To appear in Proc. FOSSACS 2000, LNCS, March 2000

Hierarchical Graph Transformation*

Frank Drewes[1], Berthold Hoffmann[1], and Detlef Plump[2] **

[1] Fachbereich Mathematik/Informatik, Universität Bremen
Postfach 33 04 40, D-28334 Bremen, Germany
{drewes, hof}@informatik.uni-bremen.de

[2] Department of Computing and Electrical Engineering
Heriot-Watt University, Edinburgh EH14 4AS, Scotland
det@cee.hw.ac.uk

Abstract. We present an approach for the rule-based transformation of hierarchically structured (hyper)graphs. In these graphs, distinguished hyperedges contain graphs that can be hierarchical again. Our framework extends the double-pushout approach from flat to hierarchical graphs. In particular, we show how to construct recursively pushouts and pushout complements of hierarchical graphs and graph morphisms. To further enhance the expressiveness of the approach, we also introduce rule schemata with variables which allow to copy and to remove hierarchical subgraphs.

1 Introduction

Recently, the idea of using rule-based graph transformation as a framework for specification and programming has received some attention, and several researchers have proposed structuring mechanisms for graph transformation systems to make progress towards this goal (see for example [2, 8, 10]). Structuring mechanisms will be indispensable to manage large numbers of rules and to develop complex systems from small components that are easy to comprehend. Moreover, we believe that it will be necessary to structure the graphs that are subject to transformation, too, in order to cope with applications of a realistic size. A mechanism for hiding (or abstracting from) subgraphs in large graphs will facilitate both the control of rule applications and the visualization of graphs.

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 well-known double-pushout approach to graph transformation [5, 3] extends smoothly to these hierarchical (hyper)graphs, 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.

* This work has been partially supported by the ESPRIT Working Group *Applications of Graph Transformation* (APPLIGRAPH) and by the TMR Research Network GETGRATS through the University of Bremen.

** On leave from Universität Bremen.

To further enhance the expressiveness of hierarchical graph transformation for programming purposes (without damaging the theory), we also introduce rule schemata containing frame variables. These variables can be instantiated with frames containing hierarchical graphs, and can be used to copy or remove frames without looking at their contents. Our running example of a queue implementation indicates that this concept is useful, as it allows to delete and to duplicate queue entries regardless of their structure and size.

Finally, we relate 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, yielding a flat graph without frames. Every transformation step on hierarchical graphs—under a mild assumption on the transformed graph—gives rise to a conventional step on the flattened graphs by using the flattened rule.

2 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 $s \in S$.

A *pushout* in a category $\mathbb{C}$ (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 $n \circ n_2 = n'_2$.

Let L be an arbitrary but fixed set of labels. A *hypergraph* H is a quintuple $(V_H, E_H, att_H, lab_H, p_H)$ such that

- V_H and E_H are finite sets of *nodes* and *hyperedges*, respectively,
- $att_H\colon E_H \to V_H^*$ is the *attachment function*,
- $lab_H\colon E_H \to L$ is the *labelling function*, and
- $p_H \in V_H^*$ is a sequence of nodes, called the *points* of H.

In the following, we will 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.

A *morphism* $m\colon G \to H$ between graphs G and H is a pair (m_V, m_E) of mappings $m_V\colon V_G \to V_H$ and $m_E\colon E_G \to E_H$ such that $m_V(p_G) = p_H$ and, 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 $m\colon G \to H$ then G and H are *isomorphic*, which is denoted by $G \cong H$. For a morphism $m\colon G \to H$ 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.

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

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

for all edges $e \in E_G \cup E_H$. (If $A_G \cap A_H \neq \emptyset$, we assume that some implicit renaming of atoms takes place.) Notice that this operation is associative but does *not* commute since $G + H$ inherits its points from the first argument.

We recall the following well-known facts about pushouts and pushout complements in the category of graphs and graph morphisms (see [5]). Let $m_1 : G \to H_1$ 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 $a \sim a'$ (i.e., H is the quotiont 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 (i.e., 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 only need the case where both of the given morphisms are injective. In this case it is sufficient to assume that the *dangling condition* is satisfied. Two morphisms $m_1 : G \to H_1$ and $n_1 : H_1 \to H$ satisfy the dangling condition 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 m_1 and n_1 satisfy the dangling condition. Furthermore, if they exist, then m_2 and n_2 are uniquely determined (up to isomorphism).

A *transformation rule* (*rule*, for short) is a pair $t \colon L \xleftarrow{l} I \xrightarrow{r} R$ of morphisms $l \colon I \to L$ and $r \colon I \to R$ such that l is injective. L, I, and R are the *left-hand side*, *interface*, and *right-hand side* of t. A graph G can be transformed into a graph H by an application of t, denoted by $G \Rightarrow_t H$, if there is an injective morphism $o \colon L \to G$, called an occurrence morphism, such that two pushouts

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

exist. It follows from the facts about pushouts and pushout complements recalled above that such a diagram exists if and only if l and o satisfy the dangling condition, and in this case H is uniquely determined up to isomorphism. Notice that we only consider injective occurrence morphisms, which is done in order to avoid additional difficulties when considering the hierarchical case. On the other hand, the morphism r of a rule $t \colon L \xleftarrow{l} I \xrightarrow{r} R$ is allowed to be non-injective.

3 Hierarchical Graphs

Graphs as defined in the previous section are flat. If someone wished to implement, say, some complicated abstract data type by means of graph transformation, there would be no structuring mechanisms available, except for the possibilities the graphs themselves provide. Thus, any structural information would have to be coded into the graphs, a solution which is usually 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. In fact, it turns out to be useful to be even more general by allowing some frames to contain variables instead of graphs. These structures will be called hierarchical graphs.

Let $\mathcal{X}$ be a set of symbols called *variables*. The class $\mathcal{H}(\mathcal{X})$ of *hierarchical graphs with variables in* $\mathcal{X}$ consists of triples $H = \langle G, F, cts \rangle$ such that G is a graph (the root of the hierarchy), $F \subseteq E_G$ is the set of *frame edges* (or just *frames*), and $cts\colon F \to \mathcal{H}(\mathcal{X}) \cup \mathcal{X}$ assigns to each frame $f \in F$ its *contents* $cts(f) \in \mathcal{H}(\mathcal{X}) \cup \mathcal{X}$. Formally, $\mathcal{H}(\mathcal{X})$ is defined inductively over the depth of frame nesting, as follows. A triple $H = \langle G, F, cts \rangle$ as above is in $\mathcal{H}_0(\mathcal{X})$ if $F = \emptyset$. In this case, H may be identified with the graph G. For $i > 0$, $H \in \mathcal{H}_i(\mathcal{X})$ if $cts(f) \in \mathcal{H}_{i-1}(\mathcal{X}) \cup \mathcal{X}$ for every frame $f \in F$. Finally, $\mathcal{H}(\mathcal{X})$ denotes the union of all these classes: $\mathcal{H}(\mathcal{X}) = \bigcup_{i \geq 0} \mathcal{H}_i(\mathcal{X})$. (Notice that $\mathcal{H}_i(\mathcal{X}) \subseteq \mathcal{H}_{i+1}(\mathcal{X})$ for all $i \geq 0$. We have $\mathcal{H}_0(\mathcal{X}) \subseteq \mathcal{H}_1(\mathcal{X})$ because an empty set of frames trivially satisfies the requirement; using this, $\mathcal{H}_i(\mathcal{X}) \subseteq \mathcal{H}_{i+1}(\mathcal{X})$ follows by an obvious induction on $i \geq 0$.) The sets $\mathcal{H}(\emptyset)$ and $H_i(\emptyset)$ $(i \geq 0)$ are briefly denoted by $\mathcal{H}$ and $\mathcal{H}_i$, respectively. These *variable-free* hierarchical graphs are those in which we are mainly interested.

Notice that, to avoid unnecessary restrictions, the definition of a hierarchical graph $H = \langle G, F, cts \rangle$ does not impose any relation between the nodes and edges of G and those of $cts(f)$, $f \in F$. Restrictions of this kind may be added for specific application areas, but the results of this paper hold in general.

Example 1 (Queue graphs). As a running example, we show how queues and their typical operations can be implemented using hierarchical graph transformation. Two kinds of frames are used to represent queues as hierarchical graphs: Unary *item frames* contain the graphs stored in the queue; binary *queue frames* contain a *queue graph*, which is a chain of edges connecting their *begin* point to their *end* point, every node in between carrying an item frame.

Figure 1 shows two queue frames. Nodes are drawn as circles, and filled if they are points. Edges are drawn as boxes, and connected to their attachments

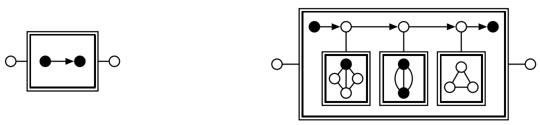
(a) 

Fig. 1. Two queue frames representing (a) an empty queue (b) a queue of length 3

by lines that are ordered counter-clockwise, starting at noon. Frames have double lines, and their contents is drawn inside.Plain binary edges are drawn as arrows from their first to their second attachment (as in simple graphs). In our examples, their labels do not matter, and are omitted. (In the item graphs, the arrowheads are omitted too.) Frame labels are not drawn either, as queue and item frames can be distinguished by their arity.

Note that item frames may contain graphs of any arity; in Figure 1 (b), they have 1, 2, and no points, respectively.

Unless they are explicitly named, the three components of a hierarchical graph H are denoted by $\overline{H}$, F_H, and cts_H, respectively. The notations V_H, E_H, att_H, lab_H, p_H, and A_H are used as abbreviations denoting $V_{\overline{H}}$, $E_{\overline{H}}$, $att_{\overline{H}}$, $lab_{\overline{H}}$, $p_{\overline{H}}$, and $A_{\overline{H}}$, respectively. Furthermore, we denote by X_H the set $\{f \in F_H \mid cts_H(f) \in \mathcal{X}\}$ 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 $A_G \cap A_H = \emptyset$. The *disjoint union* of G and H is denoted by $G + H$ and yields the hierarchical graph K such that $\overline{K} = \overline{G} + \overline{H}$, $F_K = F_G \cup F_H$, 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 $S = \{H_1, \ldots, H_n\}$ of hierarchical graphs, we denote $G + H_1 + \cdots + H_n$ by $G + \sum_{H \in S} H$. (Notice that, although the disjoint union of hierarchical graphs does not commute, this is well defined as it does not depend on the order of $H_1, \ldots, H_n$).

We will now generalize the concept of morphisms to the hierarchical case. The definition is quite straightforward. Such 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 they can also be mapped to any other frame carrying the right label.

Formally, let $G, H \in \mathcal{H}(\mathcal{X})$. A *hierarchical morphism* $h\colon G \to H$ is a pair $h = \langle \overline{h}, (h^f)_{f \in F_G \setminus X_G} \rangle$ where

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

For atoms $a \in A_G$, we usually write $h(a)$ instead of $\overline{h}(a)$. Furthermore, a hierarchical morphism $h\colon G \to H$ for which $G, H \in \mathcal{H}_0$ is identified with $\overline{h}$.

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 $\overline{l} = \overline{h} \circ \overline{g}$ 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 $\overline{g}$ is injective and, for all $f \in F_G \setminus X_G$, g^f is injective. It is *surjective up to variables* if $\overline{g}$ 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 surjective up to variables and injective. If G does not contain variables, we speak of surjective and bijective hierarchical morphisms. A bijective hierarchical morphism is also called an *isomorphism*, and $G, H \in \mathcal{H}$ are said to be *isomorphic*, $G \cong H$, if there is an isomorphism $m\colon G \to H$.

Let $\mathbb{H}$ be the category whose objects are variable-free hierarchical graphs and whose morphisms are the hierarchical morphisms $h\colon G \to H$ with $G, H \in \mathcal{H}$ (which is indeed a category, as one can easily verify). The main result we are going to establish in order to obtain a notion of hierarchical graph transformation is that $\mathbb{H}$ has pushouts. For this, looking at the inductive definition of hierarchical graphs and their morphisms, it is a rather obvious idea to proceed by induction on the depth of the frame nesting. The induction basis is then provided by the non-hierarchical case recalled in Section 2. In order to use the induction hypothesis, we have to reduce the depth of a hierarchical graph in some way. This can be done on the basis of a rather simple construction. Given a hierarchical graph $H \in \mathcal{H}_i$, we take the contents of its frames out of these frames (which, thereby, become ordinary edges) and add them disjointly to $\overline{H}$, thus obtaining a hierarchical graph in $\mathcal{H}_{i-1}$ (provided that $i > 0$). Denoting this mapping by φ, we get the desired theorem, which 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 1. *For every pair $m_1 : G \to H_1$ and $m_2 : G \to H_2$ of morphisms in $\mathbb{H}$ there are morphisms $n_1 : H_1 \to H$ and $n_2 : H_2 \to H$ in $\mathbb{H}$ (for some hierarchical graph 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 sketch. The proof works by induction on i, where $H_1, H_2 \in \mathcal{H}_i$. The case $i = 0$ is the non-hierarchical one, and it is easy to see that every pushout in the category of non-hierarchical graphs and morphisms is a pushout in $\mathbb{H}$ as well. Thus, let $i > 0$. Extending φ to morphisms in the canonical way, one obtains $\varphi(m_1) = (m'_1 : G' \to H'_1)$ and $\varphi(m_2) = (m'_2 : G' \to H'_2)$ where $H'_1, H'_2 \in \mathcal{H}_{i-1}$. By the induction hypothesis, this yields a pushout (m'_1, m'_2, n'_1, n'_2) for some $n'_j : H'_j \to H'$ $(j \in \{1, 2\})$. Now, it can be shown that $n'_j = \varphi(n_j)$ for hierarchical morphisms $n_j : H_j \to H$, yielding a commuting square (m_1, m_2, n_1, n_2). Intuitively, the parts of H' which stem from the contents of a frame f in H_j can be stored in $n'_j(f)$, turning this edge into a frame of the hierarchical graph H constructed. The main part of the proof is to show that H and the hierarchical morphisms n_j obtained in this way are well defined.

Finally, one has to verify the universal pushout property of (m_1, m_2, n_1, n_2). Let $l_1 : H_1 \to L$ and $l_2 : H_2 \to L$ be such that (m_1, m_2, l_1, l_2) commutes and let $\varphi(l_j) = (l'_j : H' \to L')$ for $j \in \{1, 2\}$. Then (m'_1, m'_2, l'_1, l'_2) commutes as well. Therefore, the pushout property of (m'_1, m'_2, n'_1, n'_2) yields a unique morphism $l' : H' \to L'$ such that $l'_j = l' \circ n'_j$. Again, l' can be turned into $l : H \to L$ with $l' = \varphi(l)$ and $l_j = l \circ n_j$ for $j \in \{1, 2\}$. Furthermore, for $k : H \to L$ with $k \neq l$ we have $\varphi(k) \neq \varphi(l)$, which shows that l is unique, by the uniqueness of l'. □

Notice that the proof of Theorem 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 a corollary for 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 1.

Corollary 1. *Let $m_1 \colon G \to H_1$ and $m_2 \colon G \to H_2$ be injective hierarchical morphisms in $\mathbb{H}$. Then, one can construct hierarchical 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, 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 $f \in F_G$, $n_1^{m_1(f)}$ and $n_2^{m_2(f)}$ are constructed recursively in such a way 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 1 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 [5].

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 is deleted as well. Let $H_1 \in \mathcal{H}(\mathcal{X})$ and $G, H \in \mathcal{H}$ (right below, we shall only use the following definition for $H_1 \in \mathcal{H}$, but later on the more general case $H_1 \in \mathcal{H}(\mathcal{X})$ will turn out to be valuable, too). Two hierarchical morphisms $m \colon I \to L$ and $n \colon L \to G$ satisfy the *hierarchical dangling condition* (*dangling condition*, for short) if

- $\overline{m}$ and $\overline{n}$ satisfy the (non-hierarchical) dangling condition,
- for every frame $f \in F_L \setminus (m(F_I) \cup X_L)$, n^f is bijective up to variables, and
- for every frame $f \in F_I \setminus X_I$, m^f and $n^{m(f)}$ satisfy the dangling condition.

Notice that this condition coincides with the usual one in the special case where m and n are ordinary graph morphisms, because in this case 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 is deleted as well (at least in the case where $L \in \mathcal{H}$; the more general case is not yet our concern). As the proof below shows, this corresponds to the last item in Corollary 1 (and is thus indeed necessary).

Theorem 2. *Let $m_1 \colon G \to H_1$ and $n_1 \colon H_1 \to H$ be injective hierarchical morphisms in $\mathbb{H}$. Then there are hierarchical morphisms $m_2 \colon G \to H_2$ and $n_2 \colon H_2 \to H$ such that (m_1, m_2, n_1, n_2) is a pushout, if and only if m_1 and n_1 satisfy the dangling condition. In this case m_2 and n_2 are uniquely determined.*

Proof. Let $G \in \mathcal{H}_i$. 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 1 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 m_1 and n_1 satisfy the dangling condition, $\overline{m_2}$ and $\overline{n_2}$ exist and are uniquely determined (since $\overline{m_1}$ and $\overline{n_1}$ satisfy the dangling condition for non-hierarchical morphisms), and (3) is satisfied for $i = 1$ (because of 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 where $i = 2$) this determines m_2 and n_2 up to isomorphism, thus finishing the proof. □

4 Hierarchical Graph Transformation

Based on the results presented in 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* $t\colon L \xleftarrow{l} I \xrightarrow{r} R$ is assumed to consist of two hierarchical morphisms $l\colon I \to L$ and $r\colon I \to R$, where $L, I, R \in \mathcal{H}$ and l is injective. The hierarchical graphs L, I, and R are called the *left-hand side*, *interface*, and *right-hand side*.

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 top level) one has to employ recursion.

Definition 1 (***Transformation of hierarchical graphs***)**.** Let $t\colon L \xleftarrow{l} I \xrightarrow{r} R$ be a rule. A hierarchical graph $G \in \mathcal{H}$ is transformed into a hierarchical graph $H \in \mathcal{H}$ 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) $\overline{H} \cong \overline{G}$ via some isomorphism $m\colon \overline{G} \to \overline{H}$, 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 $t \in T$.

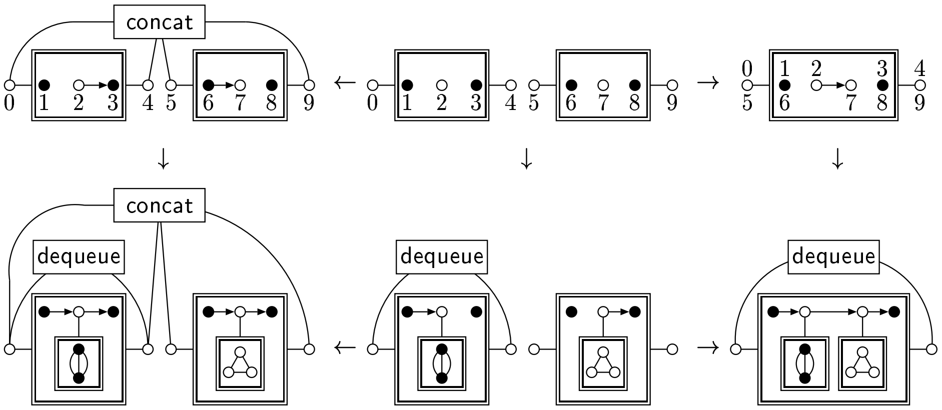


Fig. 2. The concatenation rule and its application

Example 2 (Concatenation of queues). In Figure 2, we show a concatenation rule for queues that identifies two queue frames and concatenates their contents, and a transformation with this rule. The digits in the rule indicate how the nodes of the graphs have to be mapped onto each other.

It should be noticed that the definition of transformation steps requires occurrence morphisms to be injective. Therefore, we need three variants of this rule where node 1 is identified with node 2, or 7 with 8, or both 1 with 2 and 7 with 8. (Similar variants are needed for the rules in the subsequent examples.)

Since occurrence morphisms are injective, we get the following theorem as a consequence of Theorems 1 and 2.

Theorem 3. *Let* $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ *be a rule,* $G \in \mathcal{H}$*, and* $o\colon L \to G$ *an occurrence morphism. Then the two pushouts in Definition 1(1) exist if and only if* o *satisfies the dangling condition.*[1] *Furthermore, in this case the pushouts are uniquely determined up to isomorphism.*

Proof. By Theorem 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. Finally, by Theorem 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. □

The reader should also notice that, as a consequence of the effectiveness of the results presented in Section 3, given a tranformation rule, a hierarchical graph, and an occurrence morphism satisfying the dangling condition, one can effectively construct the required pushouts.

[1] If the rule $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ in question is clear we say that o satisfies the dangling condition if l and o do.

Unfortunately, the notion of transformation of hierarchical graphs is not yet expressive enough to be satisfactory for certain programming purposes. There are some natural effects that one would certainly like to be able to implement as single transformation steps, but which cannot be expressed by rules. Consider the example of queues, for instance. It should be possible to design a rule *dequeue* which removes the first item in a queue, *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. Conversely, another rule *enqueue* should take an *item* frame, again regardless of its contents, and add it to the queue—preferably without affecting the original *item* frame. In order to implement this, one has 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 duplicate frames together with their contents.

This is where variables start to play an important role. 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. For this, a hierarchical graph H is called *linear* if no variable occurs twice in H.

A *variable instantiation for* $H \in \mathcal{H}(\mathcal{X})$ is a mapping $\imath\colon var(H) \to \mathcal{H}$. The application of $\imath$ to H is denoted by $H\imath$. It turns every variable frame $f \in X_H$ into a frame whose contents is $\imath(cts_H(f))$. By the definition of hierarchical morphisms, for every hierarchical morphism $h\colon G \to H$ such that $G \in \mathcal{H}$ and every variable instantiation $\imath$ for H, h can as well be understood as a hierarchical morphism from G to $H\imath$. In the following, this hierarchical morphism will be denoted by $h\imath$. Based on this observation, rule schemata and their application can be defined.

Definition 2 (***Transformation by rule schemata***)**.** A *rule schema*, denoted by $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$, is a pair consisting of hierarchical morphisms $l\colon I \to L$ and $r\colon I \to R$, where $L, R \in \mathcal{H}(\mathcal{X})$, $I \in \mathcal{H}$, L is linear, and $var(R) \subseteq var(L)$. If $\imath$ is a variable instantiation for L then the rule $t'\colon L\imath \overset{l\imath}{\leftarrow} I \overset{r\imath}{\rightarrow} R\imath$ is an *instance* of t.

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

Example 3 (The rule schemata enqueue *and* dequeue*).* In Figure 3, we show a rule schema that inserts a framed item graph at the tail of a queue graph, and a transformation with that rule. The item frame contains the variable x. Otherwise, it would not be possible to duplicate the item graph, and to move it into the queue frame.

In Figure 4, we show a rule schema that removes the first item frame in a queue graph. The item graph is denoted by the variable x so that it can be removed entirely.

For practical purposes Definition 2 is not very convenient because there are infinitely many instances of a rule schema as soon as it contains at least one

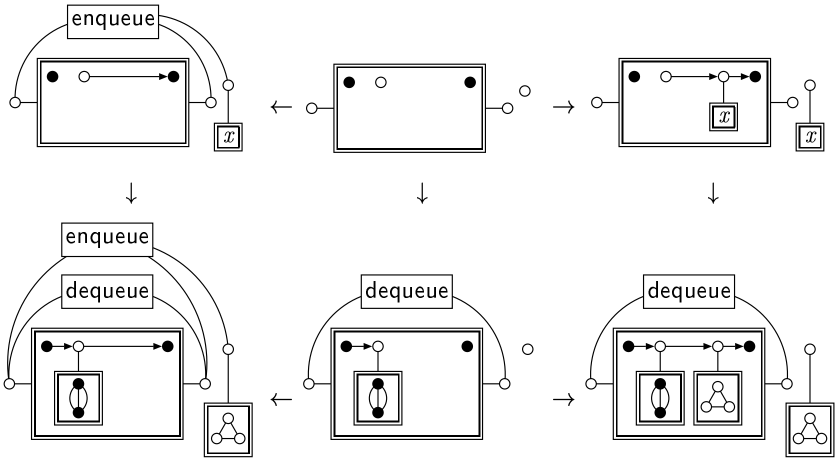


Fig. 3. The rule schema enqueue and its application

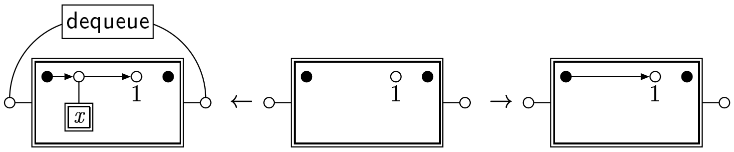


Fig. 4. The rule schema dequeue

variable. Therefore, the naive approach to implement $\Rrightarrow_t$ by constructing all its instances and then testing each of them for applicability does not work. However, there is quite an obvious way how one can do better than that. Consider some linear hierarchical graph $L \in \mathcal{H}(\mathcal{X})$ and a hierarchical graph $G \in \mathcal{H}$, and let $o\colon L \to G$ be a hierarchical morphism. Then, due to the linearity of L, o induces a variable instantiation $\imath_o\colon var(L) \to \mathcal{H}$ and an occurrence morphism $inst(o)\colon L\imath_o \to G$, as follows. For all $x \in var(L)$, if there is some $f \in X_L$ such that $cts_L(f) = x$ then $\imath_o(x) = cts_G(o(f))$. Otherwise, $\imath_o(x) = \imath_{o^f}(x)$, where $f \in F_L \setminus X_L$ is the unique frame such that $x \in var(cts_L(f))$. Furthermore, $\overline{inst(o)} = \overline{o}$ 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 theorem below states that the transformations given by a rule schema $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ can be obtained by considering occurrence morphisms $o\colon L \to G$ that satisfy the dangling condition.

Theorem 4. *Let $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ be a rule schema and $G \in \mathcal{H}$.*

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

2. *If $\imath\colon var(L) \to \mathcal{H}$ is a variable instantiation and $q\colon L\imath \to G$ is an occurrence morphism satisfying the dangling condition, then $\imath = \imath_o$ and $q = inst(o)$ (up to isomorphism) for some occurrence morphism $o\colon L \to G$ satisfying the dangling condition.*

The proof by induction on i, where $L \in \mathcal{H}_i(\mathcal{X})$, is rather straightforward and is therefore skipped in this short version.

5 Flattening

A natural operation on hierarchical graphs is the *flattening operation* which removes the hierarchy by recursively replacing every frame with its contents. For this, we use the well-known concept of hyperedge replacement (see [9, 4]) in a slightly generalized form. Flattening is similar to (a recursive version of) the operation φ considered in Section 3, but it removes all frames and identifies their attached nodes with the corresponding points of their contents. If the numbers of attached nodes and points differ, the additional nodes of the longer sequence are treated like ordinary nodes. In addition, flattening forgets about the points of its argument, so that the resulting graph is "unpointed".

It will be shown in this section that, under modest assumptions, hierarchical graph transformation is compatible with the flattening operation: A transformation $G \Rightarrow_t H$ induces a corresponding transformation $G' \Rightarrow_{t'} H'$, where G', H', and t' are the flattened versions of G, H, and t, respectively.

In order to proceed, we first need to define hyperedge replacement on hierarchical graphs. Let H be a hierarchical graph and consider a mapping $\sigma\colon E \to \mathcal{H}$ such that $E \subseteq E_H$, called a *hyperedge substitution for H*. Hyperedge replacement yields the hierarchical graph $H[\sigma]$ obtained from $H + \sum_{e\in E} \sigma(e)$ by deleting the edges in E and identifying, for all $e \in E$, the ith node of $att_H(e)$ with the ith point of $p_{\sigma(e)}$, for all i such that both these nodes exist.

Finally, for all $H \in \mathcal{H}$, let $fl(H) = H[\sigma]$ where $\sigma\colon F_H \to \mathcal{H}$ is given inductively by $\sigma(f) = fl(cts_H(f))$ for all $f \in F_H$. Then, the *flattening* of H yields the graph $flat(H) = \langle V_{fl(H)}, E_{fl(H)}, att_{fl(H)}, lab_{fl(H)}, \lambda\rangle$. For most of the considerations below, it is sufficient to study the mapping fl, which removes the hierarchy without forgetting points, instead of $flat$.

We can flatten morphisms as well. Consider a hierarchical morphism $h\colon G \to H$ with $G, H \in \mathcal{H}$ and let $\sigma = fl \circ cts_G$ and $\tau = fl \circ cts_H$. Then, $fl(h)$ is the morphism $m\colon fl(G) \to fl(H)$ defined inductively, as follows. For all $a \in A_{fl(G)}$, if $a \in A_G$ then $m(a) = h(a)$, and if $a \in A_{\sigma(f)}$ for some $f \in F_G$ then $m(a) = fl(h^f)(a)$. Furthermore, $flat(h) = (m'\colon flat(G) \to flat(H))$ is given by $m'(a) = m(a)$ for all $a \in A_{flat(G)}$. (Notice that, although the two cases in the definition of $m(a)$ above intersect, they are consistent with each other.)

Above, it was mentioned that the main result of this section holds only under a certain assumption. The reason for this 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 building $fl(G)$ 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.

For this, call a hierarchical graph $H \in \mathcal{H}$ *identification consistent* if every frame $f \in F_H$ satisfies the following:

(1) For all $i, j \in [\min(|att_H(f)|, |p_{cts_H(f)}|)]$, $att_H(f)(i) = att_H(f)(j)$ if and only if $p_{cts_H(f)}(i) = p_{cts_H(f)}(j)$, and
(2) $cts_H(f)$ is identification consistent.

The reader ought to 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 derivable hierarchical graphs are identification consistent (provided that the initial ones are).

It is not very difficult to verify the following two lemmas.

Lemma 1. *For every injective hierarchical morphism $h\colon G \to H$ $(G, H \in \mathcal{H})$ such that H is identification consistent, $fl(h)$ is injective.*

Lemma 2. *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.*

As a consequence, one obtains 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 5. *Let $t\colon L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ be a rule and let $t'\colon L' \overset{l'}{\leftarrow} I' \overset{r'}{\rightarrow} R'$ 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 sketch. Consider a transformation step $G \Rightarrow_t H$. Due to the definition of $\Rightarrow_t$ there are two cases to be distinguished. If there is a double-pushout diagram as in the first case of Definition 1, Lemmas 1 and 2 yield a corresponding "flattened" diagram. The second case to be considered is the recursive one, i.e., the transformation takes place inside a frame f. In this case it may be assumed inductively that the diagram corresponding to a transformation of the flattened contents of f exists. Due to the assumed identification consistency the flattened contents of f is injectively embedded in $flat(G)$. Therefore, the given diagram can be extended to a larger pushout diagram in the required way, retaining the injectivity of the occurrence morphism. □

It should be noticed that the flattening process implies a loss of crucial structural information so that there is no chance to prove the converse of the theorem.

6 Conclusion

We conclude this paper by briefly mentioning some related work and possible directions for future research.

Pratt [15] was probably the first to consider a concept of hierarchical graph transformation, where he used a certain kind of node replacement to define the semantics of programming languages. His graph concept was extended in [6] by allowing edges between subgraphs contained in different nodes, but without defining transformation.

A different concept of graph nesting is given by the abstraction mechanisms of the (old) graph transformation system AGG [12] and the multi-level graph grammars of [13], providing flat graphs with several views which are related by a rigid layering and a partial inclusion ordering, respectively.

An indirect nesting concept can be found in the framework of [16] and the new AGG system [7], where nesting is realized by labels and attributes, respectively.

The idea of using variables to extend the double-pushout approach with non-local effects, like copying and removal of subgraphs, is also followed in the so-called substitution-based approach to graph transformation [14] (working on flat hypergraphs).

One direction for future work on hierarchical graph transformation is to lift to the hierarchical setting the classical results of the double-pushout approach, like sequential and parallel commutativity, results on parallelism, concurrency and amalgamation, etc. Another important task is to combine hierarchical graph transformation in an orthogonal way with concepts for structuring and controlling systems of rules. As mentioned in the introduction, several such concepts (mainly for flat graphs) have recently been proposed in the literature.

A further topic of research is to develop hierarchical graph transformation towards object-oriented graph transformation, as outlined in [11]. 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". In this way frames can be seen as objects of their types that can only be manipulated by invoking the methods of the class.

Acknowledgement We thank the referees for their helpful comments.

References

[1] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. John 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. *Science of Computer Programming*, 34:1–54, 1999.

[3] 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 G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 3, pages 163–245. World Scientific, 1997.

[4] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 2, pages 95–162. World Scientific, Singapore, 1997.

[5] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1979.

[6] G. Engels and A. Schürr. Encapsulated hierachical graphs, graph types, and meta types. In A. Corradini and U. Montanari, editors, *Proc. Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[7] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 551–603. World Scientific, 1999.

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

[9] A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.

[10] R. Heckel, H. Ehrig, and G. Taentzer. Classification and comparison of module concepts for graph transformation systems. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 17, pages 669–689. World Scientific, 1999.

[11] B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl and A. Schürr, editors, *Proc. Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*AGTIVE *'99)*, Lecture Notes in Computer Science, 1999. To appear.

[12] M. Löwe and M. Beyer. AGG — an implementation of algebraic graph rewriting. In C. Kirchner, editor, *Proc. Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456, 1993.

[13] F. Parisi-Presicce and G. Piersanti. Multi-level graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretical Concepts in Computer Science (WG '94)*, volume 903 of *Lecture Notes in Computer Science*, pages 51–64, 1995.

[14] D. Plump and A. Habel. Graph unification and matching. In *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 75–89. Springer-Verlag, 1996.

[15] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[16] H.-J. Schneider. On categorical graph grammars integrating structural transformations and operations on labels. *Theoretical Computer Science*, 109:257–274, 1993.