Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 15 pages

Transformation of Shaped Nested Graphs and Diagrams[⋆]

Berthold Hoffmann[a,1] Mark Minas[b,2]

[a] *Technologiezentrum Informatik, Universität Bremen, Postfach 330 440, D-28334 Bremen, Germany*

[b] *Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg, Martensstr. 3, D-91058 Erlangen, Germany*

Abstract

This paper describes a new computational model for rule-based programming with graphs and diagrams. Using existing nesting concepts for graphs, this model defines an intuitive way of *nested graph transformation* that is based on variable matching. Shape rules are introduced for specifying structural consistency conditions on nested graphs. Shape rules set up a decidable type discipline for a refined model of *shapely nested graph transformation.* Since the refined model is compatible with the diagram editor DiaGen, it can be extended by customizable diagram interfaces so that it specifies rule-based *diagram transformation.*

Key words: nested graph, structural graph typing, nested graph transformation, diagram language

1 Introduction

The Unified Modeling Language (Uml [22]) shows that diagram languages become more and more important for system design and programming.

This paper is about a computational model of a rule-based language for programming with diagrams. The model uses *graphs* as a data model, since practically every kind of diagram can be abstractly represented as a graph, even if the diagram itself does not look graph-like [1]. In order to reach the expressive power of recursive data structures in conventional languages, graphs have earlier been extended by a concept of *nesting* [5]. In contrast to related notions of hierarchical graphs that are used for software modeling [3,7],

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

[1] Email: hof@tzi.de

[2] Email: minas@informatik.uni-erlangen.de

nested graphs are compositional since they forbid edges that cross component boundaries.

Graph transformation is a rich theory for rule-based computations on graphs [21]. We combine simple concepts of graph transformation, like edge replacement [4] and substitutive transformation [19], to a new way of *nested graph transformation* that is based on variable matching and instantiation. We find this intuitive, elegant, and particularly useful for programming, as it resembles other rule-based computational models, e.g. term rewriting [15].

Furthermore, we introduce *shape rules* that specify structural consistency conditions on nested graphs. Shape rules are similar to algebraic data type definitions in functional languages, but more powerful as sharing of substructures may be specified as well; they also go beyond typing concepts of other programming languages based on graph transformation like ProGres [24], which merely restrict the labelling and degree of nodes and edges. Nested graph transformation is refined to *shapely nested graph transformation*, by using shapes as a consistent and decidable type discipline that rules out ill-shaped graphs.

Finally, we demonstrate that shapely nested graph transformation is compatible with the diagram editor generator DiaGen [16,12]. This provides the computational model with a user interface that may be customized to the diagram notations used in particular application areas, and turns it into a model of *diagram transformation.*

This paper is structured as follows: Section 2 recalls nested graphs, to which graph transformation is extended in section 3. In section 4, we describe the specification of graph shapes, and refine transformation in section 5 so that it adheres this shape discipline. The integration of shapely nested graph transformation into the DiaGen diagram editor generator is sketched in section 6. We conclude with some remarks on related and future work (in section 7).

Acknowledgement. We thank the reviewers for valuable comments that helped us to improve the paper (hopefully).

2 Nested Graphs

Our definition of graphs is tailored to programming. It extends common notions of graphs in two respects:

- Edges may connect *any* number of nodes, not just two, so that relations of any arity can be represented.
- Edges may contain graphs, the edges of which may again contain graphs so that graphs consist of *nested* components.

The graphs occurring in rules have distinguished nodes at which they may be glued to other graphs, and may furthermore contain *variables* as placeholders where graphs may be substituted.

Nested Graph. Let Λ be a finite set of *labels* and $\mathcal{X}$ a countable set of *variable names.*

The set $\mathcal{G}$ of *nested graphs* over Λ and $\mathcal{X}$ (*graphs* for short) consists of tuples $G = \langle V, E, \mathrm{lab}, \mathrm{att}, \mathrm{cts}\rangle$ with finite sets V of *nodes* and E of *edges*, a *labelling function* $\mathrm{lab} : E \to \Lambda \cup \mathcal{X}$, an *attachment function* $\mathrm{att} : E \to V^*$ that assigns sequences of nodes to edges[3], and a *contents function* $\mathrm{cts} : E \to \mathcal{G}$ mapping edges to nested graphs.

Because of the recursion in the definition, nested graphs are defined inductively over the nesting depth: $\mathcal{G} = \bigcup_{n \geqslant 0} \mathcal{G}_n$, with $\mathcal{G}_0 = \{\langle\emptyset\rangle\}$ (where $\langle\emptyset\rangle$ is the empty graph without nodes and edges), and, for any depth $n > 0$, $G \in \mathcal{G}_n$ if $\mathrm{cts}(e) \in \mathcal{G}_{n-1}$ for every edge $e \in E$.

An edge e is called *plain* if $\mathrm{cts}(e) = \emptyset$, and a *frame* otherwise. If e has $k \geqslant 0$ attachments and label $l \in \Lambda \cup \mathcal{X}$, we qualify it as *$k$-ary*, and as an *$l$-edge.* Edges labelled by variable names are called *variable edges* (*variables*, for short). A variable e is *straight* in a graph G if its attachments are pairwise distinct. We assume that all variables in a nested graph are plain.

A graph G is called *plain* if all its edges are plain, and the *top* $\widehat{G}$ is the plain graph obtained by emptying the contents of G's frames.

A *pointed graph* $\langle G, p\rangle$ is a graph $G = \langle V, E, \mathrm{lab}, \mathrm{att}, \mathrm{cts}\rangle$ with a distinguished sequence $p \in V^*$ of *points.* If p has length k, G is called *k-ary.* Often we denote pointed graphs only by their graph component, and refer to their points by p_G.

Example 2.1 [Control Flow Graphs] Figure 1 below shows three control flow graphs of sequential imperative programs.

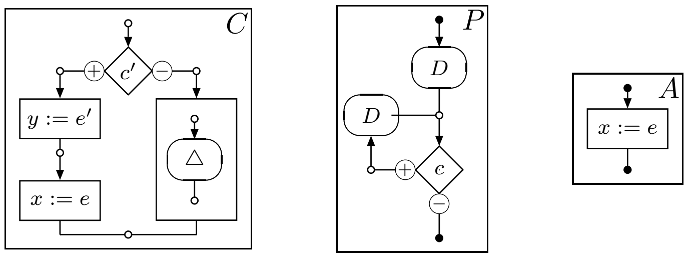

Fig. 1. Three control flow graphs

Nodes in a control flow graph represent *states of execution*; they are drawn as small circles, and filled if they are points. The topmost point in a control flow graphs indicates its *start state*, and the remaining points represent *stop states.* Constant edges represent *ececution steps*: *assignments* and *calls* are drawn as boxes, and connect one *predecessor* to one *successor* state; *branches* are drawn as diamonds, and connect one *predecessor* to one *true successor* and

[3] V^* denotes the set of *sequences* $v_1 \dots v_k$ over some vocabulary V ($v_i \in V$ for $1 \leqslant i \leqslant k$). The *empty sequence* is denoted by ε.

one *false successor* state. Assignments and branches are plain, and labelled by assignment statements, and conditions, respectively; calls are frames that contain control flow graphs of the called subroutine. Variables are drawn as ovals; they connect one predecessor to several successor states. An arrow links the predecessor state to an edge, and lines link it to its successor states; *true* and *false* successor states of branches are distinguished by attaching $\oplus$ resp. $\ominus$ to these lines. The picture of a graph is enclosed in a box and that may contain its name in the upper right corner.

Graphs P and A are plain and pointed, and C contains a single call frame. The variables (named $\triangle$ in C, and D in P) are straight.

Morphism. Structure-preserving mappings between graphs are used to identify (copies of) a graph as a subgraph in another one.

Let $G = \langle V, E, \mathrm{lab}, \mathrm{att}, \mathrm{cts}\rangle$ and $G' = \langle V', E', \mathrm{lab}', \mathrm{att}', \mathrm{cts}'\rangle$ be graphs. A *morphism* $m : G \to G'$ is a triple $m = \langle m_V, m_E, M\rangle$ where

- $m_V : V \to V'$ and $m_E : E \to E'$ are node and edge mappings preserving labels and attachments:

$$\mathrm{lab}'(m_E(e)) = \mathrm{lab}(e) \text{ and } \mathrm{att}'(m_E(e)) = m_V^*(\mathrm{att}(e)) \text{ for all } e \in E$$[4]

- $M = (m_e : \mathrm{cts}(e) \to \mathrm{cts}'(m_E(e)))_{e\in E}$ is a family of morphisms between the contents of edges.

The morphism m is surjective (injective) if its component mappings are surjective (injective, respectively). If m is surjective and injective, it is called an *isomorphism*, and the graphs G and G' are called *isomorphic*, denoted by $G \cong_m G'$. (We omit the index m if it is not relevant.)

Components. Frames are nested in a tree-like fashion. The sequence of its enclosing frames identifies a nested graph component.

The *occurrences* of edges in a graph $G = \langle V, E, \mathrm{lab}, \mathrm{att}, \mathrm{cts}\rangle$ are the edge sequences

$$\Omega_G = \{\varepsilon\} \cup \{ew \mid e \in E, w \in \Omega_{\mathrm{cts}(e)}\}$$

Then the *component* G/w at some occurrence $w \in \Omega_G$ is given as

$$G/\varepsilon = G \quad \text{and} \quad G/(ew) = \mathrm{cts}(e)/w$$

General Assumption. For the rest of this paper, we assume that the nodes and edges of all components of a graph G are pairwise disjoint, and define V_G, E_G, lab_G, and att_G as the componentwise (disjoint) union of the constituents

[4] The extension $f^* : A^* \to B^*$ of a function $f : A \to B$ maps the empty sequence ε onto itself, and a sequence $a_1 \dots a_k$ onto the sequence $f(a_1)\dots f(a_k)$.

of all components G/w of G. Likewise, the variable names occurring in the set of all components of G are denoted by $\mathcal{X}_G$.

Edge Replacement, Context Embedding, Variable Instantiation. The insertion of graphs for edges it the basic operation for defining context embedding and variable instantiation, the auxiliary operations for graph transformation.

The *replacement* of a plain k-ary edge $e \in E_G$ in a graph component G by a pointed n-ary graph $\langle U, p\rangle$ is written as $G[e/U]$, and proceeds in three steps:

(i) If G/w be the component where e is a top edge, construct the disjoint union of G/w and U, and remove e;

(ii) Glue the corresponding nodes in $\mathrm{att}_G(e)$ and p, for $1 \ldots \min(k, n)$;[5]

(iii) Replace G/w with this graph in G.

Replacement of plain edges is *commutative*: It does not matter whether some edge e is replaced by a graph U before another edge $\bar{e}$ is replaced by another graph $\bar{U}$, or afterwards.

A graph C is a *context* if it contains exactly one straight *hole variable*, say h, in one of its components. The *embedding* of a pointed graph U into C is given by $C[h/U]$.

A *substitution pair* $x \mapsto U$ associates a variable name x with a pointed graph U wherein all points are pairwise distinct. A *substitution* is a finite set

$$\sigma = \{x_1 \mapsto U_1, \ldots, x_k \mapsto U_k\}$$

of substitution pairs, with pairwise distinct variable names.

The *instantiation* of a graph P by the substitution σ is obtained by parallel replacement of all x_i-variables in $\mathcal{X}_G$ by distinct copies of the graphs U_i, for $1 \leqslant i \leqslant n$, and is denoted by $P\sigma$. Since edge replacement is commutative, this defines $P\sigma$ uniquely (up to isomorphism).

Example 2.2 [Variable Instantiation and Context Embedding] Figure 2 illustrates operations on the graphs C, P, and A shown in figure 1. The left graph results from instantiating P by the substitution $\sigma = \{D \mapsto A\}$. The right graph is the embedding of $P\sigma$ in the graph C (which is a context).

3 Graph Transformation

Graphs are transformed by embedding a pattern graph into a context, after matching its variables to suitable subgraphs, embedding a replacement graph into that same context, and instantiating the replacement's variables by the matching substitution.

[5] Thus only $\min(k, n)$ nodes are actually identified. We could enforce $k = n$, but consider this a question of typing, to be considered later.

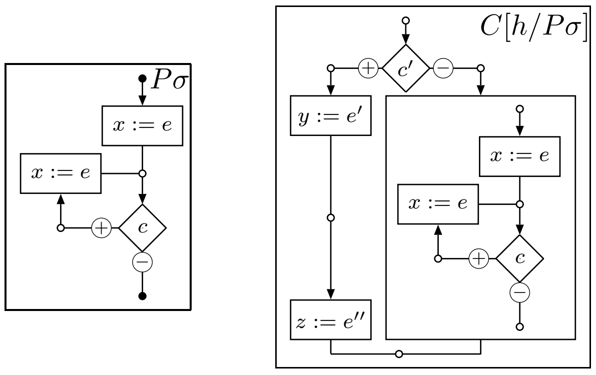

Fig. 2. A context embedding and a variable instantiation

Graph Transformation. A *graph transformation rule* $t : P \to R$ (*rule*, for short) consists of a pointed *pattern graph* P wherein all points are pairwise distinct, and a pointed *replacement graph* R.

A rule $t : P \to R$ *matches* a graph G if there is a context C and a substitution σ such that $C[h/P\sigma] \cong_m G$ for some isomorphism m. The triple $\langle t, m, \sigma\rangle$ is then called a *redex* in G; it *transforms* G into a graph $H \cong C[h/R\sigma]$, denoted by $G \Rightarrow_t H$.

If T is a set of rules, we write $G \Rightarrow_T H$ if some rule $t \in T$ transforms G to H. This relation is the least relation that contains T, and is closed under isomorphism, substitution, and context embedding. We write $\Rightarrow_T^*$ for the reflexive-transitive closure of $\Rightarrow_T$ so that $G \Rightarrow_T^* H$ expresses that T transforms G to H in $n \geqslant 0$ steps.

Example 3.1 [Control Flow Graph Transformation] Figure 3 shows two transformation rules for control flow graphs. Rule l does *loop transformation*, and rule u *unfolds* the body of a procedure call. Figure 4 shows two transformations of a control flow graph using these rules. The pattern graph P of the rule l, the context graph C and the substitution σ used for the step $H \Rightarrow_l J$ are shown in Figures 1 and 2.

Nested graph transformation is decidable.

Theorem 3.2 *Given a nested graph G and some rule $t : P \to R$, a redex $\langle t, m, \sigma\rangle$ and the transformation $G \Rightarrow_t H$ based on that redex can be computed (if they exist).*

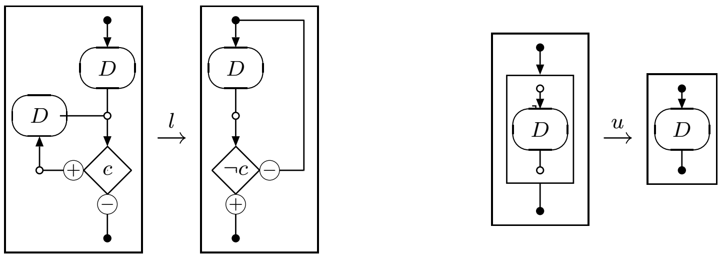

Fig. 3. Rules for loop transformation (left) and procedure unfold (right)

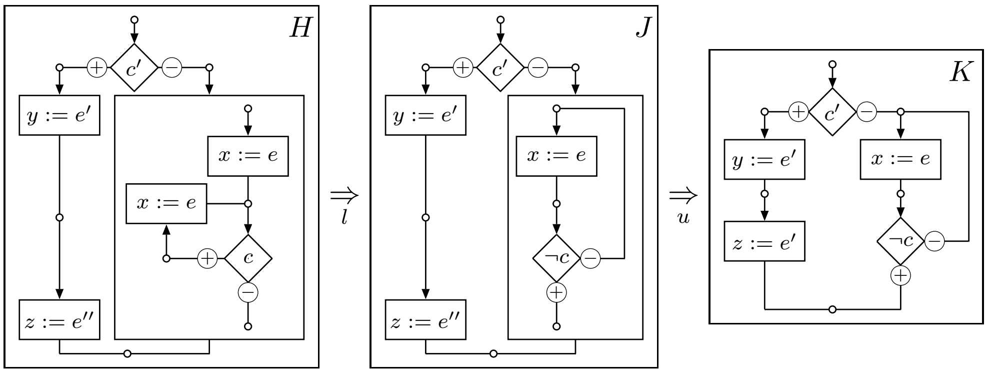

Fig. 4. Two transformation steps

Proof. Constructing a redex amounts to check, for every component $G/\bar{w}$ of G ($\bar{w} \in \Omega_G$), whether there are substitutions σ_w that make the pattern component $\widehat{P/w}$ equal to the graph components $\widehat{G/\bar{w}w}$, for every occurrence $w \in \Omega_P$. This is the *plain graph matching problem*, which is decidable [19, corollary 15]. The transformation $G \Rightarrow_T H$ can then be easily constructed, by embedding and instantiation. □

Since the *subgraph isomorphism problem* $P \cong^? S \subseteq G$ alone is NP-complete for arbitrary plain graphs P and G, we need further restrictions to make graph transformation *efficient*. (This holds for every other kind of graph transformation that does not restrict the size of patterns.) Although a deep discussion of efficiency issues is beyond the scope of this paper, this has been one motivation for the shapes introduced below.

4 Shapes

Shape analysis has been used for inferring whether pointer structures in imperative programs are shaped as doubly-linked lists, root-connected trees and the like [23]. Here we devise a means to *specify* shapes of graphs in a way that is not possible on the level of imperative languages (nor in functional or logical languages where pointers are hidden altogether). The specification is based on edge replacement [4], but we immediately adopt a terminology fitting to our purposes.

Shape Grammars. The *handle graph* of a label $l \in \Lambda$ consists of one straight k-ary l-edge that is attached to k pairwise distinct points, and is denoted by $l^\bullet$. (The arity $k \geqslant 0$ of $l^\bullet$ is arbitrary, but fixed for every l.)

Let $N \subseteq \Lambda$ be a finite set of *shape names*. A transformation rule $s : P \to R$ is a *shape rule* if its pattern P is the handle graph $n^\bullet$ of a shape name $n \in N$, and its replacement R contains no variables. We write $P \to R_1 \mid \ldots \mid R_k$ for shape rules with the same pattern. Shape rules perform edge replacement.

A finite set S of shape rules induces a *shape system* $\Sigma = \langle \mathcal{G}, N, S \rangle$. Σ derives the set $\mathcal{S}_\Sigma = \{G \in \mathcal{G} \mid n^\bullet \Rightarrow^*_S G, n \in N\}$ of *graph shapes*.

The membership problem for shape systems is known to be decidable [4, sect 2.7]:

Theorem 4.1 *There is an algorithm determining whether some graph $G \in \mathcal{G}$ is a graph shape or not.*

Example 4.2 [Shapes of Control Flow Graphs] Figure 5 shows the rules defining the shape of *structured control flow graphs* that are built (from left to right) over assignments, procedure calls, by sequential composition, conditional statement, pre-checked and post-checked loop. Edges labelled by shape names are drawn in grey.

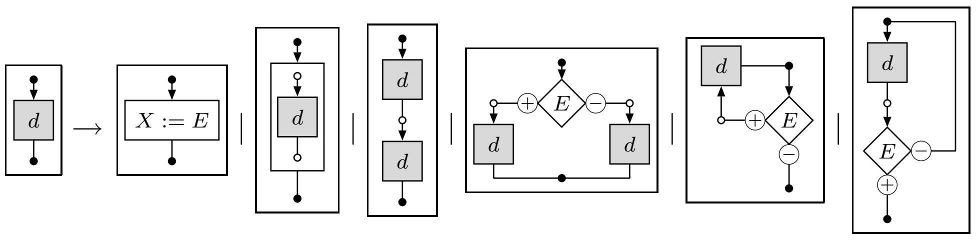

Fig. 5. The shape of structured control flow graphs

We have chosen a quite restrictive kind of flow graphs. The *semi-structured flow graphs* of [8], which are useful for data flow analysis in compilers, can be defined in a similar way [4].

Shape grammars generate a class of graph languages that seems to be one of the largest classes that can be called *context-free.* (See [4, section 2.5] for details.) In particular, they allow to define the *recursive algebraic data types* of functional languages that are constructed with disjoint union + and Cartesian product ×. Moreover, doubly-linked or cyclic lists, leaf- or root-connected trees, and other imperative data structures employing structure sharing can be specified as well [9].

Theorem 4.1 shows that shape grammars cannot define the recursively enumerable graph languages. For instance, it is not possible to define arbitrary control flow graphs in a this way. However, if a very simple form of embedding rules is allowed in addition to the shape rules used here, arbitrary control flow graphs can be defined as well, and membership can still be decided [16].

5 *Shapely* Graph Transformation

We now define shaped graphs, and refine graph transformation so that shapes are preserved. The straightforward proofs for the results in this section appear in [11].

Shaped Graphs. For the rest of this paper, we fix a shape system $\Sigma = \langle \mathcal{G}, N, S \rangle$, and require that the shape names N do not occur as labels in nested graphs $G \in \mathcal{G}$. Furthermore we assume that every variable name $x \in \mathcal{X}$ is *typed* with a shape name $\mathrm{type}(x) \in N$.

We define the *shape* $\lceil G \rfloor$ of a (pointed) graph G by relabelling every x-variable with its shape name $\mathrm{type}(x)$, for every $x \in \mathcal{X}_G$.

A graph $G \in \mathcal{G}$ is *n-shaped* if $n^\bullet \Rightarrow_S^* \lceil G \rfloor$ for some shape name $n \in N$. Then we write $\mathrm{type}(G) = n$. The set of *shaped graphs* is then given by $\mathcal{G}_\Sigma = \{G \in \mathcal{G} \mid n^\bullet \Rightarrow_S^* \lceil G \rfloor, n \in N\}$.

A substitution pair $x \mapsto \langle S, p \rangle$ is *shaped* if S is $\mathrm{type}(x)$-shaped. A *shaped substitution* σ consists of shaped substitution pairs.

Shapely Transformation. From now on, we restrict our attention to the transformation of shaped graphs by shapely rules, using shaped contexts and substitutions.

A rule $t : P \to R$ is *shapely* if its pattern and replacement graphs P and R are shaped so that $\mathrm{type}(P) = \mathrm{type}(R)$. Such a rule *matches* a shaped graph G if there is a shaped context graph $C \in \mathcal{G}_\Sigma$ with $\mathrm{type}(\triangle) = \mathrm{type}(P)$ and a shaped substitution σ such that $C[h/P\sigma] \cong G$. Then t *transforms* G to the graph $H \cong C[h/R\sigma]$, written $G \Rrightarrow_t H$.

Example 5.1 [Shapely Control Flow Graph Transformations] All control flow graphs in this paper are structured, i.e. shaped according to d. Thus the rules in Figure 3 are shapely, and the transformations in Figure 4 are shapely as well.

The shape discipline is consistent, since the result of a shapely transformation is shaped again. If the replacement graphs of rules do not contain variables that do not appear in their pattern graphs, shapely transformation keeps graphs variable-free. (this is important since in most cases, only variable-free graphs shall be transformed.)

Theorem 5.2 *(1) If $G \Rrightarrow_t H$ by some shapely rule $t : P \to R$, then $G \in \mathcal{G}_\Sigma$ implies $H \in \mathcal{G}_\Sigma$ with* $\mathrm{type}(H) = \mathrm{type}(G)$*.*
(2) If $\mathcal{X}_R \subseteq \mathcal{X}_P$ and G is variable-free, then H is variable-free as well.

Theorem 4.1 makes shapely graph transformation decidable: Since it is decidable whether some graph is shaped, it can also be decided whether substitutions are shaped, and whether rules are shapely. Once we have checked the rules, and the start graph G of a shapely transformation, checking whether the context and substitution of a redex is shaped does not cause any overhead; in contrast, it makes it even easier to determine them!

6 Visualizing Graphs as Diagrams

Graphs and graph transformations shall be used as a computational model of the language DiaPlan for programming with diagrams. DiaPlan programs perform computations on (nested) graphs, based on (shapely) graph transformation internally, but shall use diagram representations of graphs for communication with the program's user.

An implementation of DiaPlan must therefore translate diagrams to graphs, and back again:

- When the user creates diagrams as input for a program, they must be translated to their graph representation in order to be processed internally.
- If the result of some computation shall be displayed to the user, the graph representing this result internally has to be translated back into a diagram.

The DiaGen tool [16] allows to generate diagram editors from the formal specification of a diagram language that is based on graph transformation. It already supports those tasks: the diagrams edited with the generated diagram editor are internally represented by shaped graphs as described in Section 4. The shape system is the main constituent of the specification from which diagram editors are generated by DiaGen. Below we explain how DiaGen shall realizes translations between graphs and diagrams.

6.1 Translating diagrams to graphs

When a diagram is edited freely by arranging diagram components on the screen, it has to be translated into an internal representation, i.e., a graph in DiaGen. This translation is performed in two steps: First, the diagram is translated into a *layout graph* wherein edges represent diagram components, and nodes represent attachment areas of the components. Attachment areas are distinguished parts where a diagram component may be connected to other components (e.g., the endpoints of a line). Additional *spatial relationship edges* are used to represent such connections between attachment areas. In a second step, this graph is then translated into a *hypergraph model* by parallel transformation of groups of diagram components to *diagram symbols*. While the layout graph is closely related to the visual representation of the diagram, the hypergraph model abstracts from visualization details. The hypergraph model is then subject to syntax analysis which is performed by a graph parser. Since the diagram has been translated into the graph, syntax errors in the diagram can be detected by the graph parser.

The generated editor obviously provides means for translating a diagram to a graph as required for computations based on those graphs. But DiaGen also offers a solution for checking a graph shape. Generated diagram editors perform syntax analysis by a graph parser which is based on context-free graph grammars with embedding rules [1,16]. These extensions are required since context-free grammars alone are not powerful enough to describe the syntax of arbitrary diagram languages. These extensions may be used to extend the expressiveness of shape systems, too, and, at the same time, have a parser which can check graph shapes according to such extended shape syastem.

6.2 *Translating graphs to diagrams*

So far, DiaGen editors primarily translate diagrams into graphs. But, the system also offers some elementary support for creating diagrams from graphs. DiaGen allows to specify how a diagram is created from its building blocks when the syntactic structure of the representing graph has been created by the parser. This mechanism is simply based on attribute evaluation rules which are attached to the grammar productions.

Example 6.1 [Diagram Editors] The structured control flow graphs used as a running example have two representations as diagrams that are quite different: *control flow diagrams*, and *Nassi-Shneiderman diagrams.*

Figure 6 shows snapshots of a control flow diagram editor (left) and a Nassi-Shneiderman diagram editor (right). The snapshots display the graph H shown in Figure 4.

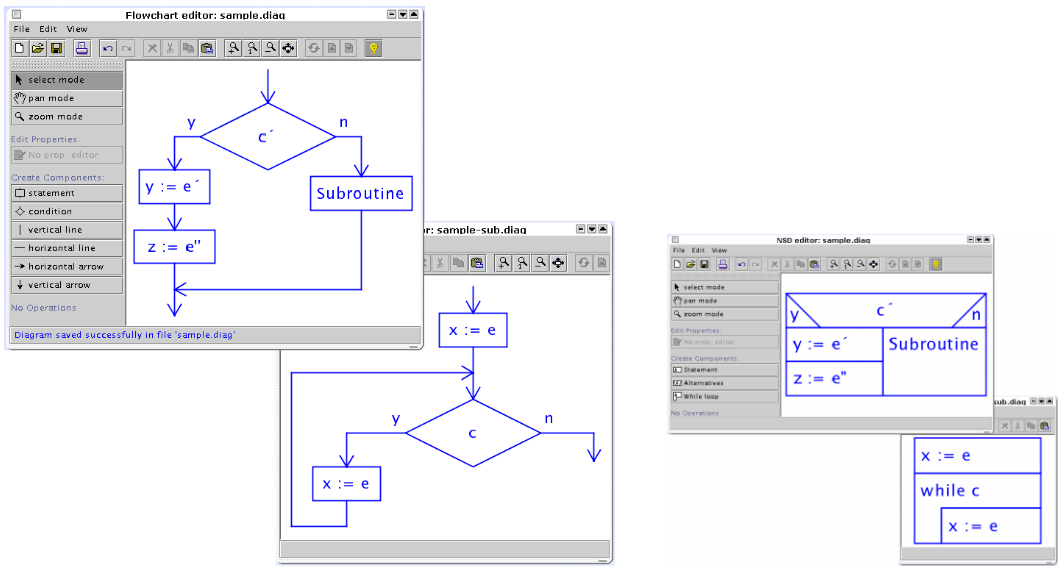

Diagram saved successfully in file 'sample.diag'

Fig. 6. Nested control flow and Nassi-Shneiderman diagrams of graph H in Figure 4

Both editors have been generated with DiaGen. Both use control flow graphs as their abstract syntactical representation. Note that the editors display the contents of frames (containing the diagrams of a procedure's body) in seperate windows.

The scenario for executing a DiaPlan program shall be as follows:

- The diagram editors provide the user interface, for creating input of the program, and displaying results of its execution.

- Operations of the program are called from this interface, e.g. via menu entries. This triggers execution of the operation, including the necessary transformations of diagrams to graphs, and back again.

7 Conclusions

We have extended graphs by a compositional notion of nesting. These graphs are no longer called *hierarchical* (as in [5,11]) since this term is mostly used for structured graphs *with* structure-crossing edges that are *not* compositional.

We have devised a notion of nested graph transformation that extends the rewriting of terms [13] (which are trees over function symbols) to trees over graphs (namely, nested graphs) in a straightforward way: Transformation substitutes variables in a rule pattern, embeds the instantiated pattern into a context, and then inserts an instantiated replacement into that context. This way of graph transformation seems to be quite intuitive from a programming point of view. Furthermore, it can easily be refined by a *shape discipline* that is consistent and decidable, and may also allow for more efficient implementation. Finally, this way of transformation integrates smoothly with diagram tools. We hope that this is accepted as excuse for proposing yet another variation of graph transformation.

Related Work. T.W. Pratt was probably the first to define nested graph languages [20]. He specified the semantics of programming languages by context-free graph grammars, but did not consider further transformation of these graph languages. Engels and Heckel [7] study hierarchical graph transformation as the basis for system modeling languages like Uml [22]. They allow edges between components (crossing frame borders), which is necessary in that application domain, but would not be adequate for programming since it would give away (de-)compositionality of nested graphs.

Substitutive graph transformation is a (modest) extension of hierarchical graph transformation [5], where variables denote the *entire* contents of frames. With our kind of transformation, it is essential that variables may denote arbitrary subgraphs.

Busatto *et al.* [3] investigate a generic notion of hierarchical graph transformation by which other approaches can be simulated [2], also that of [5], and probably shapely graph transformation as well.

The way we define shapes has been inspired by the work of P. Fradet and D. Le Métayer on *Structured Gamma* [9] that uses structured multiset rewriting, a notion that can be "translated" to graph transformation and edge replacement in a straightforward way.

Future Work. Transformation may be highly nondeterministic. This may lead to an overload of backtracking. So, nondeterminism has to be restricted to the degree that is really needed for programming. Good design of rule patterns, nesting structure, and shapes can already reduce nondeterminism.

However, even for a given context and rule, there may be many substitutions σ matching the rule to the host graph. An important goal will thus be to restrict the nondeterminism allowed by substitution. For instance, one could require that every frame in a pattern contains at most one variable. This would make rule matching nearly as deterministic as in [5].

Nested graphs consist of finite trees of components. In certain applications, components with equal contents might be *shared*, e.g. control flow graphs of the same procedure. The discussion of the adequacy of collapsed representations could draw from the results concerning collapsed representations of terms [18]. Even cyclic sharing of components makes sense, e.g. for representing control flow graphs with recursive calls. Then results concerning the cyclic representations of infinite terms could be employed [14].

Shapes are just a "structural" way of classifying values according to their (graph-ical) representation. More type discipline would be useful, for instance as in Progres [24]. Also, context-free graph languages might be too restricted for specifying the shapes of graphs that occur in certain applications. Then Church-Rosser graph languages [17] could be considered.

The Perspective. Shapely graph transformation shall become the computational model for the rule-based language DiaPlan for *programming with graphs* [10]. Further concepts of DiaPlan shall be defined on top of this model:

- *Transformation predicates* with parameters shall allow to *abstract* from transformation sequences.
- *Application conditions* and *predicate parameters* shall allow to specify *control* in an imperative or functional way.
- *Classes* shall *encapsulate* shape rules and transformation predicates.
- Primitive values like *numbers* and *strings* shall be integrated as first-class objects into the graph model.

This will make DiaPlan a mature visual programming language based on nested graphs. Together with DiaGen' user interface discussed in the previous section, it allows to implement diagram-manipulating systems that use arbitrary diagram notations that can be customized to particular application areas.

References

[1] R. Bardohl, M. Minas, A. Schürr, and G. Taentzer. Application of graph transformation to visual languages. In Engels et al. [6], chapter 3, pages 105–180.

[2] G. Busatto and B. Hoffmann. Comparing notions of hierarchical graph transformation. *Electronic Notes in Theoretical Computer Science*, 2001. to appear.

[3] G. Busatto, H.-J. Kreowski, and S. Kuske. An abstract hierarchical graph data model. Technical report, Fachbereich Mathematik-Informatik, Universität Bremen, to appear 2001.

[4] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [21], chapter 2, pages 95–162.

[5] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear 2001. (A short version appeared in number 1784 of Lecture Notes in Computer Science, pages 98–113, 2000).

[6] G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Specification and Programming.* World Scientific, Singapore, 1999.

[7] G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modelling and evolution. In U. Montanari, J. Rolim, and E. Welz, editors, *Automata, Languages, and Programming (ICALP 2000 Proc.)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer, 2000.

[8] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 42–56, Houston, Texas, 1976.

[9] P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.

[10] B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 165–180. Springer, 2000.

[11] B. Hoffmann. Shapely hierarchical graph transformation. In *Int'l Symposium on Visual Languages and Formal Methods.* IEEE Computer Press, to appear 2001.

[12] B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In J. D. P. Polim et al., editors, *ICALP Workshops 2000*, number 8 in Proceedings in Informatics, pages 443–450, Waterloo, Ontario, Canada, 2000. Carleton Scientific.

[13] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[14] R. Kennaway, J. W. Klop, R. Sleep, and F.-J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

[15] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[16] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, to appear 2001.

[17] D. Plump. Church-Rosser hypergraph languages. Talk at the Workshop "Automaten und Formale Sprachen", Schauenburg-Elmshagen, Germany, September 1999.

[18] D. Plump. Term graph rewriting. In Engels et al. [6], chapter 1, pages 3–102.

[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] T. W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, number 73 in Lecture Notes in Computer Science, pages 389–400. Springer, 1979.

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

[22] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison Wesley, 1999.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[24] A. Schürr, A. Winter, and A. Zündorf. The Progres approach: Language and environment. In Rozenberg [21], chapter 13, pages 487–550.