Towards Generic Rule-Based Visual Programming [*]

Berthold Hoffmann
Fachbereich Mathematik/Informatik
Universität Bremen
Postfach 33 04 40, 28334 Bremen, Germany
hof@informatik.uni-bremen.de

Mark Minas
Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
minas@informatik.uni-erlangen.de

Abstract

This paper outlines DIAPLAN, *a visual rule-based programming language and environment that is based on the computational model of graph transformation. Thanks to its* genericity, DIAPLAN *can generate visual environments with no restriction on their visual representation, and also supports* object-oriented programming *since its graphs are* hierarchically structured.

1. Introduction

Graphs are inherently visual, and graphs are used as an abstract model for visual representations [1, 5, 7]. Furthermore, specifying the manipulation of graphs in terms of graph transformation rules [9] is some kind of visual programming. The rule-based transformation of graphs is a common computational model for visual programming languages. However, existing programming languages based on graph transformation, such as PROGRES [10] or LUDWIG$_2$ [8], have not been as successful as one could expect. We believe that this has two major reasons:

- The *structuring* of graphs as nested graph objects, and thus *object-oriented programming* are not supported.
- It is not possible to *customize* the "standard" graph notation to the visual notation of particular application domains.

However, some approaches to these problems exist for visual environments which are not based on graphs: Object-oriented programming languages have been developed in the visual programming community [2], notably *Prograph* [3]. Furthermore, there are visual language tools which allow to use domain-specific visual representations, e.g., CALYPSO, a tool for visually defining data structures programs [11]. However, we are not aware of any language or tool that allows to *visually specify* and *generate* visual language environments which then use domain-specific visual representations.

This paper is about DIAPLAN, a visual, rule-based programming language and environment for implementing visual languages, which offers just these features. The DIAPLAN programming environment consists of

- a visual programming language which supports graph typing and structuring as well as object-oriented programming for specifying graph transformations. These rules specify the behavior of the generated visual language.
- a tool for specifying how graphs are represented by domain-specific diagrams in the generated visual language and how the user can interact with these diagrams. This makes DIAPLAN a *generic* environment.

2. DIAPLAN

Due to space limitations, only some aspects of DIAPLAN can be outlined here. Details can be found in [4, 6] and in a longer version of this paper which is available from the authors.

DIAPLAN uses *graphs* which consist of a finite number of *nodes* and *edges*, where edges may visit any number of nodes, not just two. (Such graphs are often called *hypergraphs*). For structuring purposes, we distinguish a subset of edges of a graph as *frames*: Every frame contains a nested subgraph that may contain frames again. Such graphs are called *hierarchical*.

A *Graph transformation rule* $P \rightarrow R$ consists of two hierarchical graphs, as illustrated in Fig. 1. (The symbols "// —" are explained later.) If its pattern graph P *matches* some part of a hierarchical graph, and if the variables in it *bind* appropriate subgraphs, this part is replaced by the graph R, after instantiating the variables by the bound subgraphs. The rule in Fig. 1 matches a frame containing a list graph that starts with an item graph bound to h, and re-

[*]This work has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (APPLIGRAPH).

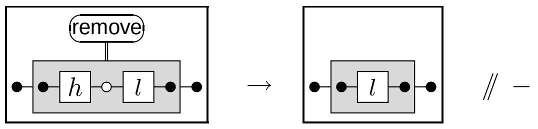

Figure 1. The predicate remove

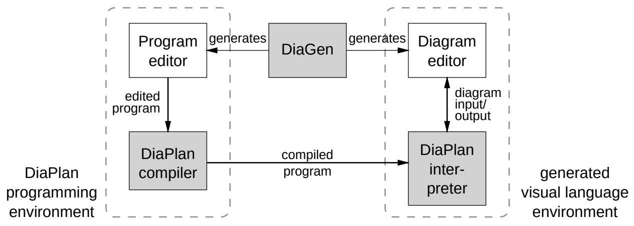

Figure 2. DIAPLAN **system architecture**

places it with a frame that contains only the tail of the list graph that was bound to l.

Furthermore, DIAPLAN offers *abstraction*, *control*, and *encapsulation* for object-oriented programming. Only abstraction is outlined here: We consider certain edges as *predicate edges* (e.g., the remove edge in Fig. 1). The links of a predicate edge indicate its *parameters*. A parameter can be just a *node*, but also an *edge*. A *predicate* named p is defined by a set of rules wherein every pattern contains exactly one p-edge, and every replacement may contain other predicate edges. A predicate p is *applied* by applying one of its rules to a p-edge in the host graph. An *otherwise part* specifies what happens if none of the rules applies. The predicate remove in Fig. 1 has just one rule, and takes a list frame as its parameter. The otherwise part "// —" indicates that its application *fails* if the frame contains an empty list graph. A predicate is *evaluated* by first applying one of its rules, and evaluating all predicates that are called in its replacement, recursively.

DIAPLAN makes use of the diagram editor generator DIAGEN for specifying domain-specific diagram representations of graphs. DIAGEN allows to generate visual editors for specified diagram languages. These editors then translate diagrams into graphs and vice versa [7].

3. Implementation

DIAPLAN is in a rather concrete phase of its design. Its implementation is only at a very early stage. Here we just outline the architecture of the implementation which is shown in Fig. 2: It shows the programming environment on the left and how it is used to make up a specific visual environment which is shown on the right.

- The *program editor* is a visual editor for writing programs according to the programming language outlined in Section 2. This editor is generated by DIAGEN.
- The *compiler* reads such programs and transforms them into an internal form that can be easily and efficiently executed by the interpreter. The compiler checks whether the program violates any lexical, syntactical or contextual rule of the language.
- The *interpreter* executes programs of the language by reading the input graph, transforming the graph according to the program, and re-displaying the modified graph, in a loop, steered by user interaction.
- The interpreter shall have an interactive *diagram editor* by which the input data is created. This editor is generated by DIAGEN to support customizable diagram notation for the graphs which are manipulated by the interpreter.

Altogether, only the heart of the system, namely its compiler and interpreter, have to be implemented anew, while we rely entirely on DIAGEN for generating its user interfaces.

References

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

[2] M. M. Burnett, A. Goldberg, and T. G. Lewis, editors. *Visual Object-Oriented Programming*. Manning, 1994.

[3] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph. In Burnett et al. [2], chapter 3, pages 45–66.

[4] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. In *Foundations of Software Science and Computation Structures (FOSSACS 2000)*, LNCS, 2000. To appear.

[5] M. Erwig and B. Meyer. Heterogeneous visual languages – Integrating visual and textual programming. In *Proc. VL'95*, pages 318–325, 1995.

[6] B. Hoffmann. From graph transformation to rule-based programming with diagrams. In *Proc. Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99)*, volume 1779 of *LNCS*, pages 165–180.

[7] M. Minas. Creating semantic representations of diagrams. In *Proc. Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE*'99)*, volume 1779 of *LNCS*, pages 209–224.

[8] J. J. Pfeiffer, Jr. LUDWIG$_2$: Decoupling program representations from processing models. In *Proc. VL'95*, pages 133–139, 1995.

[9] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[10] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Rozenberg [9], chapter 13, pages 487–550.

[11] R. Wodtli and P. Cull. CALYPSO: A visual language for data structures programming. In *Proc. VL'97*, pages 166–167, 1997.