Towards Rule-Based Visual Programming of Generic Visual Systems *

Berthold Hoffmann[1] and Mark Minas[2]

[1] Fachbereich Mathematik/Informatik
Universität Bremen
Postfach 33 04 40, 28334 Bremen, Germany
hof@informatik.uni-bremen.de
[2] Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
minas@informatik.uni-erlangen.de

Abstract. This paper illustrates how the *diagram programming language* DIAPLAN can be used to program visual systems. DIAPLAN is a visual rule-based language that is founded on the computational model of graph transformation. The language supports *object-oriented programming* since its graphs are hierarchically structured. *Typing* allows the shape of these graphs to be specified recursively in order to increase program security. Thanks to its *genericity*, DIAPLAN allows to implement systems that represent and manipulate data in arbitrary diagram notations. The environment for the language exploits the diagram editor generator DIAGEN for providing genericity, and for implementing its user interface and type checker.

1 Introduction

Many data structures can be represented by graphs, and the theory of graph transformation [22] provides a profound computational model for rule-based programming with graphs. As graphs are inherently visual, and have been used as an abstract model for visual representations [2, 9, 14], graph transformation could become widely accepted as a paradigm of *rule-based visual programming*.

However, existing programming languages based on graph transformation, such as PROGRES [23] or LUDWIG$_2$ [18], 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

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

in the visual programming community [3], notably *Prograph* [5]. Furthermore, there are visual language tools which allow to use domain-specific visual representations, e.g., CALYPSO, a tool for visually defining data structures of programs [28]. 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.

The paper is organized as follows: Section 2 describes graph transformation, the computational model of the language. Concepts for programming are illustrated in Section 3, and typing is discussed in Section 4. A complete example of a DIAPLAN program for solving graph coloring problems is discussed in Section 5. Section 6 recalls how graphs can be visualized in arbitrary diagram notations. Genericity of the language is based on this feature. In Section 7 we describe how the language can be implemented. We conclude in Section 8 with some remarks on related and future work.

Due to space limitations, our presentation can only be informal. We explain the concepts by a running example concerned with the visual representation of lists and the implementation of list operations.

2 The Computational Model

We introduce a rather general notion of graphs, and a rather simple way of applying rules to them.

2.1 Graphs

Graphs represent relations between entities as *edges* between *nodes*. Usually, the edges link two nodes of a graph, and represent binary relations. We, however, allow edges that link *any number* of nodes, and distinguish different *types* of edges by labeling them

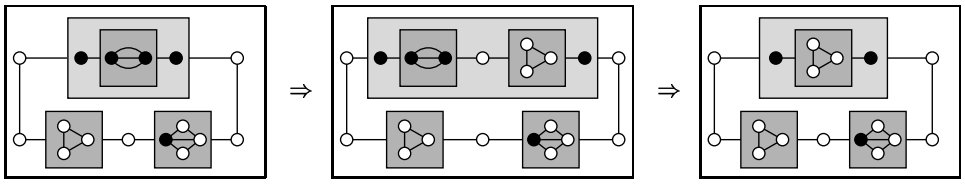

Fig. 1. Two transformation steps on list graphs

so that different relations, of any arity, can be represented in a single graph. We also distinguish a sequence of nodes as the *points* at which a graph may be connected to other graphs. Such graphs are known as *pointed hypergraphs* [6].

Usually, graphs are *flat*: Their nodes and edges are primitive; none of them may contain a nested graph. We, however, distinguish a subset of the edges in a graph H as *frames*: Every frame f *contains* a nested subgraph H_f that may contain frames again. These graphs are called *hierarchical* in [7].

Example 1 (List Graphs) The three graphs in Figure 1 show how we represent lists as graphs: A *list* frame (light gray boxes) is linked to the start and end node of a chain of item frames (dark gray boxes); every *item* frame contains an *item graph*.

Nodes are drawn as circles, and filled if they are points. Edges are drawn as boxes around their label, and are connected to their attachments by lines that are ordered counter-clockwise, starting at noon. The boxes for binary edges with empty labels "disappear" so that they are drawn as lines from their first to their second linked node. Frames are boxes (like ordinary edges), with their contents drawn inside; we distinguish list and item frames by different shades of grey, and omit their labels. The graphs in Figure 1 contain two item frames, and a list frame that contains one or two item frames. □

This representation *forbids* edges across frame boundaries (which other notions of hierarchical graphs [21, 8] allow). Only then graphs can be transformed in a modular way. However, the correspondence between the links of a frame and the points of its contents may induce an *indirect relation* between the contents and the context of that frame. The links of the list frames in Figure 1 are related to the points of their contents in that way.

2.2 Computation

In the graphs occurring in computation rules, a distinguished set X of *variable names* is used to label *variable edges* (*variables* for short). A *substitution* σ maps variables names $X_1, \ldots, X_n$ onto graphs $G_1, \ldots, G_n$.

The *instantiation* $G\sigma$ of a graph G by some substitution σ is obtained by identifiying the points of a fresh copy of $\sigma(X_i)$ with the corresponding attachments of every X_i-edge in G, and removing the edge afterwards.

A graph $C[\,]$ is called a *context* if it contains a single variable (a *hole*). We write $C[G]$ for the instantiation of the hole in $C[\,]$ by some graph G.

A *graph transformation rule* (*rule*, for short) $t : P \to R$ consists of a *pattern* P and a *replacement* R, where P and R are graphs such that the following holds:

- Every variable name occurring in R occurs in P as well.
- Every variable name occurs at most once in P.

A rule $t : P \to R$ *transforms* some host graph G into a modified graph H, written $G \Rightarrow_t H$, if there is a context $C[\,]$ and a substitution σ such that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$. [1]

[1] We write $G \cong H$ if two graphs G and H are isomorphic, i.e. equal up to the identity of their nodes and edges.

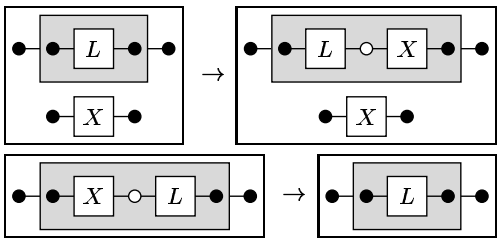

Fig. 2. Two rules specifying operations on list graphs

A transformation step can be constructed by *graph matching* (defined in [19] for flat graphs). The matching algorithm can be lifted to the hierarchical case along the lines of [7].

Example 2 (List Graph Transformation) The rules in Figure 2 specify two operations on list graphs, which are used in the transformation steps shown in Figure 1. (Variable names appear in italics.)

The first step (using the rule on the left) *enters* a copy of an item frame X at the end of a list graph L, and the second step (using the rule on the right) *removes* the first item frame from a list graph. □

Graph transformation with a set $\mathcal{T}$ of graph transformation rules considers sequences of sequential transformation steps in arbitrary order, and of arbitrary length.

By taking arbitrary graphs as input, and transforming them as long as possible, graph transformation computes a *function* on graphs. This function is *partial* if certain graphs can be transformed infinitely, and *nondeterministic* if a graph may be transformed in different ways.

3 Programming

The computational model presented in Section 2 is extended by concepts for abstraction, control, and encapsulation. This section gives only a brief account of these programming concepts. See [11] for more motivation and details.

3.1 Abstraction

We consider certain labels as *predicate names*. An edge labeled by a predicate name is depicted as an oval. 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. A predicate is *evaluated* by first applying one of its rules, and evaluating all predicates that are called in its replacement, recursively.

The links of a predicate edge indicate the *parameters* of a predicate. A parameter can be just a *node*, but also an *edge*. In particular, this edge can be a frame that contains

a *graph parameter* (as in Example 3 below), or a predicate edge that denotes a *predicate parameter* (as in Example 4 below).

The rule set of a predicate may contain an *otherwise* definition (starting with a "$/\!/$" symbol) that applies when no rule of the predicate is applicable. In Example 3, a "–" in the otherwise definition signals *failure* of the predicate, and triggers backtracking, and in Example 4, a "+" signals *success* of the predicate, so that evaluation continues. (A "$\perp$" can be used to raise an *exception* that is either caught elsewhere, or leads to program abortion).

In any case, predicate edges are always removed during the transformation because they are meta edges that are just introduced to control the program's evaluation, and are not meant to occur in its result.

Example 3 (A Predicate) Figure 3 shows the definition of a predicate remove. The

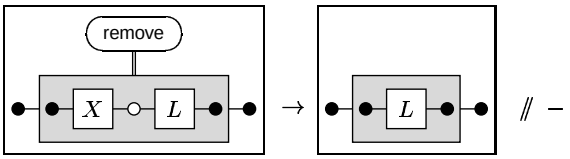

Fig. 3. The predicate remove

predicate is parameterized by a frame (which should contain a list graph). It is defined by a single rule (which appeared on the right in Figure 2). Its otherwise definition "$/\!/ -$" leads to failure if it is applied to an empty list graph.

Note that the predicate *updates* a list graph, as in imperative or object-oriented programming. However, it would be possible to define a "functional" version of remove that constructs a new list frame as a result, and leaves the input frame unchanged. Such a predicate would need more space as subgraphs have to be copied (as in functional languages). □

3.2 Control

Program evaluation is nondeterministic since predicates can be applied in arbitrary order, also concurrently. We introduce *conditional rules* by which an evaluation order for predicates can be specified. Such a rule has the form $t : P \parallel A \to R$ where the graph A is an *application condition* (or *premise*). It is applied to a graph G as follows: If $G \cong C[P\sigma]$, the graph $C = G[(A \oplus R)\sigma]$ is constructed, where $A \oplus R$ is the union of A and R that identifies only their corresponding points. If all predicate edges of $A\sigma$ in C can be evaluated, yielding a graph H, t is applicable, and its application yields H; otherwise, the rule is not applicable, and G is left unchanged. (Note that the evaluation of $A\sigma$ may thus modify the host graph. This effect is used in Example 4.)

Predicates provide some simple control mechanisms: Pattern matching and otherwise definitions allow for case distinction; applicability conditions specify an application order for predicates.

We also allow that predicates are parameterized by predicates. This allows control to be specified by *combinators* like in functional languages.

Example 4 (Control Combinators) Figure 4 shows a predicate normalize that applies to a predicate denoted by the variable T, evaluates T as an application condition, and, if that succeeds, calls itself recursively.

As T shall bind to predicate calls with any number of parameters, we use the dot notation to indicate that T links to a varying number of nodes. Where the T-edge is used as a predicate parameter, it is *disguised* as an ordinary edge by drawing a box around the oval. This prevents it from evaluation as long as it is "carried around" (in the pattern and replacement graph of the rule).

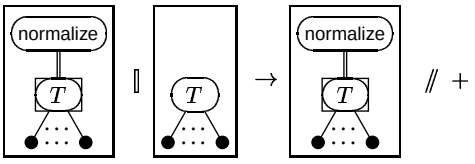

Fig. 4. The control combinator normalize

In Figure 5, normalize is applied to a disguised call of remove. Every application of normalize removes one item frame by evaluating remove as an application condition, until the list frame contains no item frame, and remove fails. (The empty list graph is represented by a single node; the numbers 1 and 2 attached to it shall indicate that this node is the first, as well as the second point of the graph.) This control combinator uses the side effect of evaluating the premise reminding of the way how the *cut* operator is used in PROLOG.

Figure 6 shows two other control combinators: Seq specifies two predicates which have to be evaluated sequentially whereas not actually does not modify anything; not fails if and only if its argument can be evaluated, i.e., not specifies an application condition which must not be satisfied. Please note that the right-hand side of its rule is "–". i.e., failure, whereas its otherwise definition specifies "++", i.e., success. For an application of these control combinators see Section 5.

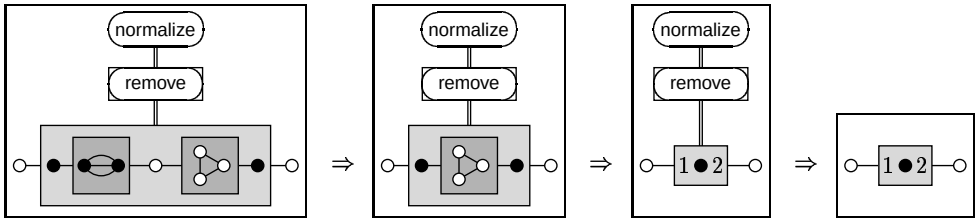

Fig. 5. An evaluation of normalize

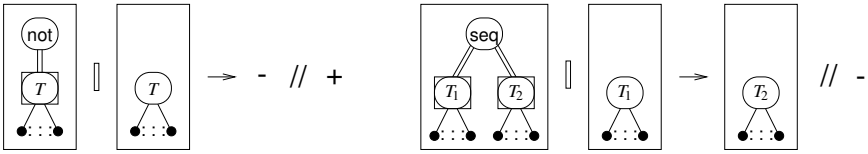

Fig. 6. The control combinators not and seq

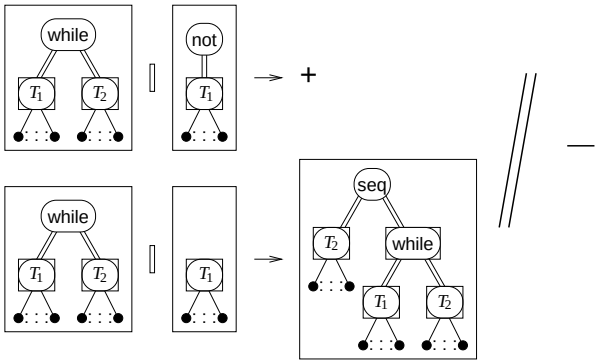

Fig. 7. The control combinator while

Finally, Figure 7 shows the two rules that define the control combinator while: It has two parameters; the second one is evaluated as long as the first one can be evaluated. The while succeeds as soon as T_1 cannot be applied (any longer). The *otherwise* definition triggers backtracking if both rules fail, i.e., T_1 can be evaluated, but the evaluation of the replacement graph of the lower rule fails. For an application of not, seq, and while see Section 5. □

3.3 Encapsulation

Programming–in–the–large relies on the encapsulation of features in modules so that only some of them are visible to the public, and the others are protected from illegal manipulation.

We consider frames as *objects* that respond to certain *messages* (which are predicate calls). The types of frames are *class names*, and a *class definition* declares predicates as its *methods*. Only the class name, and some designated methods are *public*. The graphs contained in frames, and their other methods, are *private*. This adheres to the principle of *data abstraction*.

Example 5 (The List Class) In Figure 8 we encapsulate primitive operations on list graphs within a class.

In this small example, all methods are public. However, the structure of list graphs is visible only inside for the methods remove and enter that belong to the class. Other

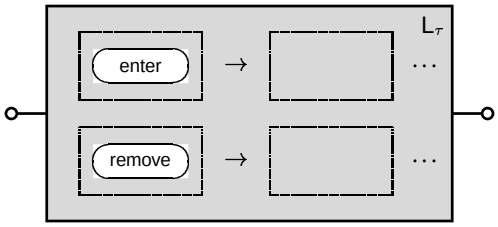

Fig. 8. The list class

objects that contain some list frame l can access its contents only by sending a message enter or remove to l. □

4 Typing

Types classify the objects of a program, and establish rules for using them. If these rules are checked, preferably *statically*, before executing the program, this ensures that a program is consistent, and may also speed up its execution.

4.1 Graph Shapes

We allow the *shape* of graphs to be specified, similar as in *Structured Gamma* [10], by *shape definitions* of the form

$$G_T ::= G_1 \mid G_2 \mid \ldots \mid G_n$$

where the graph G_T consists of an edge labeled with a *type name* T and its linked nodes, and the G_i are graphs that may contain type edges again. Every variable name, frame type, and rule has a specified type, and it is checked whether the substitutions of variables, the contents of frames, and the patterns and replacements of rules *conform* to these types. Then all graphs in a program and the diagrams which are used as visual representations are of a well-defined shape.

Example 6 (Typing Shape of List and Item Graphs) The shape definition in Figure 9 specifies list and item graphs. These graphs may be contained in list and item frames, respectively. The type L_τ of list graphs is *polymorphic*. The type parameter τ can be instantiated with any shape. In our examples, the variable L binds list graphs of type L_τ, and the variable X binds graphs of the any type τ, and is instantiated by the type I of item graphs in the transformations of Example 2 and 4. □

The rules used for shape definitions are a well-studied special case of *context-free* graph transformation [6]. Type checking thus amounts to *context-free graph parsing*, as it is implemented in DIAGEN (see Section 6).

Note that even if graph parsing may be expensive, it is done *statically*, before executing the program, and will also reduce the search space for graph matching at runtime.

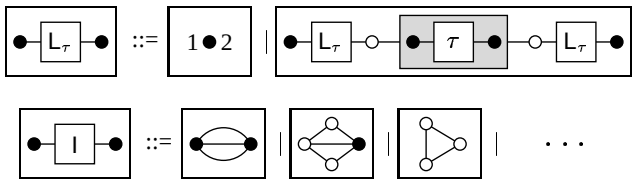

Fig. 9. The shape of list and item graphs

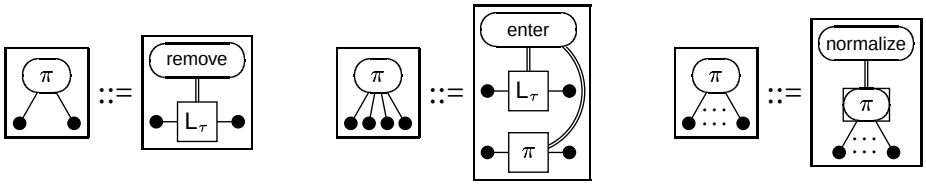

Fig. 10. The signature of list predicates

4.2 Predicate Signatures

The *signature* of a predicate shall specify to which kind of parameters it applies. As predicates are represented as graphs, their signature can be specified by shape definitions for a designated type π of predicates.

Example 7 (Signature of List Predicates) In Figure 10, we specify signatures for the predicates used in our examples. The predicate type π has a varying number of parameter nodes so that the rules of the shape definition have different left hand sides.

All predicate calls occurring in the examples of this paper can be derived with these rules (together with those of Figure 9). The predicate variable T used in example 4 is of type π. □

Note how typing increases the security of programs: Since the shape of all graphs in a program can be checked, every call of a predicate (like remove or normalize) can be type-checked at compile time. Since the input to the program, e.g. the graphs in Figure 5, can also be type-checked, a predicate will always be applied to graphs of its parameter type, and evaluation of the program may not go wrong with this respect.

5 Example: Graph Coloring

This section shows that DIAPLAN is a well-suited rule-based language for inherently graphical problems. The following DIAPLAN program searches for a solution of the graph coloring problem for an arbitrary graph which is passed as an argument to the predicate coloring (cf. Fig. 11). As specified by the while-combinator, the program alternately evaluates predicate addColor and not invalid. The program terminates as soon as addColor cannot be applied any longer (success) or when no coloring exists (failure).

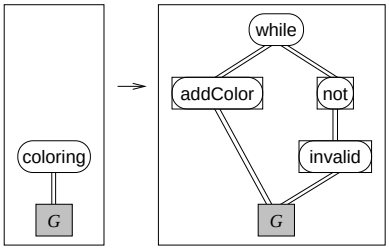

Fig. 11. The predicate coloring

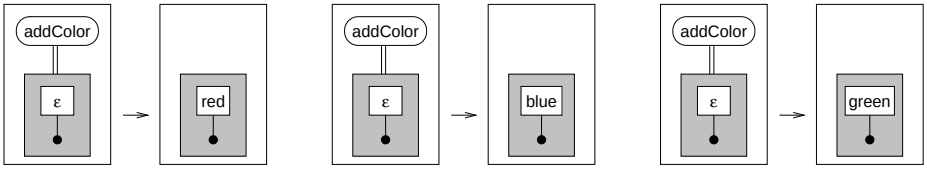

Fig. 12. The predicate addColor

Predicate addColor (Fig. 12) simply adds a color edge, i.e., *red*, *blue*, or *green* to some previously non-colored node of the graph. Non-colored nodes are indicated by ε-edges. After assigning a color to a node, predicate invalid (Fig. 13) checks whether this

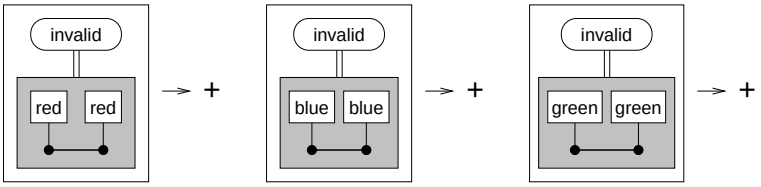

Fig. 13. The predicate invalid

action was valid: The not invalid predicate fails and triggers backtracking if previous addColor evaluation cannot yield a consistent graph coloring.

Fig. 14 shows a sample transformation sequence which terminates with a consistent coloring of a simple graph that consists of four edges and four nodes which are initially non-colored.

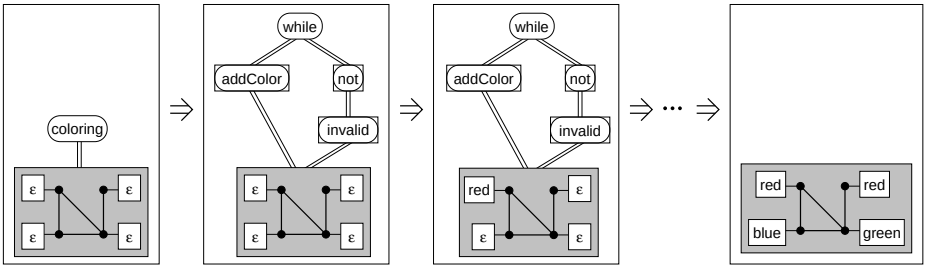

Fig. 14. An evaluation of coloring which yields a graph coloring

6 Diagrams

Genericity, the other key feature of the language, is based on diagrams as an external notation for graphs. This section briefly recalls how diagrams can be represented by graphs (e.g., in diagram editors), and, vice versa, how graphs can be visualized by diagrams.

6.1 Representation and recognition of diagrams

Andries *et al.* [1] have proposed a model for representing diagrams as graphs that has been modified in DIAGEN, a tool for generating diagram editors which support free-hand as well as syntax-directed editing [13, 16, 14]. This model shall be described here. Figure 15 shows the levels of diagram representation and the recognition steps when analyzing a given diagram.

Scanning creates a *spatial relationship graph* (*SRG* for short) for capturing the lexical structure of a diagram. This step uses edges for representing the *elements* of a diagram language (like circles, boxes, arrows, text); these *component edges* are linked to nodes representing the *attachment areas* at which these elements can be connected with

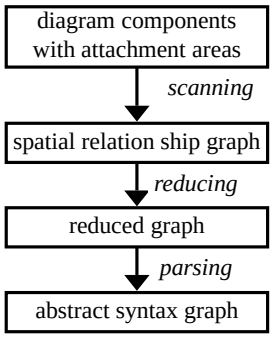

Fig. 15. Diagram representation and recognition in DIAGEN

the attachment areas of other elements, (like the *border* and *area* of circles and boxes, the *source* and *target* ends of arrows). The connection of attachment areas is explicitly represented by binary relationship edges. The type of a relationship edge reflects the types of connected attachment areas and their kind of connection (e.g., intersection). For instance, the *sources* and *targets* of arrows may have intersections with the *borders* of circles or boxes leading to specific relationship edges between the nodes of the corresponding element edges.

Syntactic analysis is performed in two steps: The SRG is first *reduced* to a more condensed graph (*reduced graph*, RG for short) which is then *parsed* according to the syntax of the diagram language. This two-step analysis transforms the spatial relationships into logical relations of the *abstract syntax graphs* of the diagrams. (The specification of diagram syntax is discussed in Section 4.) The separation into two independent steps allows for a tractable process of specifying the syntax of the diagram language and for efficient syntax analysis.

Syntax-directed editing is supported by such editors, too. DIAGEN comes with an abstract machine for the operational specification and execution of graph transformation rules (cf. Section 2.2) which are used to modify the SRG . This abstract machine already offers the full functionality which is necessary for the implementation of an interpreter for DIAPLAN which is outlined in Section 7.

This model is generic; a wide variety of diagram notations can be modeled, e.g., finite automata and control flow diagrams [17], Nassi-Shneiderman diagrams [15], message sequence charts and visual expression diagrams [16], sequential function charts [13], and ladder diagrams [14]. Actually we are not aware of a diagram language that cannot be modeled that way.

6.2 Genericity

As this generic model uses graphs for modeling very different diagram notations, we can also use diagrams for visualizing graphs. This capability allows to tackle the problem that graphs are basically a visual data structure, but using graphs for programming directly might be too abstract. Instead, we can choose an arbitrary visual syntax for *external* representations even if the programming language represents visual data as graphs internally. The user interface of a program can so be customized for the visual representations which are best suited in its application domain. This makes it possible to use the programming language of this paper which is based on graph transformations as a *generic visual programming language*. By representing very different diagram notations by graphs and operating on these graphs, many different flavors of visual (programming) languages can be described and implemented. Obvious examples are *Pictorial Janus* [12] (whose agents with ports directly correspond to our notion of typed edges) or *KidSim* [24].

7 Implementation

The programming language outlined in this paper 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. (See Figure 16 for a diagram of the system structure.)

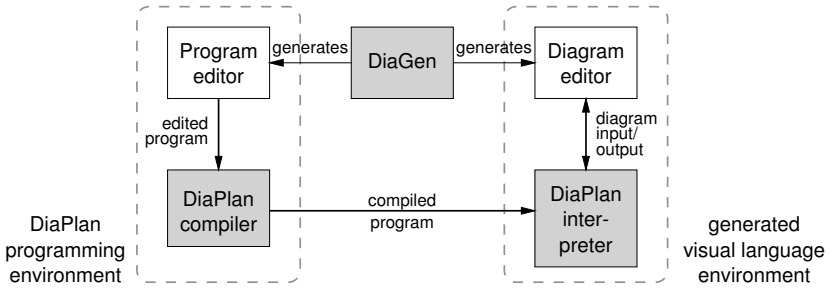

Fig. 16. System architecture

- 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. As an implementation of this interpreter, DIAPLAN will use the abstract machine for graph transformations which is part of DIAGEN.
- The *compiler* reads programs and transforms them into an internal form that can be easily and efficiently executed by the interpreter. And, very important, the compiler checks whether the program violates any lexical, syntactical or contextual rule of the language. Among the contextual rules, typing (as discussed in Section 4) plays a prominent role: The type checker shall be implemented with the graph parser built into DIAGEN [15].
- The interpreter shall have an interactive visual editor by which the input data is created. This editor shall be generated from a specification in DIAGEN in order to support customizable diagram notation for the (abstract syntax) graphs that are produced by the editor in order to be transformed internally.
- The programs will also be constructed with an interactive editor for the visual syntax of the programming language. Again, this editor shall be generated from a DIAGEN specification of graphs, rules, predicates, types, and classes.

Altogether, only the compiler has to be implemented anew, while we rely entirely on the operational graph transformation machine of DIAGEN and the capability of DIAGEN for generating the user interfaces of DIAPLAN. DIAGEN needs some extensions to meet the needs of this application:

- Other ways of specifying diagram languages, like normalizing constructor rules, have to be investigated.
- A visual user interface has to be provided for DIAGEN itself. It comes to no surprise that DIAGEN shall be used for this purpose.

The implementation of the compiler 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*. For instance, the matching of a rule explained in section 2.2 requires to check *subgraph isomorphism*, which is NP-hard in general. We hope that we will come close

the performance of logical and functional languages' implementations, at least after restricting the shapes of graphs in a suitable way. However, we cannot draw very much from the experience with implementing textual (functional or logical) languages for this aspect of the implementation.

8 Conclusion

In this paper we have presented a new programming language based on graph transformation that is rule-oriented, object-oriented, and supports structured graphs. In particular, we have discussed typing of the language, and its genericity with respect to diagram notations. Genericity allows to represent graphs by specific notations of the application domain in the user interface. This feature makes the language and its specified environment well-suited for simulations and animations.

Structured graphs have already been proposed in the context of graph transformation [21, 8], in graphical data base languages [20], and in system modelling languages [27]. Graph *shapes* exist in *Structured Gamma* [10] (for unstructured graphs).

However, we are not aware of any other language or language proposal that features structured graphs, transformations, and classes together with genericity and typing.

The precise definitions of the concepts presented in this paper has been started in [7], and needs to be continued. Some more concepts, like *concurrency* and *distribution*, have still to be considered. As these concepts have been studied by [25] and [26] in a similar setting, there is some hope that these results can be extended to our language.

Last but not least, a compiler (with type checker) has still to be implemented.

References

1. M. Andries, G. Engels, and J. Rekers. How to represent a visual specification. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, chapter 8, pages 245–260. Springer, New York, 1998.
2. R. Bardohl. GENGED: A generic graphical editor for visual languages based on algebraic graph grammars. In *Proc. VL'98*, pages 48–55, 1998.
3. M. M. Burnett, A. Goldberg, and T. G. Lewis, editors. *Visual Object-Oriented Programming*. Manning, 1994.
4. A. Corradini and U. Montanari, editors. *Proc. Joint* COMPUGRAPH/SEMAGRAPH *Workshop on Graph Rewriting and Computation*, number 2 in Electronic Notes in Theoretical Computer Science, `http://www.elsevier.nl/locate/entcs`, 1995. Elsevier.
5. P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph. In Burnett et al. [3], chapter 3, pages 45–66.
6. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [22], chapter 2, pages 95–162.
7. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. In *Foundations of Software Science and Computation Structures (FOSSACS 2000)*, LNCS, 2000. To appear.
8. G. Engels and A. Schürr. Encapsulated hierachical graphs, graph types, and meta types. In Corradini and Montanari [4].
9. M. Erwig and B. Meyer. Heterogeneous visual languages – Integrating visual and textual programming. In *Proc. VL'95*, pages 318–325, 1995.

10. P. Fradet and D. L. Métayer. Structured Ggamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.
11. 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.
12. K. M. Kahn and V. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proc. IEEE Workshop on Visual Languages (VL'90)*, pages 7–15, 1990.
13. O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *Proc. International Workshop on Graph Transformation (*GRATRA *2000), Berlin*, March 2000.
14. 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.
15. M. Minas. Diagram editing with hypergraph parser support. In *Proc. 13th IEEE Symp. on Visual Languages (VL'97), Capri, Italy*, pages 230–237. IEEE Computer Society Press, 1997.
16. M. Minas. Hypergraphs as a uniform diagram representation model. In G. Engels and G. Rozenberg, editors, *Prelim. Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98)*, Technical Report tr–ri–98–201, pages 24–31. Universität GH Paderborn, 1998. To appear in Lecture Notes in Computer Science, Springer, 2000.
17. M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In V. Haarslev, editor, *Proc. 11th IEEE Symp. on Visual Languages (VL'95), Darmstadt, Germany*, pages 203–210. IEEE Computer Society Press, 1995.
18. J. J. Pfeiffer, Jr. LUDWIG$_2$: Decoupling program representations from processing models. In *Proc. VL'95*, pages 133–139, 1995.
19. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
20. A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, 12(1):35–68, 1994.
21. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
22. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.
23. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Rozenberg [22], chapter 13, pages 487–550.
24. D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, 1994.
25. G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. Dissertation, TU Berlin, 1996. Shaker Verlag.
26. G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. In Corradini and Montanari [4].
27. J. Tapken. Implementing hierarchical graph structures. In J.-P. Finance, editor, *Proc. Formal Aspects of Software Engineering (FASE'99)*, number 1577 in Lecture Notes in Computer Science. Springer, 1999.
28. R. Wodtli and P. Cull. CALYPSO: A visual language for data structures programming. In *Proc. VL'97*, pages 166–167, 1997.