Rule-Based Programming with Diaplan

Frank Drewes [1]

Institutionen för datavetenskap
Umeå universitet
S-90187 Umeå, Sweden

Berthold Hoffmann [2,3] Raimund Klein [4]

Technologiezentrum Informatik
Universität Bremen
D-28334 Bremen, Germany

Mark Minas [5]

Fakultät für Informatik
Universität der Bundeswehr München
D-85577 Neubiberg, Germany

Abstract

Diaplan is a language for programming with graphs and diagrams that is currently being designed and implemented by the authors. In this paper, a programming example, declaration grids, shall illustrate how Diaplan supports a functional and object-oriented style of programming. The example also indicates which features are needed beyond those discussed in previous work on the language [9].

Key words: visual language, visual programming, rule-based programming, graph transformation

1 Introduction

Several tools and their underlying languages use graphs as data structures, and graph transformation as computation rules, e.g., Progres [13], Agg [5],

[1] Email: drewes@cs.umu.se
[2] This author is supported by the ESPRIT Working Group *Syntactic and Semantic Integration of Visual Modeling Techniques* (SeGraVis, www.segravis.org).
[3] Email: hof@tzi.de
[4] Email: ray@tzi.de
[5] Email: minas@acm.org

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Fujaba [6], and GReAT [10]. We are designing a new *diagram programming language*, **Diaplan**, where we exploit novel concepts, like graph hierarchies, graph shapes, and graph variables [3,8] for a better structuring and typing of graphs, and for more adequate notion of graph transformation. Further programming concepts, like functional abstraction, control, and encapsulation are added in a way that preserves the graph- and rule-based nature of the language [9]. Moreover, its implementation will be integrated with the DiaGen diagram editor generator [12] so that the graphs manipulated by **Diaplan** programs can be created and displayed as diagrams, in a notation customized to their application domains. In this paper, we illustrate the concepts by discussing a specification of *declaration grids* for the static semantic analysis of block-oriented programming languages. This discussion will also indicate some further concepts that are needed in the language.

The rest of the paper is structured as follows: The next section describes the kind of graphical data structures used in **Diaplan**, and how their types are specified. In Section 3, we show how computation is specified and performed. Then we discuss how programs can be encapsulated in classes, in Section 4. We conclude with comparing our work to related graph- and rule-based languages, sketching the ongoing implementation of a **Diaplan** interpreter, and indicating some work to be done.

2 Data

In **Diaplan**, every compound piece of data is represented as a graph. [6] For the informal view taken in this paper, it suffices to know that the nodes of a graph are drawn as circles, ovals, or boxes, and its edges as lines or arrows. Each node of a graph may be a *container* that contains some other graph. [7] Thus, graphs can be structured in a hierarchically nested fashion. This structuring concept distinguishes **Diaplan** from many other graph-based languages [13,5,6,10] where data is modeled as one monolithic graph. Nesting resembles the concept of hierarchal graphs; however, our hierarchies are *strict* in the sense that the graph contained in one node may not be connected to those contained in other nodes [8]. Extending earlier definitions, we now allow nodes to be *attributed* by primitive values, such as numbers and strings. This is frequently needed in applications, and fits neatly into the nesting concept: nodes may not only contain graphs, but primitive values as well.

[6] Actually, **Diaplan** uses hierarchical hypergraphs. Hypergraphs are generalizations of directed graphs. A hypergraph consists of nodes and hyperedges where each hyperedge connects some nodes. The number of connected nodes is determined by the label of the hyperedge. A directed graph is a hypergraph where each hyperedge connects exactly two nodes. **Diaplan**'s concept of hierarchical hypergraphs is briefly outlined in the following. For simplicity, hierarchical hypergraphs are called graphs, and hyperedges are called edges in this paper.

[7] Actually, hyperedges may be containers, too. However, this feature is not required in this paper.

Example 2.1 [Declaration Grids] Our running example is on *declaration grids*, an abstract data type that is used for efficient static semantic analysis of block-structured programming languages [14, Section 9.1.2]. In such languages, declarations may be nested so that local definitions hide global definitions of the same identifier. For the sake of simplicity, we assume first that overloading is forbidden. Thus, every identifier shall have at most one visible declaration. Figure 1 shows how declaration grids can be represented as a graph in an appropriate way.

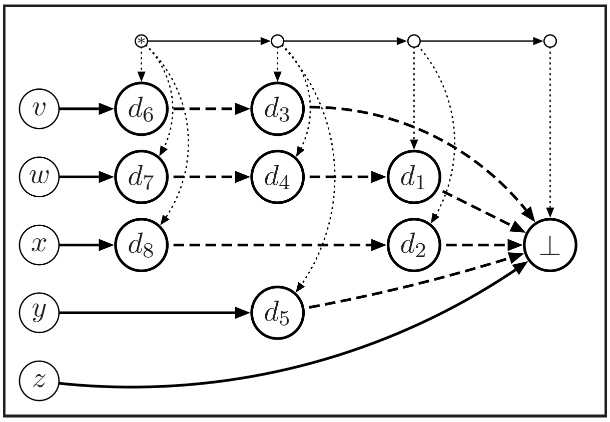

Fig. 1. A declaration grid. Its rows and columns represent the identifiers, and the blocks of a program, respectively. The cells of the grid contain declarations. These values are represented as graphs $d_1, \cdots, d_8$ whose structure depends on the actual programming language to be implemented. In addition, we assume that there is a distinguished declaration $\perp$. Identifiers are represented by nodes containing their *keys* $\{v, \ldots, z\}$ (which are typically mutually distinct integers). Every row represents the stack of declarations associated to one identifier; the top declaration (target of the solid arrow) is visible whereas the others (which are targets of dashed arrows) are hidden. The top chain of small circles represents the *stack of blocks*. Every column represents the *bag* of declarations local to a block, by a bundle of dotted edges. We assume that all identifiers have a declaration $\perp$ in the fictitious outermost block of the table. $\diamond$

Graphs are typed: Nodes and edges may be classified by labeling them with symbols or text, but also by their form, color, or other layout properties, and their edges may be restricted with respect to the types of their source and target nodes. In a declaration grid, for instance, dotted arrows may only connect block nodes (small circles) with declaration nodes (big, fat-lined circles). Unlike PROGRES [13], Dia**plan** has no multiplicity constraints specifying, e.g., that exactly one dotted edge enters a declaration node. For, in general, such constraints require dynamic type checks. Instead, the *shape* of graphs can be specified by recursive rules using *hyperedge replacement* [1]. However, extending the usual context-free rules these rules may exploit context as defined in [2], which allows for a finer typing than in most other languages. In Figure 2, we define the shape of declaration grids in this way. Neither functional,

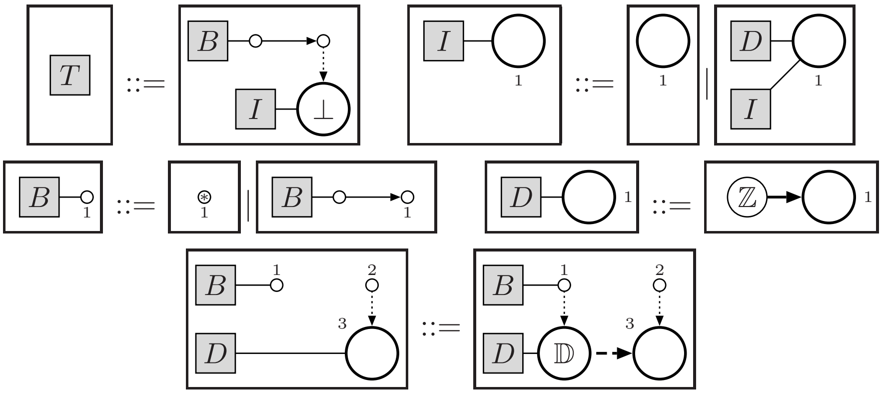

Fig. 2. The shape definition of declaration grids. The rule for T initializes the outermost block; the I rules define declaration stacks D for identifiers, starting with their outermost declaration $\perp$. The B rules define the stack of block nodes. The first rule for D generates an identifier with its visible declaration. All these rules replace a nonterminal by a graph in a context-free way. In the last rule, the nonterminal D is only replaced in the context of two distinct block nodes numbered 1 and 2. (When applying shape rules, we must find *injective* occurrences of their left hand sides in the host graph.) The labels $\mathbb{Z}$ and $\mathbb{D}$ in D rules indicate that the corresponding nodes contain numbers and declaration graphs, respectively. While $\mathbb{Z}$ is a predefined primitive type, $\mathbb{D}$ is assumed to be defined by means of another, independent set of shape rules. ◇

nor imperative languages allow for specifying the type of specific graph languages. Hence, the type of declaration grids could not be defined as precisely as in Figure 2 in such programming languages.

3 Computation

In Dia**plan**, computations are performed by *graph transformation* [8]. Programs are defined by graph transformation rules. Within rules, two kinds of graph elements play particular roles: *Variables* (labeled by uppercase letters) are edges or variables acting as placeholders for graphs or primitive values, and *predicate* edges (labeled by names in lower case) refer to sets of graph transformation rules defining them. A rule is applied by finding, in some container of a graph, the constant subgraph (i.e., the maximum subgraph without any variables) of a rule's left hand side, binding its variables to appropriate subgraphs, and replacing the match by the rule's right hand side, where variables are replaced by their bindings. [8] The evaluation strategy is as follows: A

[8] Binding of a variable to a subgraph and replacing a variable by its binding is described in [2,9] and works essentially as follows: A graph as a binding of a variable edge has some special nodes, called *points* that are arranged in a sequence. The number of points has to be equal to the number of nodes that are connected by the bound variable as a hyperedge. A variable edge is replaced by its binding by removing the variable edge and pasting in its binding where the binding's points are glued with the nodes that have been visited by the variable edge. A variable node can be bound to any node with appropriate type and

start graph containing predicates (but usually free of variables) is transformed by applying the rules for the predicates as long as possible. In every step, a rule is applied to one of the most deeply nested predicates that have been inserted most recently. Execution fails if none of the rules of a predicate match; backtracking may then explore further rules until all predicates have been completely evaluated, leaving a graph (without predicates and variables) as a result. This corresponds to *innermost evaluation* known from functional languages, and to *depth-first search* for results, as known from logic languages.[9] (See [9] for details.)

Predicates offer a means for abstraction and for control. With respect to abstraction, predicates are similar to units in GReAT [10], productions and transactions in PROGRES [13], and methods in FUJABA [6]. Nodes being visited by a predicate hyperedge correspond to ports in GReAT, and parameters in PROGRES and FUJABA. However, while existing languages based on graph transformation (e.g., GReAT, PROGRES, or FUJABA) use an imperative style for specifying flow of control, **Diaplan** uses a declarative style similar to logic or functional programming. That style reduces the need for syntactic sugar in order to specify sequences, alternatives, or procedural abstraction. Actually, using predicates together with application guards (see [9]) is sufficient.

Example 3.1 [Operations on Declaration Grids] In Figures 3 to 7, we define five operations on declaration grids as predicates: init constructs the empty grid for some set of identifiers; open pushes a new block; close pops a block with its local declarations; enter inserts a declaration for an identifier into the grid; and, lookup returns the visible declaration of an identifier.

Each predicate is represented by a sequence of rules L/R where L is the left hand side (lhs) which is replaced by the right hand side (rhs) R. Edges are represented by arrows or by rectangles. Nodes are represented by circles or rectangles with round corners. Small numbers indicate the correspondence of lhs and rhs nodes. Edges being labeled with lowercase names represent predicate edges. Each lhs of a rule belonging to a predicate contains a corresponding predicate edge in its lhs. No other predicate edge is permitted. Uppercase names represent variable edges or variable nodes. A predicate definition my be followed by $//e$ which specifies the behavior if this predicate cannot be resolved by any of its rule. $+$ in a box means *success*, i.e., the corresponding predicate edge is simply deleted. The box labeled *over* throws an exception. More details can be found in [9].

Close inspection shows that the predicates in Figures 3 to 7 preserve the shape of declaration grids specified in Figure 2. However, **Diaplan** is intended to be a statically typed language. Keeping this in mind, the example indicates

contents, i.e., a primitive value or contained graph.

9 Other choices would have been *lazy evaluation* and *breadth-first search*. They have not been chosen since lazy evaluation makes program debugging a difficult task, and breadth-first search is in general less efficient than depth-first search.

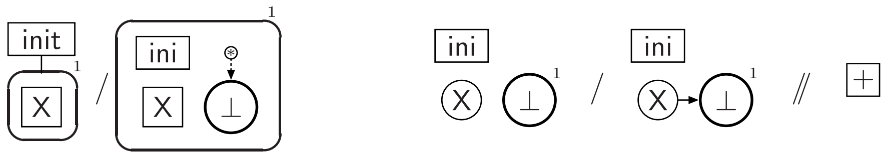

Fig. 3. The predicate init takes a discrete graph of identifier nodes (bound to the variable X), and adds a ficticious outermost block with one local declaration $\perp$ (undefined). It calls the recursive ini predicate that works inside the container and "declares" every identifier as $\perp$. Applying this rule has to preserve the $\perp$ node which is achieved by declaring this node as a point (see [9]). The predicate terminates successfully ("+") if no identifier is isolated in the grid any more. $\diamond$

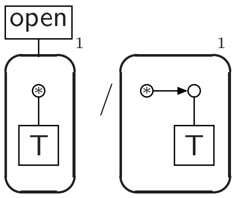

Fig. 4. The predicate *open* pushes a new block node onto the stack of blocks. $\diamond$

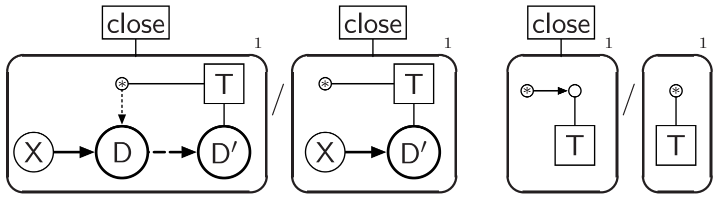

Fig. 5. The predicate close removes the innermost block with all declarations. The left rule recursively removes one local declaration from the innermost block; the second rule pops the innermost node from the block stack. $\diamond$

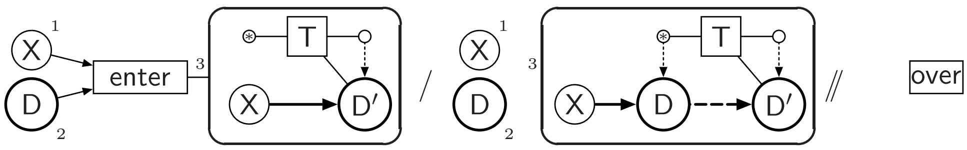

Fig. 6. The predicate enter enters a declaration D for an identifier X to a grid if the previous declaration D' of X is not local to the current block; otherwise, an exception over (for overloading) is raised. $\diamond$

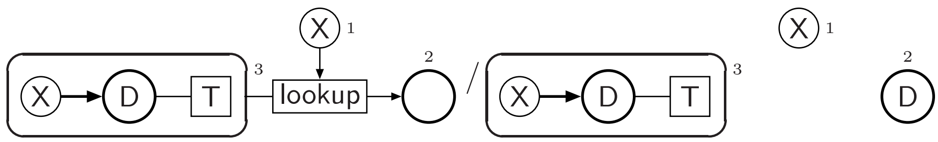

Fig. 7. The predicate lookup retrieves the declaration for X that is actually visible. The predicate init makes sure that every identifier has the declaration $\perp$ at least. $\diamond$

that future work must focus on conditions that make type checking possible. The problem is that shape preservation by the rules in Figures 3–7 cannot be statically checked in the way described in [2] because the transformation rules are not "shapely" in the sense of that paper. On the other hand, the rules are quite obviously compatible with the shape definition rules given in Figure 2, from an intuitive point of view. It should therefore be possible to extend the definition of shapely rules in a way that covers the rules above, or develop additional techniques that complement this notion. However, this is beyond the scope of the present paper and must therefore remain a topic for future research. It seems therefore unavoidable to develop additional techniques that complement each other.

The reader may have noticed that the predicate definitions of our example extend the concepts presented in [9] in that they specify the *mode* of each parameter node in an intuitive way. Ingoing and outgoing arrows indicate *in*-, and *out*-parameter, respectively.[10] Lines without arrow heads indicate *in-out*-parameters. In our example, the table grid is an in-out-parameter to all predicates, identifier nodes are in-parameters to enter and lookup, whereas the declaration node is an in-parameter to enter, and an out-parameter to lookup.

More interesting extensions of concepts described in previous papers are suggested if the example is generalized in order to allow for overloading. Figure 8 shows an example of a graph representing such a declaration grid. The shape definition rules given in Figure 2 can be adapted to this more general case in a straightforward way. The generalization of the required transformation rules (implementing open, close, enter, and lookup) is not difficult either, but gives nevertheless rise to interesting observations.

(i) In order to handle overloaded declarations in an appropriate way, enter must check whether the new declaration hides a currently visible one and, if not, whether overloading is possible. This can be done by *conditional rules* whose applicability depends on the outcome of boolean predicates hides and overloadable. Thus, enter either enters the new declaration in front of the one it hides or as an additional one besides the already existing ones, or results in an exception.

(ii) Since there can now be an arbitrary number of visible declarations for a

[10] The concept of *in*- and *out*-parameters is similar to that of Progres [13] and the use of input and output interfaces of GReAT [10].

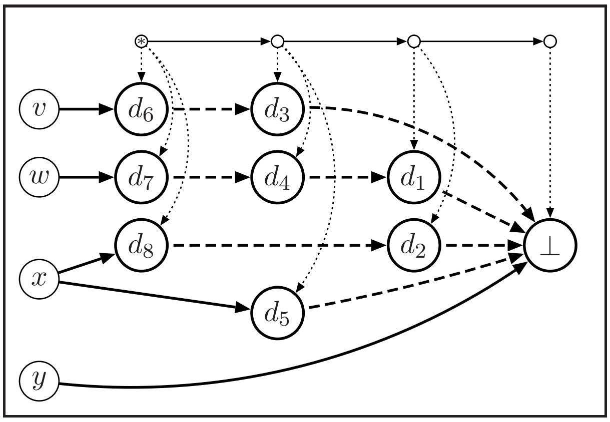

Fig. 8. A declaration grid with overloading. The shape is similar to the one in Figure 1 except that a given identifier may have any number of visible declarations which do not hide each other. In the declaration grid shown, x has visible declarations d_5 and d_8. $\diamond$

given identifier, lookup is ambiguous, i.e., it may succeed in several ways. In fact, if we modify the representation of declaration grids slightly by removing the $\perp$-labeled node, lookup may even fail, namely if there is no visible declaration for the given identifier. These observations suggest to classify predicates by *multiplicities* that indicate how often evaluation may succeed:

- If a predicate is deterministic, it has at most one evaluation. However, in general a deterministic predicate may fail, in which case it does not yield any result.
- Nondeterministic predicates may have any number of successful evaluations, but some of them are guaranteed to have at least one. Nondeterministic predicates require backtracking.

We therefore suggest to annotate predicates by "?" (deterministic, but may possibly fail), "+" (any positive number of successful evaluations), or "*" (any number of successful evaluations).[11] Predicates that do not carry any of these annotations are required to be deterministic, must not fail, and always have to yield exactly one result. Note that this is true for the rules in Figures 3–7.

Modes and multiplicities also allow for a characterization of the kinds of predicates that are used in functional and object-oriented programming, respectively:

- *Functions* have only *in*- and *out*-parameters; they do not update a parameter. One may further distinguish between partial and total functions. Total functions must not fail, but they may still raise exceptions. For example, division may be implemented as a partial function (division by 0 fails) or as a total one (division by 0 raises an exception).
- *Methods* have one *in-out*-parameter (the receiver object), and may have additional *in*-parameters (sometimes called *modifiers*), and they may have an out-parameter.

In our example, all operations are defined like methods, even if lookup could also be defined as a function. As a matter of fact, every other method could be turned into a function that has one declaration grid as an in-parameter, and yields a modified declaration grid as an out-parameter.

4 Encapsulation

Every modern programming language should provide means to group data structures and operations on that data into meaningful modules that allow to hide implementation details.

It may come as no surprise that we propose programs to be represented

[11] These multiplicites are similar to the *determinism categories* of Mercury [7] and *production* as well as *transaction qualifiers* in Progres [13].

by graphs as well. The idea is as follows. Nodes represent the *classes* and the predicates of a program. Connections between nodes represent parameters and their modes. In the *interface graph* of a program, the nodes just show the names of classes and predicates because this is the information needed to use them. In the *implementation graph* of the same program, class nodes contain their shape rules and local predicates, and predicate nodes contain their rules. Figure 9 shows the interface graph of our example.

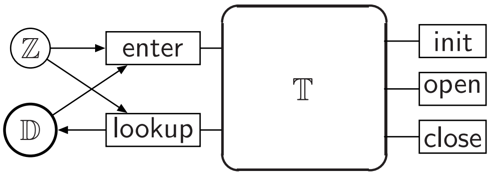

Fig. 9. The interface graph of the class $\mathbb{T}$ of declaration grids. The predicates (methods) enter and lookup of $\mathbb{T}$ refer to the predefined class $\mathbb{Z}$ of *numbers* and the class $\mathbb{D}$ defining *declarations*. The exported predicates of $\mathbb{Z}$ and $\mathbb{D}$ are omitted since they are not used in the example. In the implementation graph, $\mathbb{T}$ contains its local predicate ini and its shape rules (defined in Figure 2), and the exported predicates init, open, close, enter, and lookup contain the rules shown in Figures 3 to 7. ◇

Primitive values fit into this setting in an orthogonal way. The numbers used in our example are considered to be a predefined class $\mathbb{Z}$ with predefined operations. Other values, like characters, real numbers and strings are also considered to be predefined. Values of these classes can be used as textual literals (e.g., "42" "'Hello World!"') in values, and their operations ("+", "append") can be used in textual expressions that determine the attribute values of nodes in graph transformation rules. However, we do not compute with attribute values in our example, except for the fact that the application of the rules for enter and lookup implicitly require an equality check of the two identifiers named X.

Note that the definition of the declaration grid is *generic* with respect to the declaration information $\mathbb{D}$ that is associated with identifiers. So the class $\mathbb{T}$ should be defined as a generic class with a type parameter that can then be instantiated by $\mathbb{D}$ or by some other kind of declaration information.

5 Conclusions

In this paper we have sketched some further aspects of Dia**plan**, a graph- and rule-based language for *programming with diagrams*, which is based on shaped hierarchical graph transformation [3,8]. Our running programming example, block-oriented declaration grids, has motivated several refinements of the concepts proposed in [9], in particular a classification of predicates and their parameters, and an encapsulation concept. The resulting language supports the functional, logic and object-oriented paradigms of programming.

In comparison to the graph- and rule-based languages PROGRES [13],

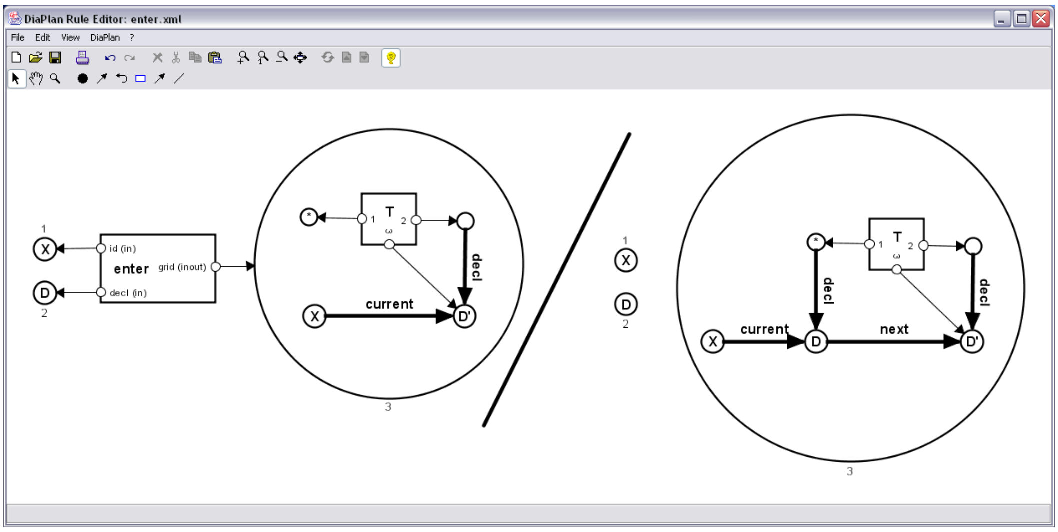

Fig. 10. Snapshot of a prototype of the **Diaplan** rule editor. The snapshot shows the rule of predicate enter (cf. Fig. 6). Please note that the editor uses names instead of dashed or dotted lines to distinguish different kinds of binary edges. Moreover, box attachments carry unique names which allows for an unambiguous interpretation of hyperedges and their attached nodes. ◇

Agg [5], and FuJaba [6] mentioned in the introduction, we note that all of them have an attribute concept for primitive values (or even for all types supported by their host languages), but none of them provides a nesting concept. Computations in those languages are thus transformations of large monolithic graphs. Progres has a rich type structure with constraints and inheritance, but no language allows the recursive definition of graph languages as in **Diaplan**. To our knowledge, Agg has no concept for functional abstraction at all; the procedures and transactions in Progres are specified in a textual language with logical and imperative concepts, and FuJaba uses Uml diagrams to specify methods. Finally, the encapsulation concepts in those languages are entirely taken from their host languages.

A prototype interpreter for **Diaplan** [11] is being implemented. It is based on the DiaGen transformation engine [12], does not consider shapes, and reads graphs and programs in textual form until now. The predicates of our running examples have been specified in that notation, and have been executed with the interpreter. Integration of the rule editor shown in Figure 10 is under way. The next step will then be to integrate the prototype with DiaGen, in order to provide customized editors for creating and displaying data of a **Diaplan** program. Finally, the prototype interpreter shall be replaced by a compiler, and an optimized **Diaplan** abstract machine.

References

[1] Drewes, F., A. Habel and H.-J. Kreowski, *Hyperedge replacement graph grammars*, in: G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, World Scientific, Singapore, 1997 pp. 95–162.

[2] Drewes, F., B. Hoffmann and M. Minas, *Context-exploiting shapes for diagram transformation*, Machine Graphics and Vision **12** (2003), pp. 117–132.

[3] Drewes, F., B. Hoffmann and D. Plump, *Hierarchical graph transformation*, Journal of Computer and System Sciences **64** (2002), pp. 249–283.

[4] Engels, G., H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, "Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages, and Tools," World Scientific, Singapore, 1999.

[5] Ermel, C., M. Rudolf and G. Taentzer, *The AGG approach: Language and environment*, in: Engels et al. [4] pp. 551–603.

[6] Fischer, T., J. Niere, L. Turunski and A. Zündorf, *Story diagrams: A new graph grammar language based on the Unified Modelling Language and Java*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in Lecture Notes in Computer Science (2000), pp. 296–309.

[7] Henderson, F., T. Conway, Z. Somogyi and D. Jeffery, *The Mercury language reference manual*, Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia (1996).

[8] Hoffmann, B., *Shapely hierarchical graph transformation*, in: *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments* (2001), pp. 30–37.

[9] Hoffmann, B., *Abstraction and control for shapely nested graph transformation*, Fundamenta Informaticae **58** (2003), pp. 39–56.

[10] Karsai, G., A. Agrawal, F. Shi and J. Sprinkle, *On the use of graph transformation in the formal specification of model interpreters*, Journal of Universal Computer Science **9** (2003), pp. 1296–1321.

[11] Klein, R., "An Interpreter for Dia**plan**," Diplomarbeit, Universität Bremen (2004, in preparation), in German.

[12] Minas, M., *Concepts and realization of a diagram editor generator based on hypergraph transformation*, Science of Computer Programming **44** (2002), pp. 157–180.

[13] Schürr, A., A. Winter and A. Zündorf, *The PROGRES approach: Language and environment*, in: Engels et al. [4] pp. 487–550.

[14] Wilhelm, R. and D. Maurer, "Compiler Design," Addison-Wesley, 1995.