# CONTEXT-EXPLOITING SHAPES FOR DIAGRAM TRANSFORMATION

Frank Drewes
*Institutionen för Datavetenskap, Umeå Universitet, S-90187 Umeå (Sweden)*
*e-mail: drewes@cs.umu.se*

Berthold Hoffmann
*Technologiezentrum Informatik, Universität Bremen, D-28334 Bremen (Germany)*
*e-mail: hof@tzi.de*

Mark Minas
*Fachbereich Informatik, Universität der Bundeswehr, D-85577 Neubiberg (Germany)*
*e-mail: minas@acm.org*

**Abstract.** DIAPLAN is a language for programming with graphs representing diagrams that is currently being developed. The computational model of the languages, nested graph transformation, supports nested structuring of graphs, and graph variables, but is—hopefully—still intuitive. This paper is about structural typing of nested graphs and nested graph transformation systems by *shape rules*. We extend the context-free shape rules proposed in earlier work to *context-exploiting shape rules* by which many relevant graph structures can be specified. The conformance of a nested graph to shape rules is decidable. If a transformation system conforms to shape rules as well, it can be shown to preserve shape conformance of the graphs it is applied to. This sets up a static type discipline for nested graph transformation.

**Key words:** Graph transformation; diagram language; visual programming; data type

## 1. Introduction

DIAGEN[1] is a tool for generating diagram editors that respect the *syntax* of particular diagram languages. Editors generated with DIAGEN work as follows:

- A drawing tool makes *free-hand editing* of diagrams possible, by arranging diagram primitives like boxes, circles, lines, and text on a drawing pane. The set of available primitives depends on the particular diagram language.
- A *scanner* analyses the spatial relationships of diagram primitives, and reduces them to *diagram symbols*, like assignments and branches in a control flow diagram.
- Finally, a *parser* checks whether the diagram symbols are related according to the syntax of the diagram language, and produces a *derivation structure*.

DIAGEN relies on graphs and graph transformation for specifying diagram languages: graphs represent diagrams, the scanner is specified by graph transformation rules, and the syntax of diagrams is specified by graph grammars (see [19] for details). Case studies with editors for control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts, state charts, and

[1] see the DIAGEN homepage `http://www2.cs.fau.de/DiaGen/`

UML class diagrams indicate that the syntax of practically every kind of diagram language—also of *design languages* in a more general sense—can be implemented with DIAGEN.

DIAGEN shall now be complemented by DIAPLAN, a diagrammatic, rule-based language for programming the semantics of diagram languages, e.g. by animating their behavior, or by transforming them into canonical forms. DIAPLAN is based on *nested graphs* wherein edges may contain graphs in a nested way (see [17]), and on a rather expressive way of graph transformation rules where *graph variables* are used to match, delete, or duplicate subgraphs of arbitrary size [13]. This should make DIAPLAN convenient for developing specialized design tools, but does not restrict it to this purpose. In contrast to standard textual languages, its level of abstraction and its underlying computing paradigm fit the requirements of design tools very nicely.

A central requirement for modern high-level programming languages is the availability of user-defined data types and type checking facilities. The term *shape analysis* has come into use for inferring invariants for the graph-like pointer structures occurring in imperative programs [8]. In DIAPLAN, we use the term *shape* for graph data types specified by graph grammars.

This work extends a result of [13] concerning *shaped transformations* $G \Rightarrow_T H$ where the nested graphs $G, H$ and the transformation rules $T$ conform to *shape rules*. Shape rules provide a static shape discipline for transformation sequences $G \Rightarrow_T^* H$: Once the input graph $G$ of such a sequence conforms to the shape rules, this holds for its result graph $H$ as well. Hence, if shape conformance can be decided for nested graphs and transformation rules, runtime shape checks (of the intermediate graphs in transformation sequences) are not necessary. In [13], shape conformance is shown for *context-free shape rules*. Even if these rules can specify rather involved shapes, like that of structured control flow diagrams, they fail to cover certain other natural and practically important diagram languages. In this paper, we will therefore introduce *context-exploiting shape rules*, which allow us to define substantially more involved shapes. As an example, we specify the shape of general control flow diagrams (of "spaghetti code") which is not context -ree. As conformance to context-exploiting shape rules can be decided by an extension of the parsing algorithm employed in DIAGEN, the shape discipline remains static.

The rest of this paper is structured as follows. We recall basic concepts of graph transformation in Section 2. In Section 3 we introduce context-exploiting shapes as a generalization of context-free shapes. Afterwards, we recall nested graph transformation (in Section 4) and demonstrate that shape rules set up a static type discipline for nested graph transformation (in Section 5). We conclude with some remarks on related and future work, in Section 6.

## 2. Graphs and their Transformation

In this section we recall a notion of graphs and of graph transformation that has proved to be useful for diagram representation and manipulation in the DIAGEN system [19].

**Graphs.** Our notion of graphs is rather general: edges may connect any number of nodes, not just two, and graphs have a sequence of interface nodes at which they may be glued together.

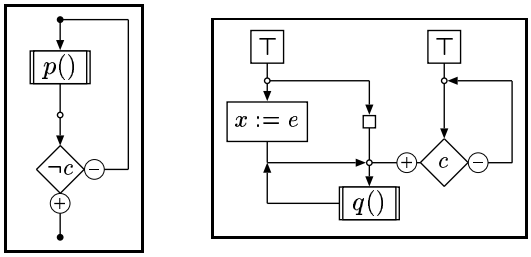


Fig. 1. Two graphs

Such graphs are known as pointed directed edge-labelled hypergraphs [5].

More precisely, let $\Lambda$ be a *ranked alphabet* where every symbol $l \in \Lambda$ comes with an *arity* $\mathrm{arity}(l) \geqslant 0$. The class $\mathcal{G}$ of *graphs* over $\Lambda$ consists of sixtuples

$$G = \langle V, E, \mathrm{lab}\colon E \to \Lambda, \mathrm{att}\colon E \to V^*, p \in V^* \rangle^2$$

with finite sets $V$ of *nodes* and $E$ of *edges*, where every edge $e \in E$ has a *labelling* $\mathrm{lab}(e)$ and a sequence $\mathrm{att}(e)$ of $\mathrm{arity}(\mathrm{lab}(e))$ *attached nodes*, and where $p$ designates a sequence of interface nodes which are called *points*. The *size* of $G$ is defined as $|G| = |V| + |E|$. Nodes of $G$ that are not points are called *inner nodes*. The length of $p$ defines the *arity* of $G$. To avoid unnecessary technicalities, we adopt the natural and practically harmless restriction that $p$ does not contain repetitions, and that the same holds for the attached node $\mathrm{att}(e)$ of every edge $e \in E$. In cases where the components of a graph $G$ are not explicitly named, we shall denote them by $V_G$, $E_G$, $\mathrm{lab}_G$, $\mathrm{att}_G$, and $p_G$ respectively.

An *isomorphism* $m\colon G \to H$ between graphs $G$ and $H$ is a pair $(m_V, m_E)$ of bijective mappings $m_N\colon N_G \to N_H$ and $m_E\colon E_G \to E_G$ with $p_H = m_V(p_G)$, $\mathrm{lab}_H(m_E(e)) = \mathrm{lab}_G(e)$, and $\mathrm{att}_H(m_E(e)) = m_V(\mathrm{att}_G(e))$ for all $e \in E_G$. $G$ and $H$ are then said to be *isomorphic*, $G \cong H$. We write $G \subseteq H$ if $G$ is a *subgraph* of $H$, i.e. if $G$'s nodes, edges, attachment function, and label function are contained in those of $H$. (Points are irrelevant for this relation.)

By $\langle l \rangle$ we denote the *handle graph* of a label $l$. It consists of a edge $e$ that is labelled with $l$ and attached to $\mathrm{arity}(l)$ points, i.e., $V_{\langle l \rangle} = \{v_1, \ldots, v_k\}$ and $E_{\langle l \rangle} = \{e\}$ where $\mathrm{att}_{\langle l \rangle}(e) = p_{\langle l \rangle} = v_1 \cdots v_k$ and $k = \mathrm{arity}(l)$. An edge $e$ is qualified as a *$k$-ary $l$-edge* if it has $k$ attached nodes and label $l$. In addition to these definitions, we use $G - e$ to denote the graph obtained by removing the edge $e$ from $G$.

**Example 2.1 (Graphs).** Figure 2.1 shows two graphs. Nodes of these graphs represent *states* of imperative programs. They are drawn as circles while edges are drawn as boxes. Unary

[2] $A^*$ denotes the set of *sequences* over some set $A$. The *empty sequence* is denoted by $\varepsilon$. The canonical extension of a function $f\colon A \to B$ to a function $f\colon A^* \to B^*$ is denoted by $f$ as well.

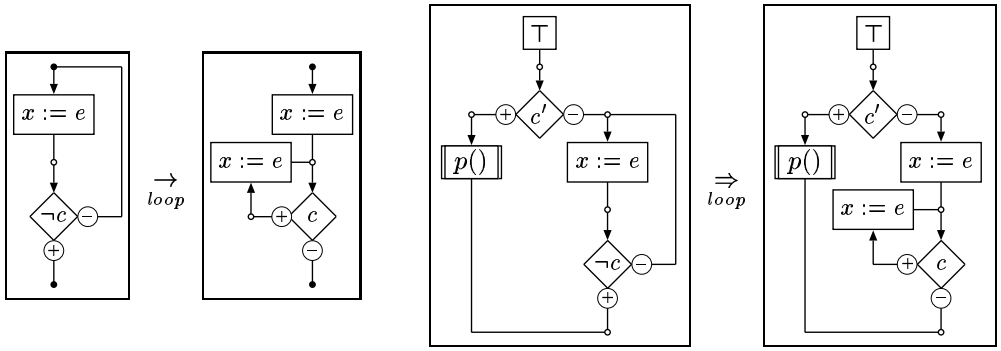


Fig. 2. A rule (on the left) and a step (on the right) transforming control flow graphs

$\top$-edges designate *start states* of programs. The remaining edges represent execution steps: small squares represent *skips* (neutral steps), rectangles represent *assignments*, rectangles with vertical stripes represent *procedure calls*, and diamonds represent *branches*; branches are ternary while the other three edge types are binary. An arrow points to a step from its predecessor state; lines link it to successor states. The true and false successors of branches are distinguished by attaching $\oplus$ and $\ominus$ to them. Finally, points are designated by filled circles. The graph on the left hand side has two of them (intuitively representing the states where the execution enters resp. leaves the diagram) while the other one has none. $\diamond$

**Transformation.** We define a rather elementary way of graph transformation that is a simple case of the algebraic theory of graph transformation [1].

A *(transformation) rule* $t = L/R$ consists of two graphs $L$ and $R$ with the same number of points. The rule can be applied to a graph $G$ if $L \cong \bar{L}$ for some subgraph $\bar{L} \subseteq G$ such that no edge in $E_G \setminus E_{\bar{L}}$ is attached to an inner node of $\bar{L}$ and no point of $G$ is an inner node of $\bar{L}$. In this case, the rule *transforms* $G$ to the graph $H$, written $G \Rightarrow_t H$, which is obtained by removing all edges and inner nodes in $\bar{L}$ from $G$, disjointly adding a fresh copy $R'$ of $R$, and identifying every point of $R'$ with the respective point of $\bar{L}$ in the remainder of $G$. The points of $R'$ loose their special status during this operation.

A finite set $T$ of transformation rules *transforms $G$ directly to $H$*, written $G \Rightarrow_T H$, if $G \Rightarrow_t H$ for some $t \in T$. If $G = G_0 \Rightarrow_T G_1 \cdots \Rightarrow_T G_n \cong H$ for some $n \geqslant 0$, we say that $T$ *transforms $G$ to $H$*, written $G \Rightarrow_T^* H$.

**Example 2.2 (Control Flow Graph Transformation).** The rule $loop$ shown on the left hand side of Figure 2 transforms loops with a body consisting of a single assignment; the right hand side shows a transformation step using $loop$. $\diamond$

## 3. Shapes

We have defined graph transformation for arbitrary graphs over $\Lambda$, although the graphs that should actually occur have a particular structure, which we call a *shape*. For example, the graph on the right hand side of Figure 1 is no "admissible" control flow graph of a sequential program: it has two start states (not one), no stop state, and the left start state is *nondeterministic* as it is the predecessor of two assignment steps.

Graph transformation itself can be used to define the shape of graphs, like Chomsky grammars define the syntax of string languages. For this, let $N$ be a ranked alphabet of *nonterminals* disjoint with the alphabet $\Lambda$. A graph transformation rule $L/R$ used to define a shape is called a *shape rule* and is written $L ::= R$, similar to string grammar rules.

Let $\Sigma$ be a finite set of shape rules. We write $\Sigma \vdash G : n$ if $\Sigma$ transforms the handle graph $\langle n \rangle$ of some nonterminal $n \in N$ into $G$, i.e. if $\langle n \rangle \Rightarrow^*_\Sigma G$. For each $n \in N$, the set of all graphs $G$ with $\Sigma \vdash G : n$ is called a *shape defined by* $\Sigma$. Note that the shapes defined in this way need not be disjoint: More than one nonterminal $n$ may satisfy $\Sigma \vdash G : n$. By definition, all graphs of a shape have the same arity, namely the arity of the respective nonterminal $n$.

By representing a string $a_1 \cdots a_n$ as a binary chain graph whose edges are labelled with $a_1, \ldots, a_n$, every type-0 Chomsky grammar can be simulated by a set of graph transformation rules. Hence, all recursively enumerable string languages (represented as graph languages) are shapes in the sense defined above, which in particular means that shapedness cannot be decided in general. So we must restrict rules in order to make membership decidable. We first recall the context-free shapes proposed in [13].

**Context-Free Shapes.** A shape rule $L ::= R$ is *context-free* (*CF*, for short) if its left hand side $L$ is the handle graph $\langle n \rangle$ of a nonterminal $n$. A set $\Sigma$ of shape rules is CF if each of its rules is CF. The shapes defined by $\Sigma$ are then said to be *CF shapes*.

**Example 3.1 (CF Shape Rules for Structured Control Flow Graphs).** Figure 3 shows a set of CF rules defining the shape of *structured control flow graphs*. The handle graph $\langle \pi \rangle$ derives all control flow graphs with a single start state (distinguished by attaching a unary edge labelled $\top$ to it) where every execution step lies on a path from the start to the stop state, without nondeterministic control flow as on the right hand side of Figure 1. Each control flow graph is built over assignments $x := e$, procedure calls $p()$, and skips (neutral state transitions) by sequential composition, conditional statement, pre-checked, and post-checked loop. In this example, $\pi$ and $\alpha$ are the only nonterminals. As usual, we write $\langle \alpha \rangle ::= R_1 | \cdots | R_n$ to abbreviate several rules $\langle \alpha \rangle ::= R_1, \ldots, \langle \alpha \rangle ::= R_n$ for the same left hand side. ◇

CF shape rules specify *hyperedge replacement*, which derives a well-studied class of context-free graph languages (see [2, 5] for details). Such rules allow the user to define recursive algebraic data types of functional and logical languages, and sophisticated pointer structures like cyclic lists, or leaf-connected trees, which cannot be defined in the type systems employed in imperative languages (see [7]). Membership in these languages is decidable:

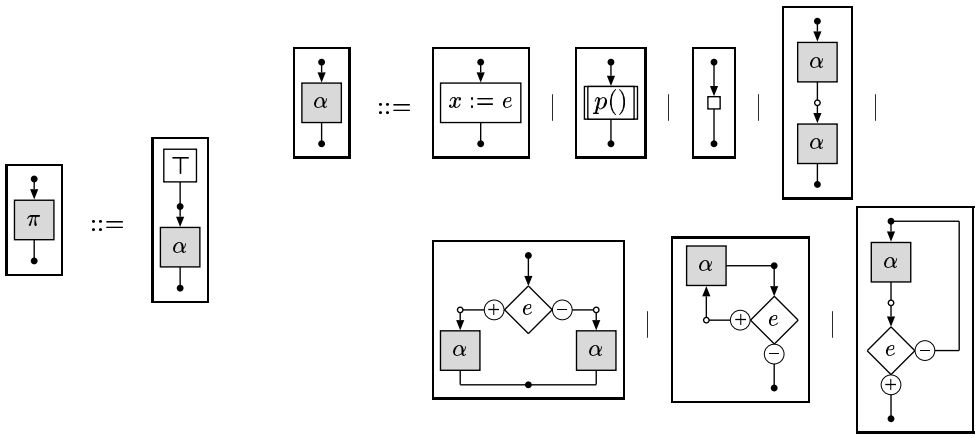


Fig. 3. Context-free shape rules for structured control fbw graphs

**Fact 3.1 (see [5, Section 2.7]).** The question "$\Sigma \vdash G : n$?" is decidable for CF sets $\Sigma$ of shape rules.

As described in [5], we can represent a context-free derivation $\langle n \rangle \Rightarrow_\Sigma^* G$ by its derivation tree, which thus represents the syntactic structure of $G$. Such derivation trees are constructed by context-free parsers [9, 6]. Derivation trees of a given graph need not be unique; similar to the string case, $\Sigma$ may be ambiguous. In a structured control flow graph, for instance, every sequential composition of more than two subdiagrams has more than one derivation tree.

The context-free parser in the DIAGEN system is a Cocke-Younger-Kasami parser that has been extended to hyperedge replacement grammars. It creates all derivation trees for any subgraph of $G$ (and therefore the set of all derivation trees of $G$ if the rules are ambiguous) in a bottom-up process by successively creating *levels* $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$ where $n$ is the number of edges of $G$. Each $\mathcal{L}_i$ of those levels is a set containing those nonterminally labelled handle graphs which can be derived to a subgraph of $G$ consisting of $i$ edges. In general, the DIAGEN parser, therefore, has an exponential time complexity. However, experience has shown that graph languages representing visual languages can usually be parsed efficiently, i.e. in polynomial time, if $\Sigma$ is chosen carefully. Efficiency is further increased by preprocessing (i.e., lexical analysis) of the graph prior to parsing and by supporting graph transformation rules with additional parsing hints (e.g., application conditions which impose an ordering of edges, e.g., for preventing ambiguity). Further work must investigate whether these approaches also can help increase the parsing efficiency for shape rules.

Shapes defined by CF shape rules have a very rigid local structure as a nonterminal edge is attached to a fixed number of nodes depending on the arity of the nonterminal. When the

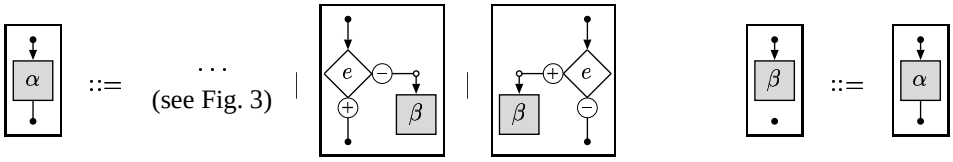


Fig. 4. Context-exploiting set of shape rules for general control flow graphs

derivation is continued, no edges can be established that link an inner node of the subgraph derived from that edge to a node outside this subgraph. This limits the generative power of CF shape rules (see also the proof of Theorem 3.1 below). Consequently, certain interesting shapes fail to be context-free. As a remedy, we devise context-exploiting shape rules. These extend CF shape rules, but for reasons of decidability we restrict them to the monotonic case (see condition (b) in the definition below).

**Context-Exploiting Shapes.** A *context-exploiting shape rule* (*CE shape rule*, for short) is a shape rule $L ::= R$ such that

(a) $L - e \subseteq R$ for some edge $e \in E_L$, where $p_L = p_R$, and
(b) $|R| \geqslant |L|$.

A set $\Sigma$ of shape rules is CE if each of its rules is CE. Each of the shapes defined by $\Sigma$ is then called a *CE shape*.

Hence, a CE shape rule replaces a single nonterminal edge $e$, but the edges of the replacing graph can be attached to nodes in the context $L - e$ of $e$. Condition (b) ensures that derivations are monotonic—the size of graphs cannot shrink during a derivation.

**Example 3.2 (CE Shape Rules for General Control Flow Graphs).** In Figure 4 we extend the CF shape rules of Example 3.1 to a CE set of shape rules for *general control flow graphs* (i.e., allowing for the description of "spaghetti code"). The additional rules for $\alpha$ introduce branches where nonterminals $\beta$ indicate places where jumps occur. The CE shape rule for $\beta$ introduces such a jump. ◇

We now show that the class of CE shapes is a proper superset of the class of CF shapes. Strictly speaking, the monotonicity requirement forces us to exclude graphs consisting of points only. Formally, for $k \in \mathbb{N}$, let $\langle\rangle_k = \langle\{v_1, \ldots, v_k\}, \emptyset, \mathrm{lab}, \mathrm{att}, v_1 \cdots v_k\rangle$, where lab and att are the empty mappings. For a shape $S$ of graphs of arity $k$ let $S^{\ominus} = S \setminus \{\langle\rangle_k\}$, and for a class $\mathcal{S}$ of shapes, $\mathcal{S}^{\ominus} = \{S^{\ominus} \mid S \in \mathcal{S}\}$.

**Theorem 3.1.** $\mathcal{S}_{\mathrm{CF}}^{\ominus} \subsetneq \mathcal{S}_{\mathrm{CE}}$, where $\mathcal{S}_{\mathrm{CF}}$ and $\mathcal{S}_{\mathrm{CE}}$ are the classes of CF resp. CE shapes.

*Proof.* Let us first argue that $S^{\ominus}$ is a CE shape for every CF shape $S$. By definition of CE shape rules, every CF shape rule $L ::= R$ is a CE shape rule unless $R = \langle\rangle_k$ for some $k \in \mathbb{N}$. However, it is shown in [2, Theorem IV.1.5] that $S^{\ominus}$ is CF if $S$ is CF, and that it can be defined

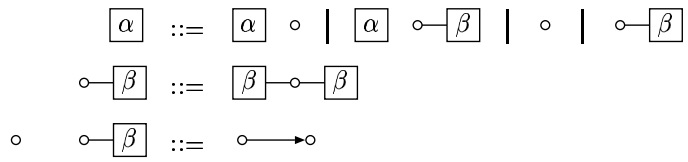


Fig. 5. Rules driving all nonempty directed graphs from a nonterminal $\alpha$.

without using this kind of rules. Hence, $S^{\ominus}$ is CE.

Now, consider the CE set of shape rules in Figure 5. Obviously, the set of all nonempty directed graphs of arity 0 (without loops) can be derived from $\langle\alpha\rangle$. However, it is well known that this shape is not CF (see, e.g., [2, Theorem IV.3.3]), which proves the theorem. ∎

In fact, the CE shape of general control flow graphs of Example 3.2 is another one that is not CF. This can be verified by observing that the bandwidth of these graphs is not uniformly bounded whereas this is the case for every CF shape (see [2, Summary V.2.6]).

The monotonicity of CE shape rules makes it easy to prove that CE shapes are decidable:

**Theorem 3.2.** The question "$\Sigma \vdash G : n$?" is decidable for CE sets $\Sigma$ of shape rules.

*Proof.* By the definition of CE shape rules, $G \Rightarrow_\Sigma H$ implies $|H| \geqslant |G|$. Consequently, $\Sigma \vdash G : n$ can be checked using a variant of the well-known membership test for monotonic Chomsky grammars: By a depth-first search one simply generates the finite set of graphs $\bar{G}$ such that $\langle n\rangle \Rightarrow^*_\Sigma \bar{G}$ and $|\bar{G}| \leqslant |G|$. This search terminates because a derivation can be pruned as soon as the derived graph is larger than $G$ or has been encountered earlier in the derivation. ∎

This procedure for deciding $\langle n\rangle \Rightarrow^*_\Sigma G$ is of course inefficient. However, the DIAGEN parser for hyperedge replacement grammars can easily be extended to CE sets of shape rules, such that the resulting parser, hopefully, is efficient in practical cases:

*Algorithm sketch.* Let $G$ be the graph that is going to be checked by this parser, $\Sigma$ be a CE set of shape rules, and $\Sigma_{\mathrm{CE}} \subseteq \Sigma$ its maximal subset of shape rules which are not CF. Without loss of generality, each of these non-CF shape rules $p \in \Sigma_{\mathrm{CE}}$ is of the form $L_p - e_p \cong R_p - e'_p$ for some edges $e_p \in E_{L_p}$ and $e'_p \in E_{R_p}$ which are called *deleted* resp. *created edges of* $p$ (otherwise, suitable CF shape rules have to be added beforehand). For the purpose of parsing, we define a *corresponding CF shape rule* $p_{\mathrm{C}} : \langle n_p\rangle ::= R_p$ of $p$ by replacing the left hand side of $p$ by the handle graph of a fresh nonterminal label $n_p$, called *CF label of* $p$. We transform the set $\Sigma$ of CE shape rules to a *corresponding* set $\Sigma_{\mathrm{C}}$ of CF shape rules by replacing each non-CF shape rule by its corresponding CF one, i.e., $\Sigma_{\mathrm{C}} = \Sigma \cup \{p_{\mathrm{C}} \mid p \in \Sigma_{\mathrm{CE}}\} \setminus \Sigma_{\mathrm{CE}}$. Using $\Sigma_{\mathrm{C}}$, the graph $G$ is then processed by the DIAGEN parser that creates the levels $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$ (see page 6.) Levels are then searched for occurrences of handle graphs with CF labels. For each shape rule $p \in \Sigma_{\mathrm{CE}}$ and for each occurrence $\langle n_p\rangle$ with its CF label $n_p$, we can conclude: $\langle n_p\rangle$ indicates that the right hand side of rule $p$ can be derived to a subgraph of $G$. $\langle n_p\rangle$ refers to a

handle graph $\langle e'_p \rangle$ containing the *created* edge $e'_p$ of rule $p$. $\langle e'_p \rangle$ belongs to one of the levels, say $\mathcal{L}_j$. Moreover, the levels contain handle graphs of the other edges of the right hand side of rule $p$. We then add a new handle graph $\langle e_p \rangle$ containing the *deleted* edge $e_p$ of rule $p$ to the same level $\mathcal{L}_j$ and add references to $\langle e'_p \rangle$ as well as to the other handle graphs of the right hand side of rule $p$. The DIAGEN parser is then used to update the sets $\mathcal{L}_{j+1}, \mathcal{L}_{j+2}, \ldots, \mathcal{L}_n$. We iterate the process of searching for new occurrences of handle graphs with CF labels, extending the levels, and running the DIAGEN parser as long as levels are modified. Finally, the information in the resulting levels together with the added references are used to create derivations in a top-down fashion. ◇

## 4. Nested Graph Transformation

Although graph transformation as defined in Section 2 is computationally complete, it needs to be extended in order to be adequate for programming with graphs and diagrams:

- Graphs need a *structuring concept* so that they can be composed and decomposed like data structures in programming languages.
- Rules need a mechanism by which subcomponents can be moved, deleted, and duplicated.

Therefore we allow that graphs can be nested, and that rules may contain variables that bind arbitrary nested subgraphs. In contrast to notions of *hierarchical graphs* that are used for system modeling [12], our nesting concept is *compositional*: it forbids edges between components so that component assignment is possible.

**Nested Graphs.** As defined below, a nested graph is a tree of graphs called packages. Although put in a different way, this definition is equivalent to that in [13].

In the following, let $X$ be an alphabet of *variable names* that is disjoint with $\Lambda \cup N$. An edge labelled with a variable name is called a *variable*. Let $\mathcal{G}$ denote the set of all graphs with edge labels in $\Lambda \cup N \cup X$. A *nested graph* is a mapping $G\colon \Delta \to \mathcal{G}$ such that

- $\Delta$ is a prefix-closed[3] set of *locations*, each location being a finite sequence of edges, and
- for every edge $e$ and every location $we \in \Delta$ it holds that $e \in E_{G(w)}$ and $\mathrm{lab}_{G(w)}(e) \in \Lambda$.

For every location $w \in \Delta$, $G(w)$ is called the *package at location* $w$. The locations of a nested graphs $G$ are often denoted by $\Delta_G$. The nested graph $G/w\colon \Delta_w \to \mathcal{G}$ is the *component* of $G$ at location $w \in \Delta$, where $\Delta_w = \{w' \mid ww' \in \Delta\}$ and $(G/w)(w') = G(ww')$ for all $w' \in \Delta_w$.

Nested graphs $G$ and $H$ are *isomorphic*, written $G \cong H$, if there is an isomorphism $m\colon G(\varepsilon) \to H(\varepsilon)$ such that, for every $e \in E_{G(\varepsilon)} \cap \Delta_G$, $G/e \cong H/m_E(e)$.

**Example 4.1 (Nested Control Flow Graphs).** The control flow graphs of Example 2.2 can be extended by designating call edges to contain control flow graphs, recursively. Figure 7 below shows three *nested control flow graphs*. Components are drawn inside the boxes of the edges containing them. ◇

[3] A set $\Delta \subseteq E^*$ of sequences over some set $E$ is *prefix-closed* if for all $w, w' \in E^*$, $ww' \in \Delta$ implies $w \in \Delta$.

**Variable Replacement.** The transformation of nested graphs is based on the notion of replacing variable edges by graphs. For this, let $G$, $H$ be nested graphs such that some package $G(w)$ at location $w \in \Delta_G$ contains a variable edge $e$ with $\mathrm{arity}(\mathrm{lab}_{G(w)}(e)) = \mathrm{arity}(H)$. The *replacement* of $e$ by $H$ is written $G[we/H]$; it yields the nested graph $G'$ with $\Delta_{G'} = \Delta_G \cup \{ww' \mid w' \in \Delta_H\}$ whose packages are given as follows:

- for all $w'' \in \Delta_G \setminus \{w\}$, $G'(w'') = G(w'')$,
- $G'(w)$ is obtained from $G(w)$ by applying the CF rule $\langle \mathrm{lab}_G(e) \rangle ::= H$ to $e$, and
- for all $w'' = ww' \in \Delta_{G'}$ with $w' \neq \varepsilon$, $G'(w'') = H(w')$.

A function $\sigma\colon X \to \mathbb{G}$ is a *substitution* if it maps variable names to nested graphs with the same arity. The *instantiation* of a nested graph $G$ according to $\sigma$ is obtained by replacing each variable $e \in E_{G(w)}$ ($w \in \Delta_G$) by the graph $\sigma(\mathrm{lab}_{G/w}(e))$; the resulting *instance graph* is denoted by $G\sigma$.

A nested graph $C$ with a single variable $e \in E_{G(w)}$ (with $w \in \Delta_G$) is called a *context*.

**Graph Transformation.** Using the notions summarized above, transformation rules and steps can be defined similar to term rewriting [3]. In doing so, it seems sensible to apply a similar restriction as for the rules of a term rewrite system: Only variable names from the left hand side are allowed to occur in the right hand side because otherwise arbitrary substructures would be created "out of thin air". (Further restrictions of the use of variables in transformation rules, which are made in [16] for the efficient construction of transformation steps, are not relevant in this paper and are therefore omitted here.)

A *nested transformation rule* $t = L/R$ consists of two nested graphs $L$ and $R$ of the same arity such that only variable names from $L$ occur in the right hand side $R$. Then $t$ *transforms* a graph $G$ into another graph $H$, written $G \Rightarrow_t H$, if $L$ and $R$ can be instantiated with a substitution $\sigma$, and embedded into some context $C$ so that $G \cong C[L\sigma]$ and $H \cong C[R\sigma]$.

**Example 4.2 (Control Flow Graph Transformation).** In Figure 6 we extend the rule *loop* of Example 2.2 so that it applies to any kind of loop body (i.e., by instantiating the variable $D$). In addition, the rule *unfold* unfolds the body of a procedure call. Figure 7 shows two transformations of a control flow graph using *loop* and *unfold*. The context $C$ for the first step

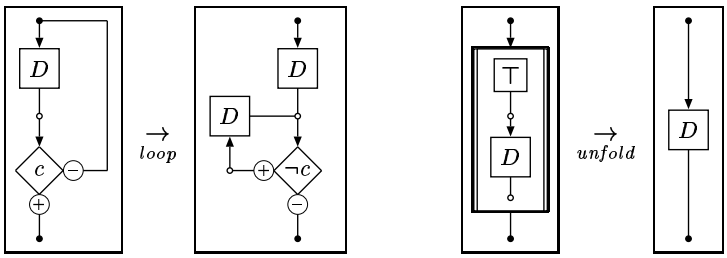


Fig. 6. Rules for general loop transformation and for unfolding procedures

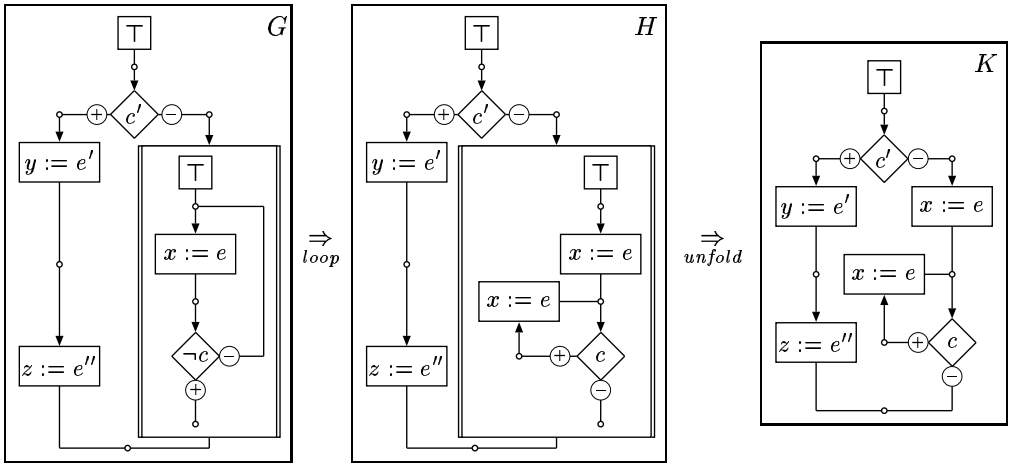


Fig. 7. Two transformation steps

has packages $C(\varepsilon) = G(\varepsilon)$ and $C/f = \langle A \rangle$ (where $\Delta_G = \Delta_C = \{\varepsilon, f\}$, and $\langle A \rangle$ is the handle graph of a binary variable named $A$), and the substitution maps $D$ to $\langle x := e \rangle$. The context for the second step equals $G[f/\langle A \rangle]$, and the substitution instantiates $D$ to $H/f$.

Let $loop^{-1}$ and $unfold^{-1}$ be obtained by interchanging the sides in rules $loop$ and $unfold$. Then $unfold^{-1}$ is a very general rule that can always be applied, to fold any control flow graph with a single entry and exit to a procedure call; $loop^{-1}$ is a transformation rule that could transform $H$ back to $G$.

Note that a single graph transformation may affect arbitrarily large subgraphs of the host graph. Every application of $loop$ duplicates the subgraph bound to the variable name $D$. Similarly, a rule deletes the subgraph bound to a variable name in its left hand side if that name does not occur in its right hand side. And, a rule may require to compare arbitrarily large subgraphs: the rule $loop^{-1}$ applies only to a host graph like $H$, where both $D$-variables on its left hand side match isomorphic subgraphs.

◇

## 5. Shapely Nested Graph Transformation

Nested graph transformation is defined for arbitrary nested graphs, although the graphs that actually occur have particular properties. Now we use the shape rules of Section 3 to impose a structural typing on transformations. For this, we employ the following assumptions.

**Assumption 5.1 (Shapes for Nested Graphs).** In the remainder of this paper, we consider a finite CE set $\Sigma$ of shape rules. We assume furthermore that the alphabets $\Lambda$ and $X$ are equipped with the following shape specifications:

1. every variable name $x \in X$ comes with a *substitution shape* $\text{sub-shape}(x) \in N$ such that $\text{arity}(x) = \text{arity}(\text{sub-shape}(x))$, and
2. every label $l \in \Lambda$ comes with a *contents shape* $\text{cts-shape}(l) \in N$.

The *shape graph* $\lceil G \rceil$ of a graph $G$ is the graph obtained by relabelling every variable $e$ in $G$ by the corresponding substitution shape $\text{sub-shape}(\text{lab}_G(e))$. Below, shapedness of nested graphs is defined packagewise. If $we$ is a location then the contents shape of (the label of) $e$ determines the shape of the package $G(we)$. Thus, a nested graph is shaped if all its individual packages are shaped. This makes it rather straightforward to extend notions and results from the previous sections to shaped nested graphs.

A nested graph $G$ is *shaped* according to some nonterminal $n \in N$, written $\Sigma \vdash G : n$, if

- $\Sigma \vdash \lceil G(\varepsilon) \rfloor : n$ for the root package $G(\varepsilon)$, and
- for all $we \in \Delta_G$, the graph $G(we)$ satisfies $\Sigma \vdash \lceil G(we) \rfloor : \text{cts-shape}(\text{lab}_{G(w)}(e))$.

**General Conformance of Transformations to Shapes.** Given the powerful mechanisms to specify context-free or context-exploiting shapes, we turn to the question whether sets of graph transformation rules preserve the shapes of graphs. As shown by Fradet and LeMétayer in [7, Section 4.1] (even for CF shape rules, graphs without nesting, and rules without variables), this question cannot be decided:

**Fact 5.1.** Given a CE set of shape rules $\Sigma$ and a nested transformation rule $t$ as input, the following property is undecidable in general:

Does $\Sigma \vdash H : n$ hold for all nested graphs $G, H$ with $\Sigma \vdash G : n$ and $G \Rightarrow_t H$?

To overcome this problem, we now refine nested transformation rules, context embedding, and rule instantiation in such a way that it preserves shapes.

**Shapely Transformation.** A variable replacement $G[we/H]$ with components as above is *shaped* if $\Sigma \vdash \lceil G \rfloor : n$ for some $n \in N$ and $\Sigma \vdash \lceil H \rfloor : \text{sub-shape}(\text{lab}_{G(w)}(e))$. As a rather direct consequence of these definitions, we get the following lemma, which we state without the straightforward proof.

**Lemma 5.1.** $\Sigma \vdash G : n$ and $\Sigma \vdash H : \text{sub-shape}(\text{lab}_{G(w)}(e))$ implies $\Sigma \vdash G[we/H] : n$.

A context $C$ is called an *$n$-context* if $\Sigma \vdash C : n'$ for some $n' \in N$ and the unique variable $e \in E_{G(w)}$ ($w \in \Delta_C$) satisfies $\text{sub-shape}(\text{lab}_{G(w)}(e)) = n$. (Note that in general, $n \neq n'$.)

A substitution $\sigma$ is *shaped* if $\Sigma \vdash \sigma(x) : \text{sub-shape}(x)$ for all $x \in X$. A nested transformation rule $t = L/R$ is *shaped* if $L$ and $R$ are shaped by the same nonterminal, say $n$. Then $t$ *transforms* a nested graph $G$ into another nested graph $H$, written $G \Rrightarrow_t H$, if $L$ and $R$ can be instantiated with a shaped substitution $\sigma$, and embedded into some $n$-context $C$ so that $G \cong C[L\sigma]$ and $H \cong C[R\sigma]$.

This yields the desired theorem: The result of a shaped transformation is shaped.

**Theorem 5.1.** If $G \Rrightarrow_t H$ using context $C$ with $\Sigma \vdash G : n$ and $\Sigma \vdash C : n$ for some nonterminal $n \in N$, then $\Sigma \vdash H : n$.

*Proof sketch.* Lemma 5.1 implies that the instantiation of a shaped nested rule preserves its shape, and that the embedding of that instance into a context preserves shapes as well. Thus the step $G \Rrightarrow_t H$ preserves shapes. ∎

Altogether, shapes set up a *type discipline* that can be *statically checked*: By Theorems 3.1 and 3.2 it is possible to decide whether a set $T$ of transformation rules is shaped or not. If the rules are shaped, and a graph $G$ (the "input") has been checked to be shaped, Theorem 5.1 guarantees that every transformation sequence $G \Rightarrow^* H$ will yield a shaped "output" graph $H$ that—as long as contexts of the same shape as $G$ are selected for each transformation step—has the same shape as $G$. Type-checking between the steps ("at runtime") is not necessary.

**Example 5.1 (Shapely Control Flow Graph Transformations).** The nested graphs in Figure 6 of Example 4.2 are shaped according to $\alpha$ with respect to the CF shape rules of example 3.1 resp. Figure 3 (and hence to the CE shape rules of Figure 4 as they are an extension of the CF ones), so the rules $loop$, $unfold$, and $loop^{-1}$ are shaped. The contexts of the transformations in Figure 7 are $\pi$-contexts, and the matching substitutions are shaped so that the transformations are shaped as well. ◇

## 6. Conclusions

In this paper, we have discussed structural types for nested graphs, called *shapes*. In particular, we have proposed *context-exploiting shapes*, by which many interesting graph languages can be specified that are relevant for diagram representation and programming with graphs. Adherence to shapes can be checked automatically. Nested graph transformation, a rather expressive way of graph transformation with graph variables, can be refined so that the application of *shaped rules* preserves shapes. This provides a static type discipline that can be directly exploited to increase the efficiency of diagram transformation: if the transformations used in a DIAGEN editor or a DIAPLAN program are shaped, their results always stay in the diagram language, making type checks at runtime obsolete.

Graph transformation often comes with some kind of typing. In typed graph grammars [4], the label of an edge determines the labels of the nodes to which it may be incident. PROGRES [11] supports multiple inheritance, and allows us to constrain the cardinality of nodes, or the degree of incident edges of a node. Such typing is *atomic* as it can be easily checked by inspecting the nodes and edges, one after the other. Labelling conditions can be checked statically, while subtyping and cardinality constraint require dynamic checks. Structural typing goes beyond atomic typing. We did already mention shape analysis [8] where shapes of pointer structures like leaf-connected trees are specified implicitly, by some kind of logic. Since programs ("rules") are not constrained, their adherence to shapes cannot be statically checked (see Theorem 5.1), but has to be proved for every individual program.

P. Fradet and D. LeMétayer [7] specify shapes for *Structured Gamma*, a rule-based language for nondeterministic programming of multisets (which are badly disguised graphs). They give a type checking algorithm for unrestricted rules that is sound, but necessarily incomplete.

A. Bakewell, D. Plump, and C. Runciman [15] study *Church-Rosser* (*CR*) shape rules where the only requirement is that their application from right to left is terminating and confluent. Thus, CR shape rules can probably describe data structures that are not CE shapes. However, CR shapes require a proof that the rules are indeed terminating and confluent. This is not always easy and cannot be done in an automated way, in general. Moreover, Lemma 5.1 does not hold for CR shape rules so that it is not clear how a reasonable static type discipline can be set up for transformation rules with variables.

Future work will address the relation of CE shapes to CR shapes. Are there practically important CR shapes that are not CE? Is it sensible to combine the defining requirements of CR and CE shapes? Such a combination may result in a subclass of CE shapes for which parsing can be done more efficiently.

The implementation of context-exploiting shape rules in DIAGEN is under way. Given the existing implementation for the case of context-free rules, this should be straightforward.

Some work remains to be done for the design of DIAPLAN. Let us just mention where the graph model has to be extended.

- Not only for the sake of design orthogonality, we must extend the approach so that nodes can contain packages as well. Then we can define abstractions as predicates (edges) that are applied to graphs (node packages), in the style of [18].
- We need an *attribute concept* for associating nodes and edges with primitive values like numbers and strings. This can be easily achieved, if frames may contain "one-dimensional" values as well as graphs. We have silently used such attributes in our running example: Assignment edges contain strings, or, more precisely, elements of a CF string language of statements; similarly, branches contain Boolean expressions. Transformation rules may contain expressions over those values, and may perform computations on them.

Finally, DIAPLAN has still to be implemented, using DIAGEN and DIAPLAN—~~w~~what else?

## References

**1979**

[1] H. Ehrig. Introduction to the algebraic theory of graph grammars. 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 1–69. Springer.

**1992**

[2] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer.

[3] 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.

**1996**

[4] A. Corradini, H. Ehrig, M. L¨owe, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunction with categories of derivations. 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 56–74. Springer.

**1997**

[5] F. Drewes, 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*, chapter 2, pages 95–162. World Scientific, Singapore.

[6] M. Minas. Diagram editing with hypergraph parser support. In *Proc. 13th IEEE International Symposium on Visual Languages (VL'97), Capri, Italy*, pages 230–237. IEEE Computer Society Press.

**1998**

[7] P. Fradet and D. Le M´etayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289.

[8] 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.

**1999**

[9] R. Bardohl, M. Minas, A. Sch¨urr, and G. Taentzer. Application of graph transformation to visual languages. In Engels et al. [10], chapter 3, pages 105–180.

[10] 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.

[11] A. Sch¨urr, A. Winter, and A. Z¨undorf. The PROGRES approach: Language and environment. In Engels et al. [10], chapter 13, pages 487–550.

**2000**

[12] 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)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer.

**2001**

[13] B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press.

[14] M. Minas and B. Hoffmann. Specifying and implementing visual process modeling languages with DIAGEN. *Electronic Notes in Theoretical Computer Science*, 44(4).

**2002**

[15] A. Bakewell, D. Plump, and C. Runciman. Specifying pointer structures by graph reduction. Technical report, Department of Computer Science, University of York.

[16] F. Drewes, B. Hoffmann, and M. Minas. Constructing shapely nested graph transformations. In H.-J. Kreowski and P. Knirsch, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*, pages 107–118.

[17] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64:249–283.

[18] B. Hoffmann. Abstraction and control for shapely nested graph transformation. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *First International Conference on Graph Transformation (*ICGT*'02)*, number 2505 in Lecture Notes in Computer Science, pages 177–191. Springer.

[19] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44:157–180.