Journal of Computer and System Sciences **64**, 249–283 (2002)
doi:10.1006/jcss.2001.1790, available online at http://www.idealibrary.com on IDEAL®

Hierarchical Graph Transformation[1]

Frank Drewes

Institutionen för Datavetenskap, Umeå Universitet, S-90187 Umeå, Sweden
E-mail: drewes@cs.umu.se

Berthold Hoffmann

Fachbereich Mathematik/Informatik, Universität Bremen, Postfach 33 04 40, D-28334 Bremen, Germany
E-mail: hof@informatik.uni-bremen.de

and

Detlef Plump

Department of Computer Science, The University of York, York YO10 5DD, United Kingdom
E-mail: det@cs.york.ac.uk

Received April 3, 2000; revised September 17, 2001

When graph transformation is used for programming purposes, large graphs should be structured in order to be comprehensible. In this paper, we present an approach for the rule-based transformation of hierarchically structured hypergraphs. In these graphs, distinguished hyperedges contain graphs that can be hierarchical again. Our framework extends the well-known double-pushout approach from flat to hierarchical graphs. In particular, we show how pushouts and pushout complements of hierarchical graphs and graph morphisms can be constructed recursively. Moreover, we make rules more expressive by introducing variables which allow us to copy and remove hierarchical subgraphs in a single rule application. © 2002 Elsevier Science (USA)

Key Words: hierarchical graphs; graph transformation.

Contents.

[1] This work has been partially supported by the ESPRIT Working Group APPLIGRAPH and the TMR Research Network GETGRATS. A short version of this paper appeared in the proceedings of FOSSACS 2000; see [14].

0. INTRODUCTION

The transformation of graphs by rules has been systematically investigated since about 1970 and has been applied in several areas of Computer Science—see the recent handbook volumes [17, 18, 40]. Graph transformation has been used, for example, as an efficient computational model for term rewriting systems and functional programming languages [5, 34] and for specifying visual languages and generating associated editors [4, 31]. Besides specific applications of graph transformation, several programming languages have been developed that are based on graph transformation rules. Examples of such languages are Progres [43], Agg [22], Gamma [3], Grrr [39], and Dactl [24].

Programming tasks of a realistic size require large numbers of rules, making it imperative to develop systems from small components that are easy to comprehend. Recently, structuring concepts for systems of graph transformation rules have received some attention. A module concept for graph transformation is proposed in [2], for example, and different proposals for module concepts of graph transformation systems are compared with each other in [27].

We believe that it will be necessary to structure not only the sets of rules but also the graphs that are subject to transformation, in order to cope with complex applications. A mechanism for hiding subgraphs—or abstracting from subgraphs—will allow us to visualize large graphs and to make them comprehensible. Moreover, such a structuring mechanism will support the control of rule applications. Graphs with a hierarchical structure have already been used in CASE tools [46] and in data base languages like Hyperlog [36].

In this paper we introduce hierarchical hypergraphs in which certain hyperedges, called frames, contain hypergraphs that can be hierarchical again, with an arbitrary depth of nesting. We show that the double-pushout approach to graph transformation [11, 15] extends smoothly to these hierarchical hypergraphs, by giving recursive constructions for pushouts and pushout complements in the category of hierarchical graphs. Hierarchical transformation rules consist of hierarchical graphs and can be applied at all levels of the hierarchy, where the "dangling condition" known from the transformation of flat graphs is adapted in a natural way.

To the best of our knowledge, this is the first time that one of the existing approaches to graph transformation has been extended by a hierarchy concept for graphs. Our extension lifts basic results on the existence and uniqueness of direct derivations in a natural way to the hierarchical case and is conservative in that on unstructured graphs the approach coincides with the conventional double-pushout approach.[2] This provides some confidence that most of the theory of the double-pushout approach will extend to the new framework in a straightforward way.

We also introduce rule schemata containing frame variables to make hierarchical graph transformation more expressive for programming purposes—without damaging the theory. These variables can be instantiated with frames containing hierarchical graphs and can be used to copy or remove frames without looking at

[2] To be precise, our approach extends the double-pushout approach with injective matching which is thoroughly studied in [26].

their contents. In particular, this makes it possible to move a copy of a frame into another frame. Our running example of a list implementation indicates that this concept is useful, as list entries can be entered, deleted, or looked up, regardless of their structure and size.

Finally, we relate our notion of hierarchical graph transformation to the conventional transformation of flat graphs by introducing a flattening operation. Flattening recursively replaces each frame in a hierarchical graph by its contents, using hyperedge replacement. The result is a flat graph without frames. Under a mild assumption on the transformed graph, every transformation step on hierarchical graphs gives rise to a conventional step on the flattened graphs by using the flattened rule.

We would like to stress that the strict hierarchical structure of our graphs provides certain advantages over approaches that allow edges between different component graphs, like [6, 21]. Edges across the component structure cause complications when it comes to reasoning about programs. In particular, they prevent compositionality of the form that component graphs can be freely replaced with equivalent components. Consider, for example, the equivalence on hierarchical graphs generated by some set of transformation rules. In our hierarchical setting it is no problem to replace some component graph by an equivalent graph: by the way we define hierarchical graph transformation it will be obvious that the resulting overall graph is equivalent to the original graph. In contrast, if there are edges across components, it is not even clear how to replace components.

The rest of this paper is structured as follows. In the next section, we recall the double-pushout approach to graph transformation. Hierarchical graphs and their morphisms are introduced in Section 2, and some key properties of the category obtained in this way are studied. These results are used in Section 3 to define the notion of hierarchical graph transformation and to prove that it behaves well. We introduce hierarchical graph transformation with variables in Section 4. In Section 5, a nontrivial example (*quicksort*) illustrates how hierarchical graph transformation can be used for programming. The flattening operation is studied in Section 6. Finally, in Section 7 and Section 8, we discuss some related work and outline directions for future work.

1. GRAPH TRANSFORMATION

If S is a set, the set of all finite sequences over S, including the empty sequence λ, is denoted by S^*. The ith element of a sequence s is denoted by $s(i)$, and its length by $|s|$. If $f\colon S \to T$ is a function then the canonical extensions of f to the powerset of S and to S^* are also denoted by f. The composition $g \circ f$ of functions $f\colon S \to T$ and $g\colon T \to U$ is defined by $(g \circ f)(s) = g(f(s))$ for $s \in S$.

We will deal with directed, edge-labelled hypergraphs in which the label of a hyperedge determines the number of the incident nodes. A *label alphabet* (or *colour alphabet*) is a set $\mathscr{C}$ such that each $l \in \mathscr{C}$ comes with a natural number $type(l) \geqslant 0$. For the rest of this paper, we consider a fixed label alphabet $\mathscr{C}$.

Definition 1.1 (Hypergraph). A *hypergraph* $H = (V_H, E_H, att_H, lab_H)$ is a system consisting of two finite sets V_H and E_H of *nodes* (or *vertices*) and *hyperedges*, a *labelling function* $lab_H: E_H \to \mathscr{C}$, and an *attachment function* $att_H: E_H \to V_H^*$ such that, for each hyperedge e, $|att_H(e)| = type(lab_H(e))$.

In the following, we simply say *graph* instead of hypergraph and *edge* instead of hyperedge. We denote by A_H the set $V_H \cup E_H$ of *atoms* of H. In order to make this a useful notation, we shall always assume without loss of generality that V_H and E_H are disjoint, for every graph H.

Remark. In order to avoid confusion we remark that the hypergraphs used in the short version of this paper [14] come with an additional component, namely a sequence of distinguished nodes called *points*. These are useful in connection with hyperedge replacement [13], which is employed by the flattening process considered in Section 6. However, as long as one is not interested in flattening, and in particular as far as the basic theory is concerned, points are of no particular value—they just complicate the whole theory in an unnecessary way. This is the reason why they are omitted in this paper until Section 6.

Example 1.1 (List Graphs). In our running example, we show how lists can be represented as graphs, and how some of their typical operations can be implemented using graph transformation. Two kinds of edges are used to represent lists as graphs: Unary I-edges designate the *item graphs* stored in the lists; binary L-edges designate the start and end node of a *list graph*. The latter consist of a chain of nodes connecting the *start* point with the *end* point, where a unique item edge is attached to every node in between.

Figure 1 shows two list graphs. Nodes are drawn as circles. Edges are drawn as boxes and are connected with their attachment nodes by lines which are ordered counter-clockwise, starting at noon. Plain binary edges are drawn as arrows from their first to their second attachment node (as in simple graphs); their labels do not matter in our examples and hence are omitted. In the item graphs, the arrowheads are omitted, too.

A *morphism* $m: G \to H$ between graphs G and H is a pair (m_V, m_E) of mappings $m_V: V_G \to V_H$ and $m_E: E_G \to E_H$ such that for all $e \in E_G$, $lab_H(m_E(e)) = lab_G(e)$ and $att_H(m_E(e)) = m_V(att_G(e))$. Such a morphism is *injective* (*surjective*, *bijective*) if both m_V and m_E are injective (respectively surjective or bijective). If there is a bijective morphism $m: G \to H$ then G and H are *isomorphic*, which is denoted by $G \cong H$. For a morphism $m: G \to H$ and $a \in A_G$ we let $m(a)$ denote $m_V(a)$ if $a \in V_G$ and $m_E(a)$ if $a \in E_G$. The composition of morphisms is defined componentwise.

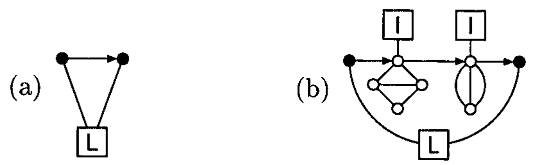


FIG. 1. Two list graphs representing (a) an empty list, (b) a list of length 2.

For graphs G and H such that $A_G \cap A_H = \varnothing$, the *disjoint union* $G+H$ yields the graph $(V_G \cup V_H, E_G \cup E_H, att, lab)$, where

$$att(e) = \begin{cases} att_G(e) & \text{if} \quad e \in E_G \\ att_H & \text{otherwise} \end{cases} \quad \text{and} \quad lab(e) = \begin{cases} lab_G(e) & \text{if} \quad e \in E_G \\ lab_H(e) & \text{otherwise} \end{cases}$$

for all edges $e \in E_G \cup E_H$. (If the assumption $A_G \cap A_H = \varnothing$ is not satisfied, it is assumed that some implicit renaming of atoms takes place.)

A *pushout* in a category $\mathbb{C}$ (for the definition of categories see, e.g., [1]) is a tuple (m_1, m_2, n_1, n_2) of morphisms $m_i\colon O \to O_i$ and $n_i\colon O_i \to O'$ with $n_1 \circ m_1 = n_2 \circ m_2$, such that for all morphisms $n_i'\colon O_i \to P$ $(i \in \{1, 2\})$ with $n_1' \circ m_1 = n_2' \circ m_2$ there is a unique morphism $n\colon O' \to P$ satisfying $n \circ n_1 = n_1'$ and $n \circ n_2 = n_2'$. Depicted as a diagram, this looks as follows:

$$\begin{tikzcd}
O \arrow[r, "m_1"] \arrow[d, "m_2"'] & O_1 \arrow[d, "n_1"'] \arrow[ddr, "n_1'"] & \\
O_2 \arrow[r, "n_2"] \arrow[drr, "n_2'"'] & O' \arrow[dr, "\exists!\, n"] & \\
& & P
\end{tikzcd}$$

We recall the following well-known facts about pushouts and pushout complements in the category of graphs and graph morphisms (see [15]). Let $m_1\colon G \to H_1$ and $m_2\colon G \to H_2$ be morphisms. Then there is a graph H and there are morphisms $n_1\colon H_1 \to H$ and $n_2\colon H_2 \to H$ such that (m_1, m_2, n_1, n_2) is a pushout. Furthermore, H and the n_i are determined as follows. Let H' be the disjoint union of H_1 and H_2, and let $\sim$ be the equivalence relation on $A_{H'}$ generated by the set of all pairs $(m_1(a), m_2(a))$ such that $a \in A_G$. Then H is the graph obtained from H' by identifying all atoms a, a' such that $a \sim a'$ (in other words, H is the quotient graph $H'/\sim$). Moreover, for $i \in \{1, 2\}$ and $a \in A_{H_i}$, $n_i(a) = [a]_\sim$, where $[a]_\sim$ denotes the equivalence class of a according to $\sim$.

In order to ensure the existence and uniqueness of pushout complements (that is, the existence and uniqueness of m_2 and n_2 if m_1 and n_1 are given), additional conditions must be satisfied. Below, we are only concerned with the case where both of the given morphisms are injective. In this case it is sufficient to assume that the *dangling condition* is satisfied. Let $m_1\colon G \to H_1$ and $n_1\colon H_1 \to H$ be morphisms. Then n_1 is said to satisfy the dangling condition with respect to m_1 if no edge $e \in E_H \setminus n_1(E_{H_1})$ is attached to a node in $n_1(V_{H_1}) \setminus n_1(m_1(V_G))$. It is well known that if m_1 and n_1 are injective, then there are m_2 and n_2 such that (m_1, m_2, n_1, n_2) is a pushout, if and only if n_1 satisfies the dangling condition with respect to m_1. Furthermore, if they exist, then m_2 and n_2 are uniquely determined (up to isomorphism).

Definition 1.2 (Transformation Rule). A *transformation rule* (*rule*, for short) is a pair $t\colon L \leftarrow I \rightarrow R$ of morphisms $l\colon I \to L$ and $r\colon I \to R$ such that l is injective. The hypergraphs L, I, and R are called the *left-hand side*, *interface*, and *right-hand side* of t, respectively.

Note that the morphism $r: I \to R$ is not required to be injective. This allows us to merge (identify) nodes or edges in transformations.

Definition 1.3 (Graph Transformation). A rule $t: L \leftarrow I \rightarrow R$ *transforms* a graph G into a graph H, denoted by $G \Rightarrow_t H$, if there are an injective morphism $o: L \to G$, called an *occurrence morphism*, and two pushouts of the following form:

$$\begin{tikzcd}
L \arrow[d, "o"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\
G & K \arrow[l] \arrow[r] & H
\end{tikzcd}$$

It follows from the facts about pushouts and pushout complements recalled above that these pushouts exist if and only if o satisfies the dangling condition with respect to l, and in this case G, t, and o determine H uniquely up to isomorphism.

We require that occurrence morphisms are injective to avoid additional difficulties for the hierarchical case. This is because non-injective occurrence morphisms have to satisfy an identification condition that is not easily extended to hierarchical morphisms. Requiring injective occurrence morphisms is no restriction, however, as graph transformation with non-injective occurrence morphisms can be simulated by using quotient rules. More precisely, a rule $t': L' \leftarrow I' \rightarrow R'$ is a *quotient* of a rule $t: L \leftarrow I \rightarrow R$ if there are two pushouts of the form

$$\begin{tikzcd}
L \arrow[d] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
L' & I' \arrow[l] \arrow[r] & R'
\end{tikzcd}$$

where the vertical morphisms are surjective. In [26] it is shown that every transformation using a non-injective occurrence morphism and a rule t can be simulated by a transformation using an injective occurrence morphism and a quotient of t. In our framework, graph transformation with non-injective occurrence morphisms corresponds to the special case that the set of rules is closed under the above quotient construction. (Note that for a finite set of rules, the latter can always be achieved by adding a finite number of quotients.) But we shall also give examples that make sense only if some quotients are omitted. Thus, injective occurrence morphisms allow us to control rule applications in a finer way than in the conventional approach. Precise results on the expressiveness gained by injective occurrence morphisms are given in [26].

Example 1.2 (Concatenation of List Graphs). In Fig. 2, we specify a concatenation rule for list graphs and show a transformation with this rule. The rule concatenates two list graphs by identifying their L-edges, start nodes, and end nodes, respectively. The digits and the letters a, b indicate the morphisms from the interface to the left- and right-hand side.

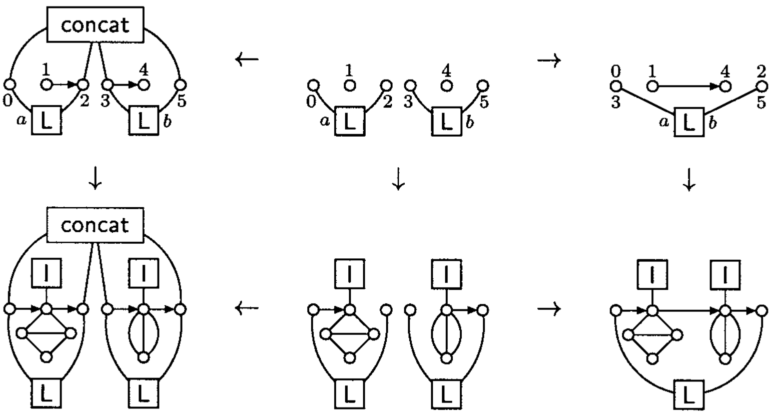


FIG. 2. The concatenation rule and its application.

As occurrence morphisms are injective, we need quotients of this rule, for the node identifications $0 = 1$, or $4 = 5$, or both of them since the list graphs may be empty. For instance, the quotient rule given by $0 = 1$ concatenates the empty list graph and a nonempty one.

A transformation step with another quotient rule is shown in Fig. 3. Here $a = b$, i.e., both list edges are identified, and their attachments as well.

This quotient rule produces a malformed list graph wherein the last item node is connected to the first item node, and the start and end nodes are isolated. This indicates that quotients need to be added with care; this particular quotient must be excluded. Note that if we had not required that occurrence morphisms are injective then we had no means to prevent undesirable transformations of this kind.

2. HIERARCHICAL GRAPHS

Graphs as defined in the previous section are flat. If some complicated abstract data type shall be implemented by graphs and graph transformation, no mechanism

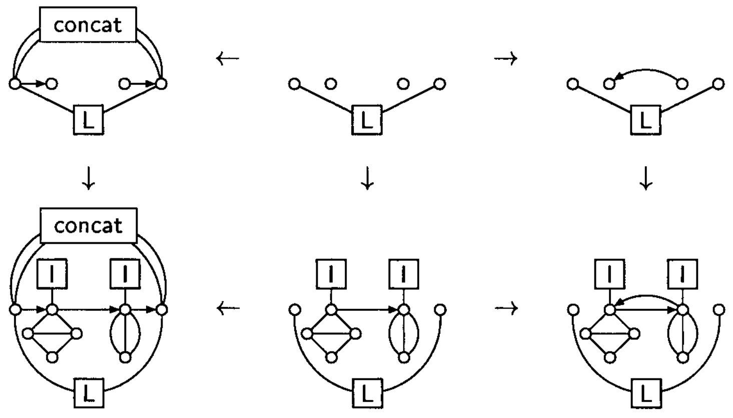


FIG. 3. A quotient of the concat rule and its application.

supports the structuring of these graphs, except for the means that graphs provide themselves. Thus, any structural information has to be coded into the graphs, a solution which is often inappropriate and error-prone. To overcome this limitation, we introduce graphs with an arbitrarily deep hierarchical structure. This is achieved by means of special edges, called frames, which may contain hierarchical graphs again. Moreover, it will be useful to allow some frames to contain variables instead of graphs. The resulting structures will be called hierarchical graphs. Thus, a hierarchical graph consists of a graph, the root of the hierarchy, a designated subset of its edges, the frames, and a mapping assigning to each frame its contents, which is either a hierarchical graph or a variable.

DEFINITION 2.1 (Hierarchical Graph). Let $\mathscr{X}$ be a set of symbols called *variables*. The class $\mathscr{H}(\mathscr{X}) = \bigcup_{i \geqslant 0} \mathscr{H}_i(\mathscr{X})$ of *hierarchical graphs with variables in* $\mathscr{X}$ consists of triples $H = \langle G, F, cts \rangle$ such that G is a graph, $F \subseteq E_G$ is the set of *frame edges* (or just *frames*), and $cts\colon F \to \mathscr{H}(\mathscr{X}) \cup \mathscr{X}$ assigns to each frame $f \in F$ its *contents* $cts(f) \in \mathscr{H}(\mathscr{X}) \cup \mathscr{X}$.

The sets $\mathscr{H}_i(\mathscr{X})$ are defined inductively, as follows. A triple $H = \langle G, F, cts \rangle$ as above is in $\mathscr{H}_0(\mathscr{X})$ if $F = \varnothing$. For $i > 0$, $H \in \mathscr{H}_i(\mathscr{X})$ if $cts(f) \in \mathscr{H}_{i-1}(\mathscr{X})$ for every frame $f \in F$.

Notice that $\mathscr{H}_i(\mathscr{X}) \subseteq \mathscr{H}_{i+1}(\mathscr{X})$ for all $i \geqslant 0$. This is due to the fact that $\mathscr{H}_0(\mathscr{X}) \subseteq \mathscr{H}_1(\mathscr{X})$, as an empty set of frames trivially satisfies the requirement; using this, $\mathscr{H}_i(\mathscr{X}) \subseteq \mathscr{H}_{i+1}(\mathscr{X})$ follows by an obvious induction on i. In the following, a hierarchical graph $\langle G, F, cts \rangle \in \mathscr{H}_0(\mathscr{X})$ will be identified with the graph G. The sets $\mathscr{H}(\varnothing)$ and $\mathscr{H}_i(\varnothing)$ $(i \geqslant 0)$ are briefly denoted by $\mathscr{H}$ and $\mathscr{H}_i$, respectively, and the hierarchical graphs in these sets are said to be *variable-free*.

EXAMPLE 2.1 (Hierarchical List Graphs). To represent lists as hierarchical graphs, we turn l-edges and L-edges into frames that *contain* item graphs and list graphs, respectively. In order to make it easy to distinguish the end nodes of a list graph from the internal ones (i.e., those to which the item frames are attached), unary edges labelled with a special symbol • are attached to the end nodes. Figure 4 shows two list frames. Frames have double lines, and their contents are drawn inside. In our figures, we omit frame labels as list and item frames can be distinguished by their arity. Furthermore, the •-labelled edges are not explicitly drawn. Instead, the nodes they are attached to are filled.

Unless they are explicitly named, the three components of a hierarchical graph H are denoted by $\bar{H}$, F_H, and cts_H, respectively. The notations V_H, E_H, att_H, lab_H, and A_H are used as abbreviations denoting $V_{\bar{H}}$, $E_{\bar{H}}$, $att_{\bar{H}}$, $lab_{\bar{H}}$, and $A_{\bar{H}}$, respectively.

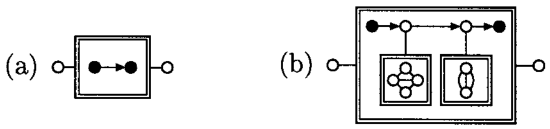


FIG. 4. Two list frames representing (a) an empty list, (b) a list of length 2.

Furthermore, we denote by X_H the set $\{f \in F_H \mid cts_H(f) \in \mathscr{X}\}$ of *variable frames* of H and by

$$var(H) = cts_H(X_H) \cup \bigcup_{f \in F_H \setminus X_H} var(cts_H(f))$$

the set of variables occurring in H.

Let G and H be hierarchical graphs such that $A_G \cap A_H = \varnothing$. The *disjoint union* of G and H is denoted by $G+H$ and yields the hierarchical graph K such that $\bar{K} = \bar{G} + \bar{H}$, $F_K = F_G \cup F_H$, and $cts_K(f)$ equals $cts_G(f)$ if $f \in F_G$ and $cts_H(f)$ if $f \in F_H$. For a hierarchical graph G and a set $S = \{H_1, ..., H_n\}$ of hierarchical graphs, we denote $G + H_1 + \cdots + H_n$ by $G + \sum_{H \in S} H$.

If H is a hierarchical graph, and $A \subseteq A_H$ is such that $e \in E_H \cap A$ implies $att_H(e) \in (V_H \cap A)^*$, then $H|_A$ denotes the *restriction of H by A*. More precisely, $H|_A$ is the hierarchical graph G such that $V_G = V_H \cap A$, $E_G = E_H \cap A$, att_G and lab_G are the restrictions of att_H and lab_H to E_G, and $F_G = F_H \cap A$ where $cts_G(f) = cts_H(f)$ for all $f \in F_G$. Notice that G is well-defined due to the assumptions on A.

We are now going to generalize the concept of a morphism to the hierarchical case. The definition is quite straightforward. A hierarchical morphism $h\colon G \to H$ consists of an ordinary morphism on the topmost level and, recursively, hierarchical morphisms from the contents of non-variable frames to the contents of their images. Naturally, only variable frames can be mapped to variable frames, but also to any other frame carrying the right label.

Definition 2.2 (Hierarchical Morphism). Let $G, H \in \mathscr{H}(\mathscr{X})$. A *hierarchical morphism* $h\colon G \to H$ is a pair $h = \langle \bar{h}, (h^f)_{f \in F_G \setminus X_G} \rangle$ where

- $\bar{h}\colon \bar{G} \to \bar{H}$ is a morphism,
- $\bar{h}(f) \in F_H$ for all frames $f \in F_G$, where $\bar{h}(f) \in X_H$ implies $f \in X_G$, and
- $h^f\colon cts_G(f) \to cts_H(\bar{h}(f))$ is a hierarchical morphism for every frame $f \in F_G \setminus X_G$.

In the following, we usually write $h(a)$ instead of $\bar{h}(a)$, for atoms $a \in A_G$. Furthermore, a hierarchical morphism $h\colon G \to H$ for which $G, H \in \mathscr{H}_0$ will be identified with $\bar{h}$.

Note that hierarchical morphisms respect and preserve the hierarchical structure. In particular, if $h\colon G \to H$ is a hierarchical morphism, then the image of G in H has the same hierarchical depth as G.

The composition $h \circ g$ of hierarchical morphisms $g\colon G \to H$ and $h\colon H \to L$ is defined in the obvious way. It yields the hierarchical morphism $l\colon G \to L$ such that $\bar{l} = \bar{h} \circ \bar{g}$ and, for all frames $f \in F_G \setminus X_G$, $l^f = h^{g(f)} \circ g^f$. The hierarchical morphism g is *injective* if $\bar{g}$ is injective and, for all $f \in F_G \setminus X_G$, g^f is injective. It is *surjective up to variables* if $\bar{g}$ is surjective and, for all $f \in F_G \setminus X_G$, g_f is surjective up to variables. Finally, g is *bijective up to variables* if it is injective and surjective up to variables. If G does not contain variables, we speak of surjective and bijective hierarchical morphisms. A bijective hierarchical morphism is also called an *isomorphism*, and G,

$H \in \mathscr{H}$ are said to be *isomorphic*, denoted by $G \cong H$, if there is an isomorphism $m: G \to H$.

Let $\mathbb{H}$ be the category whose objects are variable-free hierarchical graphs and whose morphisms are the hierarchical morphisms $h: G \to H$ with $G, H \in \mathscr{H}$. The main result we are going to establish in order to obtain a notion of hierarchical graph transformation is that $\mathbb{H}$ has pushouts. This is proved by induction on the depth of the frame nesting, where the induction basis is provided by the non-hierarchical case recalled in Section 1. In the induction hypothesis, we reduce the depth of a hierarchical graph by the following construction: Given a hierarchical graph $H \in \mathscr{H}_i$, we take the contents of its frames out of these frames (turning the frames into ordinary edges) and add them disjointly to $\bar{H}$, thus obtaining a hierarchical graph in $\mathscr{H}_{i-1}$ (provided that $i > 0$).

Formally, let $G \in \mathscr{H}$. We will assume, without loss of generality, that $A_G \cap A_{cts_G(f)} = \varnothing = A_{cts_G(f)} \cap A_{cts_G(f')}$ for all distinct $f, f' \in F_G$. On the basis of this assumption, we let $\varphi(G)$ denote the hierarchical graph $\bar{G} + \sum_{f \in F_G} cts_G(f)$ (where $\bar{G}$ is considered to be an element of $\mathscr{H}_0$). For a hierarchical morphism $h: G \to H$ (where $G, H \in \mathscr{H}$), $\varphi(h)$ denotes the hierarchical morphism $m: \varphi(G) \to \varphi(H)$ such that, for every atom $a \in A_{\varphi(G)}$,

$$m(a) = \begin{cases} h(a) & \text{if } \ a \in A_G \\ h^f(a) & \text{if } \ a \in A_{cts_G(f)} \text{ for some } f \in F_G, \end{cases}$$

and $m^{f'} = (h^f)^{f'}$ for every $f \in F_G$ and $f' \in F_{cts_G(f)}$.

The following lemma is needed for the construction of pushouts. It states conditions under which a given hierarchical morphism $m: \varphi(G) \to \varphi(H)$ can be turned into a hierarchical morphism $h: G \to H$ such that $\varphi(h) = m$.

Lemma 2.1. *Let $G, H \in \mathscr{H}$ with $G' = \varphi(G)$ and $H' = \varphi(H)$. If $m: G' \to H'$ is a hierarchical morphism such that*

(1) *for all $e \in E_G$, $m(e) \in F_H$ if and only if $e \in F_G$,*

(2) *$m(A_G) \subseteq A_H$, and*

(3) *for all $f \in F_G$, $m(A_{cts_G(f)}) \subseteq A_{cts_H(m(f))}$,*

then there is a hierarchical morphism $h: G \to H$ such that $\varphi(h) = m$. This hierarchical morphism is given by

- *$h(a) = m(a)$ for all $a \in A_G$,*
- *$h^f(a) = m(a)$ for all $f \in F_G$ and $a \in A_{cts_G(f)}$, and*
- *$(h^f)^{f'} = m^{f'}$ for all $f \in F_G$ and $f' \in F_{cts_G(f)}$.*

Proof. By the relevant definitions, if h is defined like this, then $\varphi(h) = m$. Therefore it remains to show that h is well-defined. However, this is a direct consequence of the fact that m is a hierarchical morphism and, due to the assumptions, every h^f $(f \in F_G)$ is a hierarchical morphism from $cts_G(f)$ to $cts_H(m(f))$. ∎

The next theorem is the main result of this section. It states that the category $\mathbb{H}$ has pushouts, and the proof shows how to construct them effectively.

THEOREM 2.1. *For every pair $m_1: G \to H_1$ and $m_2: G \to H_2$ of morphisms in $\mathbb{H}$ there are a hierarchical graph H and morphisms $n_1: H_1 \to H$ and $n_2: H_2 \to H$ in $\mathbb{H}$ such that (m_1, m_2, n_1, n_2) is a pushout. Furthermore, $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout in the category of graphs.*

Proof. Let $H_1, H_2 \in \mathscr{H}_i$. We proceed by induction on i, using Lemma 2.1 to exploit the induction hypothesis in the inductive step. For $i=0$ it holds that $\overline{m_i} = m_i$, i.e., both morphisms are ordinary graph morphisms. Clearly, every pushout in the category of graphs is also a pushout in $\mathbb{H}$. Thus, in this case the pushout $(m_1, m_2, n_1, n_2) = (\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ of m_1 and m_2 in the category of graphs satisfies the assertion.

Now, let $i > 0$ and consider the morphisms $(m'_j: G' \to H'_j) = \varphi(m_j)$ $(j \in \{1, 2\})$, where $G' = \varphi(G)$ and $H'_j = \varphi(H_j)$. By the induction hypothesis, their pushout (m'_1, m'_2, n'_1, n'_2) exists and $(\overline{m'_1}, \overline{m'_2}, \overline{n'_1}, \overline{n'_2})$ is a pushout in the category of graphs. Let n'_j have the form $n'_j: H'_j \to H'$. Then, we obtain a hierarchical graph H such that $H' = \varphi(H)$, by defining

- $\bar{H} = H'|_A$ where $A = n'_1(A_{H_1}) \cup n'_2(A_{H_2})$,
- $F_H = n'_1(F_{H_1}) \cup n'_2(F_{H_2})$, and
- for every $f \in F_H$, $cts_H(f) = H'|_{A_f}$, where A_f is the set of all atoms $n'_j(a)$ such that $j \in \{1, 2\}$ and $a \in A_{cts_{H_j}(f')}$ for some $f' \in F_{H_j}$ with $n'_j(f') = f$.

Let us first verify that H is well-defined. For every $e' \in E_{H'} \cap A$ there are $j \in \{1, 2\}$ and $e \in E_{H_j}$ such that $n'_j(e) = e'$ (since $(\overline{m'_1}, \overline{m'_2}, \overline{n'_1}, \overline{n'_2})$ is a pushout). Consequently, $att_{H'}(e') = n'_j(att_{H_j}(e)) \in (V_{H'} \cap A)^*$, as required. Concerning the definition of A_f for $f \in F_H$, by a similar argument as for A we get $att_{H'}(e') \in (V_{H'} \cap A_f)^*$ for all $e' \in E_{H'} \cap A_f$.

Thus, H is well-defined. Furthermore, one may check in a straightforward way that we even have $\varphi(H) = H'$ because the set A together with the sets A_f $(f \in F_H)$ form a partition of $A_{H'}$. In order to be able to apply Lemma 2.1 it remains to be noticed that, by construction, for $j \in \{1, 2\}$ and $f \in F_{H_j}$, we have $n'_j(A_{cts_{H_j}(f)}) \subseteq A_{cts_H(n'_j(f))}$. Thus, by Lemma 2.1 there are hierarchical morphisms $n_j: H_j \to H$ for $j \in \{1, 2\}$, such that $\varphi(n_j) = n'_j$, and these are given by

- $n_j(a) = n'_j(a)$ for all $a \in A_{H_j}$,
- $n_j^f(a) = n'_j(a)$ for all $f \in F_{H_j}$ and $a \in A_{cts_{H_j}(f)}$, and
- $(n_j^f)^{f'} = {n'_j}^{f'}$ for all $f \in F_{H_j}$ and $f' \in F_{cts_{H_j}(f)}$.

As claimed in the theorem, $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout since $(\overline{m'_1}, \overline{m'_2}, \overline{n'_1}, \overline{n'_2})$ is one and $m'_j(a) = m'_j(a')$ for $a \in A_G$ implies $a' \in A_G$. It remains to verify that (m_1, m_2, n_1, n_2) possesses the universal property of pushouts.

Let $L \in \mathscr{H}$ and consider hierarchical morphisms $l_1: H_1 \to L$ and $l_2: H_2 \to L$ such that (m_1, m_2, l_1, l_2) commutes. Then the square (m'_1, m'_2, l'_1, l'_2), where $(l'_j: H'_j \to L') = \varphi(l_j)$ for $j \in \{1, 2\}$, commutes as well. Thus, there is a unique hierarchical morphism $l': H' \to L'$ such that $l'_j = l \circ n'_j$ for $j \in \{1, 2\}$. Again, the aim is to exploit

Lemma 2.1 in order to turn l' into the required morphism $l: H \to L$. For this, it must be verified that l' satisfies the requirements (1)–(3), of the lemma.

As for (1), since $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout, it holds that for every $e \in E_H$ there is some $j \in \{1, 2\}$ and some $e_0 \in E_{H_j}$ such that $n_j(e_0) = e$. Therefore, $l'(e) = l'_j(e_0)$ is an element of F_L if and only if $e \in F_H$.

For requirement (2), consider some $a \in A$. Then we have $a = n'_j(a_0)$ for some $j \in \{1, 2\}$ and $a_0 \in A_{H_j}$, (again since $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout) and thus $l'(a) = l'_j(a_0) = l_j(a_0) \in A_L$.

The verification of (3) is quite similar. Consider some $f \in F_H$ and $a \in A_f$, and let $j \in \{1, 2\}$, $f_0 \in F_{H_j}$, and $a_0 \in A_{cts_{H_j}(f_0)}$ be such that $n'_j(f_0) = f$ and $n'_j(a_0) = a$. (Notice that, again, j, f_0, and a_0 must exist.) Now $l'(a) = l'_j(a_0) = l_j^{f_0}(a_0) \in A_{cts_L(l_j(f_0))} = A_{cts_L(l'(f))}$, as required. Therefore, Lemma 2.1 applies, yielding a hierarchical morphism $l: H \to L$ such that $\varphi(l) = l'$, where

- $l(a) = l'(a)$ for all $a \in A_H$,
- $l^f(a) = l'(a)$ for all $f \in F_H$ and $a \in A_{cts_H(f)}$, and
- $(l^f)^{f'} = l'^{f'}$ for all $f \in F_H$ and $f' \in F_{cts_H(f)}$.

Clearly, for $i \in \{1, 2\}$ this implies $l_j = l \circ n_j$ since $l' = l' \circ n'_j$. Furthermore, if $k: H \to L$ was another morphism with this property distinct from l' then $k' = \varphi(k)$ would, by the definition of φ, be different from l' and would also satisfy $l'_j = k' \circ n'_j$ for $j \in \{1, 2\}$, contradicting the uniqueness of l'. This finishes the proof of the theorem. ▮

Notice that the proof of Theorem 2.1 yields a recursive procedure to construct pushouts in $\mathbb{H}$, based on the construction of pushouts in the case of ordinary graph morphisms.

The construction in the proof of the theorem yields two corollaries which turn out to be useful. The first expresses the fact that, to a certain extent, pushouts are constructed "levelwise." Intuitively, the recursive part of the construction depends only on the way in which m_1 and m_2 relate the frames and their contents (and we know already from Theorem 2.1 that the top level is just the pushout in the category of graphs). Let us say that two hierarchical morphisms $h: G \to H$ and $h': G' \to H'$ *agree on frames* if $F_{G'} = F_G$, $F_{H'} = F_H$, and $h'(f) = h(f)$ as well as $h'^f = h^f$ for all $f \in F_G$. Then we have the following.

COROLLARY 2.1. *Let (m_1, m_2, n_1, n_2) be a pushout in $\mathbb{H}$ consisting of hierarchical morphisms $m_i: G \to H_i$ and $n_i: H_i \to H$, and let $m'_i: G' \to H'_i$ agree with m_i on frames $(i \in \{1, 2\})$. Then, morphisms $n'_i: H'_i \to H'$ such that (m'_1, m'_2, n'_1, n'_2) is a pushout, can be constructed as follows:*

(1) *$(\overline{m'_1}, \overline{m'_2}, \overline{n'_1}, \overline{n'_2})$ is constructed as a pushout of non-hierarchical morphisms (in the way described in Section 1),*

(2) *$F_{H'} = \overline{n'_1}(F_{H_1}) \cup \overline{n'_2}(F_{H_2})$, and*

(3) *$n'^f_i = n^f_i$ for $i \in \{1, 2\}$ and $f \in F_{H_i}$.*

The second corollary concerns the special case where m_1 and m_2 are injective. Obviously, in this case the hierarchical morphisms m'_1 and m'_2 in the proof are also injective. As a consequence, it follows that $(m_1^f, m_2^f, n_1^{m_1(f)}, n_2^{m_2(f)})$ is a pushout for every frame $f \in F_G$. This yields the following specialization of Theorem 2.1.

COROLLARY 2.2. *Let $m_1: G \to H_1$ and $m_2: G \to H_2$ be injective hierarchical morphisms in $\mathbb{H}$. Then, one can construct hierarchical morphisms $n_1: H_1 \to H$ and $n_2: H_2 \to H$ such that (m_1, m_2, n_1, n_2) is a pushout, as follows:*

- *$\overline{n_1}$ and $\overline{n_2}$ are such that $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout,*
- *for every frame $f \in F_G$ the hierarchical morphisms $n_1^{m_1(f)}$ and $n_2^{m_2(f)}$ are constructed recursively so that $(m_1^f, m_2^f, n_1^{m_1(f)}, n_2^{m_2(f)})$ is a pushout, and*
- *for every frame $f \in F_{H_i} \setminus m_i(F_G)$ $(i \in \{1, 2\})$, n_i^f is an isomorphism.*

Next, we shall see how pushout complements can be obtained. For simplicity, we consider only the case where the two given hierarchical morphisms are *both* injective. This enables us to make use of Corollary 2.2 in an easy way, whereas the more general case would be unreasonably complicated as it required a hierarchical version of the so-called identification condition [15].

Clearly, in order to ensure the existence of pushout complements, a hierarchical version of the dangling condition must be satisfied. However, for the hierarchical case it must also be required that, intuitively, no frame is deleted unless its contents are deleted as well.

DEFINITION 2.3 (Hierarchical Dangling Condition). Let $G, H \in \mathscr{H}$ and $H_1 \in \mathscr{H}(\mathscr{X})$, and let $m: G \to H_1$ and $n: H_1 \to H$ be hierarchical morphisms. Then n satisfies the *hierarchical dangling condition* (*dangling condition*, for short) with respect to m if

- $\bar{n}$ satisfies the (non-hierarchical) dangling condition with respect to $\bar{m}$,
- for every frame $f \in F_{H_1} \setminus (m(F_G) \cup X_{H_1})$, n^f is bijective up to variables, and
- for every frame $f \in F_G \setminus X_G$, $n^{m(f)}$ satisfies the hierarchical dangling condition with respect to m^f.

Right below, we shall only use the dangling condition in case $H_1 \in \mathscr{H}$, but later on the more general case $H_1 \in \mathscr{H}(\mathscr{X})$ will be needed, too.

Notice that this condition coincides with the usual one in the special case where m and n are ordinary graph morphisms, because then only the first requirement is relevant, as there are no frames. Intuitively, the second part of the condition states that, as mentioned above, a frame can be deleted only if its contents are deleted as well (at least in the case where $H_1 \in \mathscr{H}$; the more general case is not yet our concern). As the proof below shows, this corresponds to the last item in Corollary 2.2 and is thus indeed necessary.

THEOREM 2.2. *Let $m_1: G \to H_1$, and $n_1: H_1 \to H$ be injective hierarchical morphisms in $\mathbb{H}$. Then there are hierarchical morphisms $m_2: G \to H_2$ and $n_2: H_2 \to H$ such that (m_1, m_2, n_1, n_2) is a pushout, if and only if n_1 satisfies the dangling condition with respect to m_1. In this case m_2 and n_2 are uniquely determined.*

Proof. Let $G \in \mathscr{H}_i$. Again, we proceed by induction on i. Clearly, if m_2 and n_2 exist, then m_2 must be injective since $n_1 \circ m_1 = n_2 \circ m_2$ is injective. By Corollary 2.2

this means that m_2 and n_2 exist if and only if they can be constructed in such a way that the following are satisfied:

(1) $\overline{m_2}$ and $\overline{n_2}$ are such that $(\overline{m_1}, \overline{m_2}, \overline{n_1}, \overline{n_2})$ is a pushout,

(2) for every frame $f \in F_G$, the hierarchical morphisms m_2^f and $n_2^{m_2(f)}$ are constructed recursively, so that $(m_1^f, m_2^f, n_1^{m_1(f)}, n_2^{m_2(f)})$ is a pushout, and

(3) for every frame $f \in F_{H_i} \setminus m_i(F_G)$ $(i \in \{1, 2\})$, n_i^f is an isomorphism.

As n_1 satisfies the dangling condition with respect to m_1, $\overline{m_2}$ and $\overline{n_2}$ exist and are uniquely determined (since $\overline{n_1}$ satisfy the non-hierarchical dangling condition with respect to $\overline{m_1}$), and (3) is satisfied for $i = 1$ (using the second part of the dangling condition). Furthermore, the induction hypothesis yields the required hierarchical morphisms m_2^f and $n_2^{m_2(f)}$ satisfying (2), for every frame $f \in F_G$. Together with the remaining requirement in (3) (i.e., the case $i = 2$) this determines m_2 and n_2 up to isomorphism, thus finishing the proof. ▮

3. HIERARCHICAL GRAPH TRANSFORMATION

Based on the results of the previous section we are now able to define rules and their application in the style of the double-pushout approach. From now on, a *rule* $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ consists of two hierarchical morphisms $l\colon I \to L$ and $r\colon I \to R$, where $L, I, R \in \mathscr{H}$ and l is injective. The hierarchical graphs L, I, and R are called the *left-hand side*, *interface*, and *right-hand side* of t, respectively.

The application of rules is defined by means of the usual double-pushout construction, with one essential difference. In order to make sure that transformations can take place on an arbitrary level in the hierarchy of frames, rather than only on the top level, one has to employ recursion.

Definition 3.1 (Transformation of Hierarchical Graphs). Let $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ be a rule. A hierarchical graph $G \in \mathscr{H}$ is transformed into a hierarchical graph $H \in \mathscr{H}$ by means of t, denoted by $G \Rightarrow_t H$, if one of the following holds:

(1) There is an injective hierarchical morphism $o\colon L \to G$, called an *occurrence morphism*, such that there are two pushouts

$$\begin{tikzcd} L \arrow[d, "o"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\ G & K \arrow[l] \arrow[r] & H \end{tikzcd}$$

in $\mathbb{H}$, or

(2) $\bar{G} \cong \bar{H}$ via some isomorphism $m\colon \bar{G} \to \bar{H}$, and there is a frame $f \in F_G$ such that $cts_G(f) \Rightarrow_t cts_H(m(f))$ and $cts_H(m(f')) \cong cts_G(f')$ for all $f' \in F_G \setminus \{f\}$.

For a set T of rules, we write $G \Rightarrow_T H$ if $G \Rightarrow_t H$ for some $t \in T$.

Example 3.1 (Concatenation of Hierarchical List Graphs). In Fig. 5, we specify a hierarchical version of the concatenation rule of Example 1.2 and show a transformation with this rule. We also need three quotients of this rule in order to

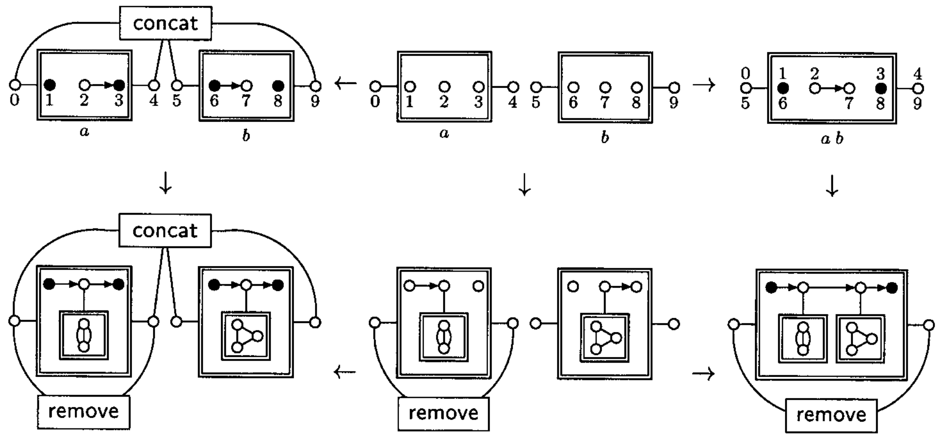


FIG. 5. The concatenation rule and its application.

handle empty lists. These quotients are characterized by the node identifications $1 = 2$ and $7 = 8$, respectively, and by both identifications together.

Note that the rule application in Fig. 5 takes place at the root level of the hierarchy, which corresponds to case (1) in Definition 3.1. Making use of Definition 3.1(2), a similar transformation could be made anywhere deeper down in the hierarchy of frames, for example, if some frame in a large hierarchical graph contained the one shown in the figure.

Since occurrence morphisms are injective, we get the following result as a consequence of Theorems 2.1 and 2.2.

THEOREM 3.1. *Let $t: L \xleftarrow{l} I \xrightarrow{r} R$ be a rule, $G \in \mathscr{H}$, and $o: L \to G$ an occurrence morphism. Then the two pushouts in Definition* 3.1(1) *exist if and only if o satisfies the dangling condition with respect to l. Furthermore, in this case the pushouts are uniquely determined up to isomorphism.*

Proof. By Theorem 2.2 the pushout on the left exists if and only if the dangling condition is satisfied, and if it exists then it is uniquely determined (up to isomorphism). Moreover, by Theorem 2.1 the pushout on the right always exists, and it is a general fact known from category theory that a pushout (m_1, m_2, n_1, n_2) is uniquely determined (up to isomorphism) by the morphisms m_1 and m_2. ▮

It is worth noting that, by the effectiveness of the results presented in Section 2, given a transformation rule, a hierarchical graph, and an occurrence morphism satisfying the dangling condition, one can effectively construct the required pushouts.

4. HIERARCHICAL GRAPH TRANSFORMATION WITH VARIABLES

Unfortunately, the concept of hierarchical graph transformation is not yet expressive enough to be satisfactory for certain programming purposes. There are some effects that one would certainly want to implement as single transformation steps, but which cannot be expressed by a single rule. In the example of lists, for instance, it should be possible to design a rule remove which deletes the first item in a list *regardless of its contents*. However, this is not possible as the dangling condition requires the occurrence morphism to be bijective on the contents of deleted frames. Similarly, a rule enter should take an *item* frame, again regardless of its contents, and add it to the list—preferably without affecting the original *item* frame. In order to realise these effects, we have to circumvent two obstacles. First, hierarchical morphisms preserve the frame hierarchy, which implies that, intuitively, rules cannot move frames across frame boundaries. Second, by now it is simply not possible to delete or duplicate frames together with their contents.

This is where variables come into play. The idea is to turn from rules to rule schemata and to transform hierarchical graphs by applying instances of these rule schemata. In order to make sure that an occurrence morphism satisfying the dangling condition always yields a well-defined transformation, we restrict ourselves to left-linear rule schemata. To this end, a hierarchical graph H is called *linear* if no variable occurs twice in H. More precisely, H must satisfy

- $cts_H(f) \neq cts_H(f')$ for all distinct variable frames $f, f' \in X_H$,
- $cts_H(f)$ is linear for every frame $f \in F_H \setminus X_H$,
- $var(cts_H(f)) \cap var(cts_H(f')) = \varnothing$ for all distinct frames $f, f' \in F_H \setminus X_H$,

and

- $cts_H(f) \notin var(cts_H(f'))$ for all variable frames $f \in X_H$ and all frames $f' \in F_H \setminus X_H$.

DEFINITION 4.1 (Variable Assignment). A *variable assignment* for $H \in \mathscr{H}(\mathscr{X})$ is a mapping $\alpha: var(H) \to \mathscr{H}$. The application of α to H is denoted by $H\alpha$ and is called the *instantiation* of H by α. It turns every variable frame $f \in X_H$ into a frame whose contents is $\alpha(cts_H(f))$. That is, $H\alpha = \langle \bar{H}, F_H, cts \rangle$ where

$$cts(f) = \begin{cases} \alpha(cts_H(f)) & \text{if } \ f \in X_H, \\ cts_H(f)\,\alpha & \text{otherwise,} \end{cases} \qquad \text{for every frame } f \in F_H.$$

By the definition of hierarchical morphisms, a hierarchical morphism $h: G \to H$ with $G \in \mathscr{H}$ can also be viewed as a hierarchical morphism from G to $H\alpha$, where α is any variable assignment for H. In the following, this hierarchical morphism will be denoted by $h\alpha$. We define rule schemata and their application based on this observation.

DEFINITION 4.2 (Transformation by Rule Schemata). A *rule schema*, denoted by $t: L \xleftarrow{l} I \xrightarrow{r} R$, is a pair of hierarchical morphisms $l: I \to L$ and $r: I \to R$, where $L, R \in \mathscr{H}(\mathscr{X})$, $I \in \mathscr{H}$, L is linear, and $var(R) \subseteq var(L)$. If α is a variable assignment for L, then the rule

$$t': L\alpha \xleftarrow{l\alpha} I \xrightarrow{r\alpha} R\alpha$$

is an *instance* of t.

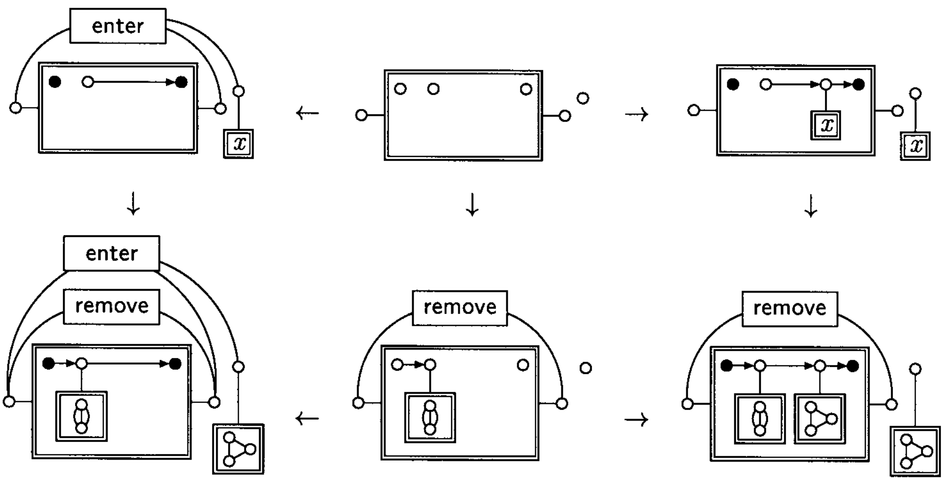


FIG. 6. The rule schema enter and its application.

A rule schema t transforms $G \in \mathscr{H}$ into $H \in \mathscr{H}$, denoted by $G \Rightarrow_t H$, if $G \Rightarrow_{t'} H$ for some instance t' of t. For a set T of rule schemata, we write $G \Rightarrow_T H$ if $G \Rightarrow_t H$ for some $t \in T$.

Example 4.1 (The Rule Schemata enter and remove). In Fig. 6, we show a rule schema that inserts a framed item graph at the tail of a list graph, and a transformation with this rule. The item frame contains a variable x which makes it possible to duplicate the item graph and to move one copy into the list frame. (We also consider the quotient of enter that merges the two left-most nodes in the list graph, and possibly other quotients that identify any of the nodes outside the list frame.)

Figure 7 shows a rule schema that removes the first item frame in a list graph, and an application of this rule to the result of the transformation in Fig. 6. The item graph contains a variable x so that it can be removed entirely. (Again we consider additional quotient rules, analogously to the case of enter.)

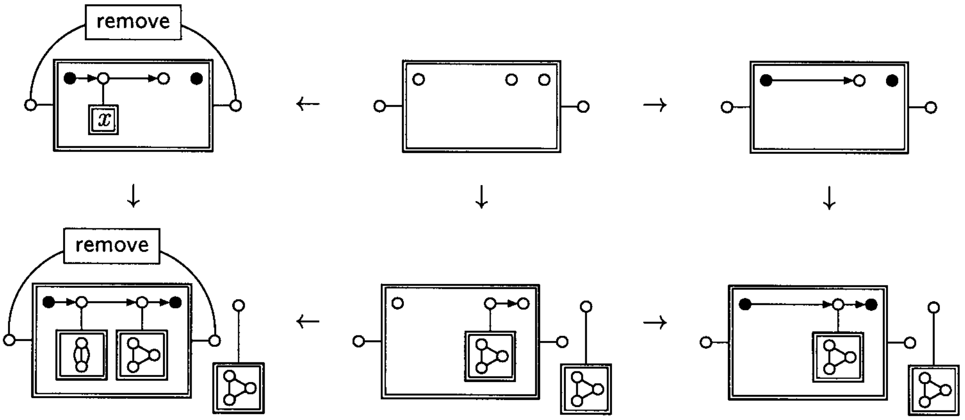


FIG. 7. The rule schema remove and its application.

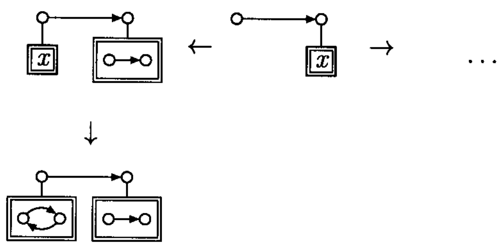


FIG. 8. A rule schema and occurrence morphism, with a variable in the interface.

Note that Definition 4.2 does not allow the interface graph I to contain variables. This avoids pathological cases while leaving the power of the formalism unaffected. To see this, assume that we would allow a "rule schema" as in Fig. 8 and consider the occurrence morphism shown there. Intuitively, the occurrence morphism instantiates x (see also Theorem 4.1 below), but if I is instantiated in this way the morphism from I to L ceases to exist. In fact, one may say that there is already a hidden instantiation of x, namely the one which is determined by the morphism from I to L, and both instantiations are in conflict with each other. Moreover, the left-linearity requirement imposed on L loses its effect. This becomes obvious if the contents of the non-variable frame in L is replaced with the empty graph or a new variable y. To avoid these unwanted effects without imposing complicated conditions, one would have to require that every variable in I is mapped to itself in L. However, then the rule schema has the same effect as the one in which all variables in I are deleted, so that nothing is gained with respect to power.

Definition 4.2 does not indicate how to implement hierarchical graph transformation with variables. The problem is that there are infinitely many instances of a rule schema as soon as it contains at least one variable. Therefore, the naive approach to implement $\Rightarrow_t$ by constructing all its instances and then testing each of them for applicability does not work. However, one can do better by constructing occurrence morphisms as follows and applying the theorem below.

Consider some linear hierarchical graph $L \in \mathscr{H}(\mathscr{X})$ and a hierarchical graph $G \in \mathscr{H}$, and let $o\colon L \to G$ be a hierarchical morphism. Then, due to the linearity of L, o induces a variable assignment $\alpha_o\colon var(L) \to \mathscr{H}$ and an occurrence morphism $inst(o)\colon L\alpha_o \to G$ as follows. For all $x \in var(L)$, if there is some $f \in X_L$ such that $cts_L(f) = x$ then $\alpha_o(x) = cts_G(o(f))$. Otherwise, $\alpha_o(x) = \alpha_{o^f}(x)$, where $f \in F_L \setminus X_L$ is the unique frame such that $x \in var(cts_L(f))$. Furthermore, $\overline{inst(o)} = \bar{o}$ and for all $f \in F_L$, $inst(o)^f$ is the identity on $cts_G(o(f))$ if $f \in X_L$ and $inst(o)^f = inst(o^f)$ otherwise.

The following theorem states that the transformations given by a rule schema $t\colon L \xleftarrow{l} I \xrightarrow{r} R$ can be obtained by considering occurrence morphisms $o\colon L \to G$ which satisfy the dangling condition with respect to l. To simplify our terminology, we may from now on say that o satisfies the dangling condition, skipping the reference to l, if the rule in question is clear from the context.

THEOREM 4.1. *Let $t\colon L \xleftarrow{l} I \xrightarrow{r} R$ be a rule schema and $G \in \mathscr{H}$.*

(1) *If $o\colon L \to G$ is an occurrence morphism satisfying the dangling condition, then $inst(o)$ is an occurrence morphism for $L\alpha_o$ satisfying the dangling condition with respect to $l\alpha_o$.*

(2) *If $\alpha: var(L) \to \mathscr{H}$ is a variable assignment and $q: L\alpha \to G$ is an occurrence morphism satisfying the dangling condition with respect to $l\alpha$, then $\alpha = \alpha_o$ and $q = inst(o)$ (up to isomorphism) for some occurrence morphism $o: L \to G$ satisfying the dangling condition with respect to l.*

Proof. Both claims are proved by induction on i, where $L \in \mathscr{H}_i(\mathscr{X})$.

(1) For $i = 0$, L is a graph. In particular, $var(L) = \varnothing$, which implies $inst(o) = o$ and $L\alpha_o = L$. Thus, there is nothing left to show.

Now consider some $i > 0$. By definition, $\overline{L\alpha_o} = \bar{L}$ and $\overline{inst(o)} = \bar{o}$. Therefore, $\overline{inst(o)}: \overline{L\alpha_o} \to \bar{G}$ is injective and satisfies the dangling condition for the nonhierarchical case. It remains to check the properties of the hierarchical morphisms $inst(o)^f$ for $f \in F_{L\alpha_o} = F_L$. There are three cases.

Case 1. $f \in l(F_I)$. Then $f \notin X_L$ because $var(I) = \varnothing$. As o^f is an occurrence morphism for $cts_L(f)$ which satisfies the dangling condition, the induction hypothesis yields that $inst(o)^f = inst(o^f)$ is an occurrence morphism for $cts_L(f)\,\alpha_{o^f} = cts_{L\alpha_o}(f)$ which satisfies the dangling condition.

Case 2. $f \in X_L$. Then $f \notin l(F_I)$ (again since $var(I) = \varnothing$). Therefore, the dangling condition requires $inst(o)^f$ to be an isomorphism, which, according to the definition of $inst(o)$, is the case.

Case 3. $f \notin X_L \cup l(F_I)$. According to the dangling condition, o^f is bijective up to variables. As one can deduce from the definition of $inst(o)$ by a straightforward induction, this implies that $inst(o)^f: cts_{L\alpha_o}(f) \to cts_G(o(f))$ is an isomorphism, as required.

(2) As in the first part, the statement is valid for $i = 0$. Therefore, let $i > 0$. Choosing $\bar{o} = \bar{q}$, we immediately get that $\bar{o}$ satisfies the (non-hierarchical) dangling condition. Now, for $f \in F_L \setminus X_L$, define o^f as follows. If $f \in l(F_I)$, then o^f is the hierarchical morphism obtained by applying the induction hypothesis to α (restricted to $var(cts_L(f))$) and q^f. Otherwise, $o^f: cts_L(f) \to cts_G(o(f))$ is chosen to be any hierarchical morphism which is bijective up to variables and satisfies $inst(o^f) = q^f$ (notice that such a morphism exists because q satisfies the dangling condition, which means that $q^f: cts_{L\alpha}(f) \to cts_G(o(f))$ is bijective as $f \notin l(F_I)$).

By construction, $inst(o) = q$ and $\alpha_o = \alpha$. Furthermore, o^f satisfies the dangling condition for every $f \in l(F_I)$ and is bijective up to variables for every $f \in F_L \setminus (l(F_i) \cup X_L)$. Together with the fact that $\bar{o}$ satisfies the (non-hierarchical) dangling condition this means that o satisfies the dangling condition, which completes the proof. ▮

Linearity of the left-hand sides of rule schemata is crucial for the instantiation of occurrence morphisms by which the pushout complement of a hierarchical graph transformation with variables is constructed. The other condition on the variables of a rule schema, namely that all variables of the right-hand side of a rule schema do already occur in its left-hand side, makes sure that the assignment used to instantiate the occurrence morphism suffices to instantiate the right-hand side of the rule to a hierarchical graph that is variable free again. (Note that similar conditions apply to term rewriting rules for similar reasons.)

5. SORTING LIST GRAPHS

In this section we demonstrate how to program by hierarchical graph transformation, by giving a quicksort algorithm for the list graphs considered before. Throughout this section, rule schemata are shown without their interface graphs. For each rule (schema) $t\colon L \xleftarrow{l} I \xrightarrow{r} R$, the morphism r is injective and the correspondence between the nodes of L and the nodes of R is indicated by the geometric arrangement of L and R. By convention, the interface graph I is the discrete subgraph of L consisting of all nodes that correspond to nodes in R.

The algorithm is given by a set of rules which falls into three parts.

Part 1. *Graph Comparison.* The rules in Fig. 9 compare the contents of two item frames with respect to their size, which is considered to be the number of their nodes and edges. The comparison works as follows:

- The first rule makes working copies of the item frames and their attachment nodes (where the attachment nodes of the left-hand side, drawn in grey, form the interface graph).
- Rules 2 through 5 remove one node or edge from the contents of the two working copies in parallel. For these rules we need quotients because in the item graph in which two nodes are connected by an edge, these nodes may be identical.
- If one of the item frames is empty, while the other has an arbitrary contents (expressed by the variable x), it is less or equal to the other in size. This is signaled, in rules 6 and 7, by connecting the item nodes of the original item frames by a "$\leqslant$" or "$\geqslant$" edge. (Note that either rule applies if both working copies are empty, so that the original graphs have been of equal size.)

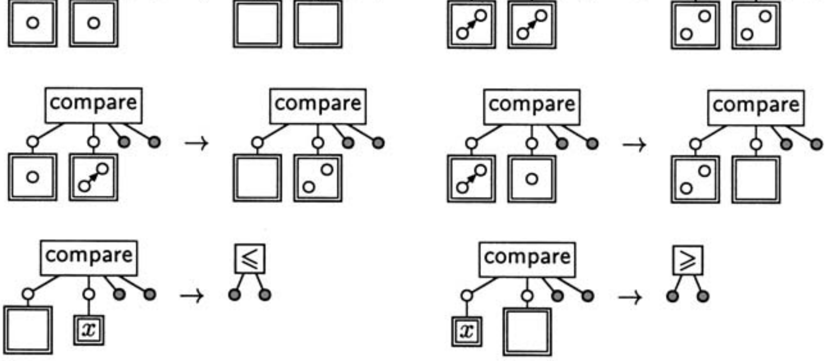


FIG. 9. The compare rules.

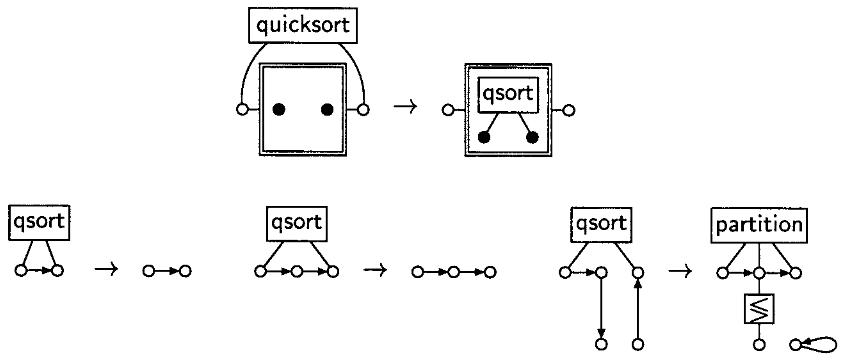


FIG. 10. The quicksort and qsort rules.

An interesting aspect of rules 6 and 7 should be noticed: Due to the hierarchical dangling condition, it is possible to test whether a frame contains a specific graph (the empty one in this case) by just attempting to delete it. Thanks to that condition, such a rule applies only if the occurrence morphism is bijective on the contents of this frame.

Part 2. *Sorting.* The rules in Fig. 10 work as follows:

• The first rule replaces an application of quicksort to a list frame by the application of qsort to the list graph it contains.

• Rules 2 and 3 cover the cases of empty and singleton lists, which need not be sorted.

• If the list contains more than one item, the first item is taken as a *pivot* to partition the remaining items into two sublists containing the items that are smaller, respectively greater, than the pivot. The unsorted items are "set aside," by disconnecting the second item node from the start node, and the last item node from the end node of the list. The end of this chain of unsorted items is marked by a loop edge. The start node is linked to the pivot, and the pivot to its end node, representing the fact that the lists of smaller respectively greater elements are empty at the beginning. We need quotients for the case where the list contains two item frames. Then the successor of the pivot and the predecessor of the end node are identified.

Part 3. *Partitioning.* These rules insert an unsorted item immediately before, or immediately after the pivot (see Fig. 11):

• Depending on whether the comparison determines that the pivot is not greater, or not less than the list element, this item is inserted after the pivot (by the upper rules), or before it (by the lower rules).

The upper shaded node is (the item node of) the pivot; the lower shaded node is the one which has just been compared with the pivot. The two rules on the left correspond to the case where the end of the list has been reached (which is signified by the loop edge). Thus, they insert the node after, respectively before, the pivot and finish the partitioning process. The other two rules also insert the node, but continue by comparing the next node with the pivot.

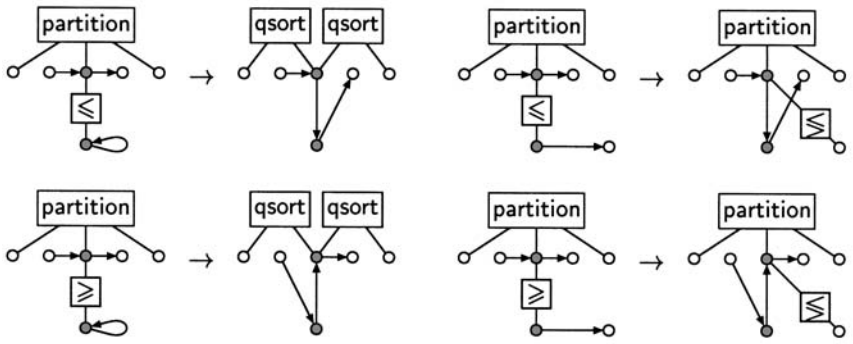


FIG. 11. The partition rules.

- If there is no further unsorted item, which is indicated by the loop edge, then qsort is called recursively to sort the sublists of smaller and greater elements (by the rules on the left). Otherwise partitioning continues by comparing the pivot with the next unsorted item.

We also need some quotients of the partition rules, since the first attachment node of the partition-edge can be identical to the pivot's predecessor, and its last attachment node can be identical to the pivot's successor.

Figure 12 shows how the quicksort program sorts a list of four item graphs. The intermediate steps of the comparison steps are omitted, and in the last line two applications of qsort, with the entailed applications of partition, compare, and qsort, are omitted as well.

6. RELATING HIERARCHICAL TO PLAIN GRAPH TRANSFORMATION

The previous section illustrates that frames are useful for programming by graph transformation. The contents of frames may be considered as hidden data, and one may think of an object-oriented system based on hierarchical graph transformation where a rule t can inspect or modify the contents of a frame f only if t belongs to the class by which f is generated (see [28]). For instance, the concat, enter, and remove rules of our running example do not inspect or modify the contents of item frames.

A hierarchical graph H may, however, also be considered as a "nested view" of a large flat graph G wherein the frames are placeholders that hide their contents. For restoring the flat graph G, it is useful to have a *flattening* operation which recursively replaces every frame with its contents. (See also [33] for hiding and restoring transformation rules working on such nested views; the views considered in [16, 45] are not nested.)

A useful notion of flattening should provide a mechanism which allows us to specify that some node of the contents of a frame f corresponds to a node v outside. More precisely, v should be an attached node of f. Thus, if f has k attached nodes, we need to designate a sequence of k nodes in the contents graph of f which the flattening process identifies with the attached nodes of the frame. We mark these k nodes by attaching a special edge to them.

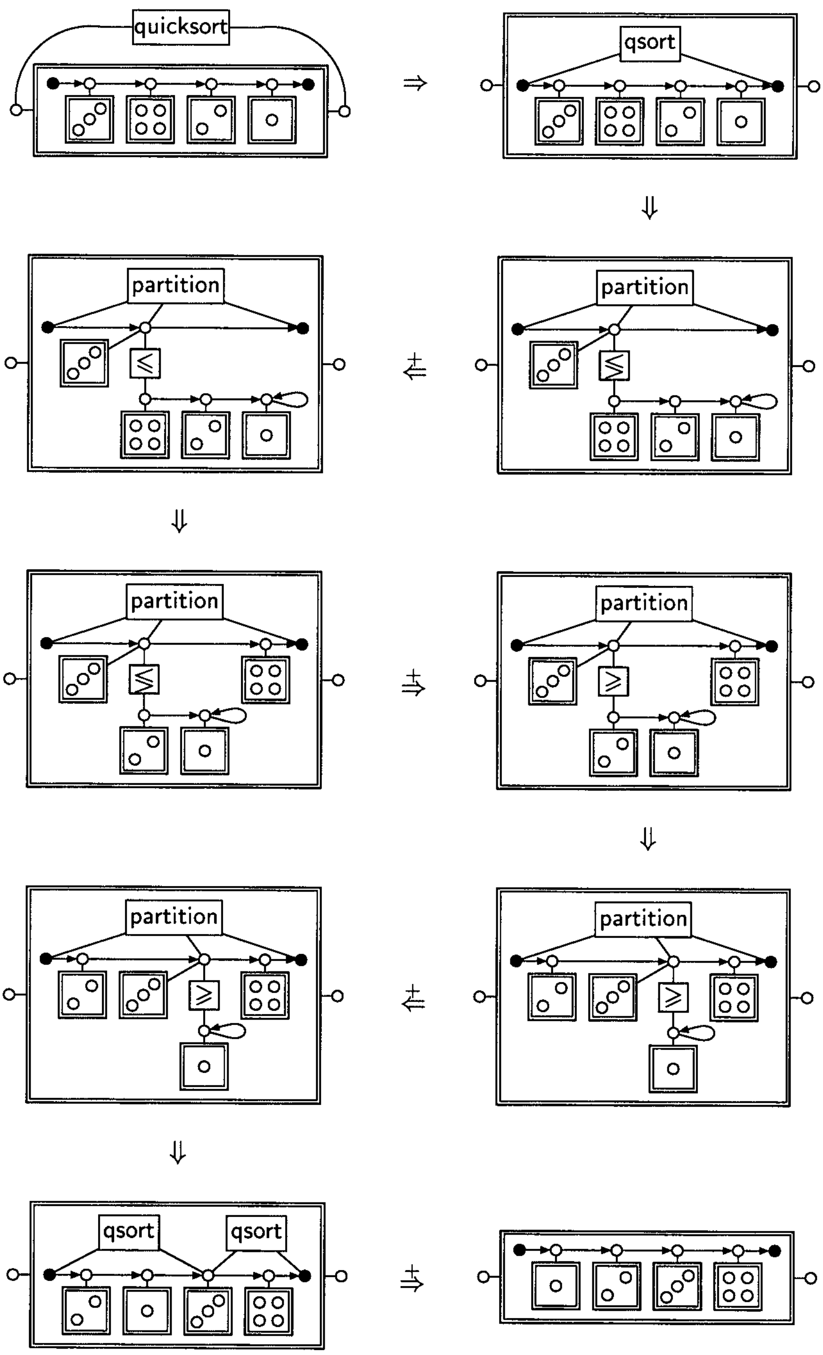


FIG. 12. Sorting a list graph by quicksort.

In the following, let $\Pi = \{\pi_k \mid k \in \mathbb{N}\} \subseteq \mathscr{C}$ be a set of special labels, where $type(\pi_k) = k$ for all $k \in \mathbb{N}$. An edge whose label is in Π will be called a Π-edge. We define three sets of *pointed hierarchical graphs*, namely $\mathscr{P}_+$, $\mathscr{P}_-$, and $\mathscr{P}$, as follows. $\mathscr{P}_+$ is the set of all hierarchical graphs G such that

(i) there is exactly one Π-edge p_G in $E_G \setminus F_G$; $point_G = att_G(p_G)$ is said to be the sequence of *points* of G and $type(G) = |point_G|$ is its *type*, and

(ii) for every frame $f \in F_G$, $cts_G(f) \in \mathscr{P}_+$ and $type(cts_G(f)) = |att_G(f)|$.

The set of all hierarchical graphs G which satisfy (ii), but where E_G does not contain a Π-edge, is denoted by $\mathscr{P}_-$, and $\mathscr{P} = \mathscr{P}_+ \cup \mathscr{P}_-$.

General Assumption. For the remainder of this section we shall only deal with hierarchical graphs in $\mathscr{P}$. To ensure that transformation preserves points, we assume in the following that $L, I, R \in \mathscr{P}_-$ for every rule $t: L \xleftarrow{l} I \xrightarrow{r} R$.

We show in this section that, under modest assumptions, hierarchical graph transformation (with rules of this kind) is compatible with the flattening operation in the following sense: A hierarchical transformation $G \Rightarrow_t H$ induces a corresponding "flat" transformation $G' \Rightarrow_{t'} H'$, where G', H', and t' are the flattened versions of G, H, and t, respectively. (Obviously, the converse cannot hold as structural information gets lost in the flattening process.)

For the definition of flattening, we extend the well-known concept of hyperedge replacement (see [13, 25]) to hierarchical graphs. Let H be a hierarchical graph. For every set $E \subseteq E_H$ of edges, $H - E$ denotes the hierarchical graph obtained by deleting all the edges in E from H, i.e., $H - E = \langle G, F_H \setminus E, cts \rangle$ where $G = \langle V_H, E_H \setminus E, att, lab \rangle$, att and lab being the restrictions of att_H and lab_H to $E_H \setminus E$, and cts is the restriction of cts_H to $F_H \setminus E$. Now, consider a mapping $\sigma: E \to \mathscr{P}$ such that $type(\sigma(e)) = |att_H(e)|$ for all $e \in E$, called a *hyperedge substitution for H*. Hyperedge replacement yields the hierarchical graph $H[\sigma]$ obtained from $(H - E) + \sum_{e \in E} (\sigma(e) - \{p_{\sigma(e)}\})$ by identifying, for all $e \in E$ with $att_H(e) = u_1 \cdots u_k$ and $point_{\sigma(e)} = v_1 \cdots v_k$, each node u_i with v_i, for all $i \in [k]$. For all atoms $a \in (A_H \setminus E) \cup \bigcup_{e \in E} (A_{\sigma(e)} \setminus \{p_{\sigma(e)}\})$ we denote by $track_{H,\sigma}(a)$ (or $track(a)$ if there is no danger of confusion) the image of a in $H[\sigma]$, if it is necessary to be particularly precise with respect to this point. Normally, however, we will implicitly assume that $track_{H,\sigma}$ is the identity, in order to avoid unnecessarily complex notation.

Finally, for all $H \in \mathscr{P}$, the *flattening* of H yields the graph $flat(H) = H[\sigma]$ where $\sigma: F_H \to \mathscr{P}_+$ is given inductively by $\sigma(f) = flat(cts_H(f))$ for all $f \in F_H$. Thus, flattening recursively replaces all frames by their contents (and removes the Π-edges from the inserted graphs), yielding a flat graph.

EXAMPLE 6.1 (Hyperedge Replacement and Flattening). Figures 13 and 14 illustrate hyperedge replacement and flattening. In these figures, π-edges are not drawn explicitly, but the nodes in the point sequence $point_G$ that they indicate are drawn as filled circles. Numbers ascribed to those nodes indicate their position in the point sequence. The ascription "$1 = 3$" in $\sigma(f)$ and G indicates that this node occurs repeatedly, at position 1 and 3 in the point sequence of the corresponding frame.

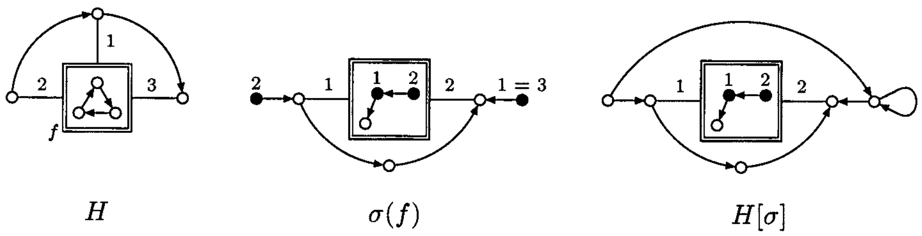


FIG. 13. Frame substitution.

Hyperedge replacement, applied to a frame as in Fig. 13, removes the replaced frame regardless of its contents. The contents gets lost. Since nodes 1 and 3 in the point sequence of $\sigma(f)$ are identical, the corresponding attached nodes of f are identified in $H[\sigma]$ (which, in this particular case, turns the edge between the two nodes into a loop). Similarly, repetitions in $att(f)$ can result in identifications of nodes in $\sigma(f)$.

Flattening, as shown in Fig. 14, does not discard the contents of frames because every frame gets replaced with its own, flattened contents. To see how this works, first replace the inner frame with its contents, and then the outer one. Of course, repetitions in point sequences or frame attachments result in similar node identifications as in Fig. 13.

The following lemma turns out to be useful in proofs. It holds by an obvious induction, using the associativity of hyperedge replacement.

LEMMA 6.1. *For all $H \in \mathscr{P}$,*

$$flat(H) = \begin{cases} H & \textit{if} \quad H \in \mathscr{H}_0, \\ flat(H[cts_H]) & \textit{otherwise.} \end{cases}$$

We can flatten morphisms as well. Consider a hierarchical morphism $h: G \to H$ with G, $H \in \mathscr{P}$ and let $\sigma = flat \circ cts_G$ and $\tau = flat \circ cts_H$. Then $flat(h)$ is the morphism m: $flat(G) \to flat(H)$ defined inductively as follows. For all $a \in A_{flat(G)}$, if $a = track_{G,\sigma}(a')$ for some $a' \in A_G$, then $m(a) = h(a')$; if $a = track_{G,\sigma}(a')$ for some $a' \in A_{\sigma(f)}$ and $f \in F_G$, then $m(a) = flat(h^f)(a')$.

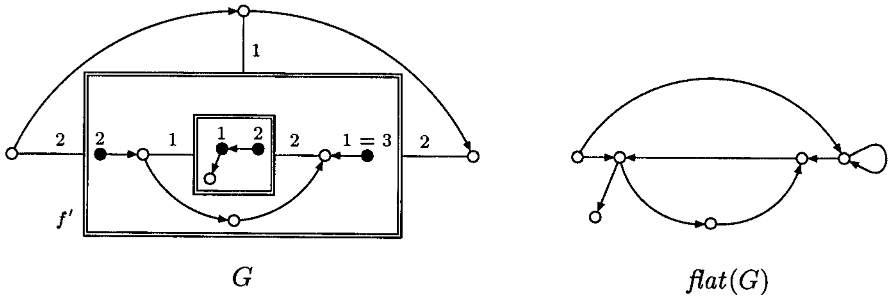


FIG. 14. Flattening.

Notice that the two cases in the definition of $m(a)$ above are consistent with each other, although they intersect. This can be seen as follows. According to the definition of hyperedge replacement, $flat(G) = G[\sigma]$ and $flat(H) = H[\tau]$ are obtained from $G' = (G - F_G) + \bigcup_{f \in F_G} (\sigma(f) - \{p_{\sigma(f)}\})$ and $H' = (H - F_H) + \bigcup_{f \in F_H} (\tau(f) - \{p_{\tau(f)}\})$, respectively, by identifying certain nodes. Thus, assuming inductively that every $flat(h^f)$ ($f \in F_G$) is a well-defined morphism, m is well-defined if $track_{G,\sigma}(u) = track_{G,\sigma}(v)$ implies $track_{H,\tau}(g(u)) = track_{H,\tau}(g(v))$ for all $u, v \in V_{G'}$, where $g\colon V_{G'} \to V_{H'}$ is given by

$$g(w) = \begin{cases} h(w) & \text{for} \quad w \in V_G, \\ flat(h^f)(w) & \text{for} \quad w \in V_{\sigma(f)}, \quad f \in F_G. \end{cases}$$

However, $track_{G,\sigma}(u) = track_{G,\sigma}(v)$ holds only if there are nodes $v_0, ..., v_n \in V_{G'}$ such that $v_0 = u$, $v_n = v$, and for every $i \in [n]$ there is some $f \in F_G$ and an index $j \in [type(\sigma(f))]$ for which $\{v_{i-1}, v_i\} = \{att_G(f)(j), point_{\sigma(f)}(j)\}$. The latter implies $\{g(v_{i-1}), g(v_i)\} = \{att_H(h(f))(j), point_{\tau(h(f))}(j)\}$, which yields the required equation $track_{H,\tau}(g(u)) = track_{H,\tau}(g(v_0)) = track_{H,\tau}(g(v_n)) = track_{H,\tau}(g(v))$.

It was mentioned above that the main result of this section holds only under a certain assumption. The reason is that a morphism $flat(h)$ may be non-injective although $h\colon G \to H$ itself is injective. This is caused by the fact that the construction of $flat(G)$ may identify some nodes in V_G because they are incident with a frame whose contents has repetitions in its point sequence. If the attached nodes of the frame are distinct, hyperedge replacement identifies them (by identifying each with the same point of the contents). Thus, flattening may turn an occurrence morphism into a non-injective morphism, making it impossible to apply the corresponding flattened rule. In fact, the dual situation where there are identical attached nodes of a frame while the corresponding points of its contents are distinct, must also be avoided. The reason lies in the recursive part of the definition of $\Rightarrow_t$. If a rule is applied to the contents of some frame, but the replacement of the frame identifies two distinct points of the contents because the corresponding attached points of the frame are identical, the flattened rule cannot be applied either.

There are several possible ways to circumvent these problems. First, one may develop the whole theory without requiring occurrence morphisms to be injective, which is probably quite a complicated task. Second, in addition to the flattened rule one could use all its quotient rules. This will probably work, and the proof should not be too difficult. However, there is a third, even simpler possibility which we choose here, namely just to exclude these situations (which appear somewhat unnatural in cases where one is interested in flattening). To this end, call a hierarchical graph $H \in \mathscr{P}$ *identification consistent* if every frame $f \in F_H$ satisfies the following condition:

(1) For all $i, j \in [type(cts_H(f))]$, $point_{cts_H(f)}(i) = point_{cts_H(f)}(j)$ if and only if $att_H(f)(i) = att_H(f)(j)$, and

(2) $cts_H(f)$ is identification consistent.

One should notice that identification consistency is preserved by the application of a rule $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ if R is identification consistent and r is injective. Thus, if we restrict ourselves to systems with rules of this kind then all hierarchical graphs derivable from some identification consistent graph will be identification consistent as well.

The following lemma holds by an obvious induction.

Lemma 6.2. *For every injective hierarchical morphism $h: G \to H$ ($G, H \in \mathscr{P}$) such that H is identification consistent, $flat(h)$ is injective.*

The main lemma of this section says that the flattening operation turns pushouts of hierarchical morphisms into pushouts of ordinary morphisms.

Lemma 6.3. *Let $m_i: G \to H_i$ and $n_i: H_i \to H$ ($i \in \{1, 2\}$) be hierarchical morphisms, where G, H_1, H_2, $H \in \mathscr{P}$. If (m_1, m_2, n_1, n_2) is a pushout in $\mathbb{H}$, then $(flat(m_1), flat(m_2), flat(n_1), flat(n_2))$ is a pushout as well.*

Proof. We may assume without loss of generality that G, H_1, H_2, $H \in \mathscr{P}_-$, as flattening treats Π-edges that occur on the top level like any other edge.

For a hierarchical morphism $h: K \to L$ with $K, L \in \mathscr{P}_-$ let $\varphi'(h)$ be the hierarchical morphism $h': K[cts_K] \to L[cts_L]$ given by

$$h'(a) = \begin{cases} h(a) & \text{if} \quad a \in A_K \\ h^f(a) & \text{if} \quad a \in A_{cts_K(f)}, \quad f \in F_K \end{cases}$$

for all $a \in A_{K[cts_K]}$, and $h'^g = h^{f^g}$ for all $f \in F_K$ and $g \in F_{cts_K(g)}$. Similar to the argumentation for the well-definedness of $flat(h)$ it follows that $\varphi'(h)$ is indeed a well-defined hierarchical morphism.

Comparing the definitions of φ' and the mapping φ considered in Section 2, it is clear that $\varphi'(h)$ is obtained from $\varphi(h) = (h_0: K_0 \to L_0)$ by

(i) deleting from $\overline{K_0}$ and $\overline{L_0}$ all edges belonging to F_K or F_L, respectively, as well as the Π-edges, and

(ii) identifying in $\overline{K_0}$ all nodes u, v such that $track_{K, cts_K}(u) = track_{K, cts_K}(v)$, and in $\overline{L_0}$ all nodes u, v such that $track_{L, cts_L}(u) = track_{L, cts_L}(v)$.

Now, consider the pushout (m_1, m_2, n_1, n_2) in the statement of the lemma. By induction on the depth of the frame hierarchy, using Lemma 6.1, it suffices to prove that $(\varphi'(m_1), \varphi'(m_2), \varphi'(n_1), \varphi'(n_2))$ is a pushout. We already know from Section 2 that $(\varphi(m_1), \varphi(m_2), \varphi(n_1), \varphi(n_2))$ is a pushout. Therefore, by Corollary 2.1 it remains to be shown that the square $(\overline{\varphi'(m_1)}, \overline{\varphi'(m_2)}, \overline{\varphi'(n_1)}, \overline{\varphi'(n_2)})$ obtained from $(\overline{\varphi(m_1)}, \overline{\varphi(m_2)}, \overline{\varphi(n_1)}, \overline{\varphi(n_2)})$ by applying (i) and (ii) to each of the morphisms, is a pushout. As we know from Section 1 how pushouts are constructed in the non-hierarchical case, this is not hard to verify. The deletion of frames in (i) does not affect the property of being a pushout because $\overline{\varphi(m_1)}$ maps frames to frames. For a similar reason, the deletion of Π-edges is consistent and preserves the pushout properties. Furthermore, (ii) does not destroy the pushout either, because it identifies only those nodes in $\overline{\varphi(H)}$ whose pre-images in $\overline{\varphi(H_1)}$ or $\overline{\varphi(H_2)}$ are identified, too. Thus, all together, $(\varphi'(m_1), \varphi'(m_2), \varphi'(n_1), \varphi'(n_2))$ turns out to be a pushout, which completes the proof. ▌

We can now prove the main theorem of this section: If a rule can be applied to an identification consistent hierarchical graph, then the flattened rule can be applied to the flattened graph, with the expected result.

THEOREM 6.1. *Let $t: L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ be a rule, and let $t': L' \overset{l'}{\leftarrow} I' \overset{r'}{\rightarrow} R'$ be the rule given by $l' = flat(l)$ and $r' = flat(r)$. For every transformation $G \Rightarrow_t H$ such that G is identification consistent, there is a transformation $flat(G) \Rightarrow_{t'} flat(H)$.*

Proof. Due to the definition of $\Rightarrow_t$ there are two cases to be distinguished. If there is a double-pushout of the form

$$\begin{tikzcd}
L \arrow[d, "o"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\
G & K \arrow[l] \arrow[r] & H
\end{tikzcd}$$

for some occurrence morphism $o: L \rightarrow G$, then Lemma 6.3 yields the following double-pushout:

$$\begin{tikzcd}
L' \arrow[d, "flat(o)"'] & I' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] & R' \arrow[d] \\
flat(G) & flat(K) \arrow[l] \arrow[r] & flat(H).
\end{tikzcd}$$

By Lemma 6.2 the morphism $flat(o)$ is injective, which by the definition of transformation means $flat(G) \Rightarrow_{t'} flat(H)$.

The second case to be considered is the recursive one. Suppose there is a frame $f \in F_G$ such that $\bar{G} \cong \bar{H}$ via some isomorphism $m: G \rightarrow H$, where $cts_G(f) \Rightarrow_t cts_H(m(f))$ and $cts_H(m(f')) \cong cts_G(f)$ for all $f' \in F_G \setminus \{f\}$. Assuming inductively that the transformation $flat(cts_G(f)) \Rightarrow_{t'} flat(cts_H(m(f)))$ exists, we obtain the following double-pushout of non-hierarchical morphisms:

$$\begin{tikzcd}
L' \arrow[d, "o"'] & I' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] & R' \arrow[d] \\
flat(cts_G(f)) & K \arrow[l] \arrow[r] & flat(cts_H(m(f)))
\end{tikzcd}$$

By our general assumption, L', I', and R' are in $\mathscr{P}_{-}$. Therefore, the unique Π-edges in $flat(cts_G(f))$, K, and $flat(cts_H(m(f)))$ cannot occur in the images of the vertical morphisms. Furthermore, by the uniqueness of these edges the two morphisms on the bottom line map Π-edges to Π-edges. Therefore, we obtain a new double-pushout diagram

$$\begin{tikzcd}
L' \arrow[d, "o"'] & I' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] & R' \arrow[d] \\
G_0 & K_0 \arrow[l] \arrow[r] & H_0
\end{tikzcd}$$

by deleting the Π-edges from these graphs: $G_0 = flat(cts_G(f)) - \{p_{cts_G(f)}\}$, $K_0 = K - \{p_K\}$, and $H_0 = flat(cts_H(m(f))) - \{p_{cts_H(m(f))}\}$. Using this, one can construct the required transformation $flat(G) \Rightarrow_{t'} flat(H)$ as follows.

Due to the assumed identification consistency, G_0 is a subgraph of $flat(G)$. More precisely, by defining $h(a) = a$ for all $a \in A_{G_0}$, one obtains an injective morphism $h\colon G_0 \to flat(G)$. Moreover, if o satisfies the dangling condition, so does $h \circ o$, since $v \in V_{cts_G(f)}$ cannot occur in $att_{flat(G)}(e)$ for some $e \in E_{flat(G)} \setminus E_{G_0}$ unless v is a point of $cts_G(f)$ (by the definition of hyperedge replacement), and is thus in the image of $o \circ l'$.

This means that the double-pushout above extends to

$$\begin{tikzcd}
L' \arrow[d, "h \circ o"'] & I' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] & R' \arrow[d] \\
flat(G) & K' \arrow[l] \arrow[r] & H'
\end{tikzcd}$$

for some graphs K' and H'. In fact, from the constructions recalled in Section 1 it follows immediately that $H' \cong flat(H)$. This shows that there is a transformation $flat(G) \Rightarrow_{t'} flat(H)$ and thus completes the proof. ▮

7. RELATED WORK

In this section we review some of the related work one can find in the literature. The contribution of the present paper has been twofold. We have introduced a hierarchical structuring mechanism for graphs together with an appropriate notion of hierarchical graph transformation. Accordingly, two categories of related work can be distinguished. The first one encompasses contributions that define an additional level of structure for graphs, usually motivated by some type of application area. The second one is about approaches that aim at the transformation of hierarchical structures, using ideas which are related to those presented here. Naturally, some papers contribute to both fields.

Let us first discuss structuring approaches for graphs. These approaches are usually motivated by the observation that large graphs are often incomprehensible both from a visual and an intellectual point of view. Despite the fact that graphs *are* structures, they lack a global structuring mechanism. Such a mechanism should allow us to designate or encapsulate subgraphs which, for semantic or pragmatic reasons, shall be considered as entities on their own. Normally, it is desirable that such a concept of graphs within graphs may be iterated. Graphs of this kind are therefore often called hierarchical graphs. (This is not to be confused with the meaning of "hierarchical" in the graph drawing community, where the term is used for a tree-like layout of flat graphs, as in [32].) Below, some of the main structuring concepts for graphs of this type are discussed.

• The nodes and/or edges of a graph can be allowed to contain other graphs, so-called nested components, which can be nested again. Such a concept has been introduced in this paper, turning a graph from a flat into a recursive structure in which the nested components are pairwise disjoint: neither do they share subgraphs, nor are there edges between their nodes. The strict hierarchical structure is particularly useful for programming by graph transformation because components can be transformed independently of each other, which guarantees that the semantics of transformation is compositional (see the discussion in the Introduction).

The nested components contained in nodes or edges can be seen as attributes. However, tools like PROGRES [43], AGG [22], and DIAGEN [31] either restrict attributes to be primitive, or to be of a (non-graphical) type in their host programming language. Even if this allows us to attribute nodes or edges of a graph by the host language implementation of another graph, transformations can then only be defined componentwise. The edges and nodes in the L-graphs proposed in [42] are labelled by (flat) graphs. By iteration, this definition can provide nesting of any fixed depth.

- Another line of research is motivated by system modelling. This motivation results in a slightly different conceptual view than the one described above. A graph is not a priori structured. Instead, the starting point is an underlying flat graph, which is then provided with a structure by associating subgraphs to so-called packages. Packages can be represented by nodes (which are either fresh, or occur in the underlying graph), and there can be edges between them which establish some kind of package dependency. Edges in the underlying graph are often allowed to cross package boundaries, and in some cases, nodes and edges may belong to several packages.

As discussed in the Introduction, such "boundary-crossing" edges prevent compositionality and are thus undesirable for programming (although the dividing-line is blurred and one may think of programming situations in which these edges are useful, similar in character to the use of GOTO-statements). In contrast to this, boundary-crossing edges are indispensable in many modelling situations and are thus of widespread use in nested visual languages like UML [41]. Therefore, it is usually unavoidable to sacrifice compositionality when hierarchical graphs shall be applied for modelling purposes.

Pratt [37, 38] was probably the first to consider such a kind of hierarchical graphs. The packages in his H-graphs may depend on each other in an arbitrary way, also recursively. A particular kind of node replacement is used to define H-graph languages that represent the semantics of programs, e.g., of Lisp expressions. (Note that this is essentially a modelling rather than a programming situation, as the semantics of a source program is represented by a set of graphs.)

The hierarchical graphs defined in [21] are equipped with an acyclic package hierarchy as well as export and import relations defining admissible package-crossing edges. However, transformational aspects are not investigated. In [6] transformation for a simpler version of such graphs is defined. The paper [20], where hierarchical graphs are defined by distinguishing "ordinary" from "hierarchy" edges, includes transformation, but without considering export, import, and conditions for the preservation of the hierarchy's consistency.

In [8], an abstract notion of hierarchical graphs is introduced where the package hierarchy, the underlying graph, and their coupling are represented by three flat graphs, and transformation is defined as componentwise application of rules to these three graphs (even heterogeneously, specified in different transformation approaches of the kind formalized in [2]). This framework is used in [7] to compare the approach introduced here, and the one by Pratt. It may also be used to equip arbitrary graph classes and transformation approaches with a hierarchy concept.

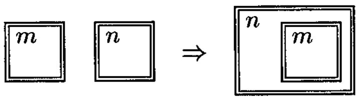
• The basic idea of distributed graph transformation is to extract several subgraphs from a graph, and to transform them independently so that a common interface remains intact [44]. This creates a shallow hierarchical structure of depth 2. The distributed encapsulated graph objects of [45] extend distributed graph transformation by export and import specifications for the local graphs.

• Several papers have investigated concepts which allow abstracting from details in graphs. Often, graphs which are obtained from an underlying graph by abstracting away certain parts are called views. In [19], two views of a graph are created so that the transformation of one view updates the other consistently. Multilevel graph grammars [33] define the transformation of graphs wherein details can be hidden by abstracting transformations that can be undone as needed. The graphs in [30] support rigidly layered views. Graphs and the morphisms between them are abstracted to nodes and edges of the next layer.

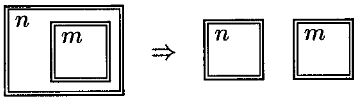
Let us finally discuss some aspects of our notion of transformation which are related to concepts discussed in the literature.

In the hierarchical setting, the ability of copying, moving, or deleting the contents of frames seems to be of central importance in order to obtain an appropriate notion of transformation. The idea of using variables in order to achieve such effects is also followed in the so-called substitution-based approach to graph transformation [35]. While this approach was originally defined for flat hypergraphs, the recent paper [29] introduces a substitution-based version of hierarchical graph transformation that employs a slightly more general concept of variables than in the present paper.

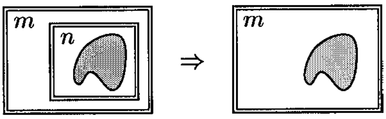
A rather new idea which has received wide interest is the ambient calculus [9]. It can be seen as a formalism which allows us to describe the transformation of hierarchical structures of a certain kind. Translated into the setting and terminology of the present paper, the calculus allows frames to execute transformations which move them from one place in the hierarchy to another. Three basic transformations, which are parameterized with the name of another frame, are possible. The transformation *in n*, executed by a frame m, moves m into n if n is (the name of) one of the frames in the neighbourhood of m:



The transformation *out n*, executed by a frame *m* whose parent frame is *n*, moves *m* out of *n*:



Finally, if *n* is a frame within a frame *m* then the transformation *open n*, executed by *m*, erases the frame border around the contents of *n* and adds it to the contents of *m*:



The ambient calculus is intended to describe the behaviour of mobile processes (called ambients) in networks such as the internet, where ambients can move from one place to another under certain conditions. The use of names plays a central role as it allow us to model restricted access: An ambient *m* cannot get into, move out of, or open another ambient unless it knows its name.

Due to the different motivations and basic assumptions it seems neither useful nor possible to make a direct comparison of hierarchical graph transformation and the ambient calculus. However, some similarities are obvious. The three transformations described above are quite similar to transformations in our sense, using appropriate rule schemata. A slightly more general concept of variables (based on the substitution-based approach mentioned above) should make it possible to model *in n*, *out n*, and *open n* by rule schemata in a rather straightforward way. However, these rule schemata could be applied everywhere; to incorporate the access restrictions provided by the role of names in the ambient calculus seems to require additional concepts.

8. CONCLUSION

In this paper we have extended graphs to hierarchical graphs wherein certain edges, called frames, contain graphs that may be hierarchical again. We have lifted double-pushout graph transformation to hierarchical graphs by showing how pushouts and pushout complements can be constructed in the category of hierarchical graphs. Furthermore, we have extended rules by frame variables by which frames can be deleted or duplicated with their entire contents, in a single transformation step. This turns out to be valuable for programming with hierarchical graph transformation.

One direction for future work on hierarchical graph transformation is to lift results of the double-pushout approach to the hierarchical setting, like sequential and parallel commutativity, results on parallelism, concurrency, amalgamation, confluence, and termination.

Hierarchical graph transformation should also be combined with concepts for structuring and controlling systems of rules. As mentioned in the Introduction, several such concepts (mainly for flat graphs) have been proposed recently [2, 27].

A further topic of research is to develop hierarchical graph transformation towards object-oriented programming, as outlined in [28]. There the idea is to restrict the visibility of frames so that only rules designated to some frame type may inspect or update the contents of frames of this type. Such frame types come close to "classes," and the designated rules correspond to "methods." Then frames can be seen as instantiations of their classes that can only be manipulated by invoking the methods of the class.

Typing is another concept to be considered if hierarchical graph transformation is to be the basis for a programming language. In [23, 29], types are context-free graph languages specified by hyperedge replacement grammars. The challenge is then to provide algorithms that can statically check whether operations in the form of general transformation rules will always preserve the specified types.

ACKNOWLEDGMENT

We thank the referees for helpful remarks and suggestions.

REFERENCES

1. J. Adámek, H. Herrlich, and G. Strecker, "Abstract and Concrete Categories," Wiley, New York, 1990.
2. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer, Graph transformation for specification and programming, *Sci. Computer Programm.* **34** (1999), 1–54.
3. J.-P. Banâtre and D. LeMétayer, Gamma and the chemical reaction model: Ten years after, *in* "Coordination Programming: Mechanisms, Models, and Semantics," pp. 3–41, World Scientific, Singapore, 1996.
4. R. Bardohl, M. Minas, A. Schürr, and G. Taentzer, Application of graph transformation to visual languages, *in* Ehrig *et al.* [17, Chap. 3, pp. 105–180].
5. E. Barendsen and S. Smetsers, Graph rewriting aspects of functional programming, *in* Ehrig *et al.* [17, Chap. 2, pp. 63–102].
6. G. Busatto, G. Engels, K. Mehner, and A. Wagner, A framework for adding packages to graph transformation approaches, *in* "Theory and Application of Graph Transformation (TAGT'98), Selected Papers," Lecture Notes in Computer Science, Vol. 1764, pp. 352–367, Springer-Verlag, New York/Berlin, 2000.
7. G. Busatto and B. Hoffmann, Comparing notions of hierarchical graph transformation, *in* "Proc. Graph Transformation and Visual Modelling Techniques," Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam/New York, Vol. 50(3), http://www.elsevier.nl/locate/entcs, 2001.
8. G. Busatto, H.-J. Kreowski, and S. Kuske, "Abstract Hierarchical Graph Transformation," Informatik-Bericht 1/01, Fachbereich Mathematik-Informatik, Universität Bremen, 2001.
9. L. Cardelli and A. D. Gordon, Mobile ambients, *Theoret. Comput. Sci.* **240** (2000), 177–213.
10. V. Claus, H. Ehrig, and G. Rozenberg (Eds.), "Graph Grammars and Their Application to Computer Science and Biology, Selected Papers," Lecture Notes in Computer Science, Vol. 73, Springer-Verlag, New York/Berlin, 1979.

11. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, Algebraic approaches to graph transformation—Part I. Basic concepts and double pushout approach, *in* Rozenberg [40, Chap. 3, pp. 163–245].

12. J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg (Eds.), "Graph Grammars and Their Application to Computer Science, Selected Papers," Lecture Notes in Computer Science, Vol. 1073, Springer-Verlag, New York/Berlin, 1996.

13. F. Drewes, A. Habel, and H.-J. Kreowski, Hyperedge replacement graph grammars, *in* Rozenberg [40, Chap. 2, pp. 95–162].

14. F. Drewes, B. Hoffmann, and D. Plump, Hierarchical graph transformation, *in* "Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)," Lecture Notes in Computer Science, Vol. 1784, pp. 98–113, Springer-Verlag, New York/Berlin, 2000.

15. H. Ehrig, Introduction to the algebraic theory of graph grammars, *in* Claus *et al.* [10, pp. 1–69].

16. H. Ehrig, G. Engels, R. Heckel, and G. Taentzer, A view-oriented approach to system modelling using graph transformation, *in* "Proc. ESEC/FSE '97," Lecture Notes in Computer Science, Vol. 1301, pp. 327–343, Springer-Verlag, New York/Berlin, 1997.

17. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (Eds.), "Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II. Applications, Languages, and Tools," World Scientific, Singapore, 1999.

18. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg (Eds.), "Handbook of Graph Grammars and Computing by Graph Transformation. Vol. III. Concurrency, Parallelism, and Distribution," World Scientific, Singapore, 1999.

19. G. Engels, H. Ehrig, R. Heckel, and G. Taentzer, A view-based approach to system modelling based on open graph transformation systems, *in* Ehrig *et al.* [17, Chap. 16, pp. 639–668].

20. G. Engels and R. Heckel, Graph transformation as a conceptual and formal framework for system modelling and evolution, *in* "Proc. Automata, Languages, and Programming (ICALP 2000)," Lecture Notes in Computer Science, Vol. 1853, pp. 127–150, Springer-Verlag, New York, 2000.

21. G. Engels and A. Schürr, Encapsulated hierarchical graphs, graph types, and meta types, *in* "Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation," Electronic Notes in Theoretical Computer Science, Vol. 2, http://www.elsevier.nl/locate/entcs, 1995.

22. C. Ermel, M. Rudolf, and G. Taentzer, The AGG approach: Language and environment, *in* Ehrig *et al.* [17, Chap. 14, pp. 551–603].

23. P. Fradet and D. Le Métayer, Structured Gamma, *Sci. Comput. Programm.* **31** (1998), 263–289.

24. R. W. Glauert, J. Kennaway, and M. Ronan Sleep, DACTL: An experimental graph rewriting language, *in* "Graph Grammars and Their Application to Computer Science, Selected Papers," Lecture Notes in Computer Science, Vol. 532, pp. 378–395, Springer-Verlag, New York/Berlin, 1991.

25. A. Habel, "Hyperedge Replacement: Grammars and Languages," Lecture Notes in Computer Science, Vol. 643, Springer-Verlag, New York/Berlin, 1992.

26. A. Habel, J. Müller, and D. Plump, Double-pushout graph transformation revisited, *Math. Structures Comput. Sci.* **11** (2001), 637–688.

27. R. Heckel, H. Ehrig, G. Engels, and G. Taentzer, Classification and comparison of module concepts for graph transformation system, *in* Ehrig *et al.* [17, Chap. 17, pp. 669–689].

28. B. Hoffmann, From graph transformation to rule-based programming with diagrams, *in* "Proc Applications of Graph Transformations with Industrial Relevance (AGTIVE '99), Selected Papers," Lecture Notes in Computer Science, Vol. 1779, pp. 165–180, Springer-Verlag, New York, 2000.

29. B. Hoffmann, Shapely hierarchical graph transformation, *in* "Proc. Int. Symposium of Visual Languages and Formal Methods (VLFM '01)," IEEE Computer Press, 2001, pp. 30–37.

30. M. Löwe and M. Beyer, AGG—An implementation of algebraic graph rewriting, *in* "Proc. Rewriting Techniques and Applications," Lecture Notes in Computer Science, Vol. 690, pp. 451–456, Springer-Verlag, New York/Berlin, 1993.

31. M. Minas, Concepts and realization of a diagram editor generator based on hypergraph transformation, *Sci. Comput. Programm.*, to appear.

32. F. Newbery Paulisch, "The Design of an Extendible Graph Editor," Lecture Notes in Computer Science, Vol. 704, Springer-Verlag, New York/Berlin, 1992.

33. F. Parisi-Presicce and G. Piersanti, Multi-level graph grammars, *in* "Graph-Theoretical Concepts in Computer Science (WG '94)," Lecture Notes in Computer Science, Vol. 903, pp. 51–64, Springer-Verlag, New York/Berlin, 1995.

34. D. Plump, Term graph rewriting, *in* "Handbook of Graph Grammars and Computing by Graph Transformation," Vol. 2, Chap. 1, pp. 3–61, World Scientific, Singapore, 1999.

35. D. Plump and A. Habel, Graph unification and matching, *in* Cuny *et al.* [12, pp. 75–89].

36. A. Poulovassilis and S. G. Hild, Hyperlog: A graph-based system for database browsing, querying, and update, *IEEE Knowledge Data Eng.* **13** (2001), 216–333.

37. T. W. Pratt, Pair grammars, graph languages and string-to-graph translations, *J. Comput. System Sci.* **5** (1971), 560–595.

38. T. W. Pratt, Definition of programming language semantics using grammars for hierarchical graphs, *in* Claus *et al.* [10, pp. 389–400].

39. P. J. Rodgers, A graph rewriting programming language for graph drawing, *in* "Proc. 14th IEEE Symposium on Visual Languages," IEEE Computer Society Press, 1998.

40. G. Rozenberg (Ed.), "Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I. Foundations," World Scientific, Singapore, 1997.

41. J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modelling Language Reference Manual," Object Technology Series, Addison–Wesley, Reading, MA, 1999.

42. H.-J. Schneider, On categorical graph grammars integrating structural transformations and operations on labels, *Theoret. Comput. Sci.* **109** (1993), 257–274.

43. A. Schürr, A. Winter, and A. Zündorf, The PROGRES approach: Language and environment, *in* "Handbook of Graph Grammars and Computing by Graph Transformation," Vol. 2, Chap. 13, pp. 487–550, World Scientific, Singapore, 1999.

44. G. Taentzer, Hierarchically distributed graph transformation, *in* Cuny *et al.* [12, pp. 304–320].

45. G. Taentzer and A. Schürr, DIEGO, another step towards a module concept for graph transformation systems, *in* "Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation," Electronic Notes in Theoretical Computer Science, Vol. 2, http://www.elsevier.nl.locate/entcs, 1995.

46. J. Tapken, Implementing hierarchical graph structures, *in* "Proc. Formal Aspects of Software Engineering (FASE'99)," Lecture Notes in Computer Science, Vol. 1577, Springer-Verlag, New York/Berlin, 1999.