Abstract Hierarchical Graph Transformation[†]

Giorgio Busatto

Carl v. Ossietzky Universität
Fachbereich Informatik
26111 Oldenburg
email: giorgio.busatto@informatik.uni-oldenburg.de

Hans-Jörg Kreowski, Sabine Kuske

University of Bremen
Department of Computer Science
P.O. Box 330440
D-28334 Bremen
email: {kreo,kuske}@informatik.uni-bremen.de

In this paper, we introduce a new hierarchical graph model to structure large graphs into small components by distributing the nodes (and edges likewise) into a hierarchy of packages. In contrast to other known approaches, we do not fix the type of underlying graphs. Moreover, our model is equipped with a rule-based transformation concept such that hierarchical graphs cannot only be used for static representation of complex system states, but also for the description of the dynamic system behaviour.

1. Introduction

Graphs are very popular structures to represent the relationship among various entities in an intuitive diagrammatic way, and they are used for this purpose in many areas of computer science and beyond. But the comprehensibility of graphs works only as long as the graphs in consideration can be kept small. Unfortunately, there are many applications where one must deal with rather large graphs. As a prominent example, think of the Internet with millions of web pages as nodes and millions of hyperlinks as edges – a graph in which one gets easily lost (see for example (Botafogo *et al.* 1992)). One must also take into account that graphs which represent states of systems may be subject to updates and transformations to reflect the dynamics of systems. Moreover, the notion of graphs is of a quite generic nature offering directed and undirected graphs

[†] This work was partially supported by the ESPRIT Working Group Application of Graph Transformation (APPLIGRAPH) and the EC TMR Network General Theory of Graph Transformation Systems (GETGRATS)

as well as hypergraphs, unlabelled and labelled graphs as well as typed and attributed graphs, and many other variations. In the literature, one encounters quite a variety of remedies for these problems.

1 To deal properly with large graphs, various hierarchical graph models are proposed (Pratt 1979; Harel 1988; Engels and Schürr 1995; Drewes *et al.* 2002; Poulovassilis and Levene 1994; Busatto *et al.* 2000).
2 To generate and update graphs in a suitable way, various rule-based models of graph transformation are developed (Rozenberg 1997; Ehrig *et al.* 1999a; Ehrig *et al.* 1999b).
3 To cover many kinds of graphs within the same framework, the concepts of high-level replacement systems (Ehrig *et al.* 1991a; Ehrig *et al.* 1991b) and of graph transformation approaches (Kreowski and Kuske 1996; Andries *et al.* 1999; Kreowski and Kuske 1999b) are introduced. While the former defines a specific type of rule application for arbitrary categories of graphs, the latter allows even different notions of rule application.

If one specifies graphs by means of rules, the further problem of managing large sets of rules may occur. Again one finds some remedy in the literature.

4. To deal with large graph transformation systems, several structuring concepts are suggested (Kaplan *et al.* 1991; Grosse-Rhode *et al.* 1998; Schürr and Taentzer 1995; Schürr and Winter 2000; Ehrig and Engels 1996; Heckel *et al.* 2000; Kreowski and Kuske 1999b; Drewes *et al.* 2000). While most of them are designed for particular types of graph transformation, the concepts of graph transformation units and modules (see (Kreowski and Kuske 1999a) for a survey) are based on the notion of graph transformation approaches, which is parametric w.r.t. the kind of graphs and rule application to be used.

All known hierarchical graph models concern specific types of graphs and do not take into account the genericity of the graph notion. Moreover, most models do not consider the transformational aspect. Hierarchical graph transformation is rarely studied. On the other hand, although the notions of high-level replacement systems and graph transformation approaches include hierarchical graphs in principle, they are not yet investigated explicitly in such a context.

In the present paper, we try to combine all four aspects. The aim is to propose a hierarchical graph model that provides a structuring principle for graphs independent of their type and that can be equipped with an adequate transformation concept. In Section 3, we introduce an abstract hierarchical graph model that is generic w.r.t. the type of graphs to be structured in a hierarchical way. A hierarchical graph in our sense consists of an underlying graph (of which kind ever), a hierarchy graph and a connection relation. The nodes of the hierarchy graph may be interpreted as packages which contain nodes of the underlying graph and its edges likewise where the containment is described by the connection relation. In this respect, we follow the approach in (Busatto *et al.* 2000) and adopt the structuring ideas of grouping and aggregation of objects as known from object orientation and the database area. Section 4 is devoted to the transformation of hierarchical graphs. If all three components stem from suitable graph transformation approaches, the hierarchical graphs form a resulting hierarchical graph transformation approach such that the rule-based transformation of hierarchical graphs is provided as well as structuring concepts for hierarchical graph transformation systems are made available. As a first major instantiation of our abstract hierarchical graph model, we show in Section 5 that one of the most frequently used graph transformation approaches, the so-called double-pushout (DPO) approach, can be adapted in a suitable way to the graph, hierarchy and connection components of hierarchical graphs such that one gets a hierarchical graph transformation approach based on the DPO approach. As the main technical result of this paper, we show in Section 6 that hierarchical graphs over the DPO approach can be flattened into ordinary graphs in such a way that the transformation of hierarchical graphs is compatible with the transformation of their flattened versions. Throughout the paper, our proposal is illustrated by an example of a simplified version of a distributed project management system starting in Section 2. The paper is concluded by a discussion of related approaches in Section 7 and of future work in Section 8.

2. A Running Example

In this section, we present a running example, that will help us illustrate our discussion in the subsequent sections. We will also give a first intuition of the basic concepts of hierarchical graphs, namely the ideas of *grouping* and *encapsulating* graph elements.

The example considers a particular case of graph modelling, namely the modelling of hypertexts in the world-wide web (WWW). The WWW is a distributed information system based on the Internet. The web contains collections of documents (*nodes*, or *pages*) with cross-references (or *hyperlinks*) between them. Pages are provided by different *web sites*. Hyperlinks are attached to pages through some kind of place-holders called *anchors*. The WWW can be modelled as a large graph, the pages and anchors being the nodes and the hyperlinks being the edges.

To give a concrete example, let us suppose that a software company maintains web pages documenting two projects: a *network traffic analysis* software (NTA), which is meant for the analysis of data traffic on a computer network, and a *network configuration* software (NC) which assists in the design and configuration of computer networks. The pages for these two projects can have links between them, but there can also be links across project boundaries (e.g. regarding software modules that are shared between the two projects). We also want to model the fact that the company has two sites, one in Trento and one in Torino, and that pages can be provided by either of the two sites.

Figure 1 shows a graphical representation of the company's web. Projects, sites, pages and anchors are represented as nodes, where Pi stands for page i and small filled squares denote anchors. Each project is documented by three pages which are not necessarily located at the same web site. Pages are linked through unlabelled directed edges to the project they document and to the site that provides them. Anchors are linked to the pages they belong to by unlabelled directed edges as well. Hyperlinks are depicted as λ-labelled directed edges.

As you can see from the picture, a graphical representation of a hypertext quickly becomes very complex, difficult to understand, and of little use for, say, an author who has to develop and maintain the projects' documentation.

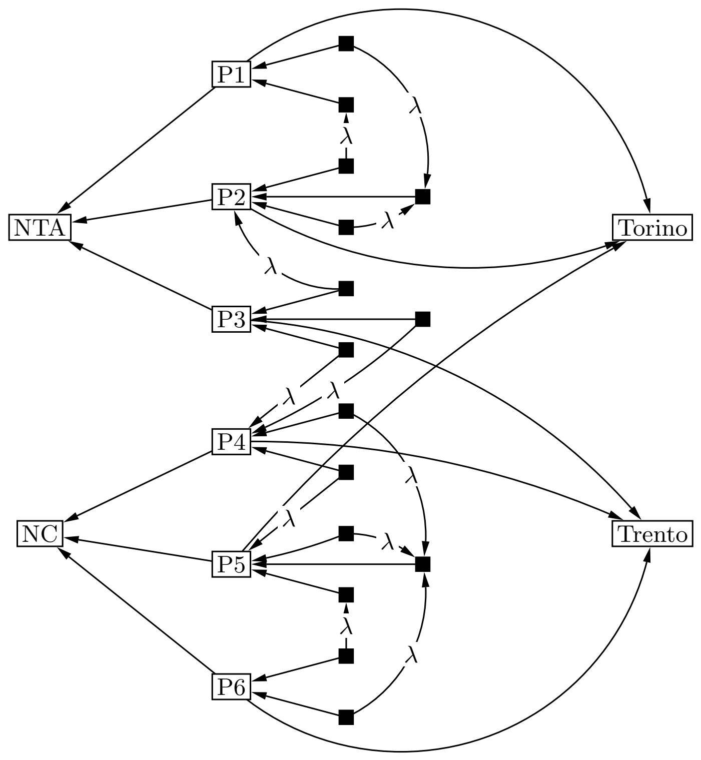

Fig. 1. Web sites, projects, pages, and anchors.

In fact, authors often adopt strategies to handle the complexity of the hypertexts they are developing, taking into account additional information. For example, pages can be grouped according to their contents and every group of pages can be given a start page or home-page, i.e. a page through which all other pages in the group can be accessed. An arbitrary user of the WWW can create hyperlinks to any page inside such a group, but the home-page is the most reliable page to which one can refer, as it is left to the maintainer of the home-page to add the necessary hyperlinks to the other pages in the group. Hence, only start pages should be accessible from the outside.

What we have seen at a very concrete level is an illustration of two general concepts, that are often found together in software engineering, and in many other areas of computer science, namely the concepts of *grouping* and *encapsulation*. These concepts are very well-known, but we wish to briefly recall them here, and to illustrate them in view of our example.

1 *Grouping*. When we have to deal with (design, understand, use) a complex system, it is useful to identify *services* that are provided by groups of components of the system. For example, pages `P4`, `P5`, `P6` provide the service of documenting project NC. By grouping these pages and making this grouping explicit, we can model the *abstraction* associated to them, namely the service they provide.

2 *Encapsulation.* When an external user wants to *use* a service, he or she is not interested in how that service is implemented, but would like to have the minimal knowledge needed to use the service. In this case it is useful to distinguish between those elements in a grouping that *provide access* to the service (the user interface) to an external user, and those that only serve to *implement* the service. The latter elements of our system should be *hidden* from the user or *encapsulated* in the grouping. In our example, we assume that page `P4` is the start page of project NC, and therefore all other pages in the project should be hidden.

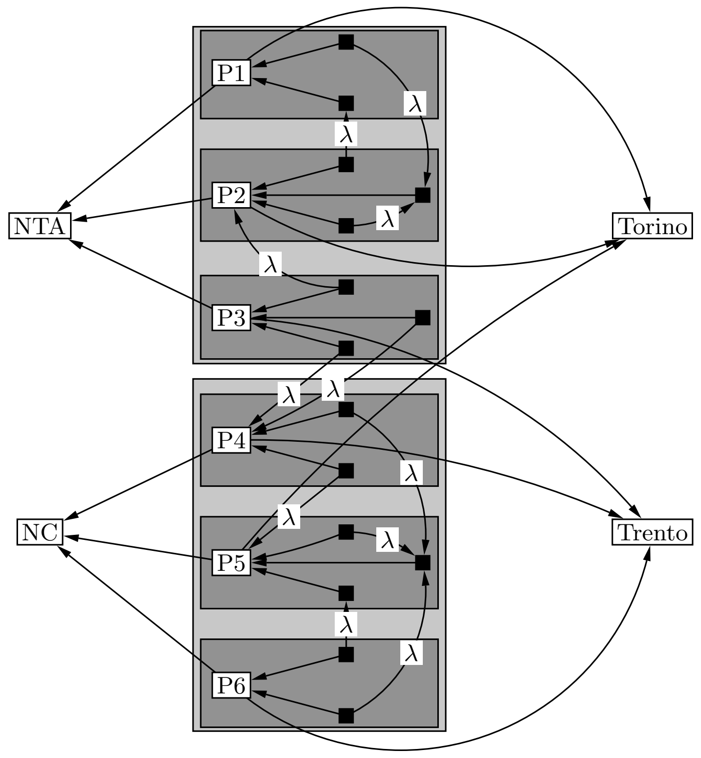

Fig. 2. Web sites, projects, pages, and anchors.

Hierarchical graphs are an extension of ordinary graphs, where the grouping of graph elements into higher-level, layered structures is modelled explicitly. The idea of encapsulation is also present in some hierarchical graph approaches, although less frequently. In this case we speak of *encapsulated hierarchical graphs*.

The hierarchical graph model presented in this paper provides a primitive kind of encapsulation since nodes and edges of a graph are only contained in certain components of the hierarchy (called *packages*). However, we do not provide explicit import/export interfaces between packages to control such containment, and therefore we prefer to use the term *hierarchical graph* instead of *encapsulated hierarchical graph*.

In Figure 2, we show a possible hierarchical decomposition for our running example. We want to capture the following abstractions:

1 *Documentation for one project.* Here we have packaged all the pages and anchors belonging to the documentation of one project in one grouping, depicted by the lightly shaded rectangular areas in the picture. For example, pages P4, P5, and P6, together with their internal anchors, form one such grouping.
2 *Structure of a page.* Here we want to consider a page together with its anchors as an abstraction on its own. This can be useful for an author who wants to change a page while having an overview of its structure. The page groupings are depicted as darker rectangular areas inside project groupings. Notice that the page groupings are nested in the project groupings, thus building a hierarchical structuring of the graph.

Although it is not explicitly depicted in the figure, one can also imagine that nodes are hidden or visible w.r.t. a given grouping. For example, one can imagine that page P4 is visible (exported) w.r.t. project NC, since it is referenced (used) by pages external to the project. On the other hand, pages P5 and P6 should be hidden in the project, since they are not used by any external page. As a consequence, since page P6 is not visible in the documentation of project NTA, it is not possible to draw any edge from an anchor inside that project to P6. This means that graph elements can be *hidden* (*encapsulated*) inside the hierarchy and that this implies *constraints* on the hierarchical graph itself by forbidding some edges.

A hierarchical graph data model must allow to represent this kind of information (hierarchy, visibility), and enforce possible constraints that derive from it (e.g. forbidden edges to hidden nodes). Looking more specifically at our hypertext example, these concepts are not supported in the basic model of the WWW, but more recent hypertext models do support such higher level structuring (see e.g. *collections* in (Maurer 1996)).

Our running example already gives us a first rough idea of the most important aspects of hierarchical graphs. They can be summarised as follows:

1 *Grouping.* Graph elements can be grouped together to form higher level structures.
2 *Encapsulation/visibility.* Graph elements can be hidden inside the hierarchy.
3 *Constraints.* The hierarchical structure and the visibility information can introduce constraints on the underlying graph

As already hinted, we do not consider encapsulation in this paper, although it is considered in some existing approaches (Engels and Schürr 1995; Busatto *et al.* 2000). It can be a future extension to investigate the transformation of hierarchical graphs with encapsulation.

In the next section, we give a formal definition of our hierarchical graph model.

3. Graph Packages and Hierarchical Graphs

In this section we introduce our notion of a *hierarchical graph.* The information that we want to model by means of hierarchical graphs concerns the grouping of graph elements. Grouping is related to the notion of *aggregation*, as known from object orientation (see e.g. (Rumbaugh *et al.* 1991)). Aggregation is a special kind of association, which describes a part-of relationship between an aggregate class and its subcomponents. In our running example, the links between anchors and pages can be considered as aggregation links.

Aggregation allows to group together the subobjects of a given object, and therefore it provides some kind of grouping. However, if we look again at the example, we see that there can be groupings which are not modelled adequately as aggregations. For example, pages are grouped according to projects they document, but they are not subcomponents of a project. Rather, we could think that the association between pages and sites is an aggregation, i.e. that a web site is made of web pages.

Besides this conceptual difference between our idea of grouping and that of aggregation, there is a technical one. In fact, aggregation always implies the existence of an aggregate object. On the other hand, a *generic* grouping mechanism should not rely on the existence of a grouping element in the graph (object, node, hyperedge, and so on), although this *can* be the case.

For these reasons, we think that a *generic* grouping mechanism should use some primitive which is distinct from nodes and edges of the graph. We call our grouping primitive a *graph package*, or simply a *package*. A graph package will be a generic container for graph elements (nodes and edges).

Our *decoupled approach* can be opposed to *coupled approaches*, where the hierarchical structure is coded in the graph itself, for example through the use of complex nodes (i.e. nodes that have an entire subgraph as their internal content, see e.g. (Engels and Schürr 1995)), or of complex edges (see e.g. (Drewes *et al.* 2002)).

A coupled approach is usually tailored for a specific application and supports a particular kind of hierarchical graph model. In our opinion, a coupled approach can hardly claim to be general enough, while a decoupled approach is better suited for defining a general hierarchical graph concept. If we define the hierarchy separately from the underlying graph, and associate elements of the graph to elements of the hierarchy, it is then much easier to specify properties of the hierarchy and of the graph separately, and study their interactions. In a decoupled approach we can decide at a later stage whether the hierarchy should be a tree or a dag, whether it should be a standalone structure or be associated to nodes, edges, or a combination of the two, and so on.

Summarising, we are going to model a hierarchical graph as an underlying *flat* graph, on top of which we add a hierarchy structure. The hierarchy is in turn modelled as a graph, namely a directed acyclic graph or dag. The elements of the hierarchy are called *graph packages*, while edges in the hierarchy graph model the containment relation between packages. The underlying graph and the hierarchy are connected to each other by a third graph, called *coupling graph*.

After giving some basic definitions in Section 3.1, we define our hierarchical-graph model in Section 3.2.

3.1. *Basic definitions*

In this section we introduce the basic definitions that we need for modelling the underlying graph of a hierarchical graph, the hierarchy, and the coupling between the two.

We first consider the *underlying graph*, i.e. the graph that is actually being structured. We have tried to make minimal assumptions on the underlying graph. In this way, our concept can be applied to several existing kinds of graphs (e.g. directed/undirected, typed/untyped, labelled graphs, hypergraphs, etc) provided that they satisfy some assumptions.

We have identified the following *essential elements*, that allow us to structure a graph in a hierarchical fashion:

— A graph should have *nodes*. No matter whether nodes are labelled, unlabelled, typed, untyped, attributed, and so on, what is essential is that a graph contains nodes that we would like to structure in some way.

— A graph should have *edges*. Edges model relations between nodes, and often some kind of proximity information, thus defining the localities in the graph. It will not be unusual that a node and all its neighbours are in the same component of the hierarchy.

We think that all other elements that can appear in a graph are marginal with respect to hierarchical structuring. For example, attributes should not be put anywhere in the hierarchy.

As long as edges are concerned, we are still left with the problem that there exist many different kinds of graphs, where nodes are connected to each other through edges in different ways. We abstract from the different kinds of graphs by only modelling the fact that edges are incident in nodes (nodes are attached to edges).

Summarising, our abstract notion of graph only assumes the existence of a set of nodes and a set of edges, and of an incidence relation between them. We call these three elements the *skeleton* of a given graph. The skeleton will serve as an interface between the underlying graph and the hierarchy dag. Any graph (structure) for which there exists a skeleton can be hierarchically structured.

Definition 3.1 (Graph and graph skeleton). A *graph skeleton* is a triple $S = (N, E, \iota)$, where N and E are finite sets, called the set of *nodes* and the set of *edges* of S, and $\iota \subseteq E \times N$ is a binary relation, called the *incidence relation* of S. A *graph* is any structure G that provides a skeleton $S_G = (N_G, E_G, \iota_G)$.

We will indicate the components of a graph skeleton S as N_S, E_S, ι_S respectively. Given a skeleton S, a node $n \in N_S$, and an edge $e \in E_S$, we will write $\iota_S(e, n)$ instead of $(e, n) \in \iota_S$. We also define the set of *items* of S as $I_S := N_S \cup E_S$. Given a graph G and its skeleton S_G, we will also write N_G for N_{S_G}, E_G for E_{S_G}, ι_G for ι_{S_G}, and I_G for I_{S_G}.

Given a graph G, $n \in N_G$, and $e \in E_G$ such that $\iota_G(e, n)$, we say that e is *incident* in n, and that n is *attached to* or an *attachment node of* e.

A class of finite graphs $\mathcal{G}$ induces a corresponding class of skeletons $\mathcal{S}(\mathcal{G}) := \{(N_G, E_G, \iota_G) \mid G \in \mathcal{G}\}$.

Notice that we do not require that the set of nodes and the set of edges of a graph be disjoint. Thus our notion captures many kinds of graphs, including those where edges between edges are allowed. Yet, in certain cases it is useful to keep the distinction between nodes and edges—e.g. when speaking of hierarchies or coupling graphs—so we explicitly speak about a set of nodes, a set of edges, and a set of items.

Items will be used to relate the underlying graph to the hierarchy graph and to the coupling graph (see below). The incidence relation will be used to express constraints on the way items are distributed over the hierarchy.

In the following example, we illustrate graph skeletons for a specific kind of graphs.

Example 3.2. Our running example uses directed graphs (to be formally defined shortly) to model hyperweb structures. Nodes are not labelled, while edges have two kinds of labels: λ for hyperlink edges, and "unlabelled" for structural links.

The skeleton of the graph in the example contains the set of nodes $N_G := \{\mathtt{NTA}, \mathtt{P1}, \mathtt{P2}, \dots\}$, the set of edges $E_G := \{\mathtt{P1\text{-}NTA}, \mathtt{P2\text{-}NTA}, \dots\}$, and the incidence relation $\iota_G := \{(\mathtt{P1\text{-}NTA}, \mathtt{P1}), (\mathtt{P1\text{-}NTA}, \mathtt{NTA}), (\mathtt{P2\text{-}NTA}, \mathtt{P2}), (\mathtt{P2\text{-}NTA}, \mathtt{NTA}), \dots\}$.

We now define labelled directed graphs, which are needed for modelling the hierarchy of a hierarchical graph.

Definition 3.3 (Directed graphs). Let Σ and Δ be two fixed sets, called the *node* and the *edge* alphabet respectively. A *labelled directed graph* is a tuple $G = (N, E, s, t, l, m)$, where N is a finite set of *nodes*, E is a finite set of *edges* of G, $N \cap E = \emptyset$, $s, t : E \to N$ are two functions mapping each edge to its *source* and *target node* respectively, and $l : N \to \Sigma$, $m : E \to \Delta$ are the *node* and *edge labelling functions* respectively.

A graph A is a subgraph of a graph B—in symbols, $A \subseteq B$—if $N_A \subseteq N_B$, $E_A \subseteq E_B$, $s_A = s_B|N_A$, $t_A = t_B|N_A$, $l_A = l_B|N_A$, $m_A = m_B|N_A$. For a function f and a set $X \subseteq \mathbf{dom}(f)$, $f|X$ denotes the restriction of f to X.

Given a labelled directed graph G, we will indicate the set of its nodes with N_G, the set of its edges with E_G, its source and target functions as s_G and t_G, and its labelling functions l_G and m_G respectively.

A *node-labelled directed graph* is a tuple $G = (N, E, s, t, l)$ defined as above, where we have dropped the edge-labelling function. Similarly, if we only provide an edge-labelling function, we obtain an *edge-labelled graph.* A tuple $G = (N, E, s, t)$, with N, E, s and t defined as above is an *unlabelled directed graph.*

Each directed graph G—may it be labelled, node-labelled, edge-labelled, unlabelled—provides a skeleton $S_G = (N_G, E_G, \iota_G)$, with $\iota_G := \{(e, s_G(e)) \mid e \in E_G\} \cup \{(e, t_G(e)) \mid e \in E_G\}$. In other words, all the introduced types of directed graphs are graphs in the sense of Definition 3.1. Without giving the technical details, undirected graphs and various kinds of hypergraphs can also be seen as graphs in this general sense.

In order to define the hierarchy of a hierarchical graph, we need a notion of directed acyclic graph, which is described in the following definition.

Definition 3.4 (Directed acyclic graphs). Let G be a directed graph. Then a *path* in G from m to n, for some $m, n \in N_G$, is a sequence of edges $e_1, \dots, e_k \in E_G$ $(k \geq 1)$ such that, for all $i = 1, \dots, k-1$, $t_G(e_i) = s_G(e_{i+1})$, $s_G(e_1) = m$, and $t_G(e_k) = n$. A *cycle* in G is a path from a node $n \in N_G$ to itself.

A *directed acyclic graph* (dag) is a directed graph that contains no cycles. Given a directed graph G, if there exists a node $n \in N_G$ such that, for all $m \in N_G - \{n\}$, there exists a path from n to m in G, then we say that n is a *root* of G and we call G a *rooted* *graph*. If G is a rooted graph which is also a dag, then we call G a *rooted dag*. Notice that a rooted dag G has exactly one root node, which we indicate as ρ_G.

Since we will use rooted dags for representing hierarchies of *graph packages* (see Definition 3.6) we will often indicate the set of nodes of a rooted dag D as P_D. We also define a relation $\succ_D$ between nodes of D as

$$\succ_D := \{(p, q) \in P_D \times P_D \mid \exists e \in E_D : s_D(e) = p \wedge t_D(e) = q\}$$

(Notice that, in general, $\succ_D$ is not transitive.)

Now that we have defined all the concepts we need for modelling the hierarchy of a hierarchical graph, we still miss a way to model the coupling between the underlying graph and the hierarchy. To this purpose we introduce coupling graphs.

Definition 3.5 (Coupling graphs). A *coupling graph* is a (bipartite) directed graph B where N_B is partitioned into the sets A_B and P_B, such that for all $e \in E_B$, we have $s_B(e) \in A_B$ and $t_B(e) \in P_B$ (i.e. if all edges are oriented from the first to the second set of nodes), and B satisfies the following *completeness condition*: For every $a \in A_B$ there exists some $p \in P_B$ and an edge $e \in E_B$ such that $s_B(e) = a$ and $t_B(e) = p$, (i.e. every node in A_B is connected to at least one node in P_B).

From now on, a coupling graph will be a system $B = (A_B, P_B, E_B, s_B, t_B, l_B, m_B)$ where A_B and P_B are two disjoint finite sets of *atoms* and *packages* respectively, E_B is the set of *edges* of B, $s_B : E_B \to A_B$, and $t_B : E_B \to P_B$ are the source and target functions, and l_B, m_B are the labelling functions. For a coupling graph B, we define a *containment relation*

$$C_B := \{(a, p) \in A_B \times P_B \mid \exists e \in E_B : a = s_B(e) \wedge p = t_B(e)\}$$

We use the notation P_B, A_B, and C_B, because coupling graphs determine in which *packages* the *atoms* of a hierarchical graph are *contained* (see Definition 3.6).

Although node and edge labels are not needed for the definition of coupling graphs, considering labelled bipartite graphs will be technically convenient in Section 5 and Section 6, where we want to use the traditional DPO transformation approach to transform these graphs.

3.2. *Hierarchical graphs*

In this section we define hierarchical graphs as a combination of three graphs that are glued to each other by means of their items.

Definition 3.6 (Hierarchical graphs). A *hierarchical graph* (HG) is a system $H = (G, D, B)$, where G is a graph in the sense of Definition 3.1, D is a rooted dag, B is a coupling graph with $A_B = I_G$ and $P_B = P_D$ satisfying, for each $p \in P_D$, the following condition: for every $e \in E_G$ and $n \in N_G$ with $\iota_G(e, n)$, $C_B(e, p)$ implies $C_B(n, p)$ i.e. if an edge e is contained in a package p of the hierarchy, then all nodes in which the edge is incident are contained in p as well.

The elements of the set P_D are called *graph packages*, or simply *packages*. D is called the *hierarchy dag*. $C_B \subseteq I_G \times P_D$ is the *containment relation* between the items and the packages of G. If $p, p' \in P_D$ and $p \succ_D p'$, we say that p' is a *superpackage* (or *parent package*) of p and that p is a *subpackage* (or *child package*) of p'. Two packages that have a common parent package are called *siblings*.

We also consider *plain hierarchical graphs*, defined as above but with $A_B = N_G$ and $N_G \cap E_G = \emptyset$ (the edges are not part of the hierarchy). The condition on edges formulated above is always trivially satisfied.

This notion is general enough to capture several other approaches to hierarchical graphs (see e.g. (Busatto and Hoffmann 2001; Busatto 2002)). The *plain* variant provides a simpler model that can be used when the location of edges in the hierarchy is not important.

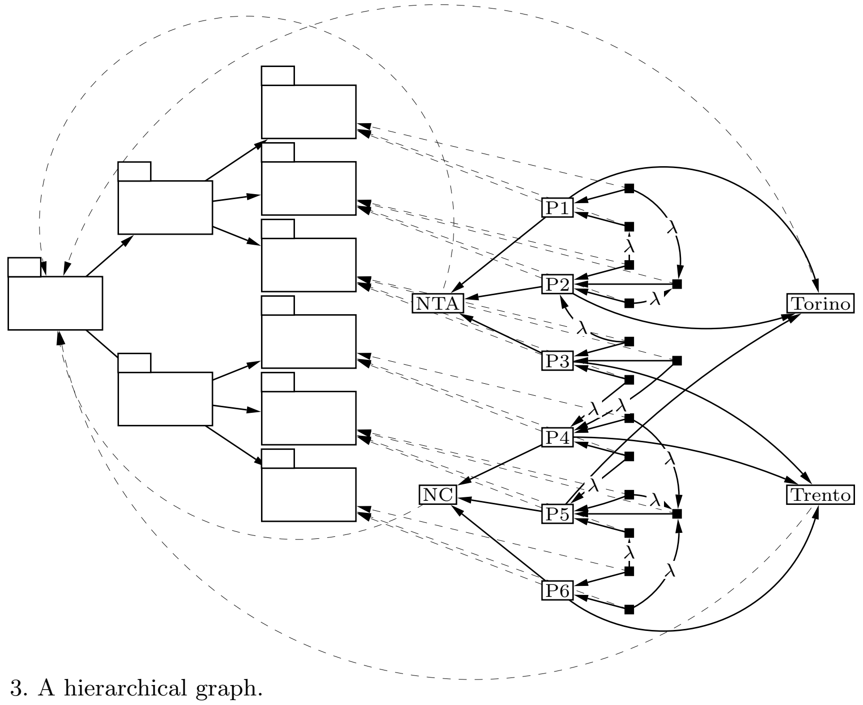

Fig. 3. A hierarchical graph.

Figure 3 depicts a hierarchical graph where the three components (hierarchy, underlying graph, coupling) are shown separately.

In Definition 3.6, we have introduced a condition relating nodes, edges, and packages saying that if an edge is contained in a package, then all its attachment nodes are contained in the same package as well. This avoids the unreasonable situation where a package "sees" an edge but does not see some of its attachment nodes.

We have also restricted the hierarchy graph to be a directed acyclic graph. Those approaches to hierarchical graphs that only allow tree-like hierarchies can be seen as a particular case. There also exist approaches (e.g. (Pratt 1979)) where arbitrary (even cyclic) hierarchies are allowed but we think that dags are closer to the usual intuition of a "layered structure." However, our model can be easily adapted to this more general notion of hierarchy by dropping the appropriate restrictions from Definition 3.6.

We now introduce some useful notation.

Definition 3.7 (Notation for hierarchical graphs). Given a hierarchical graph $H = (G, D, B)$, we denote G as G_H, D as D_H, and B as B_H.

Let $\mathcal{G}$ be a class of finite graphs in the sense of Definition 3.1, $\mathcal{D}$ be a class of rooted dags, and $\mathcal{B}$ be a class of coupling graphs. Then we define the class $\mathcal{H}(\mathcal{G}, \mathcal{D}, \mathcal{B})$ to be the smallest class containing all hierarchical graphs $(G, D, B) \in \mathcal{G} \times \mathcal{D} \times \mathcal{B}$.

On the other hand, given a class of hierarchical graphs $\mathcal{H}$, we can define the classes $\mathcal{G}_\mathcal{H} := \{G_H \mid H \in \mathcal{H}\}$, $\mathcal{D}_\mathcal{H} := \{D_H \mid H \in \mathcal{H}\}$, $\mathcal{B}_\mathcal{H} := \{B_H \mid H \in \mathcal{H}\}$. Notice that $\mathcal{H} \subseteq \mathcal{G}_\mathcal{H} \times \mathcal{D}_\mathcal{H} \times \mathcal{B}_\mathcal{H}$.

4. Hierarchical Graph Transformation

In this section we consider hierarchical graph transformation. We assume that hierarchical graphs are modelled as in Section 3, i.e. a hierarchical graph is a combination of an underlying graph, a hierarchy graph, and a containment relation between *packages* and elements of the graph (nodes and possibly edges). We model the containment relation as a graph, called the *coupling graph.*

Our main interest is on rule-based hierarchical graph transformation. We have considered two possible approaches to it:

1 Ad-hoc rules, accessing all the components (hierarchy, underlying graph, coupling graph) of a hierarchical graph at the same time, and taking advantage of the particular approach chosen for modelling hierarchical graphs. An example of such an approach is (Drewes *et al.* 2002).

2 A transformation approach which is as independent as possible from the choice of underlying graph and already existing rules for these graphs. This approach should allow to lift existing graph transformation rules to hierarchical graph transformation rules.

Following our main goal of developing an abstract model of hierarchical graph transformation, we adopt the second approach. In fact, packages are meant as a generic, approach-independent concept for structuring graphs, and we would like to specify package transformation in a similar fashion, making as few assumptions as possible about the underlying graphs, and reusing existing techniques for specifying their transformation.

By looking at our model of a hierarchical graph—namely a combination of three independent graphs with constraints between them—it is natural to apply a similar principle to graph transformation rules. In order to do this, we use the concept of a *graph transformation approach*, as described in (Kreowski and Kuske 1996) (see also (Kuske 1999; Kreowski and Kuske 1999a; Kreowski and Kuske 1999b)). We suppose to have three graph transformation approaches: one for specifying graph transformations, one for specifying hierarchy transformations, and one for specifying transformations on the coupling graphs. In this section we show how three transformation approaches like these can be combined into a hierarchical graph transformation approach. In the combined approach, the three component graphs of a HG are transformed by independent rules, but consistency conditions between them are ensured.

This section is structured as follows. In 4.1, we recall the concept of a graph transformation approach, and we introduce the notation necessary for the following subsections. In 4.2 and 4.3, we introduce the concept of a hierarchical graph transformation approach, and show how to construct such an approach, given three transformation approaches for the components of a hierarchical graph. In 4.4, we extend our running example to illustrate our hierarchical graph transformation approach.

4.1. *Graph Transformation Approaches*

The notion of a graph transformation approach (see e.g. (Kreowski and Kuske 1996; Kreowski *et al.* 1997; Kreowski and Kuske 1999a; Kuske 1999; Kreowski and Kuske 1999b)) has been introduced as an abstraction of approaches to graph transformation existing in the literature. This concept has been used for the definition of *transformation units* (see again (Kreowski and Kuske 1996)), an approach-independent mechanism that can be used to structure graph transformation systems, and to add control on rule application.

A graph transformation approach assumes the existence of a class of graphs $\mathcal{G}$, a class of rules, and a rule application operator, which associates to each rule r a binary relation $\Rightarrow^r$ on $\mathcal{G}$, such that a pair (G, G') in $\Rightarrow^r$ indicates that G' is directly derived from G via rule r. Furthermore, a graph transformation approach contains control conditions that provide control on rule application, and graph class expressions that allow to define classes of graphs (for example, initial and final graphs of a graph transformation system).

Here is the formal definition of graph transformation approaches.

Definition 4.1 (Graph transformation approach).
A system $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ is a *graph transformation approach* if:

- $\mathcal{G}$ is a class of *graphs*,
- $\mathcal{R}$ is a class of *rules*, whose semantics is provided by the *rule application operator* $\Rightarrow$ that associates to every rule $r \in \mathcal{R}$ a binary relation $\Rightarrow^r \subseteq \mathcal{G} \times \mathcal{G}$,
- $\mathcal{C}$ is a class of *control conditions* such that the semantics of each $C \in \mathcal{C}$ is a binary relation $SEM(C) \subseteq \mathcal{G} \times \mathcal{G}$,
- $\mathcal{E}$ is a class of *graph class expressions* such that the semantics of each $X \in \mathcal{E}$ is a set $SEM(X) \subseteq \mathcal{G}$.

Note that the definition of a graph transformation approach actually captures the more general situation where we have a class of objects that are transformed by rules. In fact, in the above definition no assumptions are made about the kind of graphs to be used: the class $\mathcal{G}$ can even contain other structures than graphs, provided that we have also a class of rules and a rule application operator. We will exploit the generality of this approach in the next section in order to define hierarchical graph transformation approaches. An example of a graph transformation approach will be sketched in Section 4.4. The interested reader can find more examples in (Kuske 1999, Section 2.1).

Graph transformation approaches model graph transformation from an abstract point of view, and rules and graphs are considered atomic entities. As a consequence, it is not possible to model such information as rule left- and right-hand sides, nodes and edges inside these graphs, the fact that some of these nodes and edges are preserved, deleted or created, and so on. In a similar way, it is not possible to speak about elements of the transformed graphs, and about them being preserved/deleted/newly created. On the other hand, we need this information if we want to coordinate the transformation of different components of a hierarchical graph: nodes inside different rules must be identified, and the components of a hierarchical graph must be manipulated in such a way as to respect this identification. This additional information is modelled by using the refined notion of a *tracking graph transformation approach.*

In a tracking approach, *rule skeletons* model the needed internal structure of rules by means of a left- and right-hand side graph skeleton, and a partial morphism between the two. In this way, we can speak about nodes and edges inside a rule, and—through the partial morphism from the left- to the right-hand side—we can track preserved elements of the rule. The same construct is used to refine the notion of a direct derivation step. (These ideas are borrowed from the SPO graph transformation approach, see e.g. (Ehrig *et al.* 1997).)

The notion of a rule skeleton relies on that of a partial (graph) skeleton morphism.

Definition 4.2 (Skeleton morphisms).
Given two graph skeletons $S_1 = (N_1, E_1, \iota_1)$, $S_2 = (N_2, E_2, \iota_2)$, a (*partial*) *skeleton morphism* $\mu : S_1 \to S_2$ is a pair of partial functions $\mu_{\mathrm{N}} : N_1 \to N_2$, $\mu_{\mathrm{E}} : E_1 \to E_2$, such that, for all $e \in E_1$ and $n \in N_1$, if $e \in \mathbf{dom}(\mu_{\mathrm{E}})$ and $\iota_1(e, n)$, then $n \in \mathbf{dom}(\mu_{\mathrm{N}})$ and $\iota_2(\mu_{\mathrm{E}}(e), \mu_{\mathrm{N}}(n))$.

Notice that this definition ensures that, whenever an edge belongs to the domain of a partial morphism, then also its attachment nodes belong to the domain of that morphism, i.e. no "dangling" mappings are allowed.

In the following definition, we explain how skeleton morphisms can be used as *rule* (resp. *direct derivation*) *skeletons.*

Definition 4.3 (Rule and derivation skeleton).
A *rule skeleton* is a triple $S = (LS, RS, tr)$, where LS, RS are graph skeletons, and $tr : LS \to RS$ is a partial skeleton morphism. Given two graphs G, H such that H directly derives from G in some graph transformation approach, a morphism $\varphi : S_G \to S_H$ will be called *direct derivation skeleton.*

A *tracking graph transformation approach*—see Definition 4.4—is a graph transformation approach where

— rules always provide rule skeletons,
— direct derivation steps always provide direct derivation skeletons,
— and such pairs of skeletons agree on preserved/created/deleted graph elements.

Definition 4.4 (Tracking graph transformation approach).
A graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ is *tracking* if

1 every rule $r \in \mathcal{R}$ provides a rule skeleton (LS_r, RS_r, tr_r),

2 for every rule $r \in \mathcal{R}$, for all graphs $G, G' \in \mathcal{G}$, if there is a direct derivation step $G \Rightarrow^{r} G'$, then there exists at least one tuple $\langle g, h, \varphi \rangle$, where $g : LS \to S_G$ and $h : RS \to S_{G'}$ are skeleton morphisms, and $(S_G, S_{G'}, \varphi)$ is a direct derivation skeleton such that

$$h \circ tr = \varphi \circ g$$

i.e. such that the diagram of Figure 4 commutes.

$$\begin{tikzcd}
LS \arrow[r, "tr"] \arrow[d, "g"] & RS \arrow[d, "h"] \\
S_G \arrow[r, "\varphi"] & S_{G'}
\end{tikzcd}$$

Fig. 4. Tracking derivation.

While rule skeletons and direct derivation skeletons allow to track preserved graph elements between two graphs, tracking derivation adds consistency constraints between a rule skeleton and a direct derivation skeleton by means of matching morphisms.

4.2. *Combining graph transformation approaches*

Since we model hierarchical graphs as triples of graphs, it seems a natural idea to apply graph transformation to the components of a hierarchical graph in order to define hierarchical graph transformation. This is done by means of a standard construction (Construction 1) that allows to define hierarchical graph transformation if three appropriate graph transformation approaches are available. This method has the advantage that it does not require a new notion of graph transformation and it does not force one to choose a priori an existing graph transformation approach.

To begin with, we need special names for transformation approaches where the class of graphs contains hierarchical graphs, hierarchy graphs, and coupling graphs.

Definition 4.5 (Special transformation approaches).
We call a tuple $\mathcal{A}_{\mathcal{H}} = (\mathcal{H}, \mathcal{R}_{\mathcal{H}}, \Rightarrow_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$, where $\mathcal{H}$ is a class of hierarchical graphs, and the other components are defined analogously to Definition 4.1, a *hierarchical graph transformation approach.*

We call a graph transformation approach $\mathcal{A}_{\mathcal{D}} = (\mathcal{D}, \mathcal{R}_{\mathcal{D}}, \Rightarrow_{\mathcal{D}}, \mathcal{C}_{\mathcal{D}}, \mathcal{E}_{\mathcal{D}})$ a *hierarchy transformation approach* if $\mathcal{D}$ is a class of rooted dags. We call a graph transformation approach $\mathcal{A}_{\mathcal{B}} = (\mathcal{B}, \mathcal{R}_{\mathcal{B}}, \Rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, \mathcal{E}_{\mathcal{B}})$ a *coupling transformation approach* if $\mathcal{B}$ is a class of coupling graphs.

In the following construction, we show how to build a hierarchical graph transformation approach as a combination of a graph transformation approach, a hierarchy transformation approach, and a coupling transformation approach. This construction allows to add hierarchical structuring to *any* graph transformation approach as an orthogonal concept.

Construction 1. Let $\mathcal{G}$ be a class of graphs, $\mathcal{D}$ a class of rooted dags, $\mathcal{B}$ a class of coupling graphs and, for $x \in \{\mathcal{G}, \mathcal{D}, \mathcal{B}\}$, let $\mathcal{A}_x = (x, \mathcal{R}_x, \Rightarrow_x, \mathcal{C}_x, \mathcal{E}_x)$ be a graph transformation approach, a hierarchy transformation approach, and a coupling transformation approach, respectively.

Then a *loose hierarchical graph transformation approach*

$$\mathcal{L}_\mathcal{H}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B}) = (\mathcal{H}, \mathcal{R}_\mathcal{H}, \Rightarrow_\mathcal{H}, \mathcal{C}_\mathcal{H}, \mathcal{E}_\mathcal{H})$$

is induced as follows:

— $\mathcal{H} := \mathcal{H}(\mathcal{G}, \mathcal{D}, \mathcal{B})$.
— $\mathcal{R}_\mathcal{H} := \mathcal{R}_\mathcal{G} \times \mathcal{R}_\mathcal{D} \times \mathcal{R}_\mathcal{B}$.
— Given a rule $r = (\gamma, \delta, \beta)$, its semantics relation $\Rightarrow^r \subseteq \mathcal{H} \times \mathcal{H}$ is defined for $H = (G, D, B)$, $H' = (G', D', B') \in \mathcal{H}$ as follows: $H \Rightarrow^r_\mathcal{H} H'$ iff $G \Rightarrow^\gamma_\mathcal{G} G'$ and $D \Rightarrow^\delta_\mathcal{D} D'$ and $B \Rightarrow^\beta_\mathcal{B} B'$.
— $\mathcal{C}_\mathcal{H} := \mathcal{C}_\mathcal{G} \times \mathcal{C}_\mathcal{D} \times \mathcal{C}_\mathcal{B}$ and, for $C = (C_\mathcal{G}, C_\mathcal{D}, C_\mathcal{B}) \in \mathcal{C}_\mathcal{H}$, the semantics is defined by

$$\begin{aligned} \mathit{SEM}_\mathcal{H}(C) := \{((G, D, B), (G', D', B')) \in \mathcal{H} \times \mathcal{H} \mid \\ (G, G') \in \mathit{SEM}_\mathcal{G}(C_\mathcal{G}) \wedge \\ (D, D') \in \mathit{SEM}_\mathcal{D}(C_\mathcal{D}) \wedge \\ (B, B') \in \mathit{SEM}_\mathcal{B}(C_\mathcal{B})\} \end{aligned}$$

— $\mathcal{E}_\mathcal{H} := \mathcal{E}_\mathcal{G} \times \mathcal{E}_\mathcal{D} \times \mathcal{E}_\mathcal{B}$, and, for $X = (X_\mathcal{G}, X_\mathcal{D}, X_\mathcal{B}) \in \mathcal{E}_\mathcal{H}$, the semantics is defined by

$$\mathit{SEM}_\mathcal{H}(X) := \mathit{SEM}_\mathcal{G}(X_\mathcal{G}) \times \mathit{SEM}_\mathcal{D}(X_\mathcal{D}) \times \mathit{SEM}_\mathcal{B}(X_\mathcal{B}) \cap \mathcal{H}$$

While the classes of rules, control conditions, and graph class expression are defined as the Cartesian products of the corresponding component classes, their semantics is constructed componentwise, too, but the resulting triples of graphs, dags, and connecting graphs must form hierarchical graphs in addition. A rule $r \in \mathcal{R}_\mathcal{H}$ is called a *loose hierarchical rule.*

A loose hierarchical graph transformation approach $\mathcal{L}_\mathcal{H}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B})$ is *tracking* if $\mathcal{A}_\mathcal{G}$, $\mathcal{A}_\mathcal{D}$, and $\mathcal{A}_\mathcal{B}$ are tracking.

It can be useful to assume that the three component approaches always have an identity rule (with obvious semantics), which can be used when we do not want to modify the hierarchy, the graph, or the couplings, while transforming the other components.

A hierarchical graph transformation approach built according to Construction 1 is called *loose* because there is little coordination between these transformations, leading to unexpected/unwanted results. For example, let $(G, D, B) \Rightarrow^r (G', D', B')$ be a direct derivation step defined as above, where $G \cong G'$ contains exactly one node, $D \cong D'$ contains exactly two packages, $B \cong B'$ contains the node, the two packages, and assigns the node to one of the two packages. This derivation step can easily be defined by a triple of identity rules. Now, since direct derivation steps are often defined up to isomorphism, nothing guarantees that the components of (G', D', B') are glued in the appropriate way and it could happen that the only node of the hierarchical graph is moved from one package to the other.

If the component approaches are tracking, we can extend loose hierarchical graph transformation to support *coordination*, as discussed in the following subsection. The notion of a loose hierarchical graph transformation approach can be considered an intermediate step for the definition of coordinated hierarchical graph transformation.

4.3. *Hierarchical graph transformation*

We are now ready to define (coordinated) hierarchical graph transformation approaches. We will first define coordinated rules, i.e. triple rules whose left- and right-hand side are glued in a similar way as the components of a hierarchical graph are. We then define coordinated hierarchical graph transformation and extend Construction 1 in order to take into account this additional information.

Coordinated rules are triples of graph transformation rules with additional information about correspondences between elements of the coupling graph rule and elements of the other rules. For example, we can specify that a certain package in the hierarchy rule is the same as a package in the coupling graph rule.

While the notion of a tracking graph transformation approach allows to specify correspondences between elements within a rule, coordination permits to specify external correspondences between elements of different rules. In coordinated hierarchical transformation we will require that such external and internal correspondences are compatible.

Before introducing coordinated rules, we need the notion of commutativity for certain diagrams containing two partial morphisms and two relations between graph skeletons. The two morphisms will be rule skeletons and the relations will model the gluing of skeletons along common elements.

Definition 4.6 (Gluing rule skeletons).
Let S_1 and S_2 be graph skeletons, and $f : S_1 \to S_2$ a partial skeleton morphism. We will often abuse notation and consider f as a function $f : I_{S_1} \to I_{S_2}$ defined, for all $n \in N_{S_1} \cap \mathbf{dom}(f_\mathrm{N})$, as $f(n) := f_\mathrm{N}(n)$ and, for all $e \in E_{S_1} \cap \mathbf{dom}(f_\mathrm{E})$, as $f(e) := f_\mathrm{E}(e)$.

Let S_1, S_2, S_3, S_4 be four graph skeletons, $l : S_1 \to S_2$ and $r : S_3 \to S_4$ be two partial skeleton morphisms, $\sim_{upper} \subseteq I_{S_1} \times I_{S_3}$ and $\sim_{lower} \subseteq I_{S_2} \times I_{S_4}$ be two relations. We say that the quadruple $\langle l, r, \sim_{upper}, \sim_{lower} \rangle$ *commutes* if, for all $a \in I_{S_1}$, for all $c \in I_{S_3}$, we have that if $a \sim_{upper} c$, then
— either $a \notin \mathbf{dom}(l)$ and $c \notin \mathbf{dom}(r)$,
— or $a \in \mathbf{dom}(l)$ and $c \in \mathbf{dom}(r)$ and $l(a) \sim_{lower} r(c)$.

In such a case, we say that the rule skeletons $l : S_1 \to S_2$ and $r : S_3 \to S_4$ are *glued* via the relations $\sim_{upper}$, $\sim_{lower}$.

Consider a diagram like that of Figure 5. Intuitively, the notion of commutativity introduced in Definition 4.6 states that if two elements of the upper skeletons are related through $\sim_{upper}$, then either they are both mapped to two elements of the lower skeletons that are related through $\sim_{lower}$, or they are both outside the domain of the two partial morphisms in the diagram.

$$\begin{tikzcd}
S_1 \arrow[r, dash, "\sim_{upper}"] \arrow[d, "l"] & S_3 \arrow[d, "r"] \\
S_2 \arrow[r, dash, "\sim_{lower}"] & S_4
\end{tikzcd}$$

Fig. 5. Gluing rule skeletons.

Definition 4.7 (Coordinated rules).
For $x \in \{\mathcal{G}, \mathcal{D}, \mathcal{B}\}$, let $\mathcal{A}_x = (x, \mathcal{R}_x, \Rightarrow_x, \mathcal{C}_x, \mathcal{E}_x)$ be three tracking graph transformation approaches. Furthermore, let $\mathcal{R} \subseteq \mathcal{R}_\mathcal{G} \times \mathcal{R}_\mathcal{D} \times \mathcal{R}_\mathcal{B}$ be a class of triple rules. Then a *coordinated rule* over $\mathcal{R}$ is a system

$$r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$$

where $(\gamma, \delta, \beta) \in \mathcal{R}$ and we have

$$\begin{aligned}
\sim^L_\gamma &\subseteq I_{LS_\gamma} \times N_{LS_\beta} \\
\sim^R_\gamma &\subseteq I_{RS_\gamma} \times N_{RS_\beta} \\
\sim^L_\delta &\subseteq N_{LS_\delta} \times N_{LS_\beta} \\
\sim^R_\delta &\subseteq N_{RS_\delta} \times N_{RS_\beta}
\end{aligned}$$

such that all relations are injective[†], and the two quadruples $\langle tr_\gamma, tr_\beta, \sim^L_\gamma, \sim^R_\gamma \rangle$ and $\langle tr_\delta, tr_\beta, \sim^L_\delta, \sim^R_\delta \rangle$ commute. We indicate with $\mathcal{CO}(\mathcal{R})$ the class of all coordinated rules over $\mathcal{R}$.

In Figure 6, we depict the relations and morphisms of a coordinated hierarchical graph rule $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$

$$\begin{tikzcd}
LS_\gamma \arrow[r, dash, "\sim^L_\gamma"] \arrow[d, "tr_\gamma"] & LS_\beta \arrow[r, dash, "\sim^L_\delta"] \arrow[d, "tr_\beta"] & LS_\delta \arrow[d, "tr_\delta"] \\
RS_\gamma \arrow[r, dash, "\sim^R_\gamma"] & RS_\beta \arrow[r, dash, "\sim^R_\delta"] & RS_\delta
\end{tikzcd}$$

Fig. 6. Coordinated rule diagram.

[†] We say that a relation $R \subseteq A \times B$ is injective if for all $a \neq a' \in A$, $b \neq b' \in B$, neither $(a, b), (a', b) \in R$, nor $(a, b), (a, b') \in R$ hold.

We are now ready to speak about coordinated hierarchical graph transformation and coordinated hierarchical graph transformation approaches. This completes the definition of our framework.

Definition 4.8 (Coordinated HG transformation approach).
Let us consider a tracking graph transformation approach $\mathcal{A}_\mathcal{G} = (\mathcal{G}, \mathcal{R}_\mathcal{G}, \Rightarrow_\mathcal{G}, \mathcal{C}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, a tracking hierarchy transformation approach $\mathcal{A}_\mathcal{D} = (\mathcal{D}, \mathcal{R}_\mathcal{D}, \Rightarrow_\mathcal{D}, \mathcal{C}_\mathcal{D}, \mathcal{E}_\mathcal{D})$, and a tracking coupling transformation approach $\mathcal{A}_\mathcal{B} = (\mathcal{B}, \mathcal{R}_\mathcal{B}, \Rightarrow_\mathcal{B}, \mathcal{C}_\mathcal{B}, \mathcal{E}_\mathcal{B})$. Let $\mathcal{L}_\mathcal{H}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B}) = (\mathcal{H}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be the corresponding tracking loose hierarchical graph transformation approach. Then, a *coordinated hierarchical graph transformation approach* over $\mathcal{L}_\mathcal{H}$ is a hierarchical graph transformation approach

$$\mathcal{A}_\mathcal{H} = (\mathcal{H}, \mathcal{CO}(\mathcal{R}), \Rrightarrow, \mathcal{C}, \mathcal{E})$$

where, for all $H, H' \in \mathcal{G}$, $H = (G, D, B)$, $H' = (G', D', B')$, for all rules $r \in \mathcal{CO}(\mathcal{R})$, $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$, we have that $H \Rrightarrow^r H'$ if and only if

1 $H \Rightarrow^{(\gamma,\delta,\beta)} H'$,
2 we can build a diagram like the one in Figure 7, where the front horizontal relations are defined as[‡]

$$\sim^G := \{(a, a) \mid a \in A_B\}$$
$$\sim^{G'} := \{(a, a) \mid a \in A_{B'}\}$$
$$\sim^D := \{(p, p) \mid p \in P_B\}$$
$$\sim^{D'} := \{(p, p) \mid p \in P_{B'}\}$$

and all squares in the diagram commute.

$$\begin{tikzcd}
LS_\gamma \arrow[rr, dash, "\sim^L_\gamma"] \arrow[dr, "m_\gamma"] \arrow[dd, "tr_\gamma"] & & LS_\beta \arrow[rr, dash, "\sim^L_\delta"] \arrow[dr, "m_\beta"] \arrow[dd, "tr_\beta"] & & LS_\delta \arrow[dr, "m_\delta"] \arrow[dd, "tr_\delta"] & \\
& S_G \arrow[rr, dash, "\sim^G"] \arrow[dd, "\varphi_\gamma"] & & S_B \arrow[rr, dash, "\sim^D"] \arrow[dd, "\varphi_\beta"] & & S_D \arrow[dd, "\varphi_\delta"] \\
RS_\gamma \arrow[rr, dash, "\sim^R_\gamma"] \arrow[dr, "m'_\gamma"] & & RS_\beta \arrow[rr, dash, "\sim^R_\delta"] \arrow[dr, "m'_\beta"] & & RS_\delta \arrow[dr, "m'_\delta"] & \\
& S_{G'} \arrow[rr, dash, "\sim^{G'}"] & & S_{B'} \arrow[rr, dash, "\sim^{D'}"] & & S_{D'}
\end{tikzcd}$$

Fig. 7. Skeleton for a coordinated derivation step.

If we look at the diagram in Figure 7 again, we notice that
— the back squares exist and commute since r is a coordinated rule,

‡ Recall that $A_B = I_G$, $A_{B'} = I_{G'}$, $P_B = P_D$, $P_{B'} = P_{D'}$.

— if $G \Rightarrow_{\mathcal{G}}^{\gamma} G'$, $D \Rightarrow_{\mathcal{D}}^{\delta} D'$, $B \Rightarrow_{\mathcal{B}}^{\beta} B'$, the vertical side and middle squares exist and commute since the three component approaches are tracking.

In the rest of the paper, we will drop the term coordinated, assuming that all the considered hierarchical graph transformation approaches are coordinated. Notice also that all the constructions and definitions presented in this section can be applied to plain hierarchical graphs, provided that for every hierarchical graph $H(G, D, B)$ we assume $A_B = N_G$, and for every coordinated rule $r = (\gamma, \delta, \beta, \sim_\gamma^L, \sim_\gamma^R, \sim_\delta^L, \sim_\delta^R)$ we require $\sim_\gamma^L \subseteq N_{LS_\gamma} \times N_{LS_\beta}$ and $\sim_\gamma^R \subseteq N_{RS_\gamma} \times N_{RS_\beta}$.

4.4. *An example*

We now illustrate hierarchical graph transformation by defining some transformations for our web example (see Section 2). In this example, we use the *double-pushout* (DPO) approach to graph transformation, which we briefly explain in this subsection in an informal and intuitive way, postponing formal definitions until Section 5. We use plain hierarchical graphs here, meaning that the location of edges in the hierarchy is omitted.

The traditional DPO approach uses node and edge labelled directed graphs. This means that, when we instantiate our framework to the DPO approach, we obtain hierarchical graphs that are triples of node and edge labelled directed graphs. In our specific example, we still have to choose an appropriate labelling for these graphs. For the underlying graph we use the set of node labels $\Sigma_{\mathcal{G}} = \{prj, site, page, anc\}$, so that we will be able to distinguish between four types of nodes, and the set of edge labels $\Delta_{\mathcal{G}} = \{\lambda, \sigma\}$, where λ labels *link edges* originating from anchors (see again Figure 1) and σ labels *structural edges*, i.e. all other edges. The sets of node labels and of edge labels for the hierarchy graph are $\Sigma_{\mathcal{D}} = \Delta_{\mathcal{D}} = \{\perp\}$, i.e. we have only one node and edge label, which is equivalent to having no edge labelling at all. The set of node labels for the coupling graph is $\Sigma_{\mathcal{B}} := \Sigma_{\mathcal{G}} \cup \Sigma_{\mathcal{D}}$, whereas its edge label alphabet $\Delta_{\mathcal{B}} = \{\perp\}$ contains only one element, like for hierarchy graphs. Provided that $\Sigma_{\mathcal{G}} \cap \Sigma_{\mathcal{D}} = \emptyset$ we can use labels to distinguish package nodes from atom nodes, and therefore a coupling graph $B = (P_B, A_B, E_B, s_B, t_B, l_B, m_B)$ will be represented as a directed graph $\overline{B} = (P_B \cup A_B, E_B, s_B, t_B, l_B, m_B)$.

We recall that in our example we have projects documented by web pages. We also have packages associated to projects, which contain the nodes and links related to a project's documentation. Inside project packages, we have page packages, containing the nodes and edges describing the internal structure of each page. In this setting, we consider the following two transformations:

1 *Add a new project together with its three (by now empty) pages.* Adding a project involves creating its package, creating its pages, and putting the pages inside the project package. The new package must be hung to some existing package.

2 *Add a hyperlink from a given page to a target page within the same project.* This involves adding an anchor to the source page, adding an edge from that anchor to the target page, and adding the anchor to the package of the source page.

We specify these transformations by means of hierarchical graph rules.

The first rule, depicted in Figure 8, illustrates the transformation "create project". It

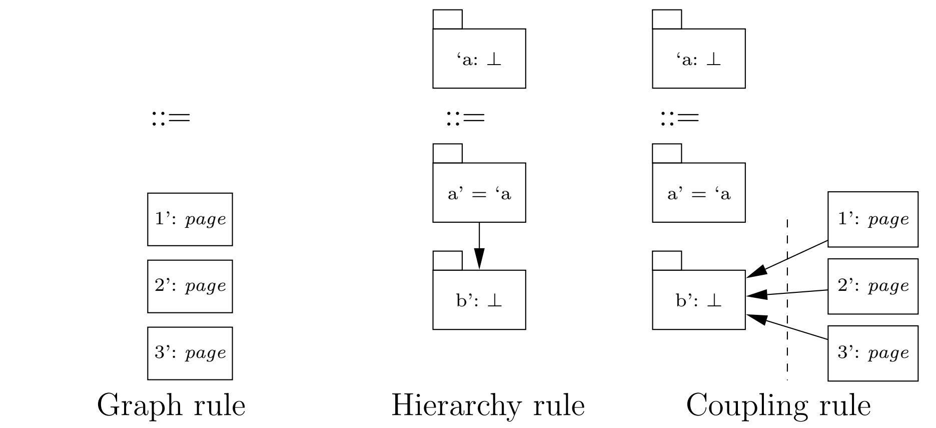

Fig. 8. Creating a package and some internal nodes.

is composed of three double-pushout rules: The graph rule, the hierarchy rule, and the coupling rule. Each DPO rule has the following elements:

— The *left-hand side*, which specifies a pattern graph to be found in the graph to be transformed (the *host graph*). In Figure 8, the graph rule has an empty left-hand side, which means that it can always be applied since the empty pattern is trivially contained in every graph. The left-hand side of the hierarchy and of the coupling rule contains one package with label ⊥ and identifier 'a. This means that we have to match a ⊥-labelled node in both the hierarchy and the coupling graph. The left-hand side is depicted on the left of (or, due to layout reasons, above) the ::= symbol.

— The *right-hand side*, which specifies the transformation to be performed on the nodes and edges of the host graph matched by the left-hand side. In Figure 8, all nodes and edges in the right-hand side of the graph rule are new, which means that they must be added to the host graph. In the right-hand side of the hierarchy and coupling rule the package with identifier a' is *preserved* from the left-hand side (notice the notation a' = 'a) while all other nodes are new. If a node or edge only appears in the left-hand side it must be deleted.

Notice that we have used the same identifiers for nodes in the graph and the coupling rule, and for packages in the hierarchy and the coupling rule. This expresses the coordination between rules. Since we are dealing with plain hierarchical graphs, there is no coordination for the edges of the underlying graph.

After this brief illustration of double-pushout rules and their semantics, we concentrate again on the first hierarchical rule. The graph rule has an empty left-hand side and three page nodes on the right-hand side, and thus it specifies the creation of three new page nodes. The hierarchy rule specifies the creation of a new package. The coupling rule specifies that new package must contain the three new pages. The vertical dashed line in the coupling rule, separates the package part (on the left) from the graph part (on the right) of the transformation. Note that we use a special notation for package nodes.

The fact that the three new pages are put in the correct package is ensured by the coordination information (they are put in package b).

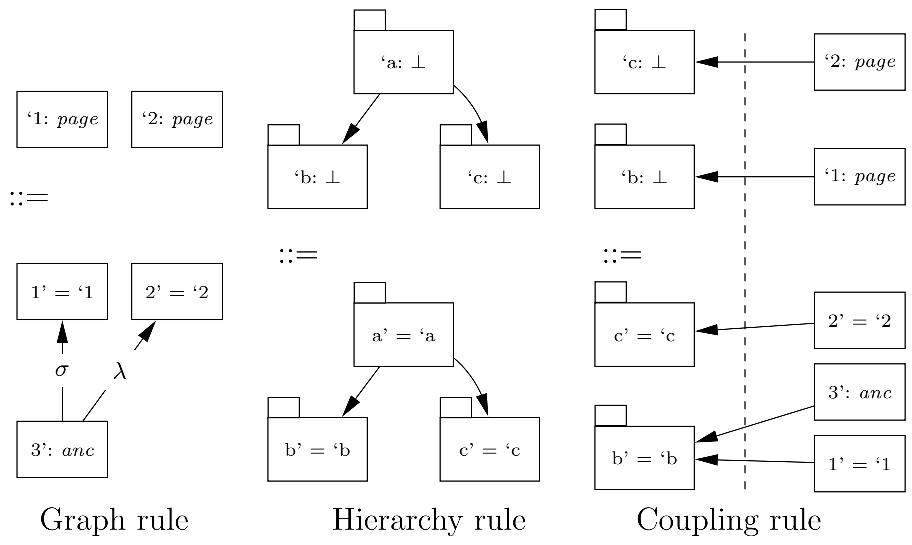

Fig. 9. Adding a hyperlink.

The second hierarchical rule, depicted in Figure 9, specifies the transformation “add hyperlink”. The graph part of the rule indicates that we should find two page nodes in the underlying graph, add a new anchor node and link it to the first page with a new structural link, and to the second page with a new hyperlink. The hierarchy rule specifies that three packages are found in the hierarchy, such that the second and the third are subpackages of the first. The rule preserves this situation. The first package represents the common project package in which the two page packages (second and third package) must be. The coupling rule specifies that there must be two page nodes contained in two different packages and that after the application there must be a new anchor node, which is put in the package that contains the target of the σ-edge. This is again ensured by the coordination between the graph and the coupling rule.

5. A Hierarchical Double-Pushout Approach

In this section, we want to instantiate our framework for hierarchical graph transformation using the double-pushout (DPO) approach to graph transformation (see (Ehrig 1979), (Corradini *et al.* 1997)). Thus we will combine three graph transformation approaches based on the double-pushout into a hierarchical graph transformation approach.

The main issue in this section concerns the restrictions that we have to impose on hierarchy and on connection transformation rules. In fact, unrestricted application of graph transformation rules to, say, hierarchy dags, can produce graphs which are no longer rooted dags, and therefore violate the required structure of the hierarchy graph. Likewise, unrestricted application of rules to connection graphs can yield graphs that are no longer connection graphs. In Section 5.2, we study two conditions that can be statically checked on DPO rules, and that together ensure that a rule always transforms hierarchy graphs into hierarchy graphs, as shown in Proposition 5.12. In Section 5.3, an analogous condition for the preservation of connection graphs is introduced, and its correctness and completeness is proved in Proposition 5.20. Thus the results presented in Section 5.2 and 5.3 provide a characterisation of DPO hierarchical graph transformation.

In Section 5.1, we give preliminary definitions that are needed in the rest of the section. These definitions, concerning the notion of DPO graph transformation, will be further exploited in Section 6, where we investigate the translation of hierarchical graph DPO rules into flat DPO rules, thus mimicking hierarchical graph transformation with flat graph transformation.

5.1. *Double-pushout graph transformation*

The double-pushout approach (DPO) to graph transformation was introduced at the beginning of the seventies (see (Ehrig *et al.* 1973)) as a generalisation of Chomsky grammars from strings to graphs. It is based on the *pushout* construction from category theory, which can be roughly described as the gluing of two objects of some kind along a common interface. Applied to graphs, such construction allows to glue two given graphs G and H together by identifying some of their nodes and edges.

The basic transformation step in the DPO approach is based on the construction of a diagram with two pushouts, where two of the graphs in the diagram represent the graph G to be transformed and the graph H derived from G. We now present the basic notions of the DPO approach, which we need in the remainder of the section.

To begin with, we need the notion of a graph morphism.

Definition 5.1 (Graph morphism, category of graphs). Given two labelled directed graphs G, G', a *graph morphism* $\varphi : G \to G'$ from G to G' is a pair of functions $\varphi = (\varphi_\mathrm{N}, \varphi_\mathrm{E})$, where $\varphi_\mathrm{N} : N_G \to N_{G'}$ and $\varphi_\mathrm{E} : E_G \to E_{G'}$, such that $\varphi_\mathrm{N} \circ s_G = s_{G'} \circ \varphi_\mathrm{E}$, $\varphi_\mathrm{N} \circ t_G = t_{G'} \circ \varphi_\mathrm{E}$, $l_G = l_{G'} \circ \varphi_\mathrm{N}$, and $m_G = m_{G'} \circ \varphi_\mathrm{E}$. Given two morphisms $\varphi : G \to H$, and $\psi : H \to K$, we define the compound morphism $\psi \circ \varphi : G \to K$ as the pair $(\psi_\mathrm{N} \circ \varphi_\mathrm{N}, \psi_\mathrm{E} \circ \varphi_\mathrm{E})$. Given a graph G, the *identity morphism* on G is the pair $id_G = \langle id_{N_G}, id_{E_G} \rangle$, where $id_{N_G} : N_G \to N_G$ and $id_{E_G} : E_G \to E_G$ are the identity functions on the set of nodes and on the set of edges of G respectively.

The category of directed graphs $\mathcal{DG}$ has labelled directed graphs as objects, graph morphisms as arrows, identity morphisms as identity arrows, and composition of morphisms as defined above as arrow composition.

A morphism φ is *injective* if φ_N and φ_E are injective functions. Given a graph morphism $\varphi = (\varphi_\mathrm{N}, \varphi_\mathrm{E})$ we write φ for φ_N (resp. φ_E) when it is clear that we speak about the node (resp. edge) function. In what follows, if $f : A \to B$ and $g : B \to C$ are functions (resp. graph morphisms), we will often abbreviate $g \circ f$ to gf.

Graph transformation rules are defined in the following.

Definition 5.2 (Graph transformation rule). A *graph transformation rule* (or, simply, a *rule*) is a tuple (L, i, K, j, R) where L, K, and R are graphs, and $i : K \to L$, $j : K \to R$ are injective graph morphisms. If we assume that K is a subgraph of both L and R, we write $r = (L \supseteq K \subseteq R)$ or (L, K, R).

Given a rule $r = (L, i, K, j, R)$, its component graphs are denoted by L_r, K_r, and R_r respectively. L_r is called the *left-hand* side of r, R_r is called the *right-hand side* of r, while K_r is called the *interface* or *gluing graph*.

Rules can be applied to graphs to derive new graphs. Applying a rule involves constructing two pushout diagrams (hence the name double-pushout). Before speaking about rule application, we define the pushout construction.

Definition 5.3 (Pushout construction). A *pushout* in the category of graphs is a tuple of graph morphisms

$$\langle i : A \to B, j : A \to C, b : B \to D, c : C \to D\rangle$$

where

— $b \circ i = c \circ j$,
— for all $\langle b' : B \to D', c' : C \to D'\rangle$ such that $b' \circ i = c' \circ j$ there exists a unique morphism $h : D \to D'$ such that $b' = h \circ b$ and $c' = h \circ c$.

In the following remark we give the intuition behind the concept of a pushout, and we show how it can be constructed in the category of graphs.

Remark 5.4. Given a pair of graph morphisms $\langle i : A \to B, j : A \to C\rangle$, the pushout graph D can be built from i and j by taking the disjoint union of the graphs B and C, and identifying the nodes and edges which have the same preimage in A. A is called the *gluing graph* because D is obtained by gluing B and C along A.

Now that we have defined the concept of a pushout, we can describe how a direct derivation step is performed.

Definition 5.5 (Direct derivation). Given a graph G and a rule $r = (L, i, K, j, R)$, if we can build a diagram as in Figure 10, i.e. if graphs D and H exist, together with morphism $g : L \to G$, $d : D \to G$, $k : K \to D$, $h : R \to H$, $d' : D \to H$, such that the two squares in the diagram are pushouts in the category of labelled directed graphs $\mathcal{DG}$, then we say that a *direct derivation* of G from H using r exists. In such a case we write $G \Rightarrow_r H$. We say that g (resp. k, h) is a *matching morphism* and that it identifies an *occurrence* of L in G (resp. K in D, R in H).

If $\mathcal{R}$ is a set of rules and G, H are two graphs, with $G \Rightarrow_{\mathcal{R}} H$ we mean that there exists $r \in \mathcal{R}$ such that $G \Rightarrow_r H$.

Intuitively, we can find a direct derivation of a graph H from a graph G using a rule $r = (L, i, K, j, R)$ by performing the following steps:

1 Choose an occurrence of L in G, i.e. a morphism $g : L \to G$;

$$\begin{tikzcd}
L \arrow[d, "g"] & K \arrow[l, "i"'] \arrow[r, "j"] \arrow[d, "k"] & R \arrow[d, "h"] \\
G & D \arrow[l, "d"] \arrow[r, "d'"'] & H
\end{tikzcd}$$

Fig. 10. DPO derivation step.

2 check the *gluing condition* on K, and g (see e.g. (Corradini *et al.* 1997) for the details), which ensures that we can

3 construct the context graph D by removing $g(L - K)$ from G, and morphisms $k : K \to D$, $d : D \to G$ such that

$$\langle i : K \to L, k : K \to D, g : L \to G, d : D \to G\rangle$$

is a pushout—the pair $\langle k : K \to D, d : D \to G\rangle$ is called the *pushout complement* of $\langle i : K \to L, g : L \to G\rangle$;

4 construct H by gluing D and R in K, i.e. by constructing the pushout of $\langle k : K \to D, j : K \to R\rangle$.

Given a diagram like in Figure 10, we have that i, j are injective and, as a consequence, that also d and d' are. Therefore, when we find it convenient, we will assume without loss of generality, that $L \supseteq K \subseteq R$ and $G \supseteq D \subseteq H$.

5.2. *A characterisation of DPO hierarchy transformation*

In this section we are going to define conditions that allow us to check whether a given DPO rule transforms rooted dags into rooted dags. This will give a characterisation of DPO hierarchy transformation.

To begin with, we observe once more that the DPO approach uses directed, node- and edge-labelled graphs. Our hierarchy graphs are indeed directed graphs—more precisely, rooted dags—where the node and edge labels are irrelevant. We will therefore assume, throughout Section 5.2, that hierarchy graphs have a fixed trivial node and edge labelling, by means of the alphabets $\Sigma = \Delta = \{\perp\}$.

As far as the graph structure is concerned, it is our task to characterise the rules that produce a correct hierarchy graph, and the ones that do not because they introduce cycles and/or loops, thus breaking the rooted-dag structure. We will tolerate rules that introduce parallel edges, since although a hierarchy with parallel edges specifies the child/parent relation on packages redundantly, we will always obtain a correct $\succ$ relation between packages.

Another important issue concerns matching morphisms upon rule application and whether they should be injective or not. The traditional DPO approach requires rule morphisms to be injective, while the matching morphism can be an arbitrary one. In (Habel *et al.* 2001) it is proved that we can restrict matching morphisms to injective ones, since the arbitrary approach can be simulated in the restricted one. Restricting ourselves to injective morphisms is technically convenient because it ensures that all paths are faithfully preserved by morphism. It can be the topic of future research to extend our results to the case where we allow non-injective morphisms. In what follows *all morphisms are injective*, unless we explicitly state that they are not.

We can now define dag preserving rules.

Definition 5.6 (Dag-preserving rules). A DPO rule $r = (L, i, K, j, R)$ is *dag-preserving* iff, for all graphs D, D', if D is a dag and $D \Rightarrow_r D'$, then D' is a dag as well.

Before proceeding, we introduce some useful terms and notation concerning paths. If p is a path in a graph G visiting the nodes $u_0, \ldots, u_h, \ldots, u_k, \ldots, u_n$ $(0 \leq h < k \leq n)$, we say that u_h *precedes* u_k and that u_k *follows* u_h in p. Given a directed graph G, we introduce the relations $\succ_G, \succ_G^+ \subseteq N_G \times N_G$, defined for all $u, v \in N_G$ as follows:

— $u \succ_G v$ if there exists $e \in E_G$ with $s_G(e) = u$ and $t_G(e) = v$;
— $u \succ_G^+ v$ if there exists a path from u to v in G.

The notation $\succ_G^+$ indicates the transitive closure of $\succ_G$.

We now consider the path-checking condition (see the following definition), which is satisfied by a rule r if whenever the right-hand side contains a path between two preserved nodes, then also the left-hand side contains a path between the same nodes. This easy-to-check condition ensures that a rule always transforms dags into dags (see Proposition 5.8).

Definition 5.7 (Path-checking condition). Given a rule $r = (L, i, K, j, R)$, the *path checking condition for* r states that

$$\forall u, v \in K : u \succ_R^+ v \Rightarrow u \succ_L^+ v$$

This condition gives us a characterisation of dag preserving rules, which is expressed by the following proposition.

Proposition 5.8. A rule (L, i, K, j, R), where L, K and R are dags, is dag preserving iff it satisfies the path-checking condition.

Before proving Proposition 5.8, we illustrate its meaning by means of an example in Figure 11. On the left side, a rule respecting the path-checking condition is depicted. Gluing nodes and their images are indicated by means of numbers. The paths between gluing nodes contained in the right-hand side have corresponding paths in the left-hand side. This ensures that no cycles are introduced in a dag to which the rule is applied. On the right side of the picture, a rule violating the path-checking condition is depicted: A new path from node 1 to node 3 is introduced. As a result, when applied to a dag that contains a path between the same nodes in the opposite direction, the rule introduces a cycle.

We now proceed with the proof of Proposition 5.8.

Proof.

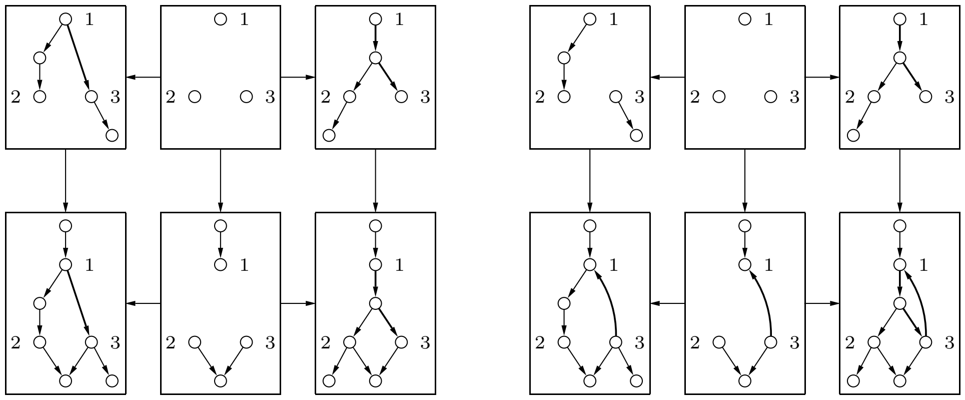

Fig. 11. Path-checking condition: examples.

If) Without loss of generality, since i and j are injective, we suppose that K is a subgraph of L and R. Let us then suppose that $(L \supseteq K \subseteq R)$ satisfies the path-checking condition, and that we have a direct derivation $G \Rightarrow_r H$, as depicted in Figure 10, where all morphisms are injective. Suppose also that H is not a dag, while G is.

Then we must have a cycle $e_1, \ldots, e_k$ $(k \geq 2)$ in H. Since R is a dag, the cycle cannot be the image of a cycle in R. This cycle cannot be the image of a cycle in D either. In fact, if D contained a cycle, then G would contain its image, contradicting the fact that G is a dag.

Then we can decompose $e_1, \ldots, e_k$ into subpaths $p_1, \ldots, p_{2n}$ (for some n, $n \leq k/2$), such that

— for all $i = 0, \ldots, n-1$: p_{2i+1} is the image of a path in R, and
— for all $i = 1, \ldots, n$: p_{2i} is the image of a path in D.

Each p_i visits a sequence of nodes $u_{i,0}, u_{i,1}, \ldots, u_{i,k_i}$, where $(u_{i,j-1}, u_{i,j})$ $(j = 1, \ldots, k_i)$ are the edges of the path in H. Then, for each p_i, we let $s_H(p_i) := u_{i,0}$ and $t_H(p_i) := u_{i,k_i}$, i.e. we call *source* (resp. *target*) of a path the source of its first (resp. the target of its last) edge.

It is clear that, for each $i = 1, \ldots, 2n-1$, $v_i := t_H(p_i) = s_H(p_{i+1})$, and that $v_{2n} := t_H(p_{2n}) = s_H(p_1)$. Then, since $v_1, \ldots, v_{2n}$ are images of nodes both in R and in D, they must be images of preserved nodes in K. We call the corresponding nodes in R, D, and K using the same names $v_1, \ldots, v_{2n}$.

Now, it is clear that, for each $i = 1, \ldots, n$, p_{2i} is the image of a path in D (by hypothesis) which has an image in G. We then call $p_2, p_4, \ldots, p_{2n}$ the corresponding paths in G.

Finally, if we look at L, it surely contains images of $v_1, \ldots, v_{2n}$ from K. Furthermore, for each $i = 1, \ldots, n-1$, since there is a path p_{2i+1} from v_{2i} to v_{2i+1} in R, by the path-checking condition there must be a path p'_{2i+1} connecting the same pairs of nodes in L. For the same reason, there exists a path p'_1 from v_{2n} to v_1 in L.

Once more, morphisms preserve paths and their ends, and therefore there are images of $p'_1, p'_3, \ldots, p'_{2n-1}$ in G, and $v_1, \ldots, v_{2n}$ are their ends. But then $p'_1, p_2, p'_3, p_4, \ldots, p'_{2n-1}, p_{2n}$ is a cycle in G, which contradicts the hypothesis that G is a dag.

As a last task, we must prove that no loops are introduced in H. If $e \in E_H$ is a loop, then e must either be the image of a loop in R or of a loop in D. However, R contains no loops, since it is a dag. D does not contain any loops either, because this would be mapped to loops in G, which is in turn a dag.

Only if) Suppose that $(L \supseteq K \subseteq R)$ does not satisfy the path-checking condition. Then there exist nodes $u, v \in N_K$, such that $u \succ_R^+ v$ but not $u \succ_L^+ v$.

Then we let $G = (N_G, E_G, s_G, t_G, l_G, m_G)$ be a dag, such that $N_G = N_L$ and $E_G = E_L \cup \{e\}$ (with $e \notin E_L$), $s_G(e) = v$, $t_G(e) = u$, $l_G = l_L$, $m_G|L = m_G$ and $m_G(e)$ has an arbitrary value. It is clear that G is a dag, since there cannot be cycles that do not pass through both u and v (otherwise the cycle would be in L already), nor can there be a cycle going through u, and v, since this would mean $u \succ_L^+ v$.

Now, since the inclusion from L into G is a morphism, we can apply r to G, deriving a graph H (see again Figure 10). Since the edge e of G is not an image of any edge in L, it must be the image of some edge, which we also came e, in the pushout complement D.

But then, $v \succ_H u$ (due to the image in H of edge e from D) and $u \succ_H^+ v$ (because $u \succ_R^+ v$), and then H contains a cycle. □

The path checking condition, which can be statically checked on DPO rules, gives a characterisation of dag preserving rules.

Remark 5.9. In (Habel *et al.* 1991), *jungles*—i.e. acyclic hypergraphs that respect some special conditions on the degree of their nodes and on their labelling—are used for representing terms over a certain signature, and (DPO) jungle rewriting for specifying term evaluation. In that paper a characterisation of DPO jungle rules is provided, i.e. of rules that transform any given jungle into a jungle. The path checking condition presented in this paper is similar to that characterisation of jungle rules and, in particular, it borrows the idea of checking that certain paths in the right-hand side of a rule correspond to paths in the left-hand side (see (Habel *et al.* 1991, Definition 4.2.b) for more details).

In addition to ensuring that a dag is transformed into a dag, we need to check that the transformed hierarchy still has a root node. We now introduce, in Definition 5.10, rooted-dag preserving rules and, in Definition 5.11, a condition on DPO rules that will ensure, together with the path-checking condition, that rooted dags are transformed into rooted dags. This last property is proved in Proposition 5.12.

Definition 5.10 (Rooted-dag preserving rule). A DPO rule r is *rooted-dag preserving* iff for all rooted dags G, for all graphs H, $G \Rightarrow_r H$ implies that H is a rooted dag.

Definition 5.11 (Root-preserving condition).
A DPO rule $r = (L, i, K, j, R)$, where K is a dag and L, R are rooted dags, respects the *root-preserving condition* if whenever there exists $\rho \in N_K$ with $\rho_L = i(\rho)$, we have $\rho_R = j(\rho)$.

This condition says that if the root ρ_L of the left-hand side is a preserved node, then its image in the right-hand side R is the root ρ_R of R. In the left side of Figure 12, we consider a rule respecting the path-checking and the root-preserving condition. Since the

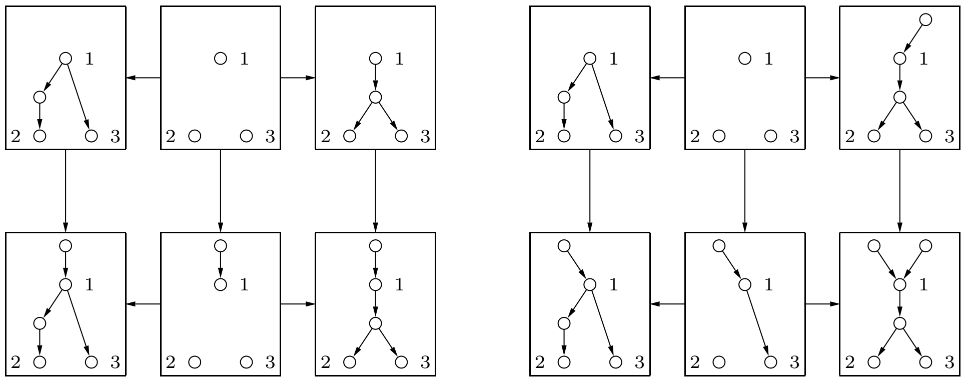

Fig. 12. Root-preserving condition: examples.

root of the left-hand side graph is a preserved node, it is also the root of the right-hand side. This ensures that a rooted-dag is transformed into a rooted-dag (see Proposition 5.12). On the right side we have a a rule that respects the path-checking condition but violates the root-preserving condition. This leads to a derivation where a second root is added to a rooted dag.

In the following proposition, we will prove that any rule r satisfying both the path-checking condition and the root-preserving condition transforms any rooted dag into a rooted dag.

Proposition 5.12 (Rooted-dag preservation).
A rule $r = (L, i, K, j, R)$, where K is a dag and L, R are rooted dags, is rooted-dag preserving iff it satisfies the path-checking condition and the root-preserving condition.

Before proving this proposition, we need one definition and a few lemmas. The proofs to the lemmas can be found in Appendix A. The definition deals with graphs whose paths are oriented w.r.t. a given subgraph.

Definition 5.13 (Oriented graphs). Given two graphs $K \subseteq G$, we say that G is *in-oriented* w.r.t. K if for all $u \in N_G - N_K$, there exists a node $v \in N_K$ such that $u \succ_G^+ v$. In such a case we write $G \succ K$.

Given two graphs $K \subseteq G$, we say that G is *out-oriented* w.r.t. K if for all $u \in N_K$, there exists a node $v \in N_G - N_K$ such that $u \succ_G^+ v$. In such a case we write $G \prec K$.

Remark 5.14. If K, G are dags with $K \subseteq G$ and $K \neq \emptyset$ then we cannot have both $G \succ K$ and $G \prec K$, because this would imply the existence of a cycle in G.

The first lemma states that the interface graph in a pushout diagram separates the two graphs that are glued together.

Lemma 5.15.
Given a pushout $\langle i : A \to B, j : A \to C, c : C \to D, b : B \to D\rangle$ in the category of directed graphs, where all morphisms are injective, the subgraph $bi(A) = cj(A)$ separates $b(B)$ from $c(C)$ in D, i.e. , for every two nodes $u \in b(N_B)$, $v \in c(N_C)$, if there exists a path $u = u_0, \ldots, u_k = v$ in D, then the path contains at least one node from $cj(N_A)$.

The second lemma studies the orientation of edges in the pushout complement of a given pushout diagram w.r.t. the interface.

Lemma 5.16. Given a pushout $\langle i : K \to L, k : K \to D, d : D \to G, g : L \to G\rangle$ in the category of directed graphs, where G is a rooted dag and all morphisms are injective, we have the following:

1 If $\rho_G \in g(N_L)$ then $D \succ k(K)$.
2 If $\rho_G \in d(N_D)$ then $L \succ i(K)$.

The following remark immediately follows from Lemma 5.16.

Remark 5.17. Given a pushout $\langle i : K \to L, k : K \to D, d : D \to G, g : L \to G\rangle$ in the category of directed graphs, where G is a rooted dag and all morphisms are injective, we always have $L \succ i(K) \vee D \succ k(K)$, because ρ_G has a preimage in at least one of the the dags D and L. Since all the considered graphs are dags, from remark 5.14 we also get that we can never have $L \prec i(K) \wedge D \prec k(K)$. This last observation will be used in the proof of Proposition 5.12.

We are now ready to prove Proposition 5.12.

Proof. We already know from Proposition 5.8 that a rule preserves dags iff it satisfies the path-checking condition. We must show that, provided that a rule satisfies the path-checking condition, it preserves rooted-dags iff it satisfies the root-preserving condition. *If*) Let $r = (L, i, K, j, R)$ be a rule satisfying the path-checking condition and the root-preserving condition, and let G be a rooted dag to which the rule can be applied. Then let $g : L \to G$, $k : K \to D$, $h : R \to H$, $d : D \to G$, and $d' : D \to H$ be graph morphisms and graphs as in Definition 5.5 (see also Figure 10). We distinguish the following four cases:

1 ρ_L is preserved—i.e. it is the image of a node of K—and $g(\rho_L) = \rho_G$. We prove that $h(\rho_R) = \rho_H$. Suppose that $v \in N_H$.

(a) If $v = h(u)$ for some $u \in N_R$ there exists a path $u \succ^+_R \rho_R$ in R. But then its image in H from v to $h(\rho_R)$ is the wanted path in H.

(b) If $v = d'(u)$ for some $u \in N_D - k(N_K)$, then $d(u) \succ^+_G \rho_G$. Since $\rho_G = g(\rho_L) = gi(\rho)$, for some $\rho \in N_K$, we can split the path from $d(u)$ to ρ_G into two subpaths $d(u) \succ^+_G dk(w)$, $dk(w) \succ^+_G \rho_G$ for some $w \in N_K$, such that all the nodes preceding $dk(w)$ in the first path belong to $d(N_D - k(N_K))$. But then $u \succ^+_D k(w)$, which implies $d'(u) \succ^+_H d'k(w)$. Now, if $w = \rho$ we are done, since $d'k(w) = hj(w) = hj(\rho) = h(\rho_R)$. Otherwise, if $w \neq \rho$, we have $j(w) \succ^+_R \rho_R$, which implies $d'k(w) = hj(w) \succ^+_H h(\rho_R)$ and, by concatenating the paths in H, we obtain $v \succ^+_H h(\rho_R)$ as required.

2 ρ_L is preserved and $g(\rho_L) \neq \rho_G$. In this case, if there existed $x \in N_L$ such that $\rho_G = g(x)$, we would have $x \succ^+_L \rho_L$, $g(x) \succ^+_G g(\rho_L) \succ^+_G \rho_G = g(x)$, and G would not be a dag. Therefore ρ_G cannot have a preimage in L, and there must exist $\overline{\rho} \in N_D \setminus k(N_K)$ such that $d(\overline{\rho}) = \rho_G$. We prove that $\rho_H = d'(\overline{\rho})$. Suppose that $v \in N_H$.

(a) If $v = h(u)$ for some $u \in N_R$, then $u \succ^+_R \rho_R$. This implies that $v = h(u) \succ^+_H h(\rho_R)$.

On the other hand, we have $g(\rho_L) \succ^+_G \rho_G$ and, since L is a dag, all nodes following $g(\rho_L)$ in this path have no preimage in L, otherwise we would have a cycle going through $g(\rho_L)$ in G. Let $\rho \in N_K$ such that $\rho_L = i(\rho)$ and $\rho_R = j(\rho)$. The path from $g(\rho_L)$ to $\rho_G = d(\overline{\rho})$ in G must have a preimage in D, which implies that $k(\rho) \succ^+_D \overline{\rho}$. As a consequence, $d'k(\rho) \succ^+_H d'(\overline{\rho})$. By concatenating paths in H, we have that $v \succ^+_H h(\rho_R) = d'k(\rho) \succ^+_H d'(\overline{\rho})$, as required.

(b) If $v = d'(u)$ for some $u \in N_D - k(N_K)$, then, since ρ_G is the root of G, we have $d(u) \succ^+_G \rho_G$ and, if $u \succ^+_D \overline{\rho}$, then $v \succ^+_H d'(\overline{\rho})$ and we are done. Otherwise, if not all the edges in the path from $d(u)$ to ρ_G have a preimage in D, we can decompose this path into paths $d(u) \succ^+_G dk(w)$, $dk(w) \succ^+_G \rho_G$, for some $w \in N_K$, such that $u \succ^+_D k(w)$. In this case, we have that $v = d'(u) \succ^+_H d'k(w)$ and $d'k(w) = hj(w) \succ^+_H d'(\overline{\rho})$ (since $j(w) \in N_R$, we fall back to case 2a above). By concatenating paths in H, we have that $v \succ^+_H d'(\overline{\rho})$, as required.

3 ρ_L is not preserved—i.e. it has no preimage in K—and $g(\rho_L) = \rho_G$. In this case we prove that $\rho_H = h(\rho_R)$ (note that the proof is very similar to case 1 above). Suppose that $v \in N_H$.

(a) If $v = h(u)$ for some $u \in N_R$ there exists a path $v \succ^+_H h(\rho_R)$ like in case 1a above.

(b) If $v = d'(u)$ for some $u \in N_D - k(N_K)$, then $d(u) \succ^+_G \rho_G$. Since $\rho_G = g(\rho_L)$, we can split the path from $d(u)$ to ρ_G into two subpaths $d(u) \succ^+_G dk(w)$, $dk(w) \succ^+_G \rho_G$ for some $w \in N_K$, such that all the nodes preceding $dk(w)$ in the first path belong to $d(N_D - k(N_K))$. But then $u \succ^+_D k(w)$, which implies $d'(u) \succ^+_H d'k(w)$. If $j(w) = \rho_R$ we are done, since $d'k(w) = hj(w) = h(\rho_R)$. Otherwise, if $j(w) \neq \rho_R$, we have $j(w) \succ^+_R \rho_R$, which implies $d'k(w) = hj(w) \succ^+_H h(\rho_R)$ and, by concatenating the paths in H, we have $v \succ^+_H h(\rho_R)$ as required.

4 ρ_L is not preserved and $g(\rho_L) \neq \rho_G$. We prove that this case never occurs. In fact, if $g(\rho_L) \neq \rho_G$, G must contain at least two nodes and at least one edge. Furthermore $g(\rho_L) \succ^+_G \rho_G$. By Lemma 5.15, there exists $w \in i(N_K)$ such that $g(w)$ lies on this path; as an extreme case, we can have $g(w) = \rho_G$, but we always have $w \neq \rho_L$, because ρ_L is not preserved. This implies that $w \succ^+_L \rho_L$, and therefore $g(w) \succ^+_G g(\rho_L) \succ^+_G g(w)$, and G would not be a dag[§].

Only-if) From Proposition 5.8, we already know that a rule that does not satisfy the path checking condition does not preserve dags. Let $r = (L, i, K, j, R)$ be a rule satisfying the path-checking condition and *not* satisfying the root-preserving condition. Then there exists a node $\rho \in N_K$, such that $\rho_L = i(\rho)$ and $\rho_R \neq j(\rho)$. In this case, let $\gamma \notin N_L$ be some node, $e \notin E_L$ be some edge, and let $G = (N, E, s, t, l, g)$ be the rooted dag with $N = N_L \cup \{\gamma\}$, $E = E_L \cup \{e\}$, $s(e) = \rho_L$, $t(e) = \rho$ (the labelling functions are irrelevant). It is clear that $\gamma = \rho_G$ is the root of G.

Then we can find a derivation from G to some graph H via rule r. Again, let $g : L \to G$, $k : K \to D$, $h : R \to H$, $d : D \to G$, and $d' : D \to H$ be graph morphisms and graphs as

[§] Notice that case 2 above is possible, since ρ_L has a preimage in the interface graph K, and therefore we can have a path $g(\rho_L) \succ^+_G \rho_G$ with no intermediate node from L.

in Definition 5.5, with g being the inclusion of L in G. Since $\rho_G \in d(N_D - k(N_K))$, we have that $D \prec k(K)$.

If we now look at the right-hand side of the rule r, we first observe that ρ_R must be a new node, i.e. there is no node $\rho' \in N_K$ such that $\rho_R = j(\rho')$. If such a node existed, then we would have that $j(\rho) \succ_R^+ j(\rho') = \rho_R$ and, since the rule also respects the path-checking condition, that $\rho_L = i(\rho) \succ_L^+ i(\rho')$. But since ρ_L is the root of L, we also have $i(\rho') \succ_L^+ \rho_L$, and therefore a cycle, against the hypothesis that L is a dag. We therefore conclude that such a $\rho' \in N_K$ does not exist, and therefore ρ_R is a new node. As a result, we also have that, for all nodes $u \in N_K$, $j(u) \succ_R^+ \rho_R \in (N_R - j(N_K))$, i.e. we have $R \prec j(K)$.

From Remark 5.17 and from the fact that $D \prec k(K)$ and $R \prec j(K)$ we conclude that H is not a rooted dag.

□

5.3. *A characterisation of DPO connection transformation*

In this section we will consider which conditions must be satisfied by a DPO rule so that it transforms connection graphs into connection graphs.

In Definition 3.5, we have defined connection graphs as special bipartite graphs with a set of packages and a set of atoms connected to each other by binary edges. Since we want to apply DPO rules to such graphs, we first need to translate connection graphs to labelled directed graphs. A hint to such a translation has already been given in Section 4.4. The main problem is that in a directed graph we cannot distinguish between the "package nodes" and the "atom nodes" of the connection graph that we want to represent. In order to do this, we use different labels to distinguish between packages and nodes from the underlying graph. As seen in Section 5.2, we label the packages of the hierarchy graph with a special symbol $\perp$. Furthermore, we assume that the underlying graph has a set of atom labels Σ_A, with $\perp \notin \Sigma_\text{A}$. Then for connection graphs we use a node alphabet $\Sigma = \{\perp\} \cup \Sigma_\text{A}$, and the edge alphabet $\Delta = \{\perp\}$ (edge labels in the connection graph are irrelevant).

One could think of an alternative representation where the atoms are the sources of coupling edges and packages are either target of such edges or isolated nodes. This solution has the drawback that one can easily turn an empty package p (isolated node) into an atom by adding a new coupling edge starting from p. So, whenever we want to add an atom a to a package q, we would have to add a new edge from a to q and check that a has an edge to some package q' already (a is actually an atom). We think that this solution would make rules unnecessarily complex and we prefer distinguishing atoms from packages by means of labels.

Let $\mathcal{BT}$ be the set of directed graphs over Σ and Δ, such that, for all $B \in \mathcal{BT}$, for all $n \in N_B$, if $l_B(n) \in \Sigma_\text{A}$, then there exists an edge $e \in E_B$ with $t_B(e) = n$ and $l_B(s_B(e)) = \perp$. It is clear that connection graphs can be mapped one-to-one to directed graphs in $\mathcal{BT}$. We will therefore encode connection graphs with graphs of $\mathcal{BT}$.

Before proceeding, we introduce some useful notation. Given a node $n \in N_B$, we define $\mathbf{indeg}_B(n) := \#\{e \mid n = t_B(e)\}$, and $\mathbf{outdeg}_B(n) := \#\{e \mid n = s_B(e)\}$. Given a label

$\lambda \in \Sigma$, we define the set $N_G^\lambda := \{n \in N_G \mid l_G(n) = \lambda\}$. Given a set of labels $M \subseteq \Sigma$, we define $N_G^M := \bigcup_{\lambda \in M} N_G^\lambda$ (which is empty if M is empty).

Given a rule $r = (L \supseteq K \subseteq R)$, we define the graphs

$$\begin{aligned} NEW(r) &:= (N_R - N_K, \emptyset, \emptyset, \emptyset, l_R|(N_R - N_K), \emptyset) \\ DEL(r) &:= (N_L - N_K, \emptyset, \emptyset, \emptyset, l_L|(N_L - N_K), \emptyset) \\ PRES(r) &:= (N_K, \emptyset, \emptyset, \emptyset, l_K, \emptyset) \end{aligned}$$

i.e. the graph containing the newly created nodes of rule r, the graph of deleted nodes, and the graph of preserved nodes. We will use this notation again in Section 6.

We can now define a condition on DPO rules, that ensures that connection graphs be transformed into connection graphs.

Definition 5.18 (Connection-graph preservation condition). A rule $r = (L \supseteq K \subseteq R)$, where L, K and R are directed graphs over $\{\perp\} \cup \Sigma_{\mathrm{A}}$ and $\{\perp\}$, satisfies the *connection-graph preservation condition* iff, for all $n \in N_{\mathrm{NEW}(r)} \cap N_R^{\Sigma_{\mathrm{A}}}$, we have $\mathbf{indeg}_R(n) > 0$ and, for all $n \in N_{PRES(r)} \cap N_R^{\Sigma_{\mathrm{A}}}$, we have that $\mathbf{indeg}_R(n) = 0$ implies $\mathbf{indeg}_L(n) = 0$.

Let us call *atom node* a node with label in Σ_{A} and *package node* a node with label $\perp$. The above condition says that a new atom node must be connected to at least one package node, and that if a preserved atom node is not explicitly connected by the considered rule to some package node, then its left-hand side must also contain no edges connecting the preserved atom node to a package node, otherwise the rule would remove that edge, possibly making the preserved node isolated.

Definition 5.19 (Connection-preserving rules). A rule $r = (L \supseteq K \subseteq R)$, where L, K and R are directed graphs over $\Sigma = \{\perp\} \cup \Sigma_{\mathrm{A}}$ and $\Delta = \{\perp\}$, is *connection preserving* iff, for all directed graphs B, B' over Σ and Δ, if $B \in \mathcal{TB}$ and $B \Rightarrow_r B'$, then $B' \in \mathcal{TB}$.

Proposition 5.20.

A rule r is connection preserving iff it satisfies the connection-graph preservation condition.

Proof.

If) Suppose that $r = (L \supseteq K \subseteq R)$ is a rule satisfying the connection-graph preservation condition, and suppose that G is a connection graph, H is a graph such that $G \Rightarrow_r H$. Then there exists a graph D and morphisms $g : L \to G$, $k : K \to D$, $h : R \to H$, $d : D \to G$, and $d' : D \to H$, such that the two resulting squares are pushouts. We have to show that every atom node in H is connected to a package node. Let $n \in N_H^{\Sigma_{\mathrm{A}}}$.

If n *does not have* a preimage in R then n has only a preimage in D, which we also denote with n. In this case there must exist an image of n in G (again, denoted with n). Since G is a connection graph, n has at least one incoming edge e in G. Furthermore, n has no preimage in L, otherwise it should have one in K and consequently in R. But then also the preimages of e and $s_G(e)$ can only be in D. These preimages have also images in H, and therefore we have found the wanted edge for n in H.

If n *does have* a preimage in R, which we also indicate with n, then we have two sub-cases:

1 If $n \in N_{NEW(r)} \cap N_R^{\Sigma_A}$, then $\mathbf{indeg}_R(n) > 0$, which means that we have the wanted edge in R, which is mapped to a corresponding edge in H.

2 If $n \in N_{PRES(r)} \cap N_R^{\Sigma_A}$, then either $\mathbf{indeg}_R(n) > 0$, and we are done, or $\mathbf{indeg}_R(n) = 0$.
In the latter case, we also have $\mathbf{indeg}_L(n) = \mathbf{indeg}_K(n) = 0$. This means that there are no incoming edges to n in any of the graphs of rule r in which n occurs.
Now, observe that n has also an occurrence in G (also denoted by n) and, since G is a connection graph, there exists an edge $e \in E_G$ with $t_G(e) = n$, which has no preimage in L, and therefore has a preimage in D. But this means that e has also an image in H, which is an incoming edge to n from some $p \in N_H^{\perp}$ as required.

Only if) Suppose that $r = (L \supseteq K \subseteq R)$ is a rule which does not satisfy the connection-graph preservation condition. Then we can have two cases:

1 There exists $n \in N_{NEW(r)} \cap N_R^{\Sigma_A}$, such that $\mathbf{indeg}_R(n) = 0$. Let G, H be two graphs over Σ and Δ, such that $G \Rightarrow_r H$. Then there exists a graph D and morphisms $g : L \to G$, $k : K \to D$, $h : R \to H$, $d : D \to G$, and $d' : D \to H$, such that the two resulting squares are pushouts.
Since n is a new node, it is not in the interface graph K. As a result, n has no image in D and $h(n)$ has no incoming edges in H. But this means that n is an atom node which has no containing package node in H, and H is not a connection graph.

2 There exists $n \in N_{PRES(r)} \cap N_R^{\Sigma_A}$, such that $\mathbf{indeg}_R(n) = 0$ and $\mathbf{indeg}_L(n) > 0$. Since we should have that r preserves connection graphs, no matter to which connection graph it is applied, let G be a graph constructed as below.
It is clear that $N_L^{\perp} \neq \emptyset$. In fact, there exists at least an edge $e \in E_L$ with $t_L(e) = n$ and $s_L(e) \in N_L^{\perp}$. We let $p := s_L(e)$ (this is an arbitrary choice: any other $\perp$-labelled node can serve for our construction).
Then let $G = (N_G, E_G, s_G, t_G, l_G, m_G)$, where

(a) $N_G := N_L$ and $l_G := l_L$,

(b) $E_G := E_L \uplus \{e_u \mid u \in N_G^{\Sigma_A} \wedge \mathbf{indeg}_L(u) = 0\}$, i.e. we add an extra edge for all atom nodes of L which have no incoming edges,

(c) $\forall e \in E_G$, we let

$$s_G(e) := \begin{cases} p & \text{if } u \in N_G^{\Sigma_A},\ \mathbf{indeg}_L(u) = 0, \text{ and } e = e_u \\ s_L(e) & \text{otherwise} \end{cases}$$

and

$$t_G(e) := \begin{cases} u & \text{if } u \in N_G^{\Sigma_A},\ \mathbf{indeg}_L(u) = 0, \text{ and } e = e_u \\ t_L(e) & \text{otherwise} \end{cases}$$

and finally we let $m_G(e) := \perp$.

G is a connection graph because it is obtained from L by adding all possibly missing edges, and the inclusion of L in G is a match, therefore we can apply r to G. Let H

the derived graph, which means that there exists a graph D, morphisms $g : L \to G$, $k : K \to D$, $h : R \to H$, $d : D \to G$, and $d' : D \to H$, such that g is the inclusion of L in G and the two resulting squares are pushouts.

Now, let us consider again node n which violates the connection-graph preservation condition. Since $\mathbf{indeg}_R(n) = 0$, we must also have $\mathbf{indeg}_K(n) = 0$, otherwise K would contain an edge which has no image in R.

All incoming edges to n in G are matched by an edge in L (these are no new edges added during the construction of G). Since these edges are not in K they are deleted by this application of r, and therefore H is not a connection graph.

□

With Proposition 5.20 we have provided a method for ensuring that a DPO rule preserves the structure of connection graphs to which it is applied. Combined with Proposition 5.12, we obtain a characterisation of DPO hierarchical graph transformation, in the sense that we are able to use DPO transformation rules to specify hierarchical graph transformation, and to check statically whether these rules transform hierarchical graphs into hierarchical graphs.

6. Comparison with the Flat Double-Pushout Approach

In this section, we compare hierarchical with non-hierarchical graph transformation in the double-pushout approach. To this aim the hierarchical graphs we consider consist of triples of directed labelled graphs where only nodes can be grouped into packages. (In general nodes and edges can be grouped into packages.) This kind of hierarchical graphs allows a straightforward translation into flat directed labelled graphs as used in the double-pushout approach by unifying the three components of a hierarchical graph. The coordinated rules considered in this section contain only double-pushout rules as described in the previous section. It turns out that under certain circumstances, for every hierarchical graph transformation $H \Rrightarrow^r H'$ there exists a direct derivation $[H] \Rightarrow_{r'} [H']$ in the double-pushout approach where $[H]$ and $[H']$ are obtained from flattening H and H', respectively. More precisely, every coordinated rule r which is based on the double-pushout approach and of a certain form can be translated into a set $DPO(r)$ of double-pushout rules such that $H \Rrightarrow^r H'$ if and only if $[H] \Rightarrow_{r'} [H']$ for some $r' \in DPO(r)$. Roughly speaking, we require that for every coordinated rule $(\gamma, \beta, \delta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ the node set of $L_\gamma - K_\gamma$ be isomorphic to the atom set of $L_\beta - K_\beta$ and the node set of $R_\gamma - K_\gamma$ be isomorphic to the atom set of $R_\beta - K_\beta$. Analogously, the "deleted/added packages" of δ must correspond to the "deleted/added packages" of β.

Assumptions. For the rest of this section, the following is assumed.

1 Every hierarchical graph (G, D, B) consists of directed labelled graphs where the node labels of G and D are disjoint sets. More precisely, B and D are interpreted as directed labelled graphs (cf. Sections 5.2 and 5.3).

2 For every coordinated rule $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$, γ, δ, and β are double-

pushout rules such that

$$\begin{array}{rcl} DEL(\gamma) & \cong & DEL(\beta)|atoms \\ DEL(\delta) & \cong & DEL(\beta)|packages \\ NEW(\gamma) & \cong & NEW(\beta)|atoms \\ NEW(\delta) & \cong & NEW(\beta)|packages. \end{array}$$

Here, $DEL(\beta)|atoms$ restricts the node set of $DEL(\beta)$ to its atoms, and $DEL(\beta)|packages$ restricts $DEL(\beta)$ to its packages. Analogously, $NEW(\beta)|atoms$ restricts the node set of $NEW(\beta)$ to its atoms and $NEW(\beta)|packages$ to its packages (cf. Section 5.3 for the definition of DEL and NEW.)

3 All occurrence morphisms in a hierarchical graph transformation step are injective. This means that for every $H \Rrightarrow^r H'$ with $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$, $H = (G, D, B)$, and $H' = (G', D', B')$, there exist derivations $G \Rightarrow_\gamma G'$, $G \Rightarrow_\delta D'$, and $B \Rightarrow_\beta B'$ in which all occurrence morphisms are injective (cf. Definition 5.5).

4 For every coordinated rule $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ and every $(a, b) \in N_{K_j} \times N_{K_\beta}$ $(j \in \{\gamma, \delta\})$, $(a, b) \notin \sim^L_j$ implies $(a, b) \notin \sim^R_j$. (It is worth noting that, according to Definition 4.8, r would not be applicable to a hierarchical graph if this condition were not satisfied.)

The required isomorphisms between deleted/added atoms or packages in the second assumption are not restrictive in the case of injective occurrence morphisms because when a hierarchical graph $H = (G, D, B)$ is transformed with a rule $r = (\gamma, \delta, \beta, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$, the rule β manipulates the node set of G and D and the edge set of B. Hence, every node of G deleted by β must also be deleted by γ and every node of D deleted by β must also be deleted by δ. Analogously, nodes added by β must also be added by either γ or δ.

The transformation of a hierarchical graph $H = (G, D, B)$ into a directed labelled graph is done by the union of its three components.

Construction 2 (Translation of hierarchical graphs). Let $H = (G, D, B)$ be a hierarchical graph. Then the corresponding directed labelled graph $[H]$ is equal to (N, E, s, t, l, m) where $N = N_G \cup N_D$, $E = E_G \cup E_D \cup E_B$, all edges keep their sources, targets, and labels, and all nodes keep their labels.

For every graph $H = [(G, D, B)]$ the component G will also be denoted by $graph(H)$. More generally, for every directed labelled graph H, the subgraph induced by the nodes labelled with node labels is denoted by $graph(H)$.[¶]

For every directed labelled graph $G = (N, E, s, t, l, m)$ we define its skeleton $skel(G)$ as (N, E, ι) where $\iota = \{(e, s(e)) \mid e \in E\} \cup \{(e, t(e)) \mid e \in E\}$ (cf. Section 3.1). Moreover, for every double-pushout rule $r = (L, K, R)$ we define its rule skeleton $skel(r)$ as $(skel(L), skel(R), tr_r)$ where the domain of tr_r is equal to $skel(K)$ and tr_r is the identity. It can be easily shown that the double-pushout approach is tracking (cf. Definition 4.4). In more detail, for every direct derivation step $G \Rightarrow_r G'$ with occurrence morphisms

[¶] For a graph H the subgraph induced by a subset N of its nodes consists of the nodes of N plus all edges which connect some nodes in N. The subgraph of H induced by a subset E of the edges of H consists of all edges in E plus all nodes which are attached to some edge in E.

$\bar{g}\colon L \to G$ and $\bar{h}\colon R \to G'$ choose the tuple $\langle g, h, \varphi\rangle$ as follows. The skeleton morphism $g\colon \mathit{skel}(L) \to \mathit{skel}(G)$ is equal to the occurrence morphism $\bar{g}$ restricted to $\mathit{skel}(L)$. Analogously, $h\colon \mathit{skel}(R) \to \mathit{skel}(G')$ is equal to $\bar{h}$ restricted to $\mathit{skel}(R)$. Moreover, choose $\varphi\colon \mathit{skel}(G) \to \mathit{skel}(G')$ such that $\mathbf{dom}(\varphi) = g(\mathit{skel}(K))$ and $\varphi(n) = h(\mathit{tr}_r(n'))$ for every n in $\mathbf{dom}(\varphi)$ with $n = g(n')$ $(n' \in K)$.

The next observation refers to a basic feature of the relations in coordinated rules. If two items are related in a coordinated rule, in every derivation step they are both either deleted, or added, or kept. This means that no item which is in $\mathit{DEL}(\gamma)$ or $\mathit{NEW}(\gamma)$ can be related to an item in K_β, and no item of $\mathit{DEL}(\beta)$ or $\mathit{NEW}(\beta)$ can be related to K_β. The same holds for the deleted or added packages of δ. This property is fundamental for the construction of double-pushout rules from coordinated rules, presented in this section.

Observation 6.1. Let $r = (\gamma, \beta, \delta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ be a coordinated rule. Then for $j \in \{\gamma, \delta\}$ we have $(n, n') \in \sim^L_j$ implies $(n, n') \in N_{K_j} \times N_{K_\beta}$ or $(n, n') \in N_{\mathit{DEL}(j)} \times N_{\mathit{DEL}(\beta)}$, and $(n, n') \in \sim^R_j$ implies $(n, n') \in N_{K_j} \times N_{K_\beta}$ or $(n, n') \in N_{\mathit{NEW}(j)} \times N_{\mathit{NEW}(\beta)}$.

Proof. Assume that the statement is false. Then the quadruples $(\mathit{tr}_j, \mathit{tr}_\beta, \sim^L_j, \sim^R_j)$ do not commute (for $j \in \{\gamma, \delta\}$). This contradicts the definition of coordinated rules. □

For the translation of a coordinated rule $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ into a set $\mathit{DPO}(r)$ of double-pushout rules, the rules γ, δ, and β are glued together. Roughly speaking, every $r' \in \mathit{DPO}(r)$ is constructed according to the following steps.

1 Build the disjoint union of δ, γ, and β.
2 Identify every deleted node of γ with a deleted atom of β in a one-to-one way such that $\sim^L_\gamma$ is satisfied, i.e. for $(n, n') \in \sim^L_\gamma$ n must be identified with n'.
3 Proceed analogously with the added nodes of γ and the deleted/added packages of δ.
4 Identify all nodes in the gluing parts of γ and β according to $\sim^L_\gamma$ and $\sim^R_\gamma$.
5 Identify all nodes in the gluing parts of δ and β according to $\sim^L_\delta$ and $\sim^R_\delta$.

For the formal construction of $\mathit{dpo}(r)$, the gluing of two graphs G and G' w.r.t. and injective (partial) graph morphism $h\colon G \to G'$, denoted by $(G + G')/\equiv_h$, is constructed by first building the disjoint union of G and G' and second identifying x with $h(x)$, for every $x \in \mathbf{dom}(h)$ (cf. also Remark 5.4). We use the notation $\equiv_h$ because h induces an equivalence relation $\equiv_h$ on the nodes and edges of G and G' consisting of every pair (x, x') where either $x = x'$, or $h(x) = x'$, or $h(x') = x$. This gluing construction can be generalised in a straightforward way to the gluing of three graphs G, G' and G'' w.r.t. injective (partial) graph morphisms $h\colon G \to G'$ and $h'\colon G'' \to G'$. The resulting graph $G + G' + G''/\equiv_{(h,h')}$ is obtained by building first the disjoint union of G, G', and G'' and by identifying then every item of G' with its preimages in G and G''. Analogously to the gluing of graphs we define the gluing of double-pushout rules w.r.t. injective (partial) rule morphisms. An injective rule morphism from $r = (L, K, R)$ to $r' = (L', K', R')$ is a pair of injective (partial) graph morphisms $\mathit{del}\colon L \to L'$ and $\mathit{new}\colon R \to R'$ where $\mathbf{dom}(\mathit{del}|K) = \mathbf{dom}(\mathit{new}|K)$ and $\mathit{del}(x) = \mathit{new}(x)$ for every x in $\mathbf{dom}(\mathit{del}|K)$. The resulting double-pushout rule is equal to $(L + L'/\equiv_{\mathit{del}}, (K + K')/\equiv_{\mathit{del}|K}, R + R'/\equiv_{\mathit{new}})$ and is denoted as $r + r'/\equiv_{(\mathit{del},\mathit{new})}$. The generalisation of the gluing of two rules to the gluing of three rules is also straightforward. More precisely, for rules $r = (L, K, R)$, $r' = (L', K', R')$, and $r'' = (L'', K'', R'')$ and injective (partial) rule morphisms $g = (g_1, g_2)$ from r to r' and $h = (h_1, h_2)$ from r'' to r', the rule $r + r' + r''/\equiv_{(g,h)}$ is defined by $L + L' + L''/\equiv_{(g_1,h_1)}, K + K' + K''/\equiv_{(g_1|K,h_1|K)}, R + R' + R''/\equiv_{(g_2,h_2)})$.

In the following construction we also consider the relation $\sim^L_\gamma$, $\sim^R_\gamma$, $\sim^L_\delta$ and $\sim^R_\delta$ as injective (partial) graph morphisms $\sim^L_\gamma\colon L_\gamma \to L_\beta$, $\sim^R_\gamma\colon R_\gamma \to R_\beta$, $\sim^L_\delta\colon L_\delta \to L_\beta$, and $\sim^R_\delta\colon R_\delta \to R_\beta$, defined in the obvious way as $\sim^L_\gamma (x) = y$ iff $(x, y) \in \sim^L_\gamma$, etc. Because of Observation 6.1, the third assumption above, and by definition of coordinated rules, the relations induce injective (partial) rule morphisms $g = (\sim^L_\gamma, \sim^R_\gamma)$ from γ to β and $h = (\sim^L_\delta, \sim^R_\delta)$ from δ to β.

Construction 3 (Translation of coordinated rules). Consider a coordinated rule $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$.

1 Let $del_\gamma\colon DEL(\gamma) \to DEL(\beta)|atoms$, $del_\delta\colon DEL(\delta) \to DEL(\beta)|packages$, $new_\gamma\colon NEW(\gamma) \to NEW(\beta)|atoms$, $new_\delta\colon NEW(\delta) \to NEW(\beta)|packages$ be isomorphisms such that for $j \in \{\gamma, \delta\}$ $del_j(n) = n'$ for all $(n, n') \in \sim^L_j$ with $n \in DEL(j)$, and $new_j(n) = n'$ for all $(n, n') \in \sim^R_j$ with $n \in NEW(j)$. Let $g = (g_1, g_2)$ with $g_1 = del_\gamma \cup \sim^L_\gamma$ and $g_2 = new^R_\gamma \cup \sim^R_\gamma$ and let $h = (h_1, h_2)$ with $h_1 = del_\delta \cup \sim^L_\delta$, and $h_2 = new_\delta \cup \sim^R_\delta$. Then the double-pushout rule $dpo(r, g, h)$ is equal to $(\gamma + \beta + \delta)/\equiv_{(g,h)}$.

2 The set of all double-pushout rules constructed as described in 1, is denoted by $DPO(r)$.

As the construction indicates we can build a set of double-pushout rules for every coordinated rule r. This means that for every different quadruple of isomorphisms del_γ, del_δ, new_γ, and new_γ, a double-pushout rule can be constructed.

The following lemma states that for every double-pushout rule r' in the above constructed set $DPO(r)$ every application of r' to a flattened hierarchical graph is equivalent to an application of r to the corresponding non-flattened hierarchical graph.

Lemma 6.1. Let $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ be a coordinated rule and let $H = (G, D, B)$, $H' = (G', D', B')$ be hierarchical graphs. Let $r' \in DPO(r)$ such that $[H] \Rightarrow_{r'} [H']$. Then

1 $G \Rightarrow_\gamma G'$, $B \Rightarrow_\beta B'$, $D \Rightarrow_\delta D'$, and

2 the diagram of Fig. 7 can be constructed.

Proof.

1 Let $r' = (L, K, R) = dpo(r, p, q)$ with $p = (p_1, p_2)$. For the derivation $[H] \Rightarrow_{dpo(r,p,q)} [H']$, let $f\colon L \to [H]$, $h\colon K \to Z$, and $g\colon R \to [H]$ be the corresponding occurrence morphisms. Let $\overline{f^\gamma}\colon L_\gamma \to L$ such that $\overline{f^\gamma}(x) = [x]_{\equiv_{p_1}}$ for all $x \in N_{L_\gamma} \cup E_{L_\gamma}$. Let $\overline{h^\gamma}\colon K_\gamma \to K$ be the restriction of $\overline{f^\gamma}$ to K. Let $h^\gamma\colon K_\gamma \to Z = h \circ \overline{h^\gamma}$. By definition we have $G = graph([H])$ and $[H]$ is isomorphic to $(Z + L)/\equiv_h$. Hence $G \cong graph((Z + L)/\equiv_h)$. Since atoms and packages are labelled disjointly $\equiv_h$ relates only atoms with atoms and only packages with packages. Different edges are not identified. Hence we get $G \cong (graph(Z) + graph(L))/\equiv_{h|graph(K)}$. Moreover, by

Construction 3 and definition of coordinated rules $\overline{f^\gamma}(L_\gamma)$ is a subgraph of *graph*(L) which is isomorphic to L_γ, $\overline{h^\gamma}(K_\gamma)$ is a subgraph of *graph*(K) which is isomorphic to K_γ and *graph*$(L) - \overline{f^\gamma}(L_\gamma)$ is a discrete graph that is equal to *graph*$(K) - \overline{h^\gamma}(K_\gamma)$. Hence, *graph*$(L) \cong (L_\gamma + \textit{graph}(K))/\equiv_{\overline{h^\gamma}}$. This implies $G \cong (\textit{graph}(Z) + L_\gamma)/\equiv_{h^\gamma}$ (because the composition of two pushouts is a pushout).
Hence, G is the result of gluing L_γ and *graph*(Z) in K_γ. In the same way it can be shown that G' is obtained by gluing R_γ and *graph*(Z) in K_γ. Hence, $G \Rightarrow_\gamma G'$. Analogously we can show that $B \Rightarrow_\beta B'$ and $D \Rightarrow_\gamma D'$.

2 Since the double-pushout approach is tracking, we get in connection with point one of this lemma that the vertical side and middle squares of Fig. 7 exist and commute. The back squares also exist and commute because r is a coordinated rule. Let m_γ and m_β be defined as the occurrence morphisms $L_\gamma \to G$ and $L_\beta \to B$ in the derivations constructed in point 1 (restricted to the skeletons of L_γ and L_β). Then for all $(n, n') \in \sim^L_\gamma$ we have $m_\gamma(n) = m_\beta(n')$. Hence $(m_\gamma(n), m_\beta(n')) \in \sim^G$ which implies that the top left square commutes. Analogously it can be shown that the top right square and the bottom squares commute. It remains to show that the front squares commute. Choose $\varphi_\gamma(n) = m'_\gamma(tr_\gamma(n'))$ for every node n in $m_\gamma(K)$ with $n = m_\gamma(n')$, where m'_γ is equal to the occurrence morphism from R_γ to G' in the above constructed derivation $G \Rightarrow_\gamma G'$ (restricted to the skeleton of G). Now let n be a node of G. Let a be a node of K_γ and let b be a node of K_β such that $m_\gamma(a) = m_\beta(b) = n$. Then by definition n is in $\mathbf{dom}(\varphi_\gamma)$ and also in $\mathbf{dom}(\varphi_\beta)$.
If $(a, b) \in \sim^L_\gamma$ we get $(a, b) \in \sim \gamma^R$ and $(m'_\gamma(a), m'_\beta(b)) \in \sim^{G'}$. Since $m'_\gamma(a) = \varphi_\gamma(m_\gamma(a))$ and $m'_\beta(b) = \varphi_\beta(m_\beta(b))$ we have $(\varphi_\gamma(m_\gamma(a)), \varphi_\beta(m_\beta(b))) \in \sim^{G'}$, i.e. $(\varphi_\gamma(n), \varphi_\beta(n)) \in \sim^{G'}$.
If $(a, b) \notin \sim^L_\gamma$ we get by definition that $\overline{f^\gamma}(a)$ and $\overline{f^\beta}(b)$ are distinct nodes of K which are mapped to the same node in $[H]$, say x. (The morphism $\overline{f^\beta}\colon L_\beta \to L$ is defined analogously to $\overline{f^\gamma}$ above.) By the third assumption above and the construction of double-pushout rules from coordinated rules $\overline{f^\gamma}(a)$ and $\overline{f^\beta}(b)$ are distinct nodes in R which by definition of rule application in the double-pushout approach are mapped to the same node in $[H']$ by the corresponding occurrence morphism of $[H] \Rightarrow_{dpo(r,p,q)} [H']$. Hence, we get that $m'_\gamma(a) = m'_\beta(b)$ which implies $(\varphi_\gamma(n), \varphi_\beta(n)) \in \sim^{G'}$.

□

The next lemma states that for every hierarchical graph transformation there exists an equivalent transformation on the flattened graphs. More precisely, for every hierarchical graph transformation from H to H' via rule r, Construction 3 gives us a double-pushout rule r' such that $[H] \Rightarrow_{r'} [H']$.

Lemma 6.2. Let $r = (\gamma, \delta, \beta, \sim^L_\gamma, \sim^R_\gamma, \sim^L_\delta, \sim^R_\delta)$ be a coordinated rule. Let H and H' be hierarchical graphs such that $H \Rrightarrow^r H'$. Then $[H] \Rightarrow_{r'} [H']$ where $r' \in DPO(r)$.

Proof. For $H = (G, D, B)$ and $H' = (G', D', B')$, $H \Rrightarrow^r H'$ implies by definition $G \Rightarrow_\gamma G'$, $B \Rightarrow_\beta B'$, and $D \Rightarrow_\delta D'$. Let $f^\gamma\colon L_\gamma \to G$, $f^\beta\colon L_\beta \to B$, and $f^\delta\colon L_\delta \to$

D, $g^\gamma \colon R_\gamma \to G'$, $g^\beta \colon R_\beta \to B'$, and $g^\delta \colon R_\delta \to D'$ be the corresponding occurrence morphisms.

For $(n,n') \in N_{L_\gamma} \times N_{L_\beta}$, let $(n,n') \in \equiv_{p_1}$ iff $f^\gamma(n) = f^\beta(n')$, and for $(n,n') \in N_{L_\delta} \times N_{L_\beta}$ let $(n,n') \in \equiv_{q_1}$ iff $f^\delta(n) = f^\beta(n')$. Analogously, for $(n,n') \in N_{R_\gamma} \times N_{R_\beta}$, let $(n,n') \in \equiv_{p_2}$ iff $g^\gamma(n) = g^\beta(n')$, and for $(n,n') \in N_{R_\delta} \times N_{R_\beta}$, let $(n,n') \in \equiv_{q_2}$ iff $g^\delta(n) = g^\beta(n')$. (By definition $\equiv_{p_1}$ is compatible with $\sim^L_\gamma$, $\equiv_{p_2}$ with $\sim^R_\gamma$, $\equiv_{q_1}$ with $\sim^L_\delta$, and $\equiv_{q_2}$ $\sim^R_\delta$.) Let $(L,K,R) = dpo(r,p,q)$.

Define $f \colon L \to [H]$ as follows. If n is a node of L_γ, $f([n]_{\equiv_{p_1}}) = f^\gamma(n)$. If n is an atom of L_β, $f([n]_{\equiv_{p_1}}) = f^\beta(n)$. Analogously, if n is a package of L_δ, $f([n]_{\equiv_{q_1}}) = f^\delta(n)$; and if n is a package of L_β, $f([n]_{\equiv_{q_1}}) = f^\beta(n)$. Moreover, define $f(x) = f_\gamma(x)$ if x is an edge of L_γ, $f(x) = f_\beta(x)$ if x is an edge of L_β, and $f(x) = f_\delta(x)$ if x is an edge of L_γ. (The graph morphism f may be non-injective but only nodes of K can be identified.) Define $g \colon R \to [H']$ analogously to f, i.e. $g([n]_{\equiv_{p_2}}) = g^\gamma(n)$ if n is a node of R_γ, etc.

Let $Z = (G - f^\gamma(L_\gamma - K_\gamma)) \cup (B - f^\beta(L_\beta - K_\beta)) \cup (D - f^\delta(L_\delta - K_\delta))$. Then by definition and assumption the following holds.

$$\begin{aligned}
[H] - f(L-K) &= (G \cup B \cup D) - f(L-K) \\
&= (G - f^\gamma(L_\gamma - K_\gamma)) \cup (B - f^\beta(L_\beta - K_\beta)) \cup (D - f^\delta(L_\delta - K_\delta)) \\
&\cong Z,
\end{aligned}$$

and

$$\begin{aligned}
[H'] - g(L-K) &= (G' \cup B' \cup D') - g(R-K) \\
&= (G' - g^\gamma(R_\gamma - K_\gamma)) \cup (B' - g^\beta(R_\beta - K_\beta)) \cup (D' - g^\delta(R_\delta - K_\delta)) \\
&\cong (G - f^\gamma(L_\gamma - K_\gamma)) \cup (B - f^\beta(L_\beta - K_\beta)) \cup (D - f^\delta(L_\delta - K_\delta)) \\
&\cong Z.
\end{aligned}$$

Hence, we get that $[H]$ is obtained from gluing Z and L in K, and $[H']$ is obtained from gluing R and Z in K. This means $[H] \Rightarrow_{dpo(r,p,q)} [H']$. □

From the previous lemmas we get the following theorem.

Theorem 6.3. Let H and H' be hierarchical graphs and let r be a coordinated rule. Then $H \Rrightarrow^r H'$ if and only if $[H] \Rightarrow_{r'} [H']$ for some $r' \in DPO(r)$.

7. Related Work

Hierarchical graphs appear in the computer science literature often as a means to model complex networks of objects with additional structure on top of them. In this overview of related work, we will first consider approaches to hierarchical graphs from the areas of object-oriented modelling and databases (with particular attention to hypermedia), and then concentrate on approaches from the graph grammar community, to which our approach belongs.

In object orientation (see e.g. (Rumbaugh *et al.* 1991)) and in databases (see e.g. (Elmasri and Navathe 1994)), it is common to model a certain domain as a graph-like structure: objects (resp. entities) correspond naturally to nodes of a graph, while links (resp. relationships) correspond to edges. In both communities, the need for additional structuring is often felt, leading to the proposal of hierarchical data models.

One of the first examples in this area is the *higraph* model, which is proposed in (Harel 1988) as a means for modelling database structures, for knowledge representation, and as the basis for statecharts, a formalism for modelling reactive systems. Compared to our model, higraphs is a coupled approach, where so-called *blobs* play both the role of packages and of nodes. The blob hierarchy is acyclic. No operations are defined for higraphs.

A second example is the Hypernode model, introduced in (Poulovassilis and Levene 1994), which was designed as a data model for databases. This model is again coupled, since the hierarchy is based on complex nodes (called *hypernodes*). Hypernodes are typed.

A last example that we would like to consider is the modelling of hypertexts and hypermedia: A hypertext can also naturally be modelled as a graph. In this area we also find examples of hierarchical structuring, like in the Hypermedia system Hyperwave (see (Maurer 1996)), where *containers* play a similar role as our packages and allow to build hierarchical hypertext structures, or in the hypermedia application design methodology OOHDM (see e.g. (Schwabe and Barbosa 1994)), where *navigation contexts* offer similar primitives to structure the navigation space of a hypermedia application at design time. Both containers and navigation contexts are added as a structuring primitive which is not part of the underlying graph structure, and can therefore be considered as decoupled approaches.

The first approach to hierarchical graphs in the graph-grammar community can be found in (Pratt 1979). Here hierarchical graphs are used as a model for data structures in the implementation of programming languages. This approach also allows to specify rewrite rules based on node replacement to generate hierarchical graphs. Compared to ours, this approach is a coupled approach (the hierarchy is built on nodes), and the transformation is restricted to the generation of hierarchical graphs, whereas our framework allows more general kinds of transformations, with both constructive and destructive operations.

In (Taentzer 1996), *distributed graphs* are introduced. These can be seen as a primitive kind of hierarchical graph, where the hierarchy has only two levels. This approach uses the double-pushout approach to graph transformation to specify both hierarchy (network) and local graph transformation.

The *encapsulated hierarchical graph* (EHG) model, proposed in (Engels and Schürr 1995) provides hierarchical structuring of graphs and encapsulation of graph elements, together with a notion of hierarchical graph typing. It is however limited in that it only allows tree-like hierarchies, does not provide any notion of transformation, and it is coupled, since the hierarchy is built using particular nodes, called *complex nodes.* The approach presented in (Busatto *et al.* 2000) is an evolution of the EHG model, with a simpler notion of typing and with the introduction of the idea of decoupling. This model also provides first ideas about operations on hierarchical graphs.

In (Drewes *et al.* 2002), another hierarchical graph model is proposed, based on hypergraphs. Here, only strict tree-like hierarchies are supported, i.e. tree-like hierarchies without edges (hyperedges) attached to nodes in different components of the hierarchy. This approach is coupled, since the hierarchy is built using special hyperedges, called *frames*. Hierarchical graph transformation is provided through an extension of the double-pushout approach to graph transformation. This approach also provides a notion of flattening for hierarchical graphs and hierarchical rules similar to ours.

In (Engels and Heckel 2000), an approach to hierarchical graphs is presented, still based on the double-pushout approach. This approach supports hierarchical graph typing and typed hierarchical graph transformation. A major weak point of this approach is that it does not provide any means to ensure that the hierarchy is well-formed (that it is indeed a tree or dag), nor any means to preserve such structure during transformation. Furthermore, this approach is coupled, since the hierarchy is built using nodes and aggregation edges between them.

In (Milner 2001; Milner 2002) *bigraphs*, a variety of hierarchical graphs, are used for modelling mobile computations. A bigraph consists of a graph (the *monograph*) and a tree-like hierarchy (the *topograph*). The hierarchy is coupled: nodes in the monograph are hierarchy components in the topograph. Boundary crossing edges are allowed. Bigraph transformations can be specified through reaction rules; these describe a local transformation of the monograph and of the topograph, thus resembling some kind of coordinated hierarchical graph transformation. It can be a topic for future research to compare reaction rules with hierarchical graph transformation rules more closely.

Finally, (Palacz 2004) proposes a hierarchical graph model that supports multiple, tree-like hierarchies where both nodes and edges can be used as hierarchy components. Hierarchical graph transformation rules are based on the DPO graph transformation approach. It is easy to verify that every rule in this approach satisfies our path-checking condition; the root-checking condition is also satisfied because hierarchical transformation rules (Definition 21) contain *root-level morphisms*.

8. Conclusions and Future Work

Although hierarchical graphs are often used in various areas of computer science, there is no common understanding of what a hierarchical graph is, nor does there a common hierarchical graph data model exist. This implies that different authors define their own hierarchical graph model, often re-discovering the same concepts over and over again, and that the particular models mix general concepts of hierarchical graphs (hierarchical structuring, constraints derived from this structuring) with concepts that are specific to a given application (attributes, node and edge labels, . . .).

We therefore felt the need for a common hierarchical graph concept and data model, which motivated the development of the hierarchical graph model proposed in this paper. In this respect, we have achieved the following goals:

— Our model does not force one to choose a particular kind of graph (directed, undirected, hypergraph) as the underlying graph to be structured.
— Our model does not force one to build the hierarchy using elements of the graph (nodes, edges, hyperedges): the hierarchy is an independent structure into which elements of a graph are distributed. We call such approaches *decoupled*.
— Our model does not force one to choose a particular graph transformation approach

to define operations on the underlying graph and on the hierarchy, it rather provides a framework, in which different graph transformation approaches can be plugged in to define concrete hierarchical graph transformation approaches. This result is achieved by combining the idea of decoupling and the notion of a graph transformation approach.

Besides defining an abstract framework for hierarchical graphs and their transformation, we have instantiated it to the double-pushout approach, which has often been applied both to graph transformation and to hierarchical graph transformation (for the latter case, see Section 7). Here we have addressed two issues: The definition of consistent transformations on the hierarchy structure, meaning that we have to choose rules that do not produce forbidden hierarchies, and the translation of hierarchical graphs and hierarchical graph transformation to flat graphs and flat double-pushout transformation.

Our decoupled approach allows us to study the hierarchy structure and its transformations separately, thus making it easier to deal with our consistency issues in a general way. In our opinion, this problem has not been satisfactorily dealt with in the literature (see e.g. (Engels and Heckel 2000)), or has been solved by considering only a restricted class of hierarchies (see e.g. (Pratt 1979; Taentzer 1996; Drewes *et al.* 2002)). We have studied this problem for generic hierarchies in the double-pushout approach, showing that it is possible to check statically whether a given double-pushout rule specifies only consistent transformations.

The flattening of hierarchical graphs shows that it is possible to *implement* our double-pushout hierarchical graph transformations using the plain double-pushout approach on flat graphs. We have shown that hierarchical double-pushout rules can be translated to sets of traditional flat rules which specify an equivalent transformation in flattened graphs. A similar result can be found in (Drewes *et al.* 2002).

As far as future work is concerned, a notion of typing and typed hierarchical graph transformation is still not available in our approach, although there are already examples in the literature (Engels and Schürr 1995; Engels and Heckel 2000). Following our decoupled approach, typing can be defined as a combination of typing on the three components of a hierarchical graph. As a consequence, typed hierarchical graph transformation could also be obtained as a combination of typed transformation on the component graphs. In spite of these first ideas, typed transformation in our model is still a subject for future work.

Encapsulation can also be a useful feature for modelling certain domains (see (Engels and Schürr 1995)). Our model allows graph elements to be distributed arbitrarily in a hierarchy, but it still provides no means to control such distribution. Import/export interfaces in the style of (Engels and Schürr 1995) seem an interesting option.

In this work we have instantiated our framework for hierarchical graph transformation to the double-pushout approach since this approach is well-known and commonly used in the literature. It is however a natural development of this research to instantiate our framework to other rule-based approaches to graph transformation (see also (Busatto and Hoffmann 2001), (Busatto 2002, Chapter 7)).

References

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A. and Taentzer, G. (1999) Graph transformation for specification and programming *Science of Computer Programming* **34** (1), 1–54 (April).

Botafogo, R. A., Rivlin, E. and Shneiderman, B. (1992) Structural analysis of hypertexts: identifying hierarchies and useful metrics *ACM Transactions on Information Systems* **10**, 142–180 (April).

Busatto, G., Engels, G., Mehner, K. and Wagner, A. (2000) A framework for adding packages to graph transformation systems. In Ehrig, H. *et al.* (editors), *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, *Lecture Notes in Computer Science* **1764**, 352–367. Springer-Verlag.

Busatto, G. and Hoffmann, B. (2001) Comparing notions of hierarchical graph transformation. In Taentzer, G., Baresi, L. and Pezzè, M. (editors), *Workshop on Graph Transformation and Visual Modelling, Satellite of ICALP'2001.*, 312–318. Elsevier.

Busatto, G. (2002) An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation. PhD thesis, Department of Computer Science, University of Paderborn, Germany.

Corradini, A. and Montanari, U. (1995, editors) *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, *Electronic Notes in Theoretical Computer Science* **2**. Elsevier.

Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R. and Löwe, M. (1997) Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In Rozenberg, G. (editor), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 3, 163–246. World Scientific.

Cuny, J., Ehrig, H., Engels, G. and Rozenberg, G. (1996, editors) *Graph Grammars and Their Application to Computer Science*, *Lecture Notes in Computer Science* **1073**. Springer-Verlag.

Drewes, F., Knirsch, P., Kreowski, H.-J. and Kuske, S. (2000) Graph transformation modules and their composition. In Nagl, M., Schürr, A. and Münch, M. (editors), *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, *Lecture Notes in Computer Science* **1779**, 15–30. Springer-Verlag.

Drewes, F., Hoffmann B. and Plump, D. (2000) Hierarchical graph transformation *Journal of Computer and System Sciences* **64** (2), 249–283 (March).

Ehrig, H., Pfender, M. and Schneider, H. J. (1973) Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, 167–180, Iowa City, 1973.

Ehrig, H. (1979) Introduction to the algebraic theory of graph grammars. In Claus, V., Ehrig, H. and Rozenberg, G. (editors), *Graph-Grammars and Their Application to Computer Science and Biology*, *Lecture Notes in Computer Science* **73**, 1–69. Springer-Verlag.

Ehrig, H., Habel, A., Kreowski, H.-J. and Parisi-Presicce, F. (1991) From graph grammars to high level replacement systems. In Ehrig, H., Kreowski, H.-J. and Rozenberg, G. (editors), *Graph Grammars and Their Application to Computer Science*, *Lecture Notes in Computer Science* **532**, 269–291. Springer-Verlag.

Ehrig, H., Habel, A., Kreowski, H.-J. and Parisi-Presicce, F. (1991) Parallelism and concurrency in high level replacement systems *Mathematical Structures in Computer Science* **1**, 361–404.

Ehrig, H. and Engels, G. (1996) Pragmatic and semantic aspects of a module concept for graph transformation systems. In Cuny, J., Ehrig, H., Engels, G. and Rozenberg, G. (editors), *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, *Lecture Notes in Computer Science* **1073**, 137–154. Springer-Verlag.

Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A. and Corradini, A. (1997) Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg, G. (editor), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*, chapter 4, 247–312. World Scientific.

Ehrig, H., Engels, G., Kreowski, H.-J. and Rozenberg, G. (1999, editors) *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific.

Ehrig, H., Kreowski, H.-J., Montanari, U. and Rozenberg, G. (1999, editors) *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution*. World Scientific.

Elmasri, R. and Navathe, S. B. (1994) *Fundamentals of Database Systems*. Benjamin/Cummings.

Engels, G. and Schürr, A. (1995) Encapsulated hierachical graphs, graph types, and meta types. In Corradini, A. and Montanari, U. (editors), *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, *Electronic Notes in Theoretical Computer Science* **2**. Elsevier.

Engels, G. and Heckel, R. (2000) Graph transformation as unifying formal framework for system modeling and model evolution. In Welzl, E., Montanari, U. and Rolim, J. D. P. (editors), *Automata, languages and programming: 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000: proceedings*, *Lecture Notes in Computer Science* **1853**, 127–150. Springer-Verlag.

Grosse-Rhode, M., Parisi-Presicce, F. and Simeoni, M. (1998) Spatial and temporal refinement of typed graph transformation systems. In *Proc. Mathematical Foundations of Computer Science*, *Lecture Notes in Computer Science* **1450**, 553–561.

Harel, D. (1988) On visual formalisms. *Communications of the Association for Computing Machinery* **31** (5), 514–530.

Habel, A., Kreowski, H.-J. and Plump, D. (1991) Jungle evaluation. *Fundamenta Informaticae* **XV**, 37–60.

Habel, A., Müller, J. and Plump, D. (2001) Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science* **11** (5), 637–688.

Heckel, R., Hoffmann, B., Knirsch, P. and Kuske, S. (2000) Simple modules for GRACE. In Ehrig, H., Engels, G., Kreowski, H.-J. and Rozenberg, G. (editors), *Proc. Theory and Application of Graph Transformations*, *Lecture Notes in Computer Science* **1764**, 383–395. Springer-Verlag.

Kaplan, S. M., Loyall, J. P. and Goering, S. K. (1991) Specifying concurrent languages and systems with Δ-Grammars. In Ehrig, H., Kreowski, H.-J. and Rozenberg, G. (editors), *Proc. Graph Grammars and Their Application to Computer Science*, *Lecture Notes in Computer Science* **532**, 475–489. Springer-Verlag.

Kreowski, H.-J. and Kuske, S. (1996) On the interleaving semantics of transformation units—A step into GRACE. In Cuny, J., Ehrig, H., Engels, G. and Rozenberg, G. (editors), *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, *Lecture Notes in Computer Science* **1073**, 89–106. Springer-Verlag.

Kreowski, H.-J., Kuske, S. and Schürr, A. (1997) Nested graph transformation units. *International Journal of Software Engineering and Knowledge Engineering* **7**, 479–502.

Kreowski, H.-J., and Kuske, S. (1999) Graph transformation units and modules. In Ehrig, H., Engels, G., Kreowski, H.-J. and Rozenberg, G. (editors), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, chapter 15, 607–638. World Scientific.

Kreowski, H.-J., and Kuske, S. (1999) Graph transformation units with interleaving semantics. *Formal Aspects of Computing* **11** (6), 690–723.

Kuske, S. (1999) *Transformation Units – A Structuring Principle for Graph Transformation Systems.* PhD thesis, Department of Mathematics and Computer Science, University of Bremen, Germany.

Maurer, H. (1996) *Hyperwave, The Next Generation Web Solutions.* Addison Wesley Longman.

Milner, R. (2001) Bigraphical reactive systems. In Larsen, K. G. and Nielsen, M. (editors), *Proc. 12th International Conference on Concurrency Theory, CONCUR 2001*, *Lecture Notes in Computer Science* **2154**, 16–35. Springer-Verlag.

Milner, R. (2002) Bigraphs as a model for mobile interaction. In Corradini, A., Ehrig, H., Kreowski, H.-J. and Rozenberg, G. (editors), *Proc. First International Conference on Graph Transformation, ICGT 2002*, *Lecture Notes in Computer Science*, **2505**, 8–13. Springer-Verlag.

Palacz, W. (2004) Algebraic hierarchical graph transformation. *Journal of Computer and System Sciences* **68**, 497–520 (March).

Poulovassilis, A. and Levene, M. (1994) A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems* **12** (1), 35–68, (January).

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

Rozenberg, G. (1997, editor) *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object Modelling and Design.* Prentice Hall.

Schürr, A. and Taentzer, G. (1995) DIEGO, another step towards a module concept for graph transformation systems. In Corradini, A. and Montanari, U. (editors), *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, *Electronic Notes in Theoretical Computer Science* **2**. Elsevier.

Schürr, A. and Winter, A. J. (2000) UML packages for PROgrammed Graph REwriting Systems. In Ehrig, H., Engels, G., Kreowski, H.-J. and Rozenberg, G. (editors), *Proc. Theory and Application of Graph Transformations*, *Lecture Notes in Computer Science* **1764**, 396–409. Springer-Verlag.

Schwabe, D. and Barbosa, S. D. J. (1994) Navigation modelling in hypermedia applications. Technical Report MCC 42/94, Departamento de Informática, PUC Rio.

Taentzer, G. (1996) *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin. Shaker Verlag.

Taentzer, G., Baresi, L. and Pezzè, M. (2001, editors) *Workshop on Graph Transformation and Visual Modelling, Satellite of ICALP'2001.* Elsevier.

Welzl, E., Montanari, U. and Rolim, J. D. P. (2000, editors) *Automata, languages and programming: 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000: proceedings*, *Lecture Notes in Computer Science* **1853**. Springer-Verlag.

Appendix A. Proofs of Lemmas

In this appendix we provide the proofs that we have omitted in Section 5.

Lemma 5.15

Given a pushout $\langle i : A \to B, j : A \to C, c : C \to D, b : B \to D\rangle$ in the category of directed graphs, where all morphisms are injective, the subgraph $bi(A) = cj(A)$ separates $b(B)$ from $c(C)$ in D, i.e. , for every two nodes $u \in b(N_B)$, $v \in c(N_C)$, if there exists a path $u = u_0, \ldots, u_k = v$ in D, then the path contains at least one node from $cj(N_A)$.

Proof.
Let $u \in c(N_C)$ and $v \in b(N_B)$, and let $u = u_0, \ldots, u_n = v$ be a path in D from u to v for some $n \in \mathbb{N} : n > 0$. Let $k \in \mathbb{N}$, $0 \leq k < n$, such that $u_k \in c(N_C)$ and $u_{k+1} \in b(N_B)$.

If either u_k or u_{k+1} are in $bi(N_A) = cj(N_A)$, then we are done. Otherwise we have that $u_k \in c(N_C - j(N_A))$ and $u_{k+1} \in b(N_B - i(N_A))$. But in this latter case, there cannot be any edge between u_k and u_{k+1} in D.

□

Lemma 5.16

Given a pushout $\langle i : K \to L, k : K \to D, d : D \to G, g : L \to G\rangle$ in the category of directed graphs, where G is a rooted dag and all morphisms are injective, we have the following:

1 If $\rho_G \in g(N_L)$ then $D \succ k(K)$.
2 If $\rho_G \in d(N_D)$ then $L \succ i(K)$.

Proof.
If $\rho_G \in g(N_L)$ then, for each $u \in N_D - k(N_K)$ we have $d(u) \succ_G^+ \rho_G$. By Lemma 5.15, we have that there exists at least one node $v \in N_K$ such that $dk(v)$ is on that path (K separates L from D). Let us choose v such that $dk(v)$ is as near as possible to $d(u)$. But then we have found a path $u \succ_D^+ k(v)$ all in D. This proves that if $\rho_G \in g(N_L)$ then $D \succ k(K)$. In a similar way, we can prove that if $\rho_G \in d(N_D)$ then $L \succ i(K)$.

□