (b), and the different models (c–f) of (a). Nodes are drawn as circles. Hyperedges are either shown as rectangles which are connected to nodes, or as arrows. Numbers indicate the order of visited nodes.

content) and the diagram structure [7].

3 Generating Abstract Models of Diagrams

The graph grammar that describes the syntax of a VL and the DDAG that is generated from a diagram can not only be used as a means of determining its correctness; in addition, they also capture larger conceptual entities in the diagram and describe the way they are related, and thus have a concrete "meaning". In the UML for instance, there exist entities like "classes", "packages" and "associations" that consist of several subparts; a parallel hierarchy exists in the grammar that describes the visual syntax of UML class diagrams. This offers an easy way to gain an abstract model of the diagram: We simply take all the nonterminals from the DDAG that bear one of a set of "meaningful" types (e. g. "package") along with all connected nodes (remember that the nonterminals are themselves hyperedges connected to nodes) and treat this subgraph as a high-level model of the diagram (an "abstract syntax graph" or ASG, cf. Fig. 1e)

In the simple case of the example, the ASG is almost identical to the rHGM. In general, the rHGM may also contain additional edges that cannot be parsed correctly or groups of edges that are combined into a single ASG entity. Also, for some UML elements there exist multiple alternative visualizations that lead to different rHGM structures, but all of them are derived from the same nonterminal type.

In the case of UML class diagrams, we would additionally like the ASG of the diagram to conform to the official definition of the "abstract model" behind UML diagrams. The UML standard describes this abstract model as an object model based on the "UML metamodel" [8]. As an example, every package in an UML diagram must be represented by a "package" object with attributes like "name" and "visibility" in its abstract model. Our approach does not create an object model for a diagram, but instead a hypergraph structure (the ASG). But it is possible to define a mapping from object structures to hypergraphs that allows a very close correspondence: For instance, objects with their attributes correspond to hyperedges with attributes, and associations between objects are mapped to nodes which the respective hyperedges attach to. The correspondence is close enough that we can write out a tree traversal of the ASG of a UML class diagram into the standardized XMI file format [8], and use the abstract description in other industry UML tools (e. g. Together, `www.togethersoft.com`).

This way of integrating abstract graph models into the syntax description (graph grammar) of a VL offers the possibility to create them from a concrete diagram and check them for correctness with standard graph parsing techniques. It should also be possible to use the graph grammar productions to create a concrete diagram from an abstract model, although we have not yet dealt with this "unparsing" problem. This would allow to generate visual representations from abstract diagram models (e. g. XMI data) or to use graph transformations to describe modifications of the abstract model (program structure transformations in the case of UML) and visualize the results directly.

4 Visualizing Abstraction

Abstraction concepts are not only useful in internal models of diagrams; they also offer a valuable aid for diagram editing when they are visualized: Entities in the diagram that are not currently of interest to the user can be replaced with (smaller) abstraction visualizations, while other parts remain visible in full detail. The resulting technique is known as “focus and context” viewing [11]; it leads to a better utilization of screen space and enhances usability [10]. Fig. 1b shows a focus and context view of the example diagram that concentrates on the “Customer” class.

Previous approaches to applying this concept were either restricted to specific applications or operated on the basis of optical display transformations only (e. g. fisheye views [5]). The use of hypergraphs as a common model for diagram languages allows us to support structure-based abstraction that is individually tailored to the specific diagram language while keeping the required programming effort small. We achieve this by treating abstraction as just a special case of diagram transformation: For UML class diagrams, we have defined two abstraction transformations that simplify classes and packages; their effect can be seen in the “Account” and “Personal” elements in the diagram (cf. Fig. 1b).

As described in [7] and mentioned in section 1, diagram transformations are specified in the form of graph transformation rules that find and replace patterns in the HGM of the diagram; higher-level transformation programs provide a control structure and allow iterated or conditional execution of subprograms or basic rules. Adding or removing component hyperedges creates or deletes visual symbols; adding or removing relation edges can cause adaption of the diagram layout.

These mechanisms must only be slightly extended for the definition of *abstraction transformations*: such a transformation receives a (user selected) diagram element as an argument; it uses one or more patterns to select all the graph parts that represent details of the desired diagram entity and (temporarily) removes them from the diagram. Of course it is necessary to add inverse “expansion” transformations that re-insert these details if they need to be considered again later. Therefore the graph elements are not actually deleted; instead they remain in the HGM structure but are marked as “hidden”, so they are neither visualized nor considered by the analysis process.

To render the specification of abstraction transformation as easy as possible, we want to make use of the information contained in the derivation structures for selecting the details of a diagram entity. To this end, we have introduced cross-model links that connect corresponding edges in the different internal graph structures (HGM, rHGM and DDAG) and can be included in graph patterns or transformation rules. Although complex graph patterns offer many possibilities for selecting the desired elements of the diagram (resp. the HGM), we have found that, in most cases, abstraction transformations need to use only a simple standard pattern: The selection starts from a diagram element selected by the user (e. g. a package frame, cf. Fig. 1a and c), it follows the cross-model links to the DDAG, and walks "up" the DDAG to the first nonterminal symbol of a certain type (e. g. "Package") representing the desired higher-level entity; finally it follows "down" the DDAG edges and against the cross-model links to include everything derived from that nonterminal. Fig. 1a–f give an idea how this selection leads through the elements with a gray background to find all parts of the "Personal" package.

Since we use freely programmable diagram transformations to create abstraction views, it is also possible to include other adaptions of the diagram that are necessary to preserve its correctness, e. g. to use language-specific abstraction symbols. Finally, the individual abstraction transformations for single diagram entities like UML classes or packages can be combined into higher-level transformations that create suitable abstraction views of the whole diagram as seen in Fig. 1b.

5 Related Work

There are several other approaches for generating visual language editors from a formal specification. The ones which are based on an abstract graph model (e. g. GenGEd [1], Kogge [4]) automatically come with "built-in" abstract representations. Other tools which avoid abstract internal models (e. g. Penguins based on *Constraint Multiset Grammars* [3]) lack this immediate availability of abstract models; they rather have to explicitely create external diagram representations by attribute evaluation.

However, as far as we know, there is no other editor-generator framework which supports visualization abstractions with varying, adjustable, structure-based detail level. The only tools with "focus and context" viewing are either special purpose tools (e. g. [9]), or they offer distorted "fisheye views" only (e. g. [5]).

References

[1] R. Bardohl. GenGEd: A generic graphical editor for visual languages based on algebraic graph grammars. In *Proc. 1998 Symp. on Visual Languages (VL'98)*, pages 48–55, 1998.

[2] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski,

and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages and Tools*, pages 105–180. World Scientific, Singapore, 1999.

[3] S. S. Chok and K. Marriott. Automatic construction of user interfaces pen-based computers. In *Proc. 3rd Int. Workshop on Advanced Visual Interfaces, Gubbio, Italy*, 1996.

[4] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in practice: a case for KOGGE. In *Advanced Information Systems Engineering, Proc. 9th Int. Conf. (CAiSE'97)*, LNCS 1250, pages 203–216. Springer, 1997.

[5] T. A. Keahey and E. L. Robertson. Techniques for non-linear magnification transformations. In *Proc. IEEE Symp. on Information Visualization, IEEE Visualization*, pages 38–45, 1996.

[6] M. Minas. Creating semantic representations of diagrams. In M. Nagl and A. Schürr, editors, *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, LNCS 1779, pages 209–224. Springer, Mar. 2000.

[7] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. Appears in *Journal of Science of Computer Programming (SCP)*, 2001.

[8] Object Management Group. *Meta Object Facility Specification.* http://www.omg.org/uml/.

[9] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot. Context and interaction in zoomable user interfaces. In *Proc. Advanced Visual Interfaces 2000 (AVI'2000)*, pages 227–231, 2000.

[10] D. Schaffer et al. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on CHI*, 3(2):162–188, 1996.

[11] M. Sheelagh, T. Carpendale, D. Coperthwaite, and F. Fracchia. Making distortions comprehensible. In *Proc. 1997 Symp. on Visual Languages (VL'97)*, pages 36–45, 1997.