A Generic Model for Diagram Syntax and Semantics

BERTHOLD HOFFMANN
Universität Bremen

MARK MINAS
Universität Erlangen

In this extended abstract, we recall how the syntax of diagrams is captured by the diagram editor generator DIAGEN [9, 10, 11], 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.

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.

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.

*This work has been partially supported by the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH).
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 transformation [4].

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

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.

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 lan-
guage 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 a *diagram editor* for the syntax of the
language with DIAGEN. (2) Program a *diagram transformer* that defines the se-
manics of the language. Editor and transformer are then composed to a *diagram
processor* that captures syntax and semantics of the diagram language. The pro-
gramming tool (2) consists of a *compiler* that transforms programs into a form
that can efficiently be executed by an *interpreter*. DIAGEN does not only gen-
erate 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 ap-
lication 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


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