A Generic Model for Diagram Syntax and Semantics*

BERTHOLD HOFFMANN
Universität Bremen

MARK MINAS
Universität Erlangen

Abstract

In this paper, we recall how the *syntax* of diagrams is captured by the diagram editor generator DIAGEN, and outline a visual, rule-based, and object-oriented programming language based on graph transformation by which DIAGEN can be extended to model the *semantics* of diagrams as well. This language is *generic* w.r.t. the diagram notation to be used in the programmed visual system and may thus be used for implementing arbitrary systems based on diagram-manipulation.

Keywords

diagram editor generator, diagram processing, graph language, graph transformation, visual programming language

Introduction

Diagrams are an effective means for visual communication between humans that have been used in many fields of science for quite a long time. In recent years, diagrams have more and more been used as input and output of software systems in order to make them user-friendlier. The use of diagrams in computers does not only call for a precise *specification* of their syntax and semantics, but also for *tools* that support their recognition, processing, and drawing.

In this paper, we first review the DIAGEN system [9, 10, 11], which allows the syntax of diagram languages to be specified by graph grammars, and generates diagram editors dedicated to these languages. We then outline a visual, rule-based, and object-oriented programming language based on graph transformation by which the semantics of diagrams can be specified on an abstract level, and discuss how a processor for the language can be incorporated in DIAGEN. We

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

illustrate that the so extended DIAGEN system is *generic* w.r.t. the diagram notation to be used in the programmed visual system, so that it may generate editors and processors for arbitrary kinds of diagrams.

Diagram Syntax

Andries *et al.* [1] have proposed a graph model for representing diagram syntax. In DIAGEN, this model has been refined as shown in Figure 1.

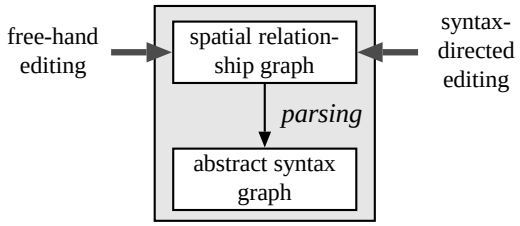

Figure 1: DIAGEN's model for diagram syntax

Scanning creates a *spatial relationship graph* that captures the lexical structure of diagrams, and uses edges for representing diagram *components* like circles and arrows. Edges are linked to nodes that represent the *attachment areas* at which diagram components can be connected with each other, like the *border* and *area* of circles, or the *source* and *target* ends of arrows. The connection of attachment areas is explicitly represented by *spatial relationship edges*. The type of a connection edge reflects the types of connected attachment areas and their kind of connection. For instance, the *sources* and *targets* of arrows may *touch* the *borders* of circles, or the *area* of a text may be included in the *area* of a circle, leading to specific connection edges between the nodes of the corresponding component edges.

Rewriting groups of diagram components to single edges yields a *reduced graph* which is then *parsed* according to the syntax of the diagram language in order to build the *abstract syntax graph*. In contrast to the approach of Andries *et al.* [1], there is no 1-1-correspondence between the spatial relationship graph and the reduced graph. This considerably reduces parsing complexity and allows for processing of diagram languages which could not be efficiently processed otherwise.

Figure 2 shows two visualizations of an abstract control flow graph. Figure 3 shows how the syntax of abstract control flow graphs is specified, by *context-free* graph grammars [4].

Diagrams can be manipulated in two ways: *Free-hand editing* manipulates diagram components, and triggers re-scanning and re-parsing of the diagram;

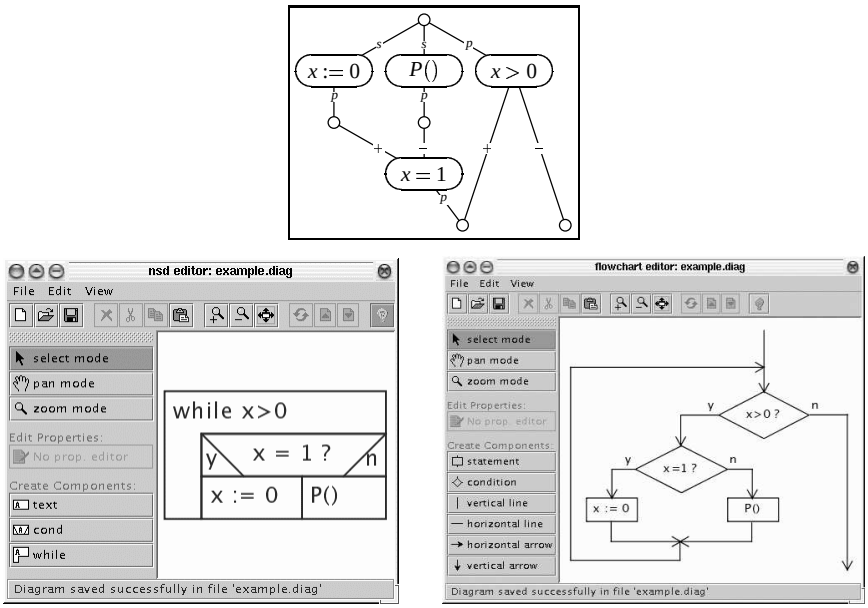

Figure 2: An abstract control flow diagram (top), visualized as a Nassi-Shneiderman diagram (bottom left), and as a control flow diagram (bottom right)

syntax-directed editing applies operations to the spatial relationship graph directly, and triggers re-parsing of the modified graph.

A wide variety of diagram notations can be captured by this model, e.g., finite automata and control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts, visual expression diagrams, sequential function charts, and ladder diagrams. Actually we are not aware of a diagram language that cannot be modeled this way.

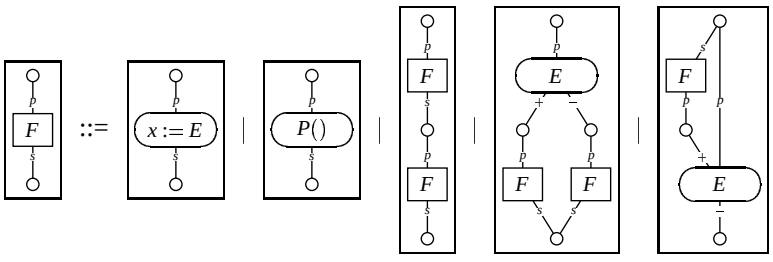

Figure 3: The syntax of abstract control flow graphs

Diagram Semantics

So far, DIAGEN handles the semantics of diagrams by translating their abstract syntax graphs "manually" into some user-defined representation that is subsequently processed by other tools. For a better integration of semantics, we plan to extend DIAGEN by a programming language that allows operations to be specified directly on the abstract syntax graphs of diagrams. The concepts of such a language have been discussed in [8], and its formal basis has been defined in [5]. Here we just state its key features, and give two small examples.

Graph Transformation Rules. Computation is based on *matching* the pattern of some rule in a host graph, *removing* the occurrence of that pattern up to those nodes where it may be clipped from the remainder of the host graph, and *gluing* the replacement of the rule at these clipping nodes. The left box in Figure 4 contains the pattern of a conditional flow graph where both branches start with equal assignments. (Clipping nodes are drawn as filled circles.) The right box in Figure 4 specifies that the assignment shall be moved in front of the condition.

Patterns may contain *graph variables*, like T in Figure 5. During matching, parts of the host graph are bound to these variables, and are used to instantiate occurrences of these variables in the replacement of the rule.

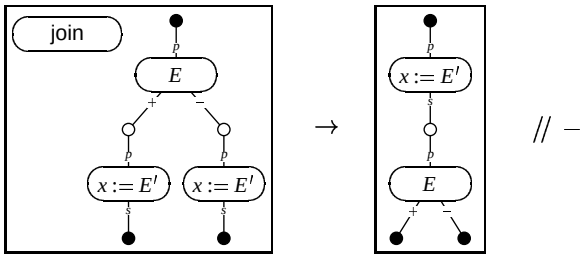

Figure 4: The predicate join

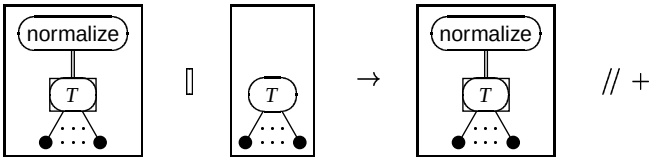

Figure 5: The predicate normalize

Graph Predicates. Sets of rules may be composed and named, providing for functional abstraction. Such compositions are called *predicates* because their evaluation may fail. Figure 4 shows a (rather primitive) predicate join, consisting of a single rule, and an otherwise definition "// -" that signals failure of the

predicate if its rule cannot be applied. The otherwise definition "// +" in Figure 5 expresses that normalize succeeds if no rule of that predicate applies.

Rules of a predicate may have applicability conditions, like the box in the center of Figure 5. Applicability conditions must evaluate successfully before the rule can be applied.

Calls to predicates are issued by inserting edges labeled with their name. The links of a predicate edge indicate its parameters. In particular, parameters may denote predicates: In Figure 5, normalize applies to a predicate denoted by the variable T, evaluates T as an applicability condition, and, if that succeeds, calls itself recursively. Thus normalize could be used to apply join to a diagram as long as possible.

Graph Nesting. The notion of graphs is extended by a concept for *nesting* graphs within graphs. We allow certain kinds of edges, called *frames*, to contain nested subgraphs. (Such graphs are called *hierarchical* in [5].) In our example, the edge labelled by the procedure call $P()$ in Figure 2 could be a frame that contains the abstract control flow graph of P.

For the sake of modularity, there may be no edges across frame boundaries (which other notions of nested graphs allow, e. g. [6]). However, every link of a frame can be associated with a node of its contents in order to express an *indirect relation* between the contents and the context of that frame.

Graph Shapes. Rules like those in Figure 3 define the shapes of graphs (and diagrams). They may be used to *type* graphs. The contents of a frame can be required to have a certain shape. For instance, the rules of Figure 3 can be used to define the shape of graphs that are contained in diagram frames. A similar way of typing is used in *Structured Gamma* [7].

Graph Objects. Every kind of frame is considered as a class, and a set of predicates is associated to it as methods. The shape of its contents, and some of its methods can be hidden. For instance, join could be one method of diagram frames (along with others). Frames are instances of a class, that is, *objects*. Their contents can only be manipulated by sending messages to them, thus adhering the principle of *data abstraction*.

The concepts sketched here support programming on a very high level, in a style reminding of functional, logical and object-oriented languages. The language is visual (based on graphs), and is generic with respect to the diagram notations used for interacting with programs developed in the language.

Towards Generic Diagram Processors

With our model of syntax and semantics, a diagram language can be implemented in two steps (see Figure 6): (1) Generate an *diagram editor* for the syntax of the language with DIAGEN. (2) Program a *diagram transformer* that defines the semantics of the language. Editor and transformer are then composed to a *diagram*

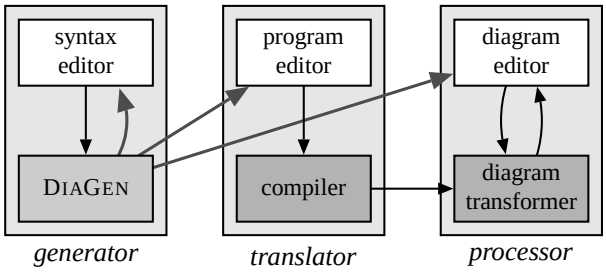

Figure 6: Modelling diagram syntax and semantics

processor that captures syntax and semantics of the diagram language. The programming tool (2) consists of a *compiler* that transforms programs into a form that can efficiently be executed by an *interpreter*. DIAGEN does not only generate the user interface of diagram processors, but also the *program editor* for the visual programming language, and the *syntax editor* of DIAGEN itself.

Note that the model is *generic* with respect to the notation used in diagram processors (and also in diagram programs). The user interfaces of the compiler and of the diagram processors can thus employ the specific notation of its application domain. This feature makes the language and its specified environment well-suited for simulation and animation. Editors for Nassi-Shneiderman diagram notation as well as flowchart notation of control flow graphs have already been built as prototypes. (Figure 2 shows two screenshots.)

We are not aware of any other tool that is generic to this extent. Graph-transformation languages like PROGRES [14], and visual programming languages like PROGRAPH [3] are more or less bound to one visual notation. Other diagram editor generators, e. g. GENGED [2] are more restricted with respect to handling the semantics of diagrams.

The implementation of compiler and interpreter for a graph- and rule-based object-oriented programming language is a challenging task. Even if we have convinced the reader that all concepts promised for the language are implementable, neither does this mean that it can be done *efficiently*, nor that this will result in *efficient systems*. However, it should be possible to reach the performance of logical and functional languages' implementations, as the aspect of user interfaces is decoupled from the kernel of the system.

References

[1] ANDRIES, M., ENGELS, G., AND REKERS, J. How to represent a visual specification. In *Visual Language Theory*, K. Marriott and B. Meyer, Eds.

Springer, New York, 1998, ch. 8, pp. 245–260.

[2] BARDOHL, R., AND TAENTZER, G. Defining visual languages by algebraic specification techniques and graph grammars. In *Proc. Workshop on Theory of Visual Languages (TVL'97)* (Capri, Italy, 1997), pp. 27–42.

[3] COX, P. T., GILES, F. R., AND PIETRZYKOWSKI, T. Prograph. In *Visual Object-Oriented Programming*, M. M. Burnett, A. Goldberg, and T. G. Lewis, Eds. Manning, Greenwich, CT, 1994, ch. 3, pp. 45–66.

[4] DREWES, F., HABEL, A., AND KREOWSKI, H.-J. Hyperedge replacement graph grammars. In Rozenberg [13], ch. 2, pp. 95–162.

[5] DREWES, F., HOFFMANN, B., AND PLUMP, D. Hierarchical graph transformation. In *Foundations of Software Science and Computation Structures (FOSSACS 2000)* (Mar. 2000), J. Tiuryn, Ed., no. 1784 in Lecture Notes in Computer Science, Springer, pp. 98–113.

[6] ENGELS, G., AND SCHÜRR, A. Encapsulated hierachical graphs, graph types, and meta types. In *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation* (`http://www.elsevier.nl/locate/entcs`, 1995), A. Corradini and U. Montanari, Eds., no. 2 in Electronic Notes in Theoretical Computer Science, Elsevier.

[7] FRADET, P., AND MÉTAYER, D. L. Structured Gamma. *Science of Computer Programming 31*, 2/3 (1998), 263–289.

[8] HOFFMANN, B. From graph transformation to rule-based programming with diagrams. In Nagl and Schürr [12].

[9] KÖTH, O., AND MINAS, M. Generating diagram editors providing freehand editing as well as syntax-directed editing. In *Proc. International Workshop on Graph Transformation (*GRATRA *2000), Berlin* (Mar. 2000).

[10] MINAS, M. Creating semantic representations of diagrams. In Nagl and Schürr [12].

[11] MINAS, M. Hypergraphs as a uniform diagram representation model. In *Theory and Application of Graph Transformation (TAGT'98), Selected Papers* (2000), H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., no. 1764 in Lecture Notes in Computer Science, Springer, pp. 281–295.

[12] NAGL, M., AND SCHÜRR, A., Eds. *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE *'99), Selected Papers* (May 2000), Lecture Notes in Computer Science, Springer.

[13] ROZENBERG, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[14] SCHÜRR, A., WINTER, A., AND ZÜNDORF, A. The PROGRES approach: Language and environment. In Rozenberg [13], ch. 13, pp. 487–550.

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

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