An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation

Giorgio Busatto

An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation

von Giorgio Busatto

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

Vorgelegt im Fachbereich 17 (Informatik)
der Universität Paderborn
im Januar 2002
Datum des Promotionskolloquiums: 21. Juni 2002

Gutachter: Prof. Dr. Gregor Engels
Prof. Dr. Hans-Jörg Kreowski
Prof. Dr. Wilhelm Schäfer

Diese Arbeit ist als Nr. 3/02 in der Reihe "Berichte aus dem Fachbereich Informatik" (ISSN 0946-2910) des Fachbereichs Informatik der Carl von Ossietzky Universität Oldenburg erschienen.

Abstract

Hierarchical graphs are an extension of ordinary graphs used to describe structural information—like grouping and encapsulation—that cannot be modeled "naturally" by means of flat graphs. Motivated by different application areas, many approaches to hierarchical graphs and their transformation exist in the literature. These share common concepts but provide different notions, and often deal with unessential, application-specific aspects.

In the present thesis, we propose an abstract model of hierarchical graphs and their transformation that tries to capture these common concepts, and to distinguish them from other aspects. This should help to better understand and to compare existing approaches, and allow to define new notions of hierarchical graphs. Rather than a specific approach, we propose a generic framework that must be instantiated in order to define concrete models.

The framework supports hierarchical graphs with many different kinds of underlying graphs (e.g. attributed, labelled, directed, undirected, hypergraphs), different kinds of hierarchies (tree-like, dag like, arbitrary), and provides typing and encapsulation. Operations are specified in a rule-based fashion: Hierarchical graph transformation rules are a combination of existing flat graph transformation rules. In this respect, our model can be considered an extension of the well-known notion of graph transformation.

We have instantiated our framework using a particular graph transformation approach—the double-pushout—and we have investigated the properties of the resulting model. On the practical side, we have applied this model for specifying consistent operations on hypermedia structures in a concrete case study.

By comparing it to other known approaches in the graph transformation field, we can show that our framework covers the static aspects of all the considered notions, and can simulate the behaviour of several of them. A wider investigation of the dynamic aspects of related approaches, especially of those outside the graph-grammar field, is an interesting topic for future research. Other directions for future research include a more powerful transformation mechanism, encapsulated hierarchical graph transformation, and further investigation on the applicability of our model in the hypermedia or in other practical fields.

Acknowledgements

In this small space I would like to remember many persons that in different—sometimes indirect—ways contributed to a successful completion of the present thesis.

First of all I wish to thank Gregor Engels for giving me the opportunity to write this thesis, and for constantly following and supporting my work. His supervision was essential for educating my skills as a researcher, for motivating me, and for completing this work successfully.

I also wish to thank Hans-Jörg Kreowski: it was a pleasure and a stimulating experience to work with him during my stay in Bremen. The friendly and familiar atmosphere in his research group offered me an excellent working environment.

I also would like to acknowledge the contribution of the European research network *GEneral Theory of GRAph Transformation Systems* (GETGRATS) and its coordinator Andrea Corradini for supporting a considerable part of the research presented in this thesis.

A special thank goes to Berthold Hoffmann for reading the preliminary versions of the thesis and for his constant support, and to Annegret Habel for her advice and help during the preparation for my defense. I also wish to remember my ex-colleagues and friends Дмитрий Чересиз, Katharina Mehner, Peter Knirsch, Pieter Jan 't Hoen, Ray Dassen, Renate Klempien-Hinricks, Sabine Kuske, Vincent Zweije. A special mention to Sebastian Maneth for our many full-fledged joint discussions.

Also, bedankt to Alessandro for introducing me to the most diverse languages and to bagna kouda, to Michela for her dear friendship and unforgotten support during my first period abroad, to Luigi, Sara and Karl for so many reasons that do not fit in this page, and to Francesco for always suggesting a new wind to fly.

Grazie to my parents Francesco and Rina for supporting my study and for encouraging me when I began my adventure abroad, and to my sister Laura for her constant affection. I also thank my grandparents Lodovico and Maria, to whose dear memory this work is dedicated, for teaching me to laugh and to see fun and paradox in life and in people.

Einen speziellen Dank an Наташа und Владик für ihr gutes Herz und liebste Freundschaft, das Erdbeben, und alles.

Finally, I wish to thank all the musicians that kept me going during these years: Bach, Elio e le Storie Tese, Josquin, Monteverdi, Mozart, Palestrina, The Cranberries, Vivaldi, and many others.

Contents

Chapter 1

Introduction

Motivation

Graphs are used very often in mathematics and in computer science, indeed every time we describe a system as a collection of entities of some kind, linked to each other in some way. Graphs received an increasing attention by mathematicians, and graph theory—whose origins can be dated back to the works of Euler (1707–1783)—had a remarkable development over the last three centuries. (For a brief overview on the history of graph theory, the reader can refer to [Har69, Chapter 1], from which the following example is taken.)

Euler's solution to a famous puzzle of his time—the problem of the bridges of Königsberg—illustrates the usefulness of this concept. This problem can be formulated as follows: "How can one walk across the seven bridges that connect the two islands of the Pregel River in Königsberg to each other and to the banks, and return to the starting point in such a way that each bridge is traversed exactly once?" (See Figure 1.1.) Euler formalized this problem by representing the banks and the islands as vertices and the bridges as edges of a graph (see Figure 1.2) and generalized it to the problem of finding a cycle where every edge is traversed exactly once (later called a *Eulerian cycle*). He then provided a criterion to determine whether a graph contains such a cycle, showing that in the particular case of the Königsberg bridges problem there is no solution.

In computer science, graph-like structures are often used for different modeling purposes. Examples are networks of objects and links in object orientation, database conceptual models, and networks of hypermedia documents with hyperlinks between them. Much of the power of graphs lies in the fact that they combine a visual, intuitive representation of a problem with a formal, precise description. Different applications have led to several variations to the basic notion of a graph, e.g. directed/undirected, unlabelled or node/edge-labelled graphs, hypergraphs, and so on. Graphs are therefore a very flexible modeling concept.

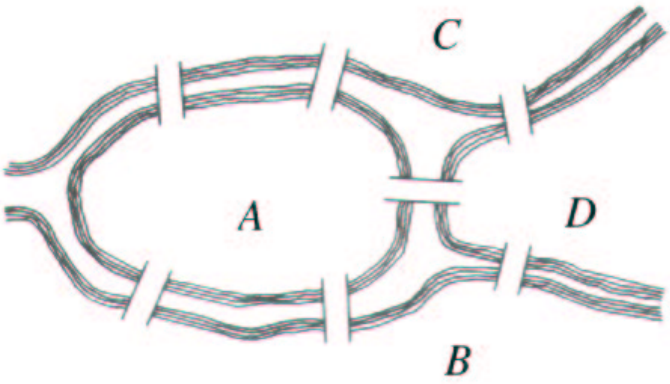


Figure 1.1: The seven bridges of Königsberg.

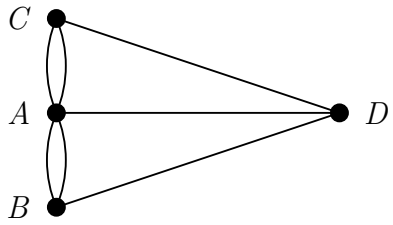


Figure 1.2: The seven bridges graph.

In spite of this, graphs tend to loose their intuitiveness and understandability when they become too large. For example, a graph representing even a small portion of the world-wide web—the well-known hypermedia system—with thousands of nodes and edges, is far from being an intuitive model. This is due to the fact that human beings can only deal with a small number of concepts or ideas at the same time. We usually tackle such situations by splitting a complex problem into smaller, more manageable units—the well-known principle of "divide et impera." Thus, we can make a graph more understandable if we add more structure to it, identifying meaningful groupings of nodes and edges that for some reason belong together, and representing such groupings as units. In addition, these groupings can be organized into a coherent, layered structure. These are the main ingredients underlying the notion of a *hierarchical graph*.

Hierarchical graphs are often used—explicitly or implicitly—in computer science, e.g. to model data structures, software system architectures, database structures, and so on. Like there are many notions of graphs, there are many notions of hierarchical graphs. First of all, the graphs that we want to structure can be of different kinds,

as mentioned above. Secondly, the structure that we add can be of different kinds as well: A hierarchy can be a tree, an acyclic graph, or even an arbitrary graph; graph elements can belong to just one component of the hierarchy, or be shared by different components, and so on. Thirdly, hierarchies can be defined in several ways: one can decide that a grouping of nodes and edges is in turn a node, or that it is an edge, or that it is some new type of entity (let us call it a *package*), distinguished from nodes and edges.

Thus, although the idea of a hierarchical graph is used in several ways and in many applications, there is no common established notion, and many approaches rely on some ad-hoc concept. This means duplication of effort, since many similar concepts are defined over and over again, and produces confusion when one tries to relate and compare different notions of hierarchical graphs.

Graphs have been often combined with rules, leading to the notion of a graph transformation system: Rule-based graph transformation allows to model complex structures and their manipulations in a declarative way. The area of graph transformation has developed in the last thirty years, and currently provides a wide theoretical background and many practical applications. (See e.g. [Roz97], [RE+99a], [RE+99b].) In this thesis, we consider rule-based hierarchical graph transformation, which can naturally be seen as an extension of (flat) graph transformation (see e.g. [Pra79], [PP95] [BEMW00], [EH00], [DHP02]). Both in the flat and in the hierarchical case, graph transformation has been used in a variety of approaches, supporting different (hierarchical) graph models and transformation styles. Even though these approaches are based on some common underlying ideas, no general model exists, resulting in the drawbacks that were mentioned for the static case.

As a consequence of these observations, we believe that an abstract model of hierarchical graphs and hierarchical graph transformation is needed. This model should help to identify the fundamental ideas underlying this concept, and serve as a basis both for comparing existing approaches to hierarchical graphs and for defining new ones. The development of such a model is the topic of this thesis.

Aims

As already said, we believe that an abstract model of hierarchical graphs and hierarchical graph transformation is needed. We now discuss the extension and the purpose of such a model.

First of all, this model should identify common "essential" aspects of hierarchical graphs, and distinguish them clearly from aspects that are specific to a particular approach or application. For example, the fact that there should be a hierarchy is a general feature, while restrictions to the hierarchy structure (tree-like, dag-like hierarchy, and so on) are approach-dependent.

The general aspects of hierarchical graphs identified in our model should also serve as dimensions according to which we can compare existing approaches and their expressive power. For example, we can classify existing approaches according to the kinds of hierarchy structures they offer.

On the other hand, these dimensions can help us in the definition of new hierarchical graph models, i.e. help us to choose the appropriate features (kind of hierarchy, kind of graph, and so on) and guide us to combine them correctly. As a special case, an abstract model should allow to define concrete models that have the same power as other existing approaches. This is a natural requirement for a model that tries to capture "essential aspects" of hierarchical graphs.

A last, but important aim of our work, is to show the usability of our model in a practical application. In this case, we want to show that we can use our abstract notion to define a concrete hierarchical graph approach that is an appropriate model for a specific application domain.

Conceptual Organization

The research presented in this thesis can be grouped into the following four themes: *static aspects* of hierarchical graphs, *dynamic aspects* of hierarchical graphs, comparison of *different hierarchical graph models*, an *example application*. We consider these themes one by one.

Static aspects By studying and comparing existing approaches to hierarchical graphs, one observes that different kinds of hierarchical structuring are applied to different kinds of graphs, and that the two concepts seem rather independent (orthogonal) from each other. Thus, an abstract model should support hierarchical structuring of different kinds of graphs, and allow to choose freely among different kinds of hierarchies.

Two further aspects of hierarchical graphs are encapsulation, i.e. hiding graph items inside the hierarchy, and typing. These will also be included in our model, since they occur in several specific approaches.

A last issue is to devise a suitable representation for hierarchical graphs, that captures all the above-mentioned aspects. Rather than providing *one* model of hierarchical graphs that includes all these aspects, we want to develop a general framework that indicates how to combine the desired aspects for producing a particular hierarchical graph model.

Dynamic aspects As far as operations are concerned, we consider rule-based hierarchical graph transformation. As for the static part, we do not want to develop a particular model of transformation using a particular model of hierarchical graphs,

rather, we aim at a framework that allows to specify transformations on different kinds of hierarchical graphs, using different approaches to (flat) graph transformation.

The issue of using different approaches to graph transformation deserves some further comments. Hierarchical graphs are graph-like structures, hence the idea of applying graph transformation to them seems very natural. Moreover, an abstract model should not be tied to a particular approach to graph transformation: different approaches are suited for different applications, and there does not exist one that encompasses all the others. Therefore, our framework will only state minimal requirements on graph transformation approaches and indicate how they can be used in order to obtain a hierarchical graph transformation approach.

Different hierarchical graph models As already sketched in the previous paragraphs, our framework should be used to define specific models of hierarchical graphs and hierarchical graph transformation. This mechanism can be used both for defining new hierarchical graph transformation approaches and for simulating and comparing existing ones. The basic idea is to instantiate our framework appropriately so that we can mimic existing approaches to hierarchical graph transformation.

Applications A last important subject of investigation concerns practical applications of our model. The idea is that the features of hierarchical graphs—which we have classified and combined in one framework—also occur in practical applications, and the validity of our hierarchical graph concept can be illustrated by modeling an appropriate application.

In this respect, the area of hypermedia provides many interesting ingredients, namely a graph-like basic model, the current trend towards hierarchically structured hypermedia models, and the strong need for specification techniques that support consistent transformations of hypertexts.

Our approach will consist in showing that we can instantiate our framework and produce a hierarchical graph model that is appropriate for representing hypermedia structures and operations in a specific industrial product.

Organization of the Chapters

This thesis is organized as follows. Chapters 2 and 3 introduce the static aspects of our hierarchical graph model, the former chapter dealing with motivation and informal introduction of the material that is then formally defined in the latter chapter. Following the same structure, Chapter 4 introduces graph transformation, motivates and describes informally our model of hierarchical graph transformation, while Chapter 5 describes the model formally. Chapter 6 investigates an instance of this model, using the double-pushout approach to graph transformation, while Chapter 7 studies how other existing

approaches to hierarchical graph transformation can be simulated in our approach. Chapter 8, illustrates an application of our model to the specification of hypermedia systems. Finally, Chapter 9 contains conclusions and outlines possible future work.

This structure is illustrated by the hierarchical graph of Figure 1.3, where an arrow indicates that the source chapter should be read before the target chapter.

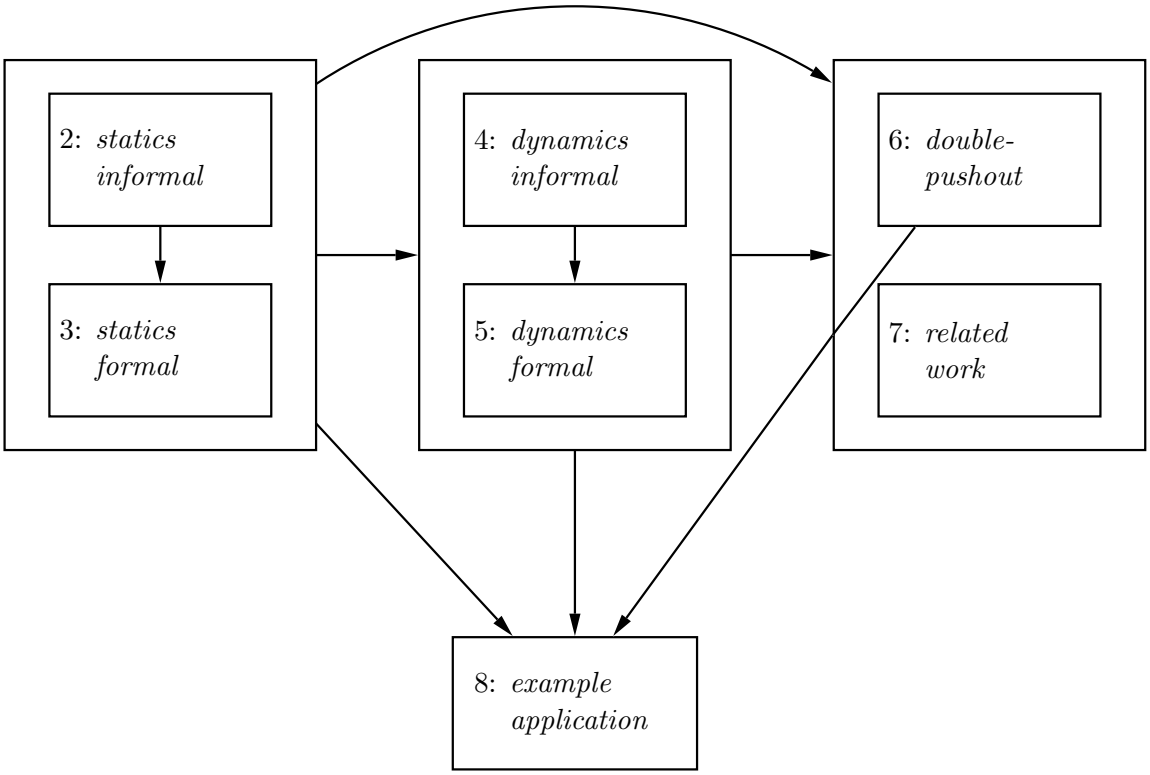


Figure 1.3: Organization of the chapters.

Chapter 2

Aspects of Hierarchical Graphs

Graphs are a very natural and intuitive approach for modeling realistic problems and have been successfully used in different fields of computer science to model, for example, database structures in information systems, data structures in software specification and in programming language semantics, and hypertext structures in hypermedia systems. Graphs provide a visual, intuitive but at the same time precise representation.

A graph consists basically of a set of nodes and a set of edges, the nodes representing entities of some kind, and the edges representing relations between them. A natural example of graph modeling is the World-Wide Web (WWW), the well-known distributed information system, in which documents (called *pages*) contain the actual information, and references (called *hyperlinks* or simply *links*) represent navigation paths between pages. It is natural to model pages as nodes of a graph, and hyperlinks as edges, thus visualizing the structure of the WWW.

Graphs, however, are difficult to manage and to understand if they contain too many nodes or edges. For example, a graph-like presentation of a large information system like the web is of little use when trying to find some specific page, and it often leads to the well known "lost in hyperspace" problem. In such situations, it is a common solution to adopt a "divide and conquer" approach: A graph is divided into smaller, more manageable components. If this decomposition is not sufficient, the same process is applied to the components as many times as needed. The result is a tree-like or dag-like (if sharing of subcomponents is allowed) decomposition of a graph, which we call a *hierarchical graph* (HG for short).

If we look again at the web, we notice that this way of organizing information is commonly used. In fact, large collections of pages are grouped into *web sites*. The pages of a web site can be further grouped into more fine grained subcollections, e.g. according to their content, or their owner. This kind of structuring is not reflected by the underlying flat data model of the WWW, but more recent hypermedia systems integrate similar ideas into their hypermedia model. (See, for example, the HyperWave system described in Chapter 8.)

The advantage of using a hierarchical structuring of information is that we can have different views of the data, according to the point in the hierarchy from which we look at it. For example, knowing that there is a web site about a certain topic helps us to abstract from the actual pages contained in that site, and concentrate on the overall content of the site. We will consider those pages individually only when visiting that site, i.e. when we decide to descend one level in the hierarchy.

In some cases the concept of hierarchical structuring needs additional features: If a graph is decomposed in a hierarchical fashion, it is, in general, not advisable to give direct access to components that are found in the inner levels of the hierarchy. For example, on the web not all the pages contained in a certain site have the same importance for an external user: A homepage is likely to be interesting, and therefore it makes sense to have direct access to it, whereas a page dealing with a very specific topic will only be interesting once the user has decided to visit the site. These features, though not currently supported by the data model of the WWW, are often implemented via ad-hoc solutions (homepages, "cookies", CGI scripts, etc). What is interesting for us, is that once we have a hierarchical decomposition of a graph, we often need to hide graph elements inside the hierarchy. This is the well-known idea of *encapsulation*, or *information hiding*, as found in programming languages (e.g. in Modula2, see [Wir85]), and software engineering (e.g. in UML, see [RJB99]).

We know numerous applications of hierarchical graphs, e.g. the definition of programming language semantics in [Pra79], the visualisation and design of hypertexts in [BRS92], graph visualisation in software systems (see e.g. [LB93], [Him94], and several commercial software or API's), and the representation of hypermedia document structures (see again our web example, and [Mau96]). We will analyze these approaches in more detail in Section 2.2.

In spite of the fact that hierarchical graphs are often used and there is an overall agreement on basic concepts like the use of hierarchies and of encapsulation, there does not exist a general approach to hierarchical graphs, nor any general-purpose hierarchical graph data model. Therefore, in different applications, ad-hoc notions of hierarchical graphs are introduced, and the term *hierarchical graph* is applied to data models that can differ a lot from each other.

Due to this variety, we feel the need for a standard hierarchical graph notion and data model, integrating the most important aspects of known approaches. This should both help to compare the features and expressive power of different approaches to hierarchical graphs, and provide a reference data model where the essential elements of hierarchical graphs are isolated from features that are specific to particular approaches. This notion should prevent from re-inventing the same concepts several times, and it could be a basis for future models. It should be (as much as possible) *general*, in the sense that it should allow to represent the information that can be modeled in other approaches to hierarchical graphs.

In this chapter, we give an overview of several representative approaches to hier-

archical graphs, and we identify and study the main aspects of hierarchical graphs, as they appear in these approaches. As a result, we can sketch an overall picture of our proposed hierarchical graph model, postponing its formal definition to Chapter 3. Operations on hierarchical graphs are considered in Chapter 4 and Chapter 5.

This chapter is organized as follows. In Section 2.1, we introduce a running example that will help us to illustrate our discussion on hierarchical graphs in Section 2.3, and in further chapters of the thesis. In Section 2.2, we give an overview of several approaches to hierarchical graphs, recalling the main references to the literature. In Section 2.3, we consider several aspects of hierarchical graphs and discuss how they should be dealt with in our hierarchical graph model. These aspects are related to the approaches considered in Section 2.2, and illustrated by means of the example introduced in Section 2.1.

2.1 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 uses graphs to model 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-holder called *anchors*. The WWW can be modeled 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 the 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 Freiburg, and that pages can be provided by either of the two sites.

Figure 2.1 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,

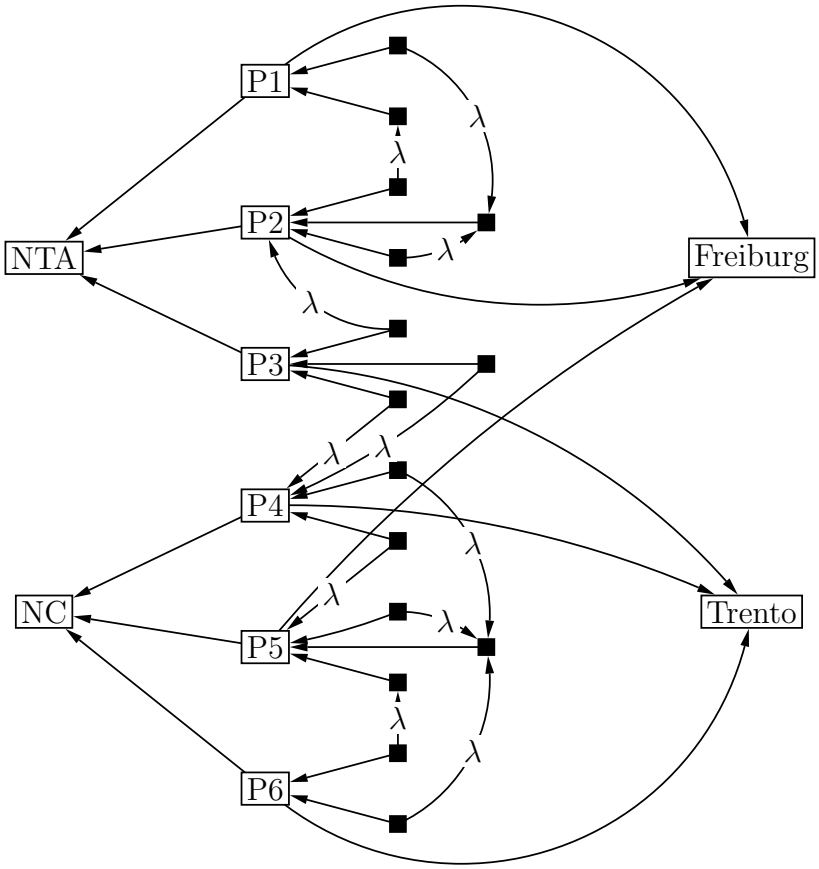


Figure 2.1: Web sites, projects, pages, and anchors.

and of little use for, say, an author who has to develop and maintain the projects' documentation.

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 homepage, 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 homepage is the most reliable page to which one can refer, as it is left to the maintainer of the homepage 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 those 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 imagine that page `P4` is the start page of project NC, and therefore all other pages in the project should be hidden.

Hierarchical graphs are an extension of ordinary graphs, where the grouping of graph elements into higher-level, layered structures is modeled explicitly. In the WWW scenario, this corresponds to making the distinction between *structural* and *referential links* explicit (see Chapter 8). The idea of encapsulation is also present in some hierarchical graph approaches, although less frequently. In this case we speak of *encapsulated hierarchical graphs*.

In Figure 2.2, we show a possible hierarchical decomposition for our running example. We have imagined that 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, we can also imagine that nodes are hidden or visible w.r.t. a given grouping. For example, we 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.

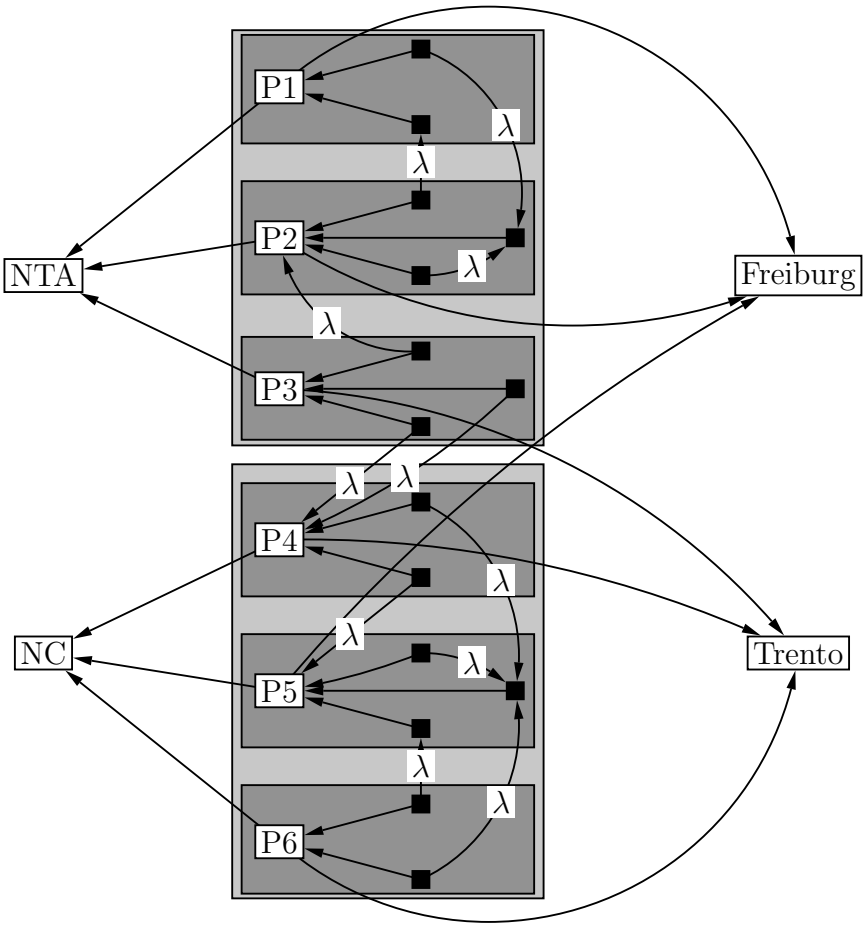


Figure 2.2: Web sites, projects, pages, and anchors.

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 (hierarchies are implemented using structural links) but more recent hypertext models do support such higher level structuring (see e.g. *collections* in [Mau96]).

Our running example already gives us a first rough impression of the most important ideas of hierarchical graphs. They can be summarized 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.

2.2 Approaches to Hierarchical Graphs

In this section we leave aside our running example for a moment (it will be used again in Section 2.3) and we look at the several existing approaches to hierarchical graphs. In the literature and in practical applications, the concept of a hierarchical graph occurs pretty frequently. In this work, we have considered approaches from different areas of computer science, choosing the most interesting ones w.r.t. the features they provide and/or the problems they raise.

In some of the considered approaches, the term *hierarchical graph* is not used explicitly, but these approaches are interesting because they use the main idea of hierarchical graphs, namely that of structuring some graph-like piece of information. An example of this kind is object-oriented modeling (see e.g. [RBP+91]), where *objects* can easily be thought of as nodes, *links* to edges, and *aggregation* can be used to build hierarchical structures of objects (see 2.2.3 further on in this chapter).

In this more general setting, the "important ideas" of hierarchical graphs, identified at the end of Section 2.1, can be formulated as follows:

1. *Grouping.* Objects/entities from a certain domain can be grouped together to form higher level structures. Groupings often model abstractions in the considered domain.

2. *Encapsulation/visibility.* Objects/entities can be hidden inside the hierarchy. Encapsulation models the fact that an abstraction can publish some of their components and keep others private.

3. *Constraints.* The hierarchical structure and the visibility information can introduce constraints on the underlying data, for example by restricting certain relationships between entities of the considered domain (like in our web example).

The approaches considered in this section are evaluated in view of these "ideas", and the results of this evaluation are summarized in Subsection 2.2.7.

In this section, we give an overview of the approaches we are interested in, considering mainly the relevant literature and highlighting their most important features. A more in-depth analysis is postponed to the next section, where *aspects* of hierarchical graphs are considered. This section contains one subsection for each considered approach. At the end of each subsection we compare the considered approaches with the ideas described at the end of Section 2.1.

2.2.1 Graph drawing

The first area we consider is that of graph drawing (see e.g. the proceedings of the latest symposia on graph drawing [TT94], [Bra96], [Whi98], [Kra99]). Graphs are often used as a visual model of information about a certain domain, and graph layout has a primary importance in helping one to understand the information contained in the graph itself. We have found several ways in which the notion of hierarchical graph occurs in graph drawing.

The first idea is that, in order to obtain a better layout, a graph can be hierarchically partitioned into several components, all of which are mapped to corresponding non-overlapping areas in the final layout, and then each component is laid out separately. This idea is used, for example, in [BBH94], and in [Ber95]. In this case, hierarchical structuring is one of the ingredients of the layout algorithm.

The second idea is that the visualized information is in some sense hierarchical—for example if we visualize the structure of an organization—and the layout algorithm should make such hierarchy visible. In this case, the layout algorithm can arrange the nodes in several (horizontal) levels representing the layers of the hierarchy, as it is done, for example, in the hierarchical layout presented in [MMPH96].

A third possibility is that a hierarchically nested graph—i.e. a graph with nodes or edges containing in turn graphs—needs to be drawn. For example, the AGG system (see [LB93], the reader should not confuse this system with the new AGG system, see [ERT99]) can be used to visualize graph transformation rules (see 2.2.6). This piece of software allows to abstract entire graphs to single nodes, and bundles of parallel edges to single edges. The mentioned graph visualisation software (see [MMPH96]), provides a similar structuring mechanism, where it is possible to define abstractions, namely nodes that can be *unfolded* to a nested graph (the nested graph is then drawn inside the containing node), or edges that can be followed in order to *navigate* to a nested graphs (the nested graph is then drawn in a different window).

In all the above mentioned cases, grouping of graph elements is the basic mechanism to add further structure to a graph. In the third case—layout of hierarchical graphs—the folding/unfolding mechanism provides a kind of encapsulation.

2.2.2 Hypermedia models

Hypermedia has become very popular in the recent years thanks to the fast development of the World-Wide Web. We briefly discuss hypermedia here, postponing a more detail discussion until Chapter 8, where a case study is considered.

The term *hypermedia* combines the terms *hypertext*, indicating a collection of documents with references between them, which is navigable by means of appropriate browser software, and *multimedia*, indicating a document that, besides text, can contain images, sound, movies, and so on. Thus a hypermedia application makes use of

a collection of multimedia documents (*nodes*), with navigable cross-references (*links*) between them (see e.g. [MSHR98] for a thorough discussion on the topic). We call such collections *hyperwebs* or *hypermedia networks*.

We distinguish between a *hypermedia application*, i.e. a piece of software that makes heavy use of hypermedia, and a *hypermedia system*, i.e. a software system that helps develop and/or maintain a hypermedia application (see e.g. [ISB95]).

It is clear that a hyperweb is essentially a graph, and in fact many modeling concepts for both hypermedia applications and hypermedia systems use some kind of underlying graph model. What is interesting for us, is that flat graph models, like that of the WWW, are often insufficient for managing complex webs: both authors and users browsing the web tend to *get lost in hyperspace* if the web is too large and poorly structured.

A solution to the "lost in hyperspace" problem, very often adopted on the web, is to distinguish between *organizational* (*hierarchical*) and *referential* (*cross-reference*) links. In this case a hypertext is given a hierarchical structure, which is coded in the hypertext itself by means of organizational links, i.e. links between nodes and their super or subnodes. In [BRS92], methods for automatically generating *good* hypertext hierarchies are proposed.

Some hypermedia systems go even further, by extending the basic node/link model with some kind of higher-level structuring, representing organizational links. For example, in HyperWave (see e.g. [Mau96]) higher level structuring mechanisms like *containers* are proposed. Containers can contain documents or, in turn, other containers. Besides grouping, containers provide encapsulation through the concept of *scope*. Higher level structuring also introduces constraints in the hyperweb, by only allowing certain link structures that are consistent with the container structure.

A concept similar to containers can be found in OOHDM (see e.g. [SR95], [SRB96], [SB94]), a hypermedia application design methodology, where *navigation contexts* are proposed as an essential tool for structuring the navigation space (i.e. the hyperweb) of an application.

From the above observations, it is clear that hypermedia is for us an interesting application area where graph-like models are frequently used, often in combination with higher-order structuring.

2.2.3 Object-oriented modeling

Object-oriented modeling is presently one of the most widely used paradigms in software modeling and design. Nowadays there are several approaches to object orientation (e.g. [RBP+91], [Boo93], [KA90], [JBR98]), and several programming languages supporting (at least some) of the concepts of object orientation, e.g. Smalltalk [GR83], C++ [Str91], Java [GJS96], [AG96], Eiffel [Mey92], Objective C [NS95], and others.

A recent development in object orientation is the effort around the Unified Modeling Language (UML, see e.g. [RJB99]), which tries to integrate the most important approaches to object-oriented modeling into a common framework and language. UML is becoming a "de facto" standard in object orientation, and therefore we will often refer to it in our discussion.

Among the basic assumptions of object orientation is the idea that our domain of interest (be it a real world situation, a software system, etc) is made of entities called *objects*. Relations between objects are modeled through the concept of a *link*.

For our discussion, it is interesting to point out that:

1. Networks of objects and links can obviously be modeled as nodes and edges of a graph, i.e. some (fundamental) aspects of object-oriented models are easily modeled as graphs.

2. Object-oriented models often add further (possibly hierarchical) structuring mechanisms on top of the object/link network. Examples of this are the notion of *aggregation*, and the higher level construct of a *package* in UML.

All three ideas of grouping, encapsulation, and additional constraints can be found in UML packages. Therefore, object-orientation is an important application area of hierarchical graphs, and we must take it into account if we want to develop a general approach to hierarchical graphs. We will discuss object-orientation in more detail later in this chapter (see 2.3.3).

2.2.4 Programming languages

In programming languages, structuring plays an important role in making program code more understandable, maintainable, and re-usable. Structuring can be found at the level of instructions (*procedures* in Pascal and Modula2, *functions* in C, etc), or at the level of variable, type and procedure declaration (e.g. *modules* in Modula2, *units* in some commercial versions of Pascal, *packages* in Ada).

Of particular interest for us is the fact that most of these structuring constructs are hierarchical, which seems to be, once more, a very natural way to structure complex objects.

Another interesting aspect of structuring in programming languages is the concept of *encapsulation* or *information hiding*. This is used to distinguish the *exported elements* of a piece of code (often called the *interface*), from the internal elements, which are only needed for implementing the functionality provided by that piece of code. An obvious advantage of this is that we can change the internal parts of a piece of code, without changing the way in which this code is used.

Although the structuring constructs of programming languages are usually not applied to graphs, it is interesting for us to consider the kind of structuring that they provide, and to see to what extent these ideas can be applied to graphs.

2.2.5 Databases

Database modeling languages have some common aspects with objects oriented modeling languages. In particular, in conceptual database modeling languages, like the *entity-relationship* model (ER, see e.g. [Che76], or any text book on databases) a domain of interest is modeled using *entities* and *relationships* between them.

Entities can be easily seen as objects (although entities are not required to have a behaviour) and relationships as links. In fact, object-oriented modeling techniques are nowadays often used for modeling databases too. Put in other words, database structures can also be seen as graphs, and the considerations relating object orientation and hierarchical graphs can be easily transfered to databases.

However, databases provide another interesting concept for us, i.e. the concept of a *view*. This concept is very useful when we want to provide different users perspectives from which they can look at a database, while keeping a unique underlying model.

Hierarchical structures can also be thought of in terms of views. Namely, we can imagine that we have a large piece of information (program code, object-oriented model, graph, etc) and that we define several points of view on it, that allow to *use* the same underlying object in different ways. Hierarchies can be obtained by composing views. In our running example of Section 2.1, we can define a view `NodesOfNTA`, which contains pages `P1`, `P2`, `P3` and their anchors. This view can be refined by composing it with, say, a view `NodesOfP1`, thus obtaining only the node `P1` and its anchors.

Therefore, the ideas of grouping (collecting) and hiding entities are somehow supported in database through the concept of a view. If a database is designed in a top-down fashion, then its views are defined over a conceptual model of the underlying database: In this case views do not impose any additional constraints on the underlying database. In a bottom-up design, where views are defined first, and the conceptual schema is obtained by integrating them, views do pose constraints on the underlying system.

2.2.6 Graph transformation approaches

The graph transformation area has developed over the last thirty years, and provides a wide theoretical background together with several practical applications. It has been applied to the description of graph properties, to the description of plant growing, to picture generation, and to the specification of data structures and operations in software systems, to mention a few examples.

Graph transformation extends the notion of string rewriting, where a substring is replaced with a new substring in a given *host* string. In graph transformation a subgraph is replaced with a new subgraph within a host graph. Such graph transformations are specified through so-called *graph transformation rules* (*graph rewrite rules*, *productions*).

Since there exist several kinds of graphs, and several ways of defining graph transfor-

mation rules, a wide variety of graph transformation approaches have been considered in the literature. For an introduction to the theory of graph transformation and to the related literature, the reader can refer to [Roz97], [RE+99a], and [RE+99b].

Most approaches to graph transformation use some notion of *flat* graph[1], but, in the recent years, the need for structured graphs has emerged, leading to several hierarchical graph models. Here, we briefly summarize the most representative among them.

Programming language semantics One of the earliest approaches, used for the definition of programming language semantics, can be found in [Pra79]. In this research, hierarchical graphs are used for modeling the *internal state* in the implementation of programming languages.

The generation of the initial hierarchical graph, which models the initial state of an execution machine, is performed by a *pair grammar*, which allows to translate the syntax of some programming language construct into its hierarchical graph representation. Operations on hierarchical graphs model the changes in the state during program execution. In this way it is possible to define and study the behaviour of a program by looking at the changes in its internal state.

This approach provides only grouping of graph elements.

Distributed systems Another approach to structuring is represented by *distributed systems* (see e.g. [Tae96]). Here the hierarchical structure is limited to two levels, thus modeling networks of local graphs. It is possible to define transformations on the network and on the local graphs by means of transformation rules based on the double-pushout approach (see [Roz97]).

In this approach grouping is supported, as well as information hiding (elements from one local graph can only be "seen" by another graph only through *interfaces*), and constraints (hidden elements cannot be accessed).

Encapsulated hierarchical graphs Encapsulated hierarchical graphs (EHG) were proposed in [ES95], and were the starting point of our research on hierarchical graphs. This approach supports grouping, encapsulation, constraints on the graph derived from the hierarchical structure, and hierarchical graph typing. Its main shortcomings are that it does not support dag-like hierarchies nor any kind of operations.

Hierarchical graph transformation In [DHP02], we find a proposal for hierarchical graph transformation, based on the categorical approach to graph transformation (see [CMR+97], [EHK+97]).

This approach supports hierarchical groupings of graph elements, it supports encapsulation by forbidding sharing of nodes or edges among different elements of the

[1] See Section 2.3.2 for a discussion on the notion of a flat graph.

hierarchy. It also forbids boundary-crossing edges, i.e. edges spanning the boundary between hierarchy components.

Hierarchical graphs for system modeling and model evolution In [EH00], Engels and Heckel propose an approach for modeling software systems based on a hierarchical graph model. This model, based on the double-pushout approach, supports aggregation, 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.

Other approaches The AGG system, which we have mentioned in Subsection 2.2.1 among graph visualisation approaches, belongs also to the area of graph grammars.

In [Him94], a model of hierarchical graphs is presented, which provides a data structure used in the Graph^{Ed} system (see [Him93]), where the hierarchy is built from a given graph through applications of replacement steps. In this approach, both nodes and edges can be expanded to a nested graph.

2.2.7 Summary

In this section, we have provided an overview on the literature of hierarchical graphs that is the basis for the definition of our hierarchical graph model. We have dedicated particular attention to the three ideas that we had introduced in Section 2.1 and formulated in a more abstract sense at the beginning of this section, namely *grouping*, *encapsulation*, and *constraints*.

Approach	*Grouping*	*Encapsulation*	*Constraints*	*Total*
Graph drawing	5	2	0	5
Hypermedia	3	2	2	3
OO modeling	2	1	1	2
Prog. languages	1	1	1	1
Databases	2	0	1	2
Graph grammars	5	3	4	5
Totals	18	9	9	18

Table 2.1: Approaches to hierarchical graphs and supported concepts.

In this respect, the information presented in this section is summarized in Table 2.1 where, for each category of approaches, we indicate the number of approaches that provides grouping, encapsulation, or some sort of constraints derived from them. In the

last column we report the total number of approaches in that category. This table thus gives a rough measure of the importance of these three notions within the considered approaches.

These figures support our claim that grouping is the most important idea behind hierarchical structuring (it is present in all approaches), while encapsulation and additional constraints play a secondary role.

2.3 Aspects of Hierarchical Graphs

In this section we want to explain the term *generic* hierarchical graph data model, mentioned at the end of the introduction to this chapter. In fact, in Section 2.2 we have seen that in the literature and in practical applications we encounter several approaches to hierarchical graphs, with different features or technical solutions. It is therefore necessary to decide which features offered by these approaches we should support in our model, always keeping in mind that we are looking for a general solution.

While we postpone the formal definition of our hierarchical graph data model to Chapter 3, here we want to consider such features (or aspects) one by one, and compare them to the approaches considered in Section 2.2.

This section has the following structure. We start by considering, in Subsection 2.3.1, the different notions of graphs that are used in the literature, and that can therefore be provided with hierarchical structuring. We then define an abstract notion of graph, to be used in the definition of our hierarchical graph model. A second issue, discussed in Subsection 2.3.2, is to compare the idea of a hierarchical structuring with the idea of typing: The two concepts, which apparently look related, serve indeed to model different aspects of data. In Subsection 2.3.3, we introduce *graph packages*, which are basic containers of graph elements, around which one can build a hierarchical structure. In Subsection 2.3.4, we evaluate the alternative choices of coding the hierarchy in the graph or of keeping it separate from the graph as a disjoint structure. In Subsection 2.3.5, we discuss how graph elements are distributed over the packages of a hierarchy, and whether they can be shared between different packages. In Subsection 2.3.6, we discuss boundary-crossing edges, i.e. edges that connect nodes from different packages. In Subsection 2.3.7, we discuss in which cases a hierarchy should allow sharing of subcomponents of the hierarchy itself, and in which cases it should be a strict tree-like hierarchy. In Subsection 2.3.8, encapsulation is considered, i.e. the possibility to hide graph elements inside a hierarchy. In Subsection 2.3.9, we consider typing, i.e. the possibility to define templates for hierarchies, and how this is related with graph typing. Finally, in Subsection 2.3.10, we consider *anchoring*, a means to associate packages to graph elements.

2.3.1 Different notions of graphs

The first problem, if we want to define a hierarchical graph data model, is that the very notion of a graph is not something fixed, but many different kinds of graphs are used in the literature and in applications, e.g. directed, undirected, node or edge labelled graphs, hypergraphs, and so on. To begin with, we review the most common notions of a graph. In this way, the reader can have an idea of the variety of features that can be associated to graphs, which motivates the need for an abstract notion of a graph. This abstract notion, which we will call a *graph skeleton*, should contain only those features that can interact with the hierarchy, and forget about the remaining ones.

Finite versus infinite graphs A finite graph is a graph which has a finite set of nodes and a finite set of edges. Finite graphs are sufficient for the application areas we are interested in, like object-oriented modeling, hypermedia modeling or the specification of data structures, where infinite graphs are never or hardly ever of any use.

Infinite graphs can be useful in certain applications, especially in theoretical computer science (for example in transition systems, see e.g. [Kel76]) but they are outside the scope of our research.

Different kinds of edges In the literature we find several different kinds of graphs and, in particular, different kinds of edges. An edge can connect exactly two nodes (*binary edge*) or an arbitrary number of nodes (*hyperedges*), it can be *directed* or *undirected*. A graph containing hyperedges is called a *hypergraph*.

In our work we only consider the essential aspect that is common to all kinds of edges: They are incident in (a certain number of) nodes. As we will see in Subsection 2.3.6, this is the only information about edges that we model in our hierarchical graph notion.

Attributed graphs Attributes are used in a graph to model information that cannot be easily, appropriately modeled as nodes and edges. In our web example, the author of a page can be such information, possibly modeled as a string-valued attribute associated to each page node.

Node attributes in graphs have their counterpart in object orientation, where attributes are associated to objects and serve to model their properties. Attributes are more often associated to objects (nodes) than to links (edges), but see object-like links, for example in [RBP+91, page 33].

Labelled graphs It is often useful to associate labels to the nodes or edges of a graph. Labels can carry some additional information which cannot be coded in the node/edge structure of the graph. For example, in our web example, we could associate labels to

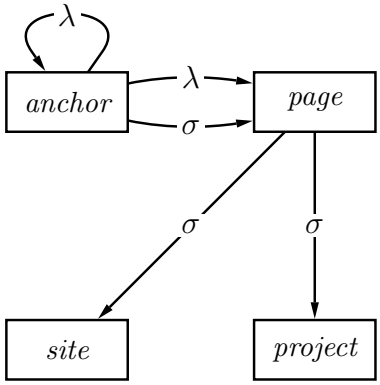


Figure 2.3: WWW type graph.

nodes, in order to distinguish project nodes from page nodes. We can then use this information to forbid, say, linking an anchor to a project.

Thus labeling can serve to provide a sort of primitive typing, in that nodes or edges with the same label are "of the same type". Usually labels are atomic entities taken from a finite set called *alphabet*. When more complex information than a plain tag needs to be associated to graph elements, attributes (see previous paragraph) are used instead of labels.

Typed graphs Here we introduce typing as one of the aspects of hierarchical graph. We will consider it again in Subsection 2.3.2, in Subsection 2.3.9, and in Section 3.4.

We first observe that, although labels can serve to classify nodes and edges and thus provide a kind of typing on them, in certain situations we need more. In our running example, it can be useful to distinguish between different kinds of nodes and edges: pages, anchors, sites, projects, structural links between pages and projects, hyperlinks, and so on. This can be achieved by means of labels, but we still need to enforce constraints like "a hyperlink can only connect an anchor to a page or to another anchor".

This kind of general constraints can be specified by means of graph types: A graph serves as the template (*type*) for a whole class of *instance* graphs. The relation between an instance and its type is specified by an appropriate homomorphism from the instance graph to the type graph.

In our running example, we define the type graph depicted in Figure 2.3. In this graph, four node types are defined, namely `project`, `site`, `anchor` and `page`. The type graph also defines four edge types, namely (`page`, `project`), (`page`, `site`), (`anchor`, `page`), and (`anchor`, `anchor`).

The edge types labelled with λ model hyperlink edges. The other edge types model

structural edges, which is indicated by the label σ. The homomorphism between the instance depicted in Figure 2.1 and the type graph, maps all page nodes to the `page` node, all project nodes to the `project` node, all hyperlink edges to the λ labelled edge, and so on.

After considering several aspects/dimensions that characterize different notions of graphs, we summarize our discussion by classifying, in Table 2.2, the approaches considered in Section 2.2 according to these dimensions. This table shows that the considered features appear in many different combinations, so that it is difficult to propose a notion of a graph that satisfies the needs of many different applications.

Approach	*Edge type*	*Node lab.*	*Edge lab.*	*Attrib.*	*Typing*
Graph visual.	bin/dir-und	YES	YES	NO	NO
Hyp. models	bin/dir	NO	NO	YES	YES
OO modeling	bin/dir	NO	NO	YES	YES
Databases	bin/und	NO	NO	YES	YES
Gr. grammars	ANY	YES	YES	YES	YES

Table 2.2: Kinds of graphs and approaches to hierarchical graphs.

In spite of this variety, we do not need all these concepts when considering hierarchical graphs from an abstract point of view. Hierarchical graphs structure the node/edge network of a graph and do not deal with other kinds of information that may be contained in it. The reason for this is that hierarchical structuring is a kind of topological structuring of a graph (it defines localities) and the basic topological information contained in a graph is provided by nodes and edges. As pointed out by Harel in [Har88], edges model relations between nodes while hierarchies (grouping) model collections of nodes (that belong together for some reason). (See also Section 9.2.)

This is reflected by the approaches found in the literature and in applications: Hierarchical structuring affects the collection of nodes and edges, regardless of additional information attached to them. Again, we want to point out that this "information attached" to nodes and edges, must be flat itself: If we allow node labels to contain graphs, then we are already providing a hierarchical structuring of the graph.

Typing may (only apparently) interact with hierarchical structuring, since typing and sub-typing of node elements can induce a hierarchical decomposition of a graph. In Subsection 2.3.2, we discuss why typing must be considered an orthogonal concept to hierarchical structuring.

The conclusion of this discussion is the following: A general formalism for hierarchically structuring graphs should not rely on a concrete kind of graph, since

- either it would have a restricted applicability, because it does not provide enough features,

- or it should contain all imaginable features, resulting in an unnecessarily overloaded model.

Instead, we propose a very simple model of a graph—called a *graph skeleton*—which will serve as an interface between the actual graph that we want to structure and the hierarchical structure. We will always assume that a mechanism exists to extract a skeleton from a given graph G. This skeleton will contain a set of nodes N_G, a set of edges E_G, together with some mechanism to link them together (this mechanism will be further discussed in Subsection 2.3.6). Only when considering specific approaches do we need to specify how nodes and edges are related to each other, and whether the graph contains additional information.

2.3.2 Grouping versus typing

An important issue that we want to consider in this subsection is the comparison between two kinds of hierarchies: grouping hierarchies and type hierarchies. This is also related to the question of what we mean with the term *flat graph*.

Typing, as known from many areas of computer science, is some kind of grouping: it allows to group together entities that have the same *type*. To have the same type means to share common properties, operations, behaviour, common relations to entities of other types, and so on. Thus a type groups together entities.

In our example, pages have the same type because they all contain some text, they all serve to document projects, they can all contain anchors, they all play a *similar role* in the model. As another example, numbers 2 and 3 have the same type because they both belong to the set of integers, and common operations—e.g. the successor operation—can be applied to them.

Typing can also induce hierarchies on a collection of entities through the mechanism of sub-typing. If a type T_1 is a sub-type of a type T_2, then an entity of type T_1 should have all the properties of an entity of type T_2 (and possibly more). In our example, we could define a `homepage` type, as a sub-type of `page`. Both `page` and `anchor` could have a super-type `navigation-element`. In this way, we can build a hierarchy of types.

In spite of this, we think that a type hierarchy has little or nothing to do with the kind of structuring that characterizes hierarchical graphs. To see this, suppose that we want to use typing to model the hierarchy in our running example. Since we have pages, and we want to build two groups of pages, we must create two sub-types, `NTA-page` and `NC-page`. We then want to create three sub-groups for each project, one for each page, and we have to put the right anchors into them. In order to do this, we must create six sub-types of `anchor`, one for each page.

We think that this solution—the only solution we could think of for mimicking our hierarchy using typing—is very artificial and difficult to use. For example, suppose that we want to change our graph and add a new page. As a consequence we should create

a new grouping to accommodate its anchors, i.e we should create a new sub-type of `anchor`. Thus, dynamic changes to the hierarchy would require dynamic changes to the type system.

From a conceptual point of view, the problem with this approach is that we are using typing in an artificial way, to model something else. Pages of different projects, even though they *lie* in different portions of the graphs, still have the same properties, and so do anchors in different pages. On the other hand, pages and anchors are of different types, but pages and anchors belonging to the documentation of the same project are in the same project grouping because they provide the same service.

We are now able to identify what are the key differences between these two kinds of grouping. On one hand we want to define hierarchical structures *in* a certain domain, i.e. to add more information to the domain itself, expressing that certain entities form higher level structures with certain properties. On the other hand we collect information *on* the domain and the entities that it contains, i.e. we deal with meta-information. In the first case hierarchical structures are something that we want to store, manipulate, query as part of the data, in the second case they are a (more or less static) description of the domain which we use to model the domain itself.

The kind of grouping found in hierarchical graphs is some kind of *topological*[2] grouping, which identifies *localities* (where objects are located in the data)—rather than *types*—in the graph by specifying that certain graph elements, for some reason, belong together. An example in object orientation, where the hierarchy represents information about localities *inside* the domain of interest can be found in [EH00], where hierarchies model software components in a software system.

Summarizing, given a domain of entities that we want to model, we identify the following kinds of grouping:

Typing Groupings of objects according to common structure/behaviour. The groupings are normally static and defined at design time. Types can be structured into hierarchies. This is meta-information about a certain domain of interest that can be used when modeling it.

Topological grouping Groupings provide information about the localities into which a graph is split. Localities can change dynamically if this is required by the modeled domain. Grouping/localities can be structured into hierarchies. This is additional information that is stored in the domain of interest and which is manipulated as part of the data that is contained in it.

From the above discussion we conclude that hierarchical structuring of a graph has little to do with the typing of its elements, and therefore it should be an orthogonal

[2]We use the term *topological* in an informal way. The idea is that elements of the same grouping are considered "near" to each other while elements of different groupings are "far". It is not our intention to interpret this term in a mathematical sense.

concept to typing. From now on we will call the notion of grouping found in hierarchical graph *topological grouping*. We will consider a graph without this kind of *grouping* as *flat*, even though it may have some type hierarchy defined on it.

2.3.3 Graph packages: aggregation and more

The notion of topological grouping, introduced in Subsection 2.3.2, is related to that of *aggregation*, as known from object-orientation (see Subsection 2.2.3). Aggregation is a special kind of association, which describes a part-of relationship between an aggregate class and its sub-components. Each component object must be part of exactly one aggregate object, whereas an aggregate object may contain a variable number of sub-objects. We speak of *strong aggregation* (*composition* in UML, see [RJB99]) when the deletion of a composite object implies the deletion of its components. In our running example, the links between pages and sites can be considered aggregation links, whereas links between anchors and pages can be considered strong aggregation links.

Aggregation provides some kind of grouping but, if we look again at the example, there can be groupings which are not modeled adequately as aggregations. For example, pages are grouped according to projects they document, but they are not sub-components 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.

Another problem with aggregation is that it always implies the existence of an aggregate object, whereas a *general-purpose* grouping mechanism, which is what we are aiming at, should not rely on the existence of a grouping element in the graph (object, node, edge, hyperedge, and so on), although this *can* be the case. As an example, in a UML model elements can be grouped using *packages*, but packages must not be confused with other elements (classes, associations, etc) of the model: they serve to structure it. Aggregation models only a particular case of topological grouping, namely one in which the localities are identified by aggregate object boundaries.

Since we aim at identifying *essential* elements of hierarchical graphs and at separating them from those that are specific to a specific approach, we prefer a UML-like solution, where this separation is reflected by the model. We will use a grouping primitive called a *graph package*, or simply a *package* (from time to time we will use the term *hierarchy component*), as a general-purpose container of graph elements (nodes and edges). This idea does not forbid that, in a particular approach, this structuring be tightly coupled with some elements of the graph: In our running example, it is natural that certain packages be associated to projects. In Subsection 2.3.10 we come back to this issue.

Summarizing, besides nodes and edges, a hierarchical graph in our model will have a set P of graph packages, in which nodes and edges are distributed. Packages are a more general concept than aggregation in object-orientation, and similar concepts in other approaches to hierarchical graphs. By "more general" we intend that while aggregation

is a grouping mechanism with a predefined conceptual interpretation, our packages do not assume any interpretation of this kind. In order to avoid similar situations, in our model the hierarchy is *decoupled* from the graph being structured. This idea is further developed and motivated in Subsection 2.3.4.

2.3.4 Coupling versus decoupling approaches

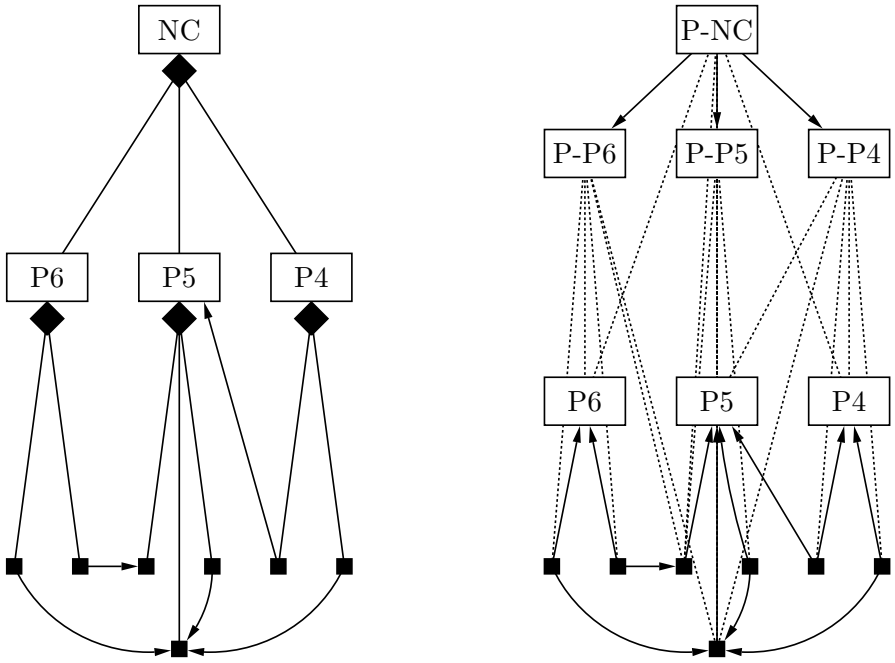
In Subsection 2.3.3, we have concluded that the hierarchical structure of a hierarchical graph should be *decoupled* from the underlying graph, by means of graph packages. In this subsection, we further motivate this approach, and compare it with so-called *coupled* approaches.

To begin with, we want to introduce some terminology. An approach to hierarchical graphs, where nodes, edges or both of them serve as a grouping primitive, is called a *coupling approach* (or *coupled approach*): In such an approach the hierarchical structure is coded in the graph itself and strictly integrated with it. For example, in [ES95] subgraphs of the overall underlying graph are contained in *complex nodes*, i.e. in nodes that have a graph as their internal content. A nesting relation between complex nodes allows to build a hierarchy. In [DHP02], an approach based on hypergraphs, special hyperedges called *frames* contain either a flat or a hierarchical graph, thus building a hierarchical structure. AGG (see [LB93]) uses a combination of these approaches, by nesting graphs inside both nodes and edges.

On the other hand, a *decoupling approach* (*decoupled approach*) uses a grouping primitive (packages in our case) distinct from nodes and edges: The hierarchy is a separate structure from the graph, since elements of the hierarchy are not elements of the graphs and vice-versa. UML models give such an example: packages can be used to structure, say, a class diagram containing classes and associations, but packages are neither classes nor associations.

In Figure 2.4, we compare an example of coupled hierarchical graph based on aggregation (left half of the figure) with an example of a decoupled approach (right half). In the coupled approach, the hierarchy relies on aggregate nodes, and a node belongs to the grouping of another node if it is linked to it by an aggregation link. In the decoupled approach, we use a *package* `P-NC` to group together the page nodes (note that the node `NC` can be omitted if it is not necessary to model a project in the graph), and three packages to model the groups of anchors internal to a page (note that we still keep page nodes in the graph). The nodes are assigned to the packages through dashed lines (see Subsection 2.3.5).

Aggregation is just one example of a coupled approach, and several other possibilities exist to construct a hierarchical structure based on the nodes and/or the edges of a graph. These approaches overload graph elements with the additional task of representing the hierarchical structure, and are therefore less suited for our model, where we want to identify and make explicit the *essential aspects* of hierarchical graphs.



a) Coupled representation. b) Decoupled representation.

Figure 2.4: Coupling versus decoupling.

This overloading makes a hierarchical graph model less appropriate as a general-purpose model: A general-purpose hierarchical graph model should allow an easy representation of different kinds of hierarchical graphs, and a straightforward translation from existing hierarchical graph models.

In this respect, we illustrate the problems of coupled approaches with an example. Let us suppose that we want to translate a hierarchical graph from an approach which uses edges as grouping primitives (e.g. [DHP02]) to another which uses nodes for the same purpose (e.g. [EH00]). We can proceed as follows.

1. We map grouping edges from the first approach to dummy grouping nodes in the second approach, i.e. to nodes which do not correspond to nodes in the original graph.

2. We represent the fact that the original edges (which are also mapped to the second approach because they are also *normal* edges) are related to the corresponding

grouping nodes in the second approach. If we do not model this, we are discarding the information that a certain grouping is related to a certain edge: This information would be stored in the first hierarchical graph but not in the second.

Point 2 above is rather tricky. We can code this information with an additional edge from one of the attachment nodes of the grouping edge to the dummy node in the second graph, although this is not a very elegant solution. An alternate solution can be to represent graphs from the first approach as bipartite graphs in the second, and use nodes to model both nodes and edges from the first approach. In such a way, nodes modeling edges would allow to define the hierarchical structure. (This solution is rather tricky, too.)

The problem with coupled approaches is that they use one primitive to represent distinct pieces of information. For example, a complex node represents

- the fact that there is a node in the graph,
- the fact that there is a grouping of graph items (nodes and edges),
- the fact that this grouping is associated to that particular node,

thus forcing one to represent all these aspects together. The resulting representation will be compact and *natural* only if this clustering of information is appropriate for the data being modeled. When we want to model, say, edges that abstract bundles of edges we are in trouble.

In a decoupled approach, we provide a compromise to different coupling representations by keeping all these pieces of information separate, so that no particular clustering is a-priori enforced. If we want to translate the same graph as above to a decoupled approach, we can proceed as follows.

1. Build the underlying graph corresponding to the considered hierarchical graph, i.e. the graph without the hierarchy information.
2. Associate to each grouping edge a package, put all nodes and edges in the appropriate packages (see Subsection 2.3.5), and nest the packages according to the nesting of grouping edges (see Subsection 2.3.7).
3. Anchor every package to the corresponding grouping edge (see Subsection 2.3.10).

The advantage is that we do not need to invent an ad-hoc coding because every primitive of a decoupled approach represents exactly one aspect of a hierarchical graph.

From the considerations made in this and the previous subsection, we conclude that a decoupling approach is better suited for defining a general hierarchical graph concept. Decoupling allows to associate a specific role to every primitive of the model, thus giving

more flexibility in the kind of hierarchical graphs that we can store, and making it easier to translate hierarchical graphs from existing models into ours.

Further advantages of a decoupling approach are gained when defining hierarchical graph transformations, as discussed in Chapter 5.

2.3.5 Associating graph elements to packages, sharing

Up to now we have established that a hierarchical graph shall contain a set of nodes N, a set of edges E, and a set of packages P. The next issue we consider is how the elements of the graph are related to packages.

Packages are meant as "general-purpose containers of graph elements", therefore we need to define a *containment relation* (or *association relation*) $\mathrm{C} \subseteq P \times (N \uplus E)$, which associates each package to the nodes and edges that it contains. As a variation, we consider the case where only nodes are contained in packages, whereas edges are "floating" outside the hierarchy. In the latter case we have a containment relation $\mathrm{C} \subseteq P \times N$, and we speak about (*plain*) *hierarchical graphs*. The former kind of hierarchical graph, which we will use in the rest of this chapter, is called *full hierarchical graphs*. In either case, we can see the structure formed by a set of packages, a set of nodes (resp. nodes and edges) and the containment relation, as a bipartite graph, which we call a *coupling graph*.

In the literature and in applications, we find different kinds of containment relations for hierarchical graphs. We consider several possibilities, as illustrated in Figure 2.5. These different possibilities motivate the use of the two terms containment and association. In the extreme case where a hierarchical graph is a sort of modular decomposition of a graph (see below), nodes and edges are actually "contained" in packages, in the sense that it is possible to speak about the inside and the outside of a package, and to tell where a graph element is located. The other extreme is unrestricted containment (see again below), where nodes and edges are freely distributed over the hierarchy, and packages are very close to views defined on top of the underlying graph. In this case, it is more appropriate to say that an item is "associated" to a package.

For these reasons, we have allowed both terms in our definitions. In all informal discussions, we have tried to choose the term that is more appropriate for the kind of hierarchical graph being considered.

Modular decomposition A first possibility, which we find for example in [DHP02], is that each graph element be associated to exactly one element of the hierarchy (i.e. to exactly one package in our approach). In this case, the graph is partitioned into as many (possibly empty) subgraphs, as many packages there are in the hierarchy. Package boundaries are completely opaque and isolate the content of a package from the rest of the world.

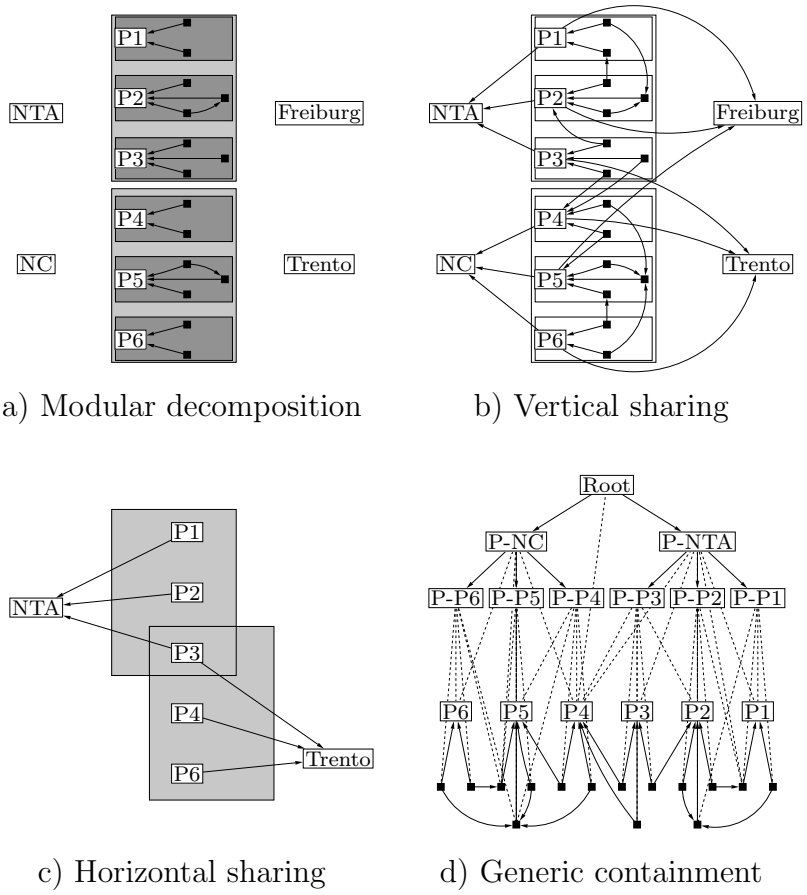


Figure 2.5: Several kinds of containment relations.

This approach is particularly appropriate if we think of a hierarchical graph as a modular decomposition of a graph. This approach would not be suitable for our running example, since we need at least edges across package boundaries to model, for example, hyperlinks from one project to another. See Subsection 2.3.6 for a discussion on boundary crossing edges.

An example of modular decomposition is shown in Figure 2.5.a. We suppose that there is a surrounding *root* package that contains the two projects and the two sites, which is however not depicted. As in Figure 2.2, we use shading for indicating the different packages, namely *white* for the root package, *light grey* for the project packages, and *dark grey* for the page packages. Here, a node or edge in a certain shade indicates that it belongs exactly to that package.

Notice that we are using the web graph from our running example, but we have removed the forbidden edges. For example, the edge between `P1` and `NTA` can neither

be contained in the package of `P1`, because its target is outside that package, nor outside because its source is inside (again, see 2.3.6). Therefore the underlying graph is partitioned into several subgraphs, one for each package.

Vertical sharing A second possibility is to consider packages transparent w.r.t. their containing packages. This means that if some graph element x is associated to a package p and p is nested in a package q, then x can also be associated to q.

This idea is closer to the semantics of aggregation, as found e.g. in [RBP+91], where part-of relations are transitive. Notice however that in object orientation we have relations between classes or between objects, whereas here we have a relation between graph elements and packages, which is propagated upwards along the hierarchy. When a situation like the one described above occurs, we say that a graph element is *vertically shared* in the hierarchy, and we speak of *vertical sharing*.

In our running example, vertical sharing means that all anchors, besides being contained each in one page package, can also be contained in the surrounding project package, and in the root package. Pages would be contained in the corresponding project package and in the root package.

Vertical sharing is depicted in Figure 2.5.b, where the absence of shading indicates that all nodes from sub-packages are also contained in super-packages. Thus page `P1` is contained in the first page package, in the surrounding project package, and in the (not depicted) root package.

In the same picture, an edge is contained in a package if its line is all contained within the borders of the package. If an edge is contained in a package p, then it is contained in all super-packages of p. For example, the edge between the upper anchor of `P1` and the middle anchor of `P2` is contained in the `NTA` project package, but in no page package. It is also contained in the root package.

This approach is compatible with our web example, in the sense that it allows all wanted edges, still maintaining the original idea of structuring the underlying graph. It is however not good to allow unrestricted vertical sharing as in Figure 2.5.b, since, as pointed out in Section 2.1, it often makes sense to hide graph elements inside the hierarchy, i.e. to control vertical sharing. This issue is discussed further on in this section, namely in Subsection 2.3.8.

Horizontal sharing While vertical sharing deals with sharing of graph elements between packages that are in a child-parent relation, *horizontal sharing* considers sharing of graph elements between sibling packages.

In our running example, we could imagine that we want to group pages not only according to corresponding projects, but also according to the sites that host them. In the resulting package structure, some pages are shared by sibling packages, namely by project and site packages. In Figure 2.5.c, we show the `NTA` project package and the

`Trento` site package, together with their contained page nodes. Page `P3` is *horizontally shared* among the two packages. The edges in the picture all belong to the root package.

When we allow this kind of sharing, we can no longer see the package structure as a modular decomposition of the graph. On the other hand, packages with horizontal sharing are closer to views in the database sense, where a view contains an arbitrary collection of records from a given underlying database, and two unrelated views may well have a non-empty intersection.

Although it does not provide a modular decomposition of the underlying information, horizontal sharing is still a meaningful way of organizing data. As another example, in the hypermedia system HyperWave (see again Subsection 2.2.2), an existing document can be inside different containers, not necessarily nested into each other.

Unrestricted containment As a last possibility of assigning the content of an underlying graph to a hierarchy of packages, we consider a relation $\mathrm{C} \subseteq P \times (N \uplus E)$ with no restrictions. Such a relation is, in our opinion, even closer to the notion of a view. Then a hierarchical graph can be seen as a collection of views on an underlying graph, with some ordering between them.

In Figure 2.5.d, we show an example of such an unrestricted collection of views, inspired from our running example. Again, we have left out the project and site nodes to simplify the picture. We have also separated the package hierarchy from the underlying graph, and represented the containment relation explicitly, using dashed lines between graph elements and packages. The containment of edges is not depicted.

We think that such an unrestricted containment relation can be useful for certain applications (see again the example of HyperWave). We also think that, in other applications, some control over the propagation of information along the hierarchy — i.e. over the containment relation — is needed. This has already been illustrated in Section 2.1 and Section 2.2, when speaking about encapsulation. We look at encapsulation in more detail in Subsection 2.3.8.

Summary In this subsection, we have discussed how graph elements can be distributed over the package hierarchy of a hierarchical graph through a containment relation. We have considered modular decomposition and unrestricted distributions (unrestricted containment relations). We have also discussed vertical and horizontal sharing.

In our model, we will formalize explicitly unrestricted distributions (the basic model) as well as modular decomposition as a particular case. We also formalize an intermediate kind of distribution based on import/export interfaces between packages. This is discussed in Subsection 2.3.8.

2.3.6 Edges, edge containment, boundary-crossing edges

In this subsection, we are interested in the way edges are attached to nodes, and in the interaction between node and edge containment in a hierarchical graph, i.e. whether the fact that an edge is contained in a package should influence where its attached nodes are contained.

In our hierarchical graph model, as sketched up to now, we have introduced the notion of a graph skeleton as a pair of sets (N, E), where N are the nodes and E are the edges of the graph, and we have been very vague about their linking mechanism since, as discussed in Subsection 2.3.1, this depends on the considered kind of graphs. We now want to be more specific about the linking mechanism, and we make the reasonable assumption that, for each graph $G = (N, E)$, there exists an incidence relation $I \subseteq E \times N$, that tells us whether a given edge is *incident* in a given node. If an edge e is incident in a node n we also say that e is *attached* to n. Therefore, we refine our notion of a graph skeleton, as introduced in Subsection 2.3.1, by considering a skeleton as a triple (N, E, I), where I is an incidence relation.

Now, suppose that a graph G is given, and that a hierarchical graph is built on top of its skeleton $S = (N, E, I)$. A reasonable restriction is that, whenever an edge is contained in a package, all its attached nodes are contained in the same package. The idea is that if a package knows of the existence of an edge, it should know of the existence of all its attached nodes.

In our example, edges linking anchors to their page pose no problems, since they are contained in the corresponding page package. But where should, for example, the hyperlink between the upper anchor of page `P1` and the middle anchor of page `P4` be contained? We can suppose that this edge is in the package of project `NTA`. Now, does it make sense that one of the two anchors or both are not contained in the same package?

In our opinion such a situation should be forbidden: Package `NTA` would know the link but not its ends. The solution is that the two anchors be contained in their original page packages *and* in the surrounding package. Such an edge, linking nodes from different packages, is called a *boundary-crossing edge*.

Notice that, when we only allow a modular decomposition of the underlying graph (see Subsection 2.3.5), no boundary-crossing edges are allowed, since at least one end of a boundary-crossing edge must be contained in at least two packages: its original package, and the package containing the edge. Notice also that boundary-crossing edges are always allowed in a plain hierarchical graph.

2.3.7 Different kinds of hierarchies

In general, a hierarchy involves having several *packages* with some relation between them indicating that a certain component p is on a *level* (*layer*) immediately above another component q. Thus a hierarchy can be seen as a graph—the *hierarchy graph*—

induced by such a relation between components. The reader should not confuse the term *hierarchy graph* with the term *hierarchical graph*.

Up to now, we have implicitly spoken about a package hierarchy, without specifying what kind of hierarchy, i.e. of graph, we had in mind. In our running example, the hierarchy of packages forms a tree, but this can be too restrictive in certain situations. For example, in HyperWave (see again Subsection 2.2.2) a container can be nested in two or more different containers. More generally, *horizontal sharing* is needed at the hierarchy level as well as at the underlying graph level.

From these considerations, we conclude that a generic hierarchical graph model should allow not only tree-like hierarchies, but also dag-like[3] ones, where sharing is allowed. Since tree-like hierarchies play an important role in modeling, we consider them as a relevant subcase of the more general dag-like ones. Therefore, we will speak about *dag-like* and *tree-like hierarchical graphs*.

We do not allow more general hierarchy graphs than dags, i.e. graphs containing cycles, since we think that the very concept of a hierarchy (a layered structure) would be lost if we allowed components to (possibly indirectly) contain themselves. There are, however, approaches (see e.g. [Pra79]) where cyclic hierarchies are allowed, and our decoupled approach makes it very easy to allow them in case they should be needed: It suffices to relax the constraints on hierarchy graphs.

From a modeling point of view, a tree-like hierarchical graph is closer to the idea of a modular decomposition, already discussed in Subsection 2.3.5. However, requiring a tree-like hierarchy does not necessarily involve that we forbid sharing at the graph level (see e.g. [ES95], [BEMW00]): We may allow a freer policy w.r.t. the distribution of graph elements along the hierarchy. In our example, although the hierarchy is a tree, it makes sense to share nodes like anchors and pages among different packages.

On the other hand, the dag-like approach, where *sharing* of subpackages is allowed, is more in the direction of the view interpretation, where a hierarchical graph is seen as a layered collection of views on top of an existing graph. We think that this interpretation, and therefore a dag-like model, can be appropriately combined with unrestricted distribution of graph elements in the hierarchy.

A last feature that we require about hierarchies is that they always have a *root* package, which, directly or indirectly, contains all other packages. This means that we want *rooted dags* (i.e. dags in which all nodes are reachable from a root node through a path) in the general case, and trees in the restricted case.

In our running example, this means that we always model the hierarchy as in Figure 2.5.d, where we have added a root package to model the *environment* containing all the outer nodes, edges, and packages. As we can see, this both makes sense from a modeling point of view, and is technically convenient, since it allows us to have all nodes and edges in at least one package.

[3]Dag stands for *directed acyclic graph*.

Summarizing, in our model we support two kinds of hierarchies (the second being a restricted case of the first), namely rooted dag-like and tree-like hierarchies.

2.3.8 Encapsulation

In Subsection 2.3.5, we have spoken about sharing of graph elements among packages in a hierarchical graph. We have also pointed out that mechanisms are needed to control this sharing. In this section, we consider such a mechanism, based on *import/export interfaces* between packages, an idea taken from software engineering and programming languages. A hierarchical graph with encapsulation will be called *encapsulated hierarchical graph*.

The basic idea of import/export interfaces is that every graph element should be *owned* by exactly one package, and that knowledge about graph elements should be spread up and down the hierarchies through the interfaces between the packages. Thus a package can *import* a graph element that is either *exported* by a direct subpackage or contained (in the sense of containment relations) by a direct superpackage. Contained elements are owned and imported elements. A package can export a contained element only if that element was not imported from a superpackage: it does not make sense to export to the environment what is already there! It is clear that a package *encapsulates* (*hides*) a graph element which is internal to it (owned or imported from a subpackage) if it does not export it.

From a modeling point of view, this encapsulation concept seems pretty natural when applied to tree-like hierarchies. In this case, a hierarchical graph is close to the idea of a modular decomposition of the underlying graph, with import/export interfaces between the different modules.

In the case of dag-like hierarchies, the import/export mechanism does not provide true encapsulation. In fact, it can happen that a graph element x, which is imported by two packages p_1 and p_2 from a common subpackage q, is, say, exported by p_1 and hidden by p_2. Then, knowledge about x can spread upwards through the hierarchy, although p_2 is trying to hide it. Although we have not investigated practical applications of import/export interfaces for dag-like hierarchies, we think that they can be an interesting way of organizing information in a hierarchical graph.

Import/export interfaces can again be seen as views in the database sense (see Subsection 2.3.5 and Subsection 2.3.7). More precisely, they can be seen as views on views (i.e. on the content of packages) and a hierarchical graph can be seen as a complex structure of views on a common underlying graph, with constraints between them (the import/export constraints described above).

Summarizing, in order to provide a finer control on the containment relation of a hierarchical graph, we define an encapsulation concept for hierarchical graphs by means of import/export interfaces between packages. In the tree-like case, encapsulation provides a refinement to the idea of a modular decomposition of a graph. In the dag-like

case, encapsulation is only partial, and can be interpreted as the definition of a complex system of view on top of an underlying graph: Interfaces filter the spreading of information between the levels of the hierarchy. Encapsulated hierarchical graphs are defined in Section 3.3.

2.3.9 Typing

In Subsection 2.3.1, we have considered typing in general, and we have briefly considered typed graphs. In this subsection, we consider typing with more detail, and we discuss the extension of graph typing to hierarchical graph typing.

Types are a fundamental concept for programming languages (see e.g. [Wat90]). Programming languages deal with *values* (i.e. elementary pieces of data), which can be grouped into sets called *types*, according to common properties and operations defined on them. Thus we can define an *integer* type, with $\{\dots, -2, -1, 0, 1, 2, \dots\}$ as set of elements, and $+$, $-$, $\times$, / (the integer division) as operations.

According to the "object-oriented" or "database-oriented" view, typing is used to capture common properties and/or behaviour of entities inside a given domain (see e.g. [RBP+91]). Such common properties/behaviour allow us to partition our domain into *classes* or *entity types*. In our running example, we can identify four entity types: Pages, anchors, projects, and sites.

The latter concept of typing is very close to graph typing. In fact, a graph can be easily thought of as a collection of objects (the nodes) with links between them (the edges), and the notions of graph typing that we find in the literature—in particular in the graph-grammar literature—are very close to the corresponding notions in object orientation. Besides node and edge types, graph types allow to express constraints like "an edge of type ε can only be incident to nodes of types $\nu_1, \dots, \nu_k$" (see e.g. [CEMP96]), or cardinality constraints w.r.t. the number of edges of a certain type that can be incident on nodes of a certain node type (see e.g. [SWZ99]). (Further typing information that can be found in the literature on graph grammars concerns node attributes (see again [SWZ99], [LB93]). In this work, we are not interested in node attributes since, as discussed in Subsection 2.3.1, we focus on the node/edge structure of a graph.)

Since a hierarchical graph contains a graph as a substructure, we consider hierarchical graph typing as an extension of graph typing. Thus, besides node and edge types and constraints on them, we would like to model package types, constraints on package nesting and on the containment relation between packages and graph elements.

In our WWW model, we can define four types of nodes—*anchors*, *pages*, *packages*, and *sites*—and several edge types, for example from page to site, from page to project, and so on. We have already illustrated this graph type in Subsection 2.3.1: See Figure 2.3 on page 21. For example, in our WWW model, we have two types of packages, namely page and project packages: page packages are always nested in project packages, whereas the inverse is not allowed. Furthermore, anchors are contained in page packages

(the coupling graph is also typed), and so on.

Summarizing, a hierarchical graph type takes the following information into account:

- The underlying graph of a hierarchical graph must be typed.
- The hierarchy graph must be typed. This means that we can establish a priori what kinds of packages can be present in a hierarchy, and how they are allowed to be nested.
- The coupling graph (i.e. the connection relation between packages and graph elements) must be typed. This models constraints such as which packages can contain which nodes or edges of the underlying graph.

A hierarchical graph type will be a triple of graph types, one for each component of the hierarchical graph. The formal definition, the corresponding notation, as well as further illustration of hierarchical graph types can be found in Section 3.4.

2.3.10 Associating packages to graph elements

A last topic that we want to address in this section is the issue of associating packages to graph elements. As already seen in this chapter, there are several *coupling* approaches to hierarchical graphs. This means that the hierarchy is coded in the graph and elements of the hierarchy are strictly bound to elements of the graph, like nodes, edges, hyperedges, and so on.

Although it is not always good to couple packages and graph elements, this can often be a useful piece of information to model. For example, in our WWW model we should associate every page package to the corresponding page node. Then, although a generic hierarchical graph model should not force one to use coupling, it should allow to model such coupling when needed. We achieve this by means of *anchoring* or *qualification*, i.e. we define a relation that *anchors* packages to nodes or edges of the underlying graph. A package that is anchored to some graph element is also said to be *qualified* by that element. As we will see in Chapter 3, this relation will be stored, together with the containment relation, in a graph called the coupling graph. (Compare Subsection 2.3.5.)

We say that a hierarchical graph where no package is qualified is *loosely coupled*, while one where every package is qualified is called *tightly coupled*. Notice that we allow more than one package to be anchored to the same node or edge (compare Subsection 9.2 and [Con94, page 51]).

2.4 Summary

In this chapter we have discussed the static aspects of an abstract hierarchical graph model.

In Section 2.2, we have considered several approaches that use hierarchical graphs either explicitly or implicitly. The result of this investigation is that different approaches share common ideas, like that of grouping and encapsulating graph elements, and that of defining constraints based on this additional structure.

The following step was to further analyze these approaches, and extract the features that we want to provide with our hierarchical graph model. Instead of a concrete model, we provide a generic framework that can be instantiated for specific applications as needed. This framework allows to combine freely the features previously identified.

The first feature, considered in Subsection 2.3.1, concerns the notion of graph that should be used in a hierarchical graph. The result of this investigation is a minimal model of a graph, called a *graph skeleton*, that serves as an interface between the actual *underlying graph* being structured and the superimposed hierarchy.

In Subsection 2.3.3 and Subsection 2.3.4, the idea that the hierarchy and the underlying graph should be separate (decoupled) structures is motivated and elaborated. By comparing coupling and decoupling approaches to hierarchical graphs, we have argued that a decoupling approach is more convenient, both from a conceptual and from a technical point of view, for an abstract model like ours. As a consequence, we have proposed the concept of a *graph package* as a general-purpose container of graph elements.

In Subsection 2.3.7, we have discussed the allowed structures of the package hierarchy (called the *hierarchy graph*), namely tree-like and dag-like hierarchies. In dag-like hierarchies sharing of subpackages is allowed, whereas in tree-like ones it is not. It is anyway forbidden that a package, even indirectly, contains itself. This constraint can be removed in particular applications.

In Subsection 2.3.5, we have considered several ways of distributing the elements of a graph along a hierarchy, which lead to different restrictions on the sharing of graph elements among packages. The distribution of graph elements influences the possibility to define boundary-crossing edges, i.e. edges that are incident in nodes contained in different packages, as discussed in Subsection 2.3.6. Such edges can only be contained in a package that also contains all their attached nodes. Technically, the containment relation between graph elements and packages, is modeled as a graph, called the *coupling graph*.

In Subsection 2.3.10, we have discussed *qualification*, a mechanism to anchor packages to graph elements. In this way, the coupling information that is mandatory in coupled approaches can still be modeled in our approach. From a technical point of view, this relation will also be stored in the coupling graph. As a result, a hierarchical graph will be modeled as a triple of graphs.

In Subsection 2.3.8, we have discussed encapsulation, a mechanism that allows to restrict the visibility of graph elements inside a hierarchical graph by means of import/export interfaces.

Finally, in Subsection 2.3.9, we have considered typed hierarchical graphs. Since a hierarchical graph is represented as a triple of graphs, a hierarchical graph type will

be defined as a combination of three graph types. As discussed in Subsection 2.3.2, the idea of hierarchical structuring and the notion of typing, which apparently seem related, serve indeed to model different aspects of data.

In this chapter, we have illustrated our discussion using a running example based on the world-wide web. We will continue using and further developing this example in later chapters of the thesis.

Chapter 3

Graph Packages and Hierarchical Graphs

In this chapter, we formally define our hierarchical graph model. In this model, a hierarchical graph consists of an underlying flat graph, on top of which we add a hierarchy structure. The hierarchy is in turn a graph, either a directed acyclic graph (dag) or, in a more restricted case, a tree. The components of the hierarchy are called (*graph*) *packages* and are general-purpose containers of nodes and edges from the underlying graph. The expression "general-purpose" indicates that there is no predefined interpretation for such containers: the user of a hierarchical graph will decide the meaning of the hierarchical decomposition in his/her model. The underlying graph and the hierarchy graph are connected to each other by a third graph, called *coupling graph*. The notions of an underlying graph, of a hierarchy graph, and that of a coupling graph, have been discussed informally in Subsection 2.3.1, 2.3.7, and 2.3.10, respectively. This *decoupled approach*, where the hierarchy and the underlying graph are modeled as separate structures, has been discussed and motivated in Subsections 2.3.3 and 2.3.4.

The formal definition of our model includes several steps. First of all, in Section 3.1, we fix some basic notions. These include:

- *graph skeletons*, which serve to abstract from the actual underlying graph being structured and to provide a uniform interface with the hierarchy defined on top of it (see the discussion in Subsection 2.3.1 and Subsection 2.3.6);

- *directed acyclic graphs* and *trees*, which serve to model the hierarchy structure of a hierarchical graph (see Subsection 2.3.7);

- *bipartite graphs*, which serve to model the connection between the underlying graph and the hierarchy (see also the informal discussion in Subsection 2.3.5 and Subsection 2.3.10).

Using these notions, in Section 3.2 we define *hierarchical graphs* as a combination of a graph, a hierarchy graph (dag or tree of packages) and a bipartite coupling graph. We give particular stress to boundary-crossing edges and to the way they interact with the hierarchy.

A refinement of hierarchical graphs leads to *encapsulated hierarchical graphs* (see Section 3.3), where the coupling graph is restricted by means of import/export interfaces between packages.

Subsequently, in Section 3.4, we define *typed hierarchical graphs*. Following our decoupled approach, hierarchical graph types are a combination of a graph skeleton type, of a hierarchy graph type, and of a coupling graph type.

In Section 3.5, we speak about *qualification*, a mechanism which allows to associate packages to graph elements (see also the discussion in Subsection 2.3.10).

3.1 Basic Definitions

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

We first consider what kind of graph we should use to model the *underlying graph* of a hierarchical graph, i.e. the graph that is actually being structured. We would like to use our hierarchical graph model with several different kinds of graphs, e.g. directed/undirected, labelled/unlabelled graphs, and so on. We therefore looked for some "minimal" assumptions that a graph should satisfy in order to be used as the underlying graph in a hierarchical graph.

We must address the question: "What are the essential elements of a graph that allow us to structure it in a hierarchical fashion?" This issue has been discussed in Subsection 2.3.1, where we have identified the following *essential elements*:

- A graph should have *nodes*. It does not matter whether nodes are labelled, unlabelled, typed, untyped, attributed, and so on: It is essential that a graph contains nodes that we would like to structure in some way.

- A graph should have *edges*. Without edges we would be structuring just a collection of nodes, which would be tantamount to structuring any collection of entities. Edges are very important because they model relations between nodes, and often some kind of proximity information, thus defining the localities in the graph. It is not unusual that a node and all its neighbours are in the same component of the hierarchy.

- The way nodes and edges are connected to each other is also important because it defines the "topological structure" of a graph. Since there exist many different

kinds of graphs where nodes are connected to each other through edges in different ways, in our abstract hierarchical graph notion we only model the fact that edges are incident in nodes (nodes are attached to edges).

We consider all other elements that can appear in a graph as marginal with respect to hierarchical structuring. For example, attributes should not have anything to do with the hierarchy. Even though attributes can be used to implement a hierarchy—by means of graph-valued attributes—we believe that this is only an ad-hoc solution, and that the hierarchy should be kept separate from attributes.

Summarizing, 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 such a triple a *graph skeleton*. A graph skeleton allows us to interface any graph with the hierarchical structure defined over it, or, in other words, any graph (directed, undirected, typed, attributed, and so on) that provides a graph skeleton (possibly by forgetting some of its elements) can be lifted to a hierarchical graph using our concepts.

We now define graph skeletons formally.

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

We will indicate the components of a graph skeleton S as N_S, E_S, I_S respectively. Given a skeleton S, a node $n \in N_S$, and an edge $e \in E_S$, we will often write $I_S(e, n)$ instead of $(e, n) \in I_S$. We also define the set of *atoms* of S as $A_S := N_S \cup E_S$. 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}, I_G for I_{S_G}, and A_G for A_{S_G}.

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

Given a class of finite graphs $\mathcal{G}$, we define the class of skeletons $\mathcal{S}(\mathcal{G}) := \{(N_G, E_G, I_G) \mid G \in \mathcal{G}\}$.

We have grouped nodes and edges together into a set of atoms, because atoms will be put into components of the hierarchy in a hierarchical graph. Notice that the notion of a skeleton allows many kinds of edges, like directed/undirected, binary edges or hyperedges, dangling edges[1]. Edges between edges are forbidden, but they could be allowed by removing the constraint that the set of nodes and the set of edges should be disjoint.

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

[1] A *dangling edge* is an edge for which one of the attachment nodes is missing.

Example 3.2 Our running example uses directed graphs (to be formally defined shortly) to model hyper-web 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 $N := \{\texttt{NTA}, \texttt{P1}, \texttt{P2}, \dots\}$, the set of edges $E := \{\texttt{P1-NTA}, \texttt{P2-NTA}, \dots\}$, and the incidence relation

$$I := \{(\texttt{P1-NTA}, \texttt{P1}), (\texttt{P1-NTA}, \texttt{NTA}), (\texttt{P2-NTA}, \texttt{P2}), (\texttt{P2-NTA}, \texttt{NTA}), \dots\}$$

Notice that labels are forgotten in the skeleton.

We now define directed graphs, which are needed to model 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 *node- and edge-labelled, directed graph* is a tuple $G = (N, E, s, t, l, л)$, where N is a set of *nodes*, E is a 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 $l : N \to \Sigma$, $л : E \to \Delta$ are the *node* and the *edge labeling function* respectively.

A *loop* is any edge $e \in E$ with $s(e) = t(e)$. The element of a set $\{e_1, \dots, e_k\}$ of edges with, for all $i \neq j$, $e_i \neq e_j$, $s(e_i) = s(e_j)$ and $t(e_i) = t(e_j)$, are called *parallel edges*. Parallel edges with the same label are called *multiple edges*.

The skeleton of a directed graph G is a tuple $S = (N_G, E_G, I_G)$, where $I_G := \{(e, s_G(e)) \mid e \in E_G\} \cup \{(e, t_G(e)) \mid e \in E_G\}$.

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

A *simple graph* is an unlabelled directed graph $G = (N, E, s, t)$, with no loops and no parallel edges. A simple graph can also be represented as a pair $G = (N, E)$, where N is a finite set and $E \subseteq N \times N \setminus \{(n, n) \mid n \in N\}$[2].

Given a 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 labeling functions l_G and $л_G$ respectively.

We now define paths and cycles, which we need for the definition of directed acyclic graphs and for the definition of trees.

Definition 3.4 (Paths and cycles)
Let $G = (N, E, s, t, l, л)$ be a directed graph. Then a *path* in G from u to v, for some

[2] E is some *irreflexive* relation over N.

$u, v \in N$, is a sequence of edges $e_1, \ldots, e_k$ ($k \geq 1$) such that, for all $i = 1, \ldots, k-1$, $t(e_i) = s(e_{i+1})$, $s(e_1) = u$, and $t(e_k) = v$. A *cycle* in G is a path from a node $n \in N_G$ to itself.

Given a directed graph G, we introduce the relations $\succ_G, \succ_G^+ \subseteq N_G \times N_G$, defined as follows:

- For all $u, v \in N_G$, $u \succ_G v$ iff there exists $e \in E_G$ with $s_G(e) = u$ and $t_G(e) = v$;
- For all $u, v \in N_G$, $u \succ_G^+ v$ iff there exists a path from u to v in G.

Notice that, if G is a simple graph, then $\succ_G = E_G$. The intuition behind the notation $u \succ_G v$ (resp. $u \succ_G^+ v$) is that some node u *precedes* some node v w.r.t. some edge (resp. some path).

In order to define the hierarchy of a hierarchical graph, we need a notion of directed acyclic graph and a notion of tree.

Definition 3.5 (Directed acyclic graphs and trees)
A *directed acyclic graph* (dag) is a simple graph $G = (N, E)$ that contains no cycles. Given a directed graph $G = (N, E)$, if there exists a node $n \in N$ such that, for all $m \in N \setminus \{n\}$, there exists a path from n to m in G ($n \succ_G^+ m$), then we say that n is a *root* of G and we call G a *rooted directed graph*. Notice that if such a node n exists for a dag G, then it is unique. If a dag G has a root we denote it as ρ_G, and we call G a *rooted dag*.

Let G be a rooted dag, with root ρ. If for all $m \in N_G$ there exists a *unique* path from ρ to m in G, then G is called a *tree*. The node ρ is called the *root* of G (seen as a tree). Given a tree T, we indicate its root with ρ_T.

Now that we have defined all the concepts we need for modeling the hierarchy of a hierarchical graph, we introduce coupling graphs as a particular kind of bipartite graphs. Note that graph skeletons, as described in Definition 3.1, are also a particular example of bipartite graphs, where the two sets of nodes are interpreted as the set of nodes and the set of edges in some other graph. Due to their different interpretation, we want to have skeletons and coupling graphs as distinct concepts, and to use a fresh definition and notation for the latter.

We require two conditions on coupling graphs. The completeness condition states that every atom must be associated to at least one package, i.e. we forbid (orphan) atoms that are nowhere in the hierarchy. The injectivity condition states that packages can be anchored to at most one atom, and that each atom has no more than one package anchored to it. The latter condition is motivated by the fact that anchoring should model the coupling of packages and graph elements found in coupling approaches. This condition could be relaxed in order to model approaches like *hygraphs*, as suggested in Subsection 9.2 on page 195.

Definition 3.6 (Coupling graph)
A directed graph B is *bipartite* if N_B can be partitioned into two sets of nodes X, Y, such that, for all edges $e \in E_B$, either $s_B(e) \in X$ and $t_B(e) \in Y$ or $s_B(e) \in Y$ and $t_B(e) \in X$. If B is simple, we represent it as a triple $B = (X, Y, E)$, with $E \subseteq (X \times Y) \cup (Y \times X)$.

A *coupling graph* is a bipartite simple graph $B = (P_B, A_B, C_B)$, that satisfies the following conditions:

1. *Completeness condition*: For every $n \in A_B$ there exists some $p \in P_B$ and an edge $e = (n, p) \in C_B$, i.e. every node in A_B is connected to at least one node in P_B.

2. *Injectivity condition*: For every $p \in P_B$, if there exists some $x \in A_B$ and an edge $e = (p, x) \in C_B$, then for all $e' = (p', x') \in C_B$, if $e' \neq e$ then $p' \neq p$ and $x' \neq x$, i.e. every node in P_B is connected to at most one node in A_B and no two nodes in P_B are connected to the same node in A_B.

We call P_B the set of *packages* and A_B the set of *atoms*. For a coupling graph B, we define an *association relation*

$$\leftharpoonup_B := \{(p, x) \in P_B \times A_B \mid (x, p) \in C_B\}$$

and a *correspondence relation*

$$\approx_B := \{(p, x) \in P_B \times A_B \mid (p, x) \in C_B\}$$

If p is a package and x is an atom of B, if $p \leftharpoonup_B x$ we say that x *is associated to* or *contained in* p, and if $p \approx_B x$ we say that p is *qualified by* or *anchored to* x. If x is a node we say that x is the *qualifying node* of p. Similarly, if x is an edge we say that x is the *qualifying edge* of p. If $\approx_B = \emptyset$, we say that B is a *loose coupling graph*. If for all $p \in P_B$, there exists an atom $x \in A_B$ such that $p \approx_B x$, we say that B is a *tight coupling graph*. From now on, the expression coupling graph will indicate a loose coupling graph, unless otherwise indicated.

We use the notation P_B, A_B and C_B because the edges C_B represent the *coupling* between the *packages* P_B and the *atoms* A_B of a hierarchical graph (see below). Loose coupling graphs will be used for defining our basic notion of a hierarchical graph (see Section 3.2), whereas non-loose and tight coupling graphs are used for representing qualified hierarchical graphs (see Section 3.5).

3.2 Hierarchical Graphs

In this section, we give the definitions of hierarchical graphs. We introduce an important distinction, namely between *full hierarchical graphs* where both nodes and edges belong

to the hierarchy, and (*plain*) *hierarchical graphs* where nodes are distributed in the hierarchy while edges are not.

Besides defining plain and full hierarchical graphs, we provide some useful notation for them. Subsequently, we discuss boundary-crossing edges, i.e. edges across different packages. The last part of the section is dedicated to possible refinements of the hierarchical graph model, concerning special kinds of hierarchies (tree-like), and special kinds of coupling graphs.

To begin with, we define hierarchical graphs.

Definition 3.7 (Hierarchical graph)
A (*plain*) *hierarchical graph* (HG) is a triple $H = (G, D, B)$, where G is a graph in the sense of Definition 3.1, $D = (P_D, \succ_D)$ is a rooted dag, $B = (P_B, A_B, C_B)$ is a loose coupling graph with $A_B = N_G$ and $P_B = P_D$.

The elements of the set P_D are called *graph packages*, or simply *packages*. The graph D is called the *hierarchy graph*, while B is the *coupling graph* of H.

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. If $p \succ_D^+ p'$ we say that p is an *ancestor* of p' and that p' is a *descendant* of p or p' is *nested* in p. The superpackages of a given package p form its *context*. Two packages that have a common parent package are called *siblings*.

A *full hierarchical graph* (FHG) is a tuple $H = (G, D, B)$, defined as a plain hierarchical graph, with the difference that $A_B = A_G = N_G \cup E_G$[3] and that we require the additional condition that

$$\forall p \in P : \forall e \in E_G : p \leftharpoondown_B e \Rightarrow (\forall n \in N_G : I_G(e, n) \Rightarrow p \leftharpoondown_B n)$$

i.e. if an edge e is contained in a component p of the hierarchy, then all nodes in which the edge is incident are contained in p as well.

The terminology and notation for an FHG is the same as for an HG.

Figure 3.1 depicts a hierarchical graph where the three components (hierarchy graph, underlying graph, coupling graph) are shown separately. The underlying graph is depicted on the right side of the picture, while the hierarchy graph is depicted on the left side. Packages are depicted using a UML-like notation (see [RJB99]). The nodes of these two graphs, together with the dashed edges between them, represent the coupling graph.

The main difference between HG's and FHG's is that in a hierarchical graph, edges are not put in any specific package of the hierarchy, while in a full hierarchical graph edges are also contained in packages. As a consequence, we require an additional condition on FHG's, namely that if an edge is contained in a package, then all its attached

[3]Recall that we assume, for each graph G, that $N_G \cap E_G = \emptyset$.

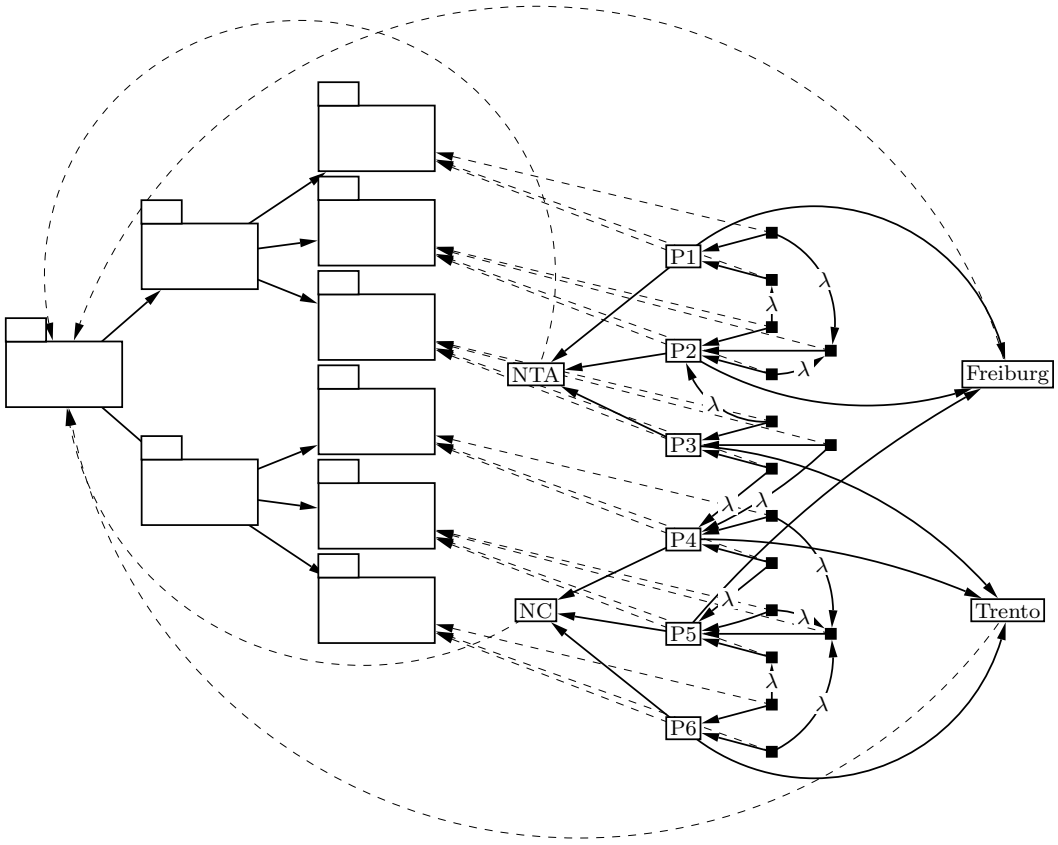


Figure 3.1: A hierarchical graph.

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 attached nodes.

We now introduce some useful notation for HG's and FHG's.

Definition 3.8 (Notation for HG and FHG)
Let $H = (G, D, B)$ be a FHG. If $x \in A_B$, and $p \in P_D$ we will write $p \leftarrowtail_H x$ if $p \leftarrowtail_B x$. Furthermore, we define the *set of contained nodes* of p as $C^{\mathrm{N}}(p) := \{n \in N_G \mid p \leftarrowtail_H n\}$, and the *set of contained edges* of p as $C^{\mathrm{E}}(p) := \{e \in E_G \mid p \leftarrowtail_H e\}$. Finally, we define the *content* of p as $C(p) := C^{\mathrm{N}}(p) \cup C^{\mathrm{E}}(p)$ (remember that we assume the sets N_G and E_G to be disjoint).

If H is a (plain) HG, then we use a similar notation. The only difference is that, for each package p, the set $C^{\mathrm{E}}(p)$ does not exist, and we have $C(p) := C^{\mathrm{N}}(p)$.

Let $\mathcal{G}$ be a class of finite graphs in the sense of Definition 3.1, $\mathcal{D}$ be a class of hierarchy graphs, 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 (plain) hierarchical graphs $(G, D, B) \in \mathcal{G} \times \mathcal{D} \times \mathcal{B}$. We also define the class $\mathcal{F}(\mathcal{G}, \mathcal{D}, \mathcal{B})$ to be the smallest class containing all full hierarchical graphs $(G, D, B) \in \mathcal{G} \times \mathcal{D} \times \mathcal{B}$.

Some HG models (e.g. [DHP02]) allow only edges between nodes that are in the same component of the hierarchy (in the same package in our case). Although this is sufficient for certain applications, in general it is required to allow edges that span across package boundaries. For example, hyperlinks in a hierarchical hypertext can point to any other node in the hypertext, provided that the target is somehow known to the source. In our running example, we want allow page `P3` to link to page `P4`, although they are in different parts of the hierarchy.

In the rest of this section, we consider such *boundary-crossing edges*, i.e. edges across package boundaries (see Subsection 2.3.6), and we see how these interact with the hierarchy when the containment relation is restricted in some way. We formalize boundary-crossing edges in the following definition.

Definition 3.9 (Boundary crossing edges)
Given a hierarchical graph $H = (G, D, B)$, an edge $e \in E_G$ is called a *boundary-crossing edge* of H if there exist $p, q \in P_D$, with $p \neq q$, and two nodes $u, v \in N_G$, with $u \neq v$, such that

$$p \leftarrowtail_B u \wedge q \leftarrowtail_B v \wedge I_G(e, u) \wedge I_G(e, v)$$

i.e. there exist distinct packages of P_D that contain distinct nodes in which e is incident.

Boundary crossing edges in a FHG are defined likewise, since containment of edges is irrelevant for this definition.

Remark 3.10 Notice that we can have boundary-crossing edges in a hierarchical graph, since there are no restrictions on the edges at all.

In the case of a full hierarchical graph F, we can still have a boundary-crossing edge e, but the distinct attachment nodes which belong to distinct packages of F must also belong to the package containing the crossing edge.

Our model supports the sharing of graph elements between different components of the hierarchy, since we use generic containment relations that allow nodes (and in full hierarchical graphs also edges) to be contained by different packages.

In some cases, we want to consider particular hierarchical graphs, where graph elements are contained by exactly one package (see [DHP02] for an example). These model some kind of modular decomposition of a graph, and thus graph elements must be owned by one package only.

Definition 3.11 (Ownership)
Given a hierarchical graph $H = (G, D, B)$, an *ownership relation* for H is a relation $\mathbf{own} \subseteq \leftarrowtail_B$ such that for all $n \in N_G$ there exists a unique $p \in P_D$ such that $\mathbf{own}(p, n)$.

Given a full hierarchical graph $F = (G, D, B)$, an *ownership relation* for F is a relation $\mathbf{own} \subseteq \leftharpoonup_B$ such that for all $x \in A_G$ there exists a *unique* $p \in P_D$ such that $\mathbf{own}(p, x)$.

Fact 3.12
If $F = (G, D, B)$ is a full hierarchical graph, and $\leftharpoonup_B$ is an ownership relation for F, then F does not contain any boundary-crossing edges.

Proof Every edge $e \in E_G$ must be "owned" by exactly one package $p \in P_D$. Then all its attachment nodes must be contained in p, for the condition on containment of nodes, in Definition 3.7. But then all the attachment nodes of e can only be contained in p (the containment relation is an ownership relation), and therefore e is not a boundary-crossing edge.

□

On the other hand, if $H = (G, D, B)$ is a (plain) hierarchical graph, and $\leftharpoonup_B$ is an ownership relation for H, then H can still contain boundary-crossing edges because edge restrictions only apply to full hierarchical graphs.

Besides sharing of graph elements, we can consider sharing of hierarchy components (i.e. of packages). Such sharing is allowed by our model, provided that the obtained hierarchy graph is a dag. A useful restriction when we want to consider a HG as a modular decomposition of a graph (see again [DHP02], [ES95]) is to forbid such sharing completely by requiring that the package hierarchy be a tree. If we forbid both node and package sharing, we obtain the notion of a strict hierarchical graph.

Definition 3.13 (Tree-like and strict hierarchical graphs)
If $H = (G, T, B)$ is a hierarchical graph and T is a tree, then H is called a *tree-like hierarchical graph*. Likewise, a full hierarchical graph with a tree-like hierarchy is called a *tree-like full hierarchical graph*.

If $H = (G, D, B)$ is a tree-like hierarchical graph (full tree-like hierarchical graph) and $\leftharpoonup_B$ is an ownership relation for H, then H is called a *strict hierarchical graph* (*strict full hierarchical graph*).

A strict full hierarchical graph H has no boundary-crossing edges, and therefore it does not allow any kind of relation between different parts of the hierarchy, by forbidding sharing of node, of edges, of packages, and boundary-crossing edges. In [DHP02] we find an example of such graphs.

In Section 3.3, we refine the idea of an ownership relation, by assuming that nodes and edges from the underlying graph are owned by exactly one package, but can still be exported/imported along the package hierarchy. Thus the ownership relation is a refinement of the association relation, and the association relation is controlled by the ownership relation and by import/export interfaces between packages that are adjacent in the hierarchy graph.

3.3 Encapsulated Hierarchical Graphs

As announced in Subsection 2.3.8, in this section we extend the notion of a hierarchical graph to that of an encapsulated hierarchical graph (EHG). In an EHG, all graph elements (nodes and possibly edges) are owned by exactly one package. It is however possible that graph elements are exported by one package and imported by another one via import and export interfaces.

This mechanism can be used to define the containment relation of a hierarchical graph: a graph element is contained in a package if it is owned by it or imported into it. Thus EHG's are a refinement of hierarchical graphs, where we let each graph element be owned by exactly one package and we leave the packages the responsibility to spread information about their content along the hierarchy.

The import/export mechanism of EHG's is very similar to that used in programming languages like Modula2 (see e.g. [Wir85]) or in modeling languages like UML (see [RJB99]). We first describe and illustrate encapsulation informally, and then give the formal definitions.

The rules of encapsulation in EHG's are the following:

1. **Unique ownership.** We suppose that every graph element is *owned* by exactly one package. The owned element is also contained in the owner package.

2. **Export mechanism.** Every package has an *export interface*, which allows to spread information about graph elements upwards in the hierarchy. Contained elements of a package can be in its export interface, provided that restriction 5 below is respected. The root package has no exported elements.

3. **Import mechanism.** Every package p has an *import interface* and every graph element in this interface is contained by p. A graph element can be imported in two ways:

   (a) If x is contained in a package p, p is nested in q, and x is exported by p, then x can be imported in q. In such a case we say that x is *internally imported* in q.

   (b) If a package p is nested in a package q, any graph element x that is contained in q can be put in the import interface of p, provided that restriction 5 is respected. Such a graph element x is said to be *imported from the context* of p.

   Notice that the root package has no context, and therefore can only import graph elements from its subpackages.

4. **Containment, ownership and import.** A graph element x is contained in a package p iff it is either owned by p, or it is made available to p through the import/export mechanism, i.e. if it is in its import interface.

5. **Restriction on import/export.** If p is a package and x is a graph element, then x can be exported by p only if it is owned by p or internally imported in it. If p is a package and x is a graph element, then x can be imported by p only if it is not exported by p itself.

The last condition ensures that a package does not export an element x that it has imported from the context, and that it does not import (from the context) an element x that it also exports and is therefore already inside the package. We say that ownership and import/export interface define the *visibility* of graph elements in a hierarchical graph.

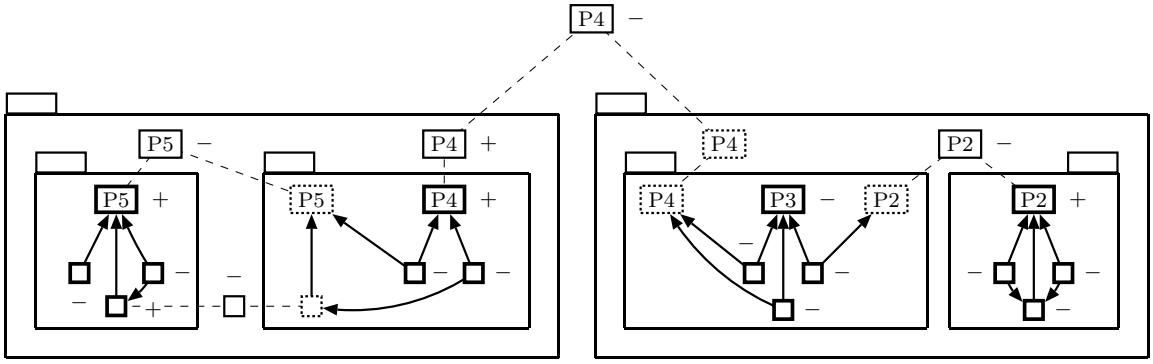


Figure 3.2: Import/export mechanism.

In our running example, we suggested that only certain pages, like home-pages, should be accessible from outside the project and access to other pages should be restricted. We can use ownership and import/export interfaces to model this situation and restrict the containment relation accordingly. Each page is then owned by its corresponding package. In Figure 3.2, nodes (pages and anchors) are depicted inside their owning package using a thick border. Public (exported) nodes are tagged with +, while private (non exported) nodes are tagged with −.

When needed, a node is exported by its owning package to the surrounding project package. For example, nodes `P2`, `P4`, `P5`, and the public anchor of `P5` are public. We depict these nodes as copies of the original nodes, with thinner borders, and with a tag (+/−) indicating whether the exported node should be exported further towards the root of the hierarchy. Since we suppose that `P4` is a homepage, we export this node further to the root package, whereas all other nodes are hidden inside the containing project package.

Since `P4` is exported to the root package, it can be imported down the hierarchy until the package of page `P3`. We depict such imported nodes as copies of the original nodes with dashed border. These nodes are not tagged with visibility information, since

a package cannot hide something that it has imported from the outside. Since P4 is now known inside the package of page P3, we are allowed to place the corresponding hyperlinks (boundary-crossing edges, see Definition 3.9) in the same package. The nodes P2, P5 and the anchor attached to P5 are imported in a similar way, allowing the existence of further hyperlinks.

As we can see in the example, the import/export mechanism allows to restrict the association relation and provides *encapsulation* of graph elements inside the hierarchy. Encapsulation is already provided by the association relation, but in our general hierarchical graph model there is no means to structure this relation in some coherent way. Ownership and import/export interfaces provide this kind of structuring.

We now give the formal definition of encapsulated hierarchical graphs (EHG). We first consider plain EHG's, and subsequently full EHG's.

Definition 3.14 (Encapsulated hierarchical graph)
A (*plain*) *encapsulated hierarchical graph* (EHG) is a tuple

$$H = (G, D, B, \mathbf{own}, \mathbf{exp}, \mathbf{imp})$$

where

- (G, D, B) is a hierarchical graph and **own** is an ownership relation for it,
- $\mathbf{exp}, \mathbf{imp} \subseteq P_D \times N_G$ are two relations, associating to each package its *exported* resp. *imported* nodes,

such that H satisfies the *visibility conditions* (see Definition 3.15 below), and we have $\leftarrowtail_B = \mathbf{own} \cup \mathbf{imp}$.

For all $p \in P_D$, we let $\mathbf{own}(p) := \{n \mid n \in N_G \wedge \mathbf{own}(p, n)\}$, and we call it the *set of own nodes* of p. In a similar way, we define $\mathbf{exp}(p)$, the *set of exported nodes* of p, and $\mathbf{imp}(p)$, the *set of imported nodes* of p. Since **own** is an ownership relation, we can define a function $\mathbf{owner} : N_G \to P_G$ where, for all $n \in N_G$, we let $\mathbf{owner}(n)$ be the unique package p such that $\mathbf{own}(p, n)$.

The existence of a relation **own** satisfies rule 1 on page 51. The existence of relations **exp** and **imp** satisfy rules 2 and 3, likewise. Rule 4 is expressed by the condition that $\leftarrowtail_B = \mathbf{own} \cup \mathbf{imp}$. In the definition of EHG's we refer to a visibility condition, which we are going to define now. This condition states constraints on owned, imported and exported nodes, according to the hierarchical structure of the packages. This condition refers to rules 3.3a, 3.3b and 5.

Definition 3.15 (Visibility conditions for EHG's)
Let $H = (G, D, B, \mathbf{own}, \mathbf{exp}, \mathbf{imp})$ be a tuple as in Definition 3.14. Then the *visibility conditions* for H state that:

1. For all $q \in P_D$ we have

$$\mathbf{imp}(q) \subseteq \bigcup_{q \succ_D r} \mathbf{exp}(r) \cup \bigcup_{p \succ_D q} \mathbf{own}(p) \cup \left(\bigcup_{p \succ_D q} \mathbf{imp}(p) \setminus \mathbf{exp}(q) \right)$$

   in particular, for ρ_D we have

$$\mathbf{imp}(\rho_P) \subseteq \bigcup_{\rho_D \succ_D r} \mathbf{exp}(r)$$

2. For all $q \in P_D$ we have $\mathbf{exp}(q) \subseteq \mathbf{own}(q) \cup \left(\bigcup_{q \succ_D r} \mathbf{exp}(r) \cap \mathbf{imp}(q) \right)$.

3. $\mathbf{exp}(\rho_D) = \emptyset$.

Intuitively, Definition 3.15 states that: (1) An imported node of a package q must either be an exported node of some child package r, or be an owned node of some parent package p, or an imported node of p (but not an exported node of q itself, i.e. p imports the node from one of its parent packages or from a sibling of q). (2) The exported nodes of a package q must either be owned nodes of q or be imported from one of the subpackages q. (3) The root package has no exported nodes.

Like EHG's are an extension of plain hierarchical graphs, *full encapsulated hierarchical graphs* are an extension of full hierarchical graphs. Although the two concepts are very similar (again, the fundamental difference is that the latter allows to distribute the edges of a graph over the hierarchy), we define them separately to keep the single definitions simpler.

Definition 3.16 (Full encapsulated hierarchical graph)
A *full encapsulated hierarchical graph* (FEHG) is a tuple

$$H = (G, D, B, \mathbf{own}, \mathbf{exp}, \mathbf{imp})$$

where

- (G, D, B) is a hierarchical graph and $\mathbf{own}$ is an ownership relation for it,
- $\mathbf{exp}, \mathbf{imp} \subseteq P_D \times (N_G \cup E_G)$ are two relations[4], associating to each package its *exported* resp. *imported* nodes and edges,

such that H satisfies the *visibility conditions* (see Definition 3.17 in the sequel), and $\leftharpoonup_B = \mathbf{own} \cup \mathbf{imp}$.

For all $p \in P_D$, we let $\mathbf{own}_{\mathrm{N}}(p) := \{n \mid n \in N_G \wedge \mathbf{own}(p, n)\}$, and we call it the *set of own nodes* of p, we let $\mathbf{own}_{\mathrm{E}}(p) := \{e \mid e \in E_G \wedge \mathbf{own}(p, e)\}$, and we call it the *set*

[4] As already said, we assume that $N_G \cap E_G = \emptyset$.

of own edges of p, we let $\mathbf{own}(p) := \mathbf{own}_{\mathrm{N}}(p) \cup \mathbf{own}_{\mathrm{E}}(p)$, and we call it the *set of own elements* of p.

In a similar way, we define $\mathbf{exp}_{\mathrm{N}}(p)$, $\mathbf{exp}_{\mathrm{E}}(p)$, $\mathbf{exp}(p)$, $\mathbf{imp}_{\mathrm{N}}(p)$, $\mathbf{imp}_{\mathrm{E}}(p)$, $\mathbf{imp}(p)$. Since $\mathbf{own}$ is an ownership relation, we can define a function $\mathbf{owner} : N_G \cup E_G \to P_D$ where, for all $x \in N_G \cup E_G$, we let $\mathbf{owner}(x)$ be the unique package p such that $x \in \mathbf{own}(p)$.

The relation between the rules on page 51 and the definitions of FEHG's (Definition 3.14 and 3.15) are analogous to the case of EHG's (Definition 3.16 and 3.17). We now define the missing condition for FEHG's. It states constraints on owned, imported and exported nodes, according to the hierarchical structure of the packages.

Definition 3.17 (Visibility conditions for FEHG's)
Let $S = (G, D, B, \mathbf{own}, \mathbf{exp}, \mathbf{imp})$ be a tuple as in Definition 3.14. Then the *visibility conditions* for S state that:

1. For all $q \in P_D$ we have

$$\mathbf{imp}(q) \subseteq \bigcup_{q \succ_D r} \mathbf{exp}(r) \cup \bigcup_{p \succ_D q} \mathbf{own}(p) \cup \left(\bigcup_{p \succ_D q} \mathbf{imp}(p) \setminus \mathbf{exp}(q) \right)$$

   and for ρ_D we have

$$\mathbf{imp}(\rho_D) \subseteq \bigcup_{\rho_D \succ_D r} \mathbf{exp}(r)$$

2. For all $q \in P$ we have

$$\mathbf{exp}(q) \subseteq \mathbf{own}(q) \cup \left(\bigcup_{q \succ_D r} \mathbf{exp}(r) \cap \mathbf{imp}(q) \right)$$

   and, for all $e \in \mathbf{exp}_{\mathrm{E}}(q)$, we have that

$$\forall n \in N : q \succ_D^* \mathbf{owner}(n) \wedge I(e, n) \to n \in \mathbf{exp}_{\mathrm{N}}(q)$$

3. $\mathbf{exp}(\rho_D) = \emptyset$.

Intuitively, Definition 3.17 states that: (1) Every imported element of a package q must either be exported by one of the subpackages $r_1, \ldots, r_n$ of q, or be owned by one of the parent packages $p_1, \ldots, p_k$ of q, or imported into some p_i (but not imported from q itself). (2) The exported elements of a package q must either be owned elements of q or be imported from one of the subpackages $r_1, \ldots, r_n$ of q. Furthermore, for each exported edge of q all its attachment nodes that are owned by q or by some descendant

package of q, must also be exported. (3) The root package has no exported nodes or edges.

The condition on exported edges ensures that whenever we export an edge from a package p, its ends can also be imported by the parent packages. Notice that simply exporting the ends of that edge is not always possible, since some of them may be imported from some superpackage, and therefore cannot be exported. Therefore, it is necessary to check that the exported ends of an exported edge come originally from q or from a subpackage of q.

3.4 Typed Hierarchical Graphs

Typing is normally used to classify entities from a certain domain, according to common structure and/or behaviour. When we consider hierarchical graph typing, we first have to decide to which aspects of a hierarchical graph the notion of typing is applied.

In Subsection 2.3.9, we have made general observations on the desired features of a hierarchical graph type concept, which we recall here:

- A hierarchical graph contains an underlying graph, which can be typed.
- The hierarchy can also be typed. This means that we can establish a priori what kinds of packages can be present in a hierarchy, and how they are allowed to be nested.
- The coupling graph can also be typed. This models constraints such as which packages can contain which nodes or edges of the underlying graph, and to which nodes or edges a package can be anchored.

These are the dimensions according to which we define our notion of hierarchical graph typing.

Our decoupled approach to hierarchical graphs allows us to model the different kinds of typing above separately. This means that we combine a notion of typing on the hierarchy graphs with a notion of typing on the underlying graph and a notion of typing on the coupling graph.

Since the hierarchy and the coupling graphs are particular instances of directed graphs, we will first define typed directed graphs, and then typed hierarchies and typed coupling graphs. Typed directed graphs also provide an example of typed graphs that can be used as the underlying graphs, although in the general model, the type of these graphs is represented by a skeleton type (see further on in this section for more details). These three notions will be combined into the notion of a typed hierarchical graph.

The type of a directed graph allows us to express the fact that

1. nodes are classified according to (a finite set of) *node types*,

2. edges are classified according to (a finite set of) *edge types*,

3. node and edge types are compatible with each other, meaning that the graph type prescribes that edges of a certain type ε can only be incident in source and target nodes of appropriate types σ_ε, τ_ε,

4. edge cardinality respects constraints that are part of the type specification.

Notice that nodes and edges can be classified as in 1 and 2 by means of node and edge labels. For constraints like 3, however, we need more. We use a notion of typed graph commonly found in the graph grammar literature (see e.g. [CEMP96], [SWZ99]), which relies on *type graphs*, and *graph morphisms* to relate *instance graphs* to their type. This solution addresses points 1, 2 and 3: node and edge types in the type graph represent the classes into which nodes and edges of the instance graph are partitioned. A graph morphism maps nodes and edges to their types, ensuring that constraint 3 on nodes and edges be respected.

Type graphs and graph morphisms are however not enough to provide cardinality constraints on the edges of the instance graphs. These kinds of constraints are important for application areas like databases (see the entity relationship model, see e.g. [Che76]) and object-oriented modeling (see e.g. [RBP+91], [RJB99]), and are provided also by some graph data models, for example in Progres (see [SWZ99]). We provide these constraints by enriching type graphs with *cardinality constraints*, and requiring that a morphism from an instance graph to its type graph respect them. (See also e.g. [SWZ99]).

We now give the formal definition of graph homomorphisms, graph types and typed graphs. We begin with graph homomorphisms.

Definition 3.18 (Graph homomorphism)
Given a node alphabet Σ, an edge alphabet Δ, and two directed graphs $G = (N_G, E_G, s_G, t_G, l_G, \text{л}_G)$, $H = (N_H, E_H, s_H, t_H, l_H, \text{л}_H)$, labelled over these alphabets, a *graph homomorphism* (or *graph morphism*) $f : G \to H$ from G to H is a pair of functions $\langle f_\text{N} : N_G \to N_H, f_\text{E} : E_G \to E_H \rangle$, such that $f_\text{N} \circ s_G = s_H \circ f_\text{E}$, $f_\text{N} \circ t_G = t_H \circ f_\text{E}$, $l_G = l_H \circ f_\text{N}$, $\text{л}_G = \text{л}_H \circ f_\text{E}$.

We are now ready to define typed graphs.

Definition 3.19 (Typed graphs)
Let Σ be a node alphabet and Δ be an edge alphabet. Let $T = (N, E, s, t, l, \text{л})$ be a directed graph over Σ and Δ. Let $K := \{(x, y) \in \mathbb{N} \times (\mathbb{N} \uplus \{\infty\}) \mid x \leq y \vee y = \infty\}$ be the set of *cardinality constraints*. Given a finite set A, we indicate with $|A|$ its cardinality. Then a tuple $\tau = (T, \text{c}^\text{s}, \text{c}^\text{t})$ is a (*directed*) *graph type* if $\text{c}^\text{s}, \text{c}^\text{t} : E \to K$ are two functions associating to each edge its *source* and *target cardinality constraints*.

Let τ as above be a graph type, G be a directed graph, $f : G \to T$ a homomorphism, and let, for all $n \in N_G$, for all $\varepsilon \in E_T$,

$$\mathrm{adt}(n, \varepsilon) := \{m \in N_G \mid \exists e \in E_G : n = s_G(e) \wedge m = t_G(e) \wedge \varepsilon = f_{\mathrm{E}}(e)\}$$

and

$$\mathrm{ads}(n, \varepsilon) := \{m \in N_G \mid \exists e \in E_G : n = t_G(e) \wedge m = s_G(e) \wedge \varepsilon = f_{\mathrm{E}}(e)\}$$

be the sets of nodes *adjacent* to n in G as *targets* (resp. *sources*) of edges of type ε. Then, we say that G is *of type* τ if there exists a graph homomorphism $f : G \to T$, satisfying the cardinality constraints, i.e. such that, for all $n \in N_G$, we have

$$\begin{aligned}&\forall \varepsilon \in E_T : \forall x, y \in \mathbb{N} : \forall n \in N_G : \\ &\qquad f_{\mathrm{N}}(n) = s_T(\varepsilon) \wedge \mathrm{c}^{\mathrm{t}}(\varepsilon) = (x, y) \Rightarrow x \leq |\mathrm{adt}(n, \varepsilon)| \leq y\end{aligned}$$

and

$$\begin{aligned}&\forall \varepsilon \in E_T : \forall x \in \mathbb{N} : \forall n \in N_G : \\ &\qquad f_{\mathrm{N}}(n) = s_T(\varepsilon) \wedge \mathrm{c}^{\mathrm{t}}(\varepsilon) = (x, \infty) \Rightarrow |\mathrm{adt}(n, \varepsilon)| \geq x\end{aligned}$$

and, similarly, we have

$$\begin{aligned}&\forall \varepsilon \in E_T : \forall x, y \in \mathbb{N} : \forall n \in N_G : \\ &\qquad f_{\mathrm{N}}(n) = t_T(\varepsilon) \wedge \mathrm{c}^{\mathrm{s}}(\varepsilon) = (x, y) \Rightarrow x \leq |\mathrm{ads}(n, \varepsilon)| \leq y\end{aligned}$$

and

$$\begin{aligned}&\forall \varepsilon \in E_T : \forall x \in \mathbb{N} : \forall n \in N_G : \\ &\qquad f_{\mathrm{N}}(n) = t_T(\varepsilon) \wedge \mathrm{c}^{\mathrm{s}}(\varepsilon) = (x, \infty) \Rightarrow |\mathrm{ads}(n, \varepsilon)| \geq x\end{aligned}$$

We can interpret a hierarchy graph $D = (P_D, \succ_D)$ and a coupling graph $B = (P_D, A_D, C_B)$, as directed graphs $\overline{D} = (N_D, \succ_D, s_{\overline{D}}, t_{\overline{D}}, l_{\overline{D}}, \pi_{\overline{D}})$ and $\overline{B} = (P_B \cup A_B, C_B, s_{\overline{B}}, t_{\overline{B}}, l_{\overline{B}}, \pi_{\overline{B}})$ labelled over the alphabets $\Sigma = \Delta = \{\bot\}$, and defined in an obvious way. Then, a *typed hierarchy graph* is a hierarchy graph D such that $\overline{D}$ is a typed directed graph w.r.t.some type graph DT, and a *typed coupling graph* is a coupling graph B that such that $\overline{B}$ is a typed directed graph w.r.t.some type graph BT.

In Figure 3.3, we show the graph type for our running example (see also Figure 2.3 in Chapter 2). This graph contains four node types, namely `Page`, `Anchor`, `Project`, and `Site`, and five edge types, namely

- `Anchor` $-\lambda\rightarrow$ `Page`,
- `Anchor` $-\lambda\rightarrow$ `Anchor`,
- `Page` $-\sigma\rightarrow$ `Project`,
- `Anchor` $-\sigma\rightarrow$ `Page`,

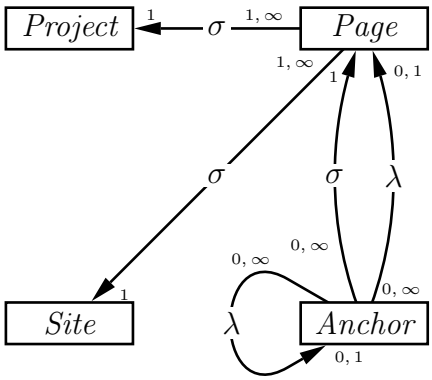


Figure 3.3: WWW type graph.

– `Page` $-\sigma\rightarrow$ `Site`.

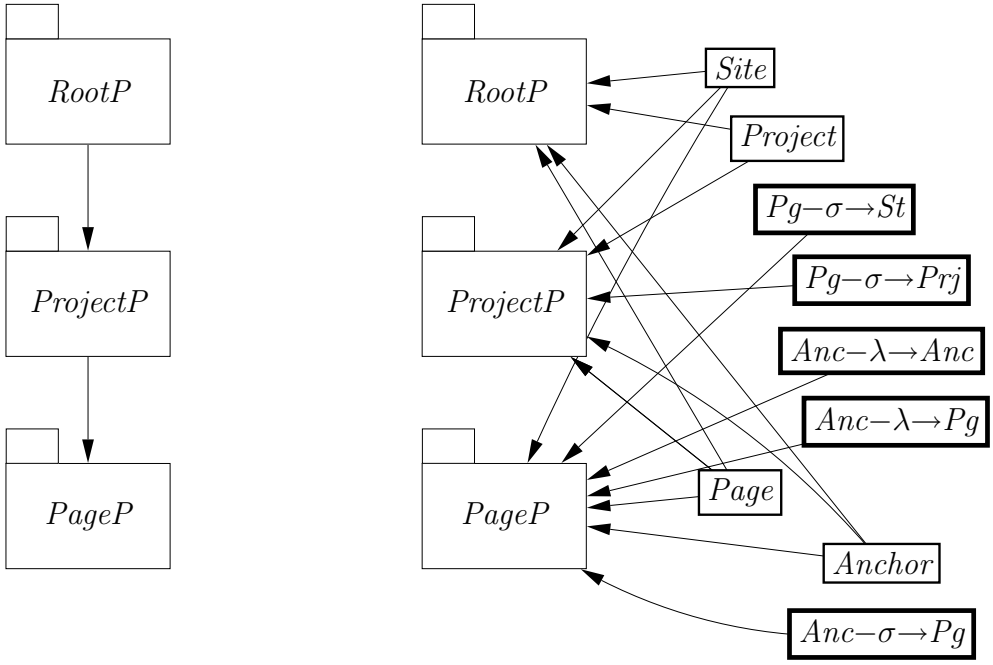
Page nodes in an instance graph are mapped to type `Page`, projects to `Project`, and so on. Structural edges are mapped to σ labelled edges in the type graph, and hyperlinks to λ labelled edges.

Cardinality constraints are written as pairs x, y ($x \in \mathbb{N}$, $y \in \mathbb{N} \cup \{\infty\}$) next to source and target of edges. A cardinality constraint of the type (x, x) ($x \in \mathbb{N}$) is written simply as x. For example, an anchor node *must* be linked to exactly one page through a structural (σ-labelled) edge (constraint "1"), and *can* be linked to one page through a reference (λ-labelled) edge (constraint "$0, 1$"). On the other hand, a page can have an arbitrary number (constraint "$0, \infty$") of anchors either with σ or with λ edges. When a cardinality constraint is omitted in the diagram of a graph type, we assume that the constraint is "$0, \infty$", i.e. that it poses no restrictions on the corresponding edge cardinality.

As further examples, in Figure 3.4, we display the type graphs for the hierarchy dag and for the coupling graph of our running example. The hierarchy type graph contains three types of packages, namely `PageP`, `ProjectP`, and `RootP`, and two edge types. We do not need labels here (we can assume that all nodes have the same label, as well as edges). The coupling type contains one node type for each package type in the hierarchy type, and one node type for each node and edge type in the underlying graph type (see again Figure 3.3). The latter have connections to the former, indicating when a graph element of a certain type can be contained in a certain package type. Node types in the coupling type that represent node (types) of the underlying graph are drawn with a thicker border.

In the running example we have the following connections:

- Sites should exist independently of any package, and therefore they can be put in the root package. Sites also need to be imported into page packages, in order to link the corresponding page to its site (recall the condition on forbidden edges in Definition 3.7 . A *page to site* edge is contained in the package of the corresponding page.



Hierarchy type graph Coupling type graph

Figure 3.4: Hierarchy and coupling graph types.

- A project node can be in the root package and in its own package. In this way its pages can be linked to it. In fact, *page to project* edges are in the package of the project they point to.
- *Anchor to anchor* and *anchor to page* link edges are contained in the package of the page to which the source anchor belongs.
- The target of a link must be imported into the correct page package, therefore such targets—namely anchors and pages—are contained in page, project, and the root package(s).
- An *anchor to page* structural edge must be contained in the package of the page owning the anchor.

In order to speak about hierarchical graph types, we still need an ingredient. In fact, we cannot embed a graph into a hierarchical graph as it is, we must first extract its skeleton from it. Likewise, we cannot assume a standard typing mechanism for all existing graph types, so that it is difficult to interface the type of the underlying graph

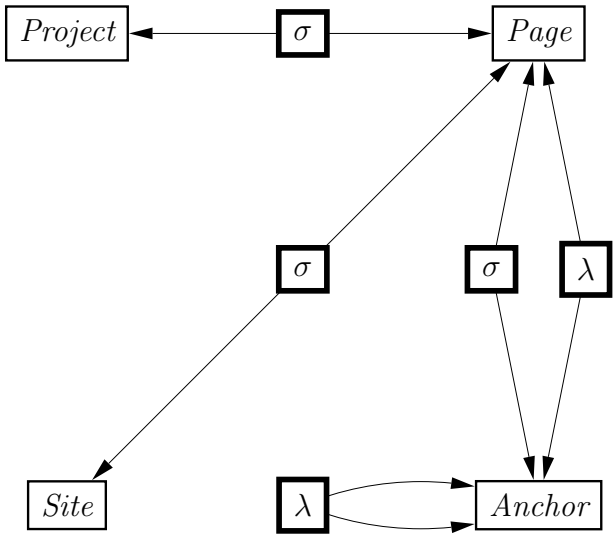


Figure 3.5: WWW skeleton type.

with the type of the coupling graph. Our solution is to introduce *skeleton types* (see Definition 3.20), and to assume that for a given class of graphs $\mathcal{G}$ such that, according to some typing mechanism, all $G \in \mathcal{G}$ have the "same type", we can provide a skeleton type T, such that all skeletons $S_G, G \in \mathcal{G}$ have the same type T.
In Figure 3.5, we show the skeleton type for our running example. This skeleton contains four node types (compare Figure 3.3), namely `Page`, `Anchor`, `Project`, and `Site`.

It also contains five edge types, depicted as nodes with thick borders, and using two kinds of labels to distinguish structural edges from link edges. The arrows in the picture indicate the incidence relation between node and edge types. Notice that, for reasons of space, we have tagged edges with their labels rather than with their names (compare this to the right part of Figure 3.4).

We are now ready to introduce skeleton types formally.

Definition 3.20 (Skeleton morphisms, typed skeletons)
Given two skeletons S and T, a *skeleton morphism* $f : S \to T$ is a pair of functions $\langle f_{\mathrm{N}} : N_S \to N_T, f_{\mathrm{E}} : E_S \to E_T \rangle$, where, for all $e \in E_S$, for all $n \in N_S$, $I_S(e, n) \Rightarrow I_T(f_{\mathrm{E}}(e), f_{\mathrm{N}}(n))$. A skeleton S is *of type* T, where T is a skeleton, if there exists a skeleton homomorphism $f : S \to T$.

We are now ready to introduce typed hierarchical graphs. A HG type will be a tuple, containing a skeleton type, a hierarchy type, and a coupling graph type.

Definition 3.21 (Typed hierarchical graphs)
A (*plain*) *hierarchical graph type* is a triple $\tau = (\tau_{\mathrm{S}}, \tau_{\mathrm{H}}, \tau_{\mathrm{C}})$, where τ_{S} is a skeleton, $\tau_{\mathrm{H}} = (T_{\mathrm{H}}, \mathrm{c}^{\mathrm{s}}_{\mathrm{H}}, \mathrm{c}^{\mathrm{t}}_{\mathrm{H}})$ is a hierarchy type, $\tau_{\mathrm{C}} = (T_{\mathrm{C}}, \mathrm{c}^{\mathrm{s}}_{\mathrm{C}}, \mathrm{c}^{\mathrm{t}}_{\mathrm{C}})$ is a coupling type, and $(T_{\mathrm{S}}, T_{\mathrm{H}}, T_{\mathrm{C}})$ is a (plain) hierarchical graph.

A hierarchical graph $H = (G, D, B)$ is *of type* τ if S_G is of type T_{S}, D is of type T_{H}, and B is of type T_{C}.

Notice that the fact that $(T_{\mathrm{S}}, T_{\mathrm{H}}, T_{\mathrm{C}})$ is a (plain) hierarchical graph ensures that the coupling type contains exactly the node, edge, and package types defined in the skeleton type and in the hierarchy type.

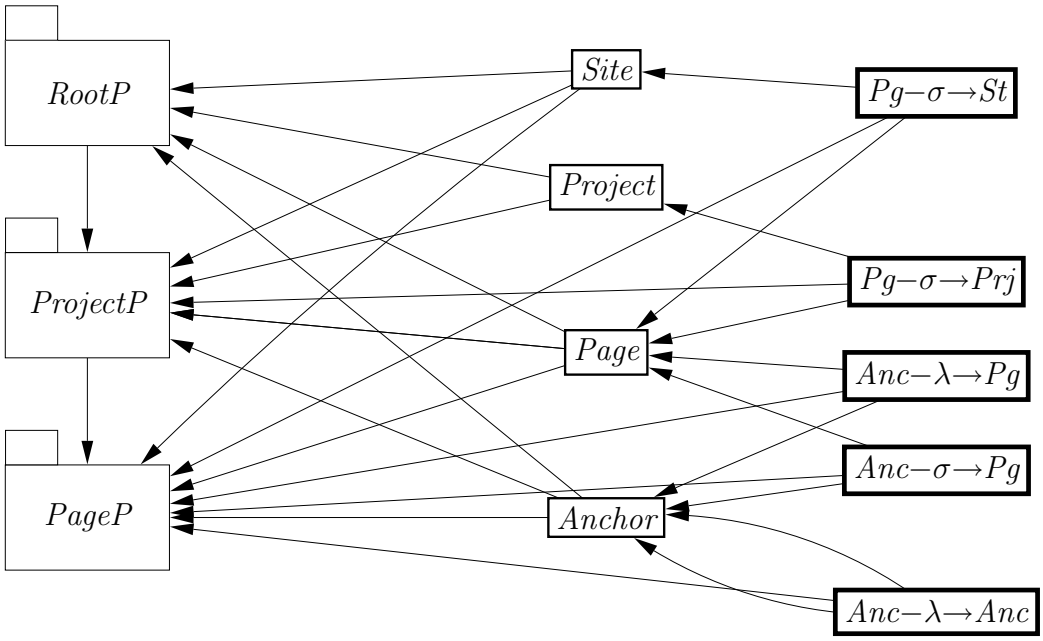


Figure 3.6: WWW HG type.

In Figure 3.6, we combine the notations for hierarchy types, coupling types and skeleton types into a notation for hierarchical graph types. We use three different notations for package types, skeleton node types, and edge types. The figure illustrates a hierarchical graph type for our running example.

Summarizing, in this section we have introduced a notion of *hierarchical graph types* and of *typed hierarchical graphs*. The former notion is a combination of a skeleton type, a hierarchy type, and a coupling type, in the spirit of our decoupled approach. We have explained these concepts by means of our running example.

3.5 Qualified Hierarchical Graphs

Up to now, we have considered hierarchical graphs where the coupling graph is loose, i.e. where packages are not anchored to any nodes or edges of the underlying graph. In this section, we briefly discuss non-loose coupling graphs.

As seen in Chapter 2, we have adopted a decoupling approach, i.e. we have separated the hierarchy information from the underlying graph, as one of the main design decision of our hierarchical graph model, even though many coupled approaches exist that force one to anchor *all* packages to some node or edge of the underlying graph.

We have not enforced this feature in our hierarchical graph model, because it is in general not needed, and sometimes even inappropriate. For example, a UML class diagram (see [RJB99]) can be structured by means of so-called *packages*, but packages have nothing to do with the underlying classes and associations: There are no classes that are related to packages, nor can packages be the source or target of associations in the class model.

In spite of this, in other situations it turns out to be useful to anchor packages to graph elements. In our running example this is very clear: project packages are anchored to project nodes and page packages to page nodes. As another example, in [DHP02] the hierarchy is coupled with frames, i.e. with special hyperedges.

This kind of anchoring is already provided by our model, through the $\approx$ relation that is associated to every coupling graph. Using this mechanism—which we call *qualification*, or *anchoring*—we can model the information that is implicitly stored in all coupling approaches, if this turns out to be useful. We say that a package is *anchored to* or *qualified by* a node or an edge. A hierarchical graph is *qualified*, if some of its packages are anchored to nodes or edges of its underlying graph. It is called *fully qualified* or, better, *tightly coupled*, if every package is anchored to a node or edge.

The hierarchical graph of our running example is qualified, but not fully qualified, since the root package is not qualified. All coupled approaches consider only fully qualified hierarchical graphs.

3.6 Summary

In this chapter, we have presented the formal definition of our hierarchical graph model. A *hierarchical graph* (HG) is the combination of an *underlying graph*, a *hierarchy dag* whose elements are called (*graph*) *packages*, and a *coupling graph*. The interface between the underlying graph and the hierarchy is provided by a *graph skeleton*.

We distinguish between (*plain*) *hierarchical graphs* and *full hierarchical graphs*: In the former nodes are distributed in the hierarchy, while edges are not; in the latter both nodes and edges are contained in the hierarchy.

We have also considered *boundary-crossing edges*, i.e. edges that link nodes contained

in different packages. Furthermore, we have considered *tree-like hierarchical graphs*, where the hierarchy graph is restricted to be a tree, and *strict hierarchical graphs*, which are tree-like HG's where nodes and edges are contained in exactly one element of the hierarchy. A strict HG contains no boundary-crossing edges.

We have extended hierarchical graphs to *encapsulated hierarchical graphs* (EHG) by introducing *ownership relations* and *import/export interfaces*, which can be seen as a particular mechanism for defining the coupling graph of a hierarchical graph. EHG's provide encapsulation in that nodes and edges from the underlying graph can be hidden inside the hierarchy.

We have also introduced *hierarchical graph types*. These are a combination of *skeleton types*, *hierarchy graph types*, and *coupling graph types*, and allow to express structural constraints on hierarchical graphs.

Finally, we have discussed *qualification*, a mechanism to associate packages to graph elements. This turns out to be useful when modeling certain domains, and it is the default for coupled approaches to hierarchical graphs.

Chapter 4

Introduction to Hierarchical Graph Transformation

In the previous chapters, we have considered hierarchical graphs as a data model to represent complex structures in various application areas within computer science, and we have concentrated on their static aspects, i.e. on what kind of information they allow to model, and how. The other important aspect of a data model concerns the operations that it permits to define. This is the topic of Chapter 4 and of Chapter 5.

As a basis for the definition of operations on hierarchical graphs we adopt graph transformation systems (GTS), a rule-based approach to the specification of graph transformation that has been developed over the last thirty years (see Section 4.1 below or, for a thorough treatment of the subject, the three volumes of the *Handbook on Graph Grammars and Computing by Graph Transformation* [Roz97], [RE+99a], [RE+99b]). While graphs allow to model complex structures in an intuitive way, graph transformation systems add a description of the transformations/behaviour of such structures, thus making them applicable to a wide variety of situations. In fact, GTS's have several applications both in theoretical and in practical computer science, for example the generation of sets of graphs, the specification of data structures and their operations, the specification of CASE tools, and even in biology for modeling plant growth. In this thesis, we are interested in hierarchical graph transformation as an extension of an extension of rule-based graph transformation.

As already discussed in Chapter 2, one of the main goals of our hierarchical graph model is to abstract from particular aspects of already existing approaches. In fact, although there are several ways to define hierarchical graphs and their transformations, we are convinced that different approaches embody (and hide) common concepts, which we try to identify and to make explicit. For these reasons, in the static part of our model, we have decoupled the hierarchy from the underlying graph and made hierarchical structuring independent of the different kinds of graphs to which it is applied. Following this philosophy, we would like to consider hierarchical graph transformation as the

combination of traditional graph transformation and of a new orthogonal concept added to it.

As a consequence, we do not aim at the definition of a particular hierarchical graph transformation approach, but at the development of a framework that allows to lift (flat) graph transformation to hierarchical graph transformation. This framework should carry the following benefits with it:

1. It should grasp the common concepts upon which hierarchical graph transformation, as known from the literature, is based.

2. It could provide a basis for the development of new approaches to hierarchical graph transformation.

3. It should help to compare existing approaches to hierarchical graph transformations.

In Chapter 9, we will discuss to what extent our framework fulfills these requirements.

Since hierarchical graphs are represented as a combination of different graphs (underlying graph, hierarchy graph, coupling graph), we specify hierarchical graph transformation as a combination of graph transformations. This has the following consequences:

- One can reuse an existing *flat graph* transformation approach—possibly as it is—to define transformations of the underlying graph, of the hierarchy, and of the coupling graph. This means that no new graph transformation concepts need to be invented.

- One can use different approaches to graph transformation for the different components of a hierarchical graph. This flexibility can be useful in certain situations, as suggested in Subsection 7.1.5.

- Hierarchical-graph transformation can easily be seen as an extension of ordinary graph transformations, namely of transformations on the underlying graph. Therefore we can say that the hierarchy information and its transformations are added orthogonally.

Summarizing, our framework will allow to combine existing graph transformation approaches in order to obtain hierarchical graph transformation approaches (see Section 4.3 and Chapter 5 for more details).

After further motivating and describing our framework in the present chapter, we define it formally in Chapter 5. In Chapter 6, we consider a case study on how it can be instantiated using the double-pushout approach to graph transformation (see point 2 above). In Chapter 7, we use the framework to classify and compare existing approaches to hierarchical graph transformation (see point 3 above).

This chapter has the following structure. In Section 4.1, we give a brief overview on graph transformation systems and on their most important concepts. In Section 4.2, we classify existing approaches to graph transformation and try to apply the same classification to hierarchical graph transformation approaches. In Section 4.3, we collect the requirements for our framework, and discuss informally the concepts that will be formalized in Chapter 5. In Section 4.4, we provide a summary of the whole chapter.

4.1 Graph Transformation

In this section, we illustrate the general idea of rule-based graph transformation, and we give a taste of different approaches to it. The interested reader can find a thorough treatment of graph transformation (systems), as well as a good starting point in the related literature, in [Roz97], [RE+99a], [RE+99b].

Graph grammars are one of the first examples of GTS's, and they are a good starting point for illustrating graph transformation. They have been developed originally as an extension of string grammars to graphs: Like Chomsky grammars are based on the substitution of substrings, graph grammars use graph substitution as the basic mechanism to derive new graphs from given ones. In the last thirty years graph grammars have found several applications both in theoretical and in practical areas of computer science. Examples are the description of graph languages (see e.g. [ER97], [DHK97]), picture generation [DK99], the specification of data structures in software systems (e.g. [SWZ99], [NZ00]), the modeling of plant growth (see e.g. [RS92]), and so on.

The basic means for defining graph substitution is a *graph transformation* (or *graph rewrite*) *rule*. Several rules can be combined into more complex devices, called *graph transformation systems*, of which graph grammars are a particular case. In spite of this common idea of graph substitution, several different approaches have emerged in the literature, motivated by different application areas. These approaches differ for the kinds of graphs that they use (e.g. directed/undirected, labelled/unlabelled, ordinary graphs/hypergraphs, and so on), and for the particular rewrite mechanisms by means of which they define a graph transformation step (different approaches to graph transformation are considered in Section 4.2).

Graph transformation rules and graph transformation systems are the main topic of this section: The former are considered in Subsection 4.1.1, while the latter are dealt with in Subsection 4.1.2. The presentation is derived from that of [AEH+99] and in particular from Section 2. Our presentation here is however less detailed, and considers only the aspects that are relevant to us.

4.1.1 Graph transformation rules and direct derivation

Graph transformation stems from the combination of two well-known concepts in computer science:

1. The use of graphs as a model for complex structures, like flow-charts, entity relationship diagrams, class diagrams in object orientation, and so on. Graphs combine a formal description of a system with an intuitive graphical representation, making them applicable to many different areas.

2. The use of rules to specify complex structures or computations, like for example in language syntax definition, in logic programming languages, in functional programming language implementation, in expert systems, and several others. As far as computations are concerned, rule-based approaches favor a declarative style of specification, as opposed to the more traditional imperative paradigm.

Rule-based graph transformation combines these two ideas into one computational paradigm, and allows to define local transformations on graphs in an intuitive, yet mathematically precise way.

Graph rules are a generalization of string rewrite rules to the graph case[1]. A string rule is a pair of strings $r = \langle x, y \rangle$, and the application of r to a string α consists in decomposing α (if possible) into substrings uxv, replacing x by y and producing the string $\beta = uyv$. In a similar way, graph transformation is based on the substitution of subgraphs in a given host graph, but while in the case of strings there is *one* natural way to perform this substitution, in the case of graphs we have many possibilities (as we will see in Section 4.2). This has led to different approaches to graph transformation, motivated by the applications for which they were used.

It is not our intention to describe such approaches in detail (see e.g. [Roz97]) but we want to give the reader a taste of how a graph transformation system operates. To this purpose, we have chosen a concrete—although quite general—notion of graph rewrite rule and graph transformation, which is used in [AEH+99] (see also [KR90]) and which generalizes several approaches to graph transformation. We dedicate the rest of this section to the illustration of graph transformation using this approach.

In this section, a *graph* is a node- and edge-labelled directed graph, as already defined in Definition 3.3 in Chapter 3. A *subgraph* of a graph H is a graph G such that $N_G \subseteq N_H$, $E_G \subseteq E_H$, and s_G, t_G, l_G and λ_G are the restriction to G of the corresponding functions of H. Given two graphs G and H, a *homomorphism* $f : G \to H$ from G to H is a pair $\langle f_{\mathrm{N}}, f_{\mathrm{E}} \rangle$ of functions $f_{\mathrm{N}} : N_G \to N_H$ and $f_{\mathrm{E}} : E_G \to E_H$, compatible with the source, target, and labeling functions. When there exists a homomorphism $f : G \to H$ from G to H, we say that G has an *occurrence* in H.

[1] Notice also that edge-labelled graphs can be considered a natural extension of strings, since strings can be represented as linear, edge-labelled graphs.

A *graph rewrite rule*[2] is a tuple $r = (L, K, R, glue, emb, appl)$, where L, K and R are three graphs, called the *left-hand side*, the *interface*, and the *right-hand side* of the rule; K is a subgraph of L; $glue : K \to R$ is a homomorphism; $emb \subseteq N_L \times N_R$ is a relation between the nodes of the left-hand side and the nodes of the right-hand side, and *appl* is a set of *application conditions*[3]. Some authors (see e.g. [ER97]) call L the *mother graph* and the right-hand side R the *daughter graph*.

An *application* of rule r to a graph G (the *host graph*) is performed according to the following steps:

1. CHOOSE L in G, i.e. find an occurrence (also called a *match*) $m : L \to G$ of L in G.

2. CHECK the application conditions of r on G and m.

3. REMOVE $L - K$ from G, as well as all the resulting dangling edges. The resulting graph D is called the *context graph*.

4. GLUE R and D in K, i.e. construct the disjoint union of R and D and identify the nodes and edges which have a common preimage in K via *glue* and m respectively. The resulting graph E is called the *gluing graph*.

5. EMBED R in D according to *emb*, i.e. for every edge linking a node $u \in N_D$ with the image $m(v) \in N_G$ of a node $v \in N_L$, establish an edge with the same label, linking u to a node $w \in N_R$ iff $(v, w) \in emb$.

If such a *derivation step* is successful, we say that H is (*directly*) *derived* from G through r and we write $G \Rightarrow_r H$.

We illustrate graph rewrite rules by considering again our web example (see Chapter 2), and specifying a simple transformation on the web structure. The transformation consists in splitting an existing page with three internal anchors into two pages, and assigning two anchors to the former page, and one to the latter. This transformation can be specified by the rule depicted in Figure 4.1. This rule contains the following elements:

- The left-hand side L is a graph with four nodes. The notation '1: *page* indicates that the node's identifier is 1, that its label is *page*, and the fact that it belongs to the left-hand side (the symbol "'"). The same notation is used for the other nodes. Edges are labelled with the symbol σ (they are *structural* edges, linking anchors to their pages).

[2]The terms *graph transformation rule* and *production* are also used.
[3]The notion of application condition is not further formalized.

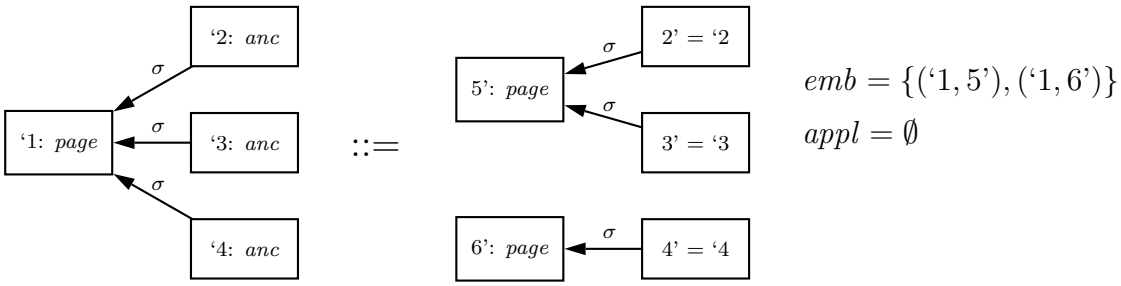


Figure 4.1: A graph transformation rule.

- The right-hand side graph R is separated from the left-hand side by the ::= symbol. The graph R contains two nodes with label *page*, three nodes with label *anc*, and three σ-labelled edges. The notation 5': *page* indicates that the upper left node has identifier 5, label *page*, and that it is a new node. The lower left node is a new node likewise. The notation 2' = '2 indicates that the upper right node is *not* a new node, but it is the same as the corresponding node in the left-hand side[4].

- The interface graph K and the *glue* homomorphism are not represented explicitly in the picture. However, the notation 2' = '2 can be read: "The node in the left-hand side with identifier 2 is also a node of K and it is mapped by *glue* to the upper right node in the right-hand side." Therefore, K has three *anc*-labelled nodes and *glue* maps them to the right-most nodes of R.

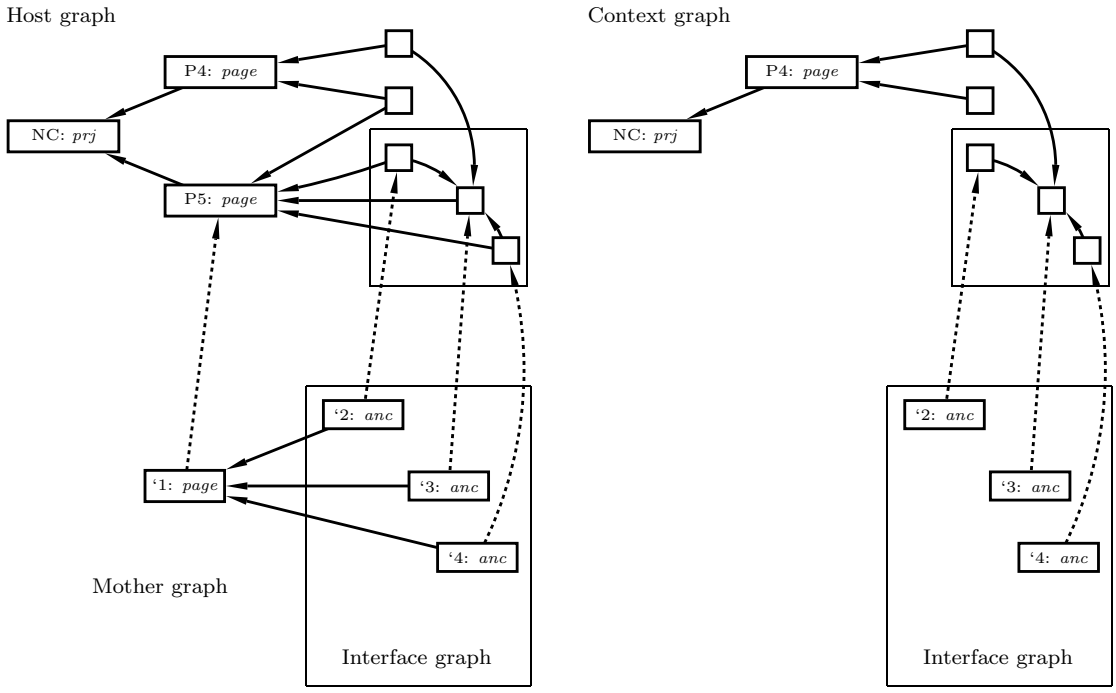
- The rule has an embedding relation $emb = \{(\text{'}1, 5\text{'}), (\text{'}1, 6\text{'})\}$, indicating that possible removed edges that were incident in the left-hand side page node must be redirected to both newly created page nodes[5]. Finally, there are no application conditions.

We now illustrate rule application by applying this rule to a graph derived from our web example (see Figure 2.1). The graph consists of project `NC` and two pages, `P4` and `P5`. The rule will split page `P5` into two new pages, `P5'` and `P6`.

The first step (CHOOSE) of the application of the rule is illustrated in Figure 4.2.a. In the upper part of the figure we can see the host graph, while the left-hand side of the rule (the mother graph) is depicted in the lower part. The interface graph is highlighted by means of a rectangular frame. Edge labels are not shown. The dashed arrows from

[4]This notation is adapted from PROGRES, see e.g. [SWZ99].

[5]This ensures that the new pages be added to the same project as the old page, and that all hyperlinks pointing to the old page in the old graph point to both of the new pages in the new graph.



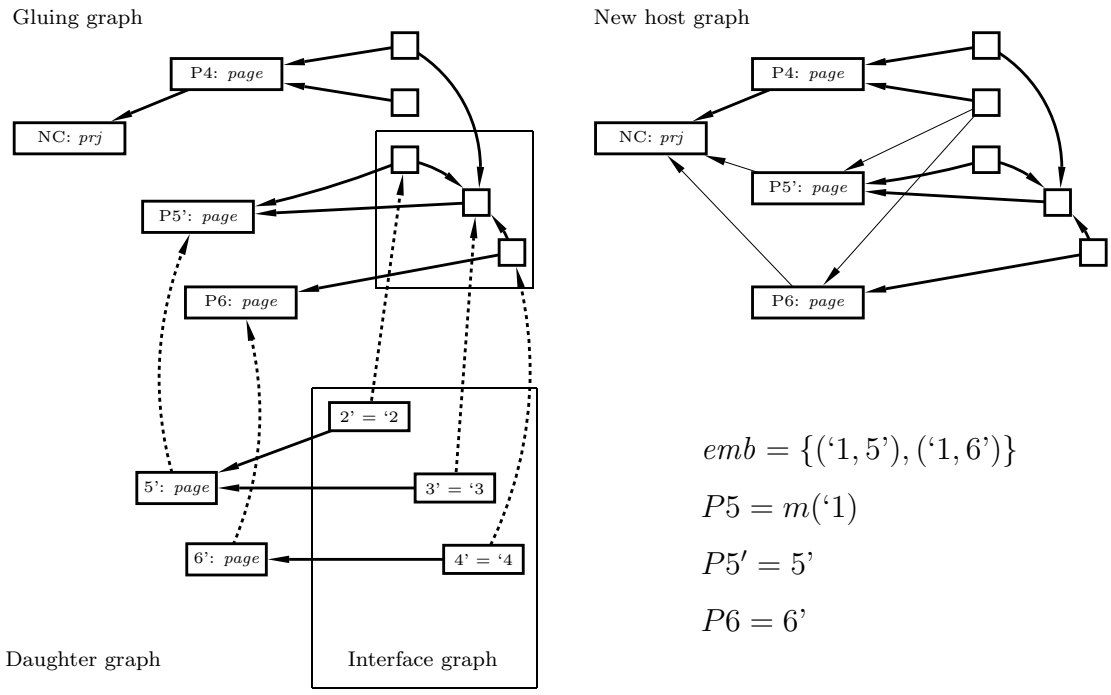
a) CHOOSE b) REMOVE

Figure 4.2: Direct derivation: MATCH, (CHECK) and REMOVE.

the mother graph to the host graph represent the occurrence morphism for this rule application. The arrows between edges of the mother graph and edges of the host graph are omitted. However, we assume that the edge between '2 and '1 is mapped to the edge between the corresponding nodes of the host graph. The other edges are mapped to edges of the host graph in a similar way.

The rule has no application condition, so the corresponding step in the rule application is trivial. We then consider the third step, which is illustrated in Figure 4.2.b. During this step (REMOVE), all the nodes of the host graph that are matched by nodes in the mother graph, but not by nodes of the interface graph are removed. The edges incident in the removed nodes are removed as well. The resulting graph, the context graph, is depicted in the upper part of the figure.

In the fourth step (GLUE), a copy of the daughter graph is glued to the context graph along the interface graph. This is depicted in Figure 4.3.c, where the gluing graph is shown in the upper part, and the daughter graph in the lower part. The arrows between the daughter graph and the gluing graph indicate the correspondences between nodes of the former and nodes of the latter graph.



c) GLUE d) EMBED

Figure 4.3: GLUE and EMBED.

In the fifth and last step (EMBED), edges are established in the gluing graph between the nodes that belong to the context graph and nodes that belong to the daughter graph (but not to the interface graph). In our example (see Figure 4.3.d), the embedding relation specifies that for each removed edge that was incident in the removed node matched by '1, we should establish a new edge incident in the copy of 5' in the gluing graph, and a new edge incident in the copy of 6'. Thus, two new edges linking the two new pages to their project are established, and two new hyperlinks are established from the lower anchor of page `P4` to `P5` and `P6`.

This completes our illustration of graph rewrite rules and their application to graphs.

4.1.2 Graph transformation systems

While a rule allows to define a direct derivation step on a graph, in order to define more complex computations we normally need to perform several such steps, possibly applying different rules. We then need to use several rules together, and to provide means to control their application.

A *graph transformation system* T is a first step in this direction: It consists of a set of rules R, which can be freely applied to a given graph, i.e., at any derivation step, we can choose which rule $r \in R$ is applied. A *graph grammar* is a graph transformation system that additionally provides a start graph S and distinguishes between *terminal* and *non-terminal* node and edge labels. In a graph grammar all derivations begin with the start graph, and end when (and if) we derive a graph which is labelled only with terminal labels. Graph grammars are particularly suited when we want to generate sets of graphs and study the properties of the generated graphs, or if we want to parse existing graphs and check whether they belong to the language of a certain grammar (see e.g. [ER97], which deals with node-replacement grammars, and [DHK97], dedicated to hyperedge-replacement grammars).

In a wider context of application, where we want to use graph transformation for programming generic computations, we may need more complex ways to control the derivation process. For example we can specify a *set* of initial graphs, instead of *one* initial graph, so that a computation can be applied to a domain of possible input states. Terminal graphs can also be specified in more general ways than using terminal labels, for example by specifying some property that they have to satisfy. Another important extension concerns the control of rule application: For example, we can specify that only certain sequences of rule applications are allowed, leading to the notion of a *programmed graph transformation system*. More on this topic can be found in the cited [AEH+99] and in [Sch97].

Another example of programmed graph transformation systems can be obtained by using *transformation units* (see e.g. [KK96]).

4.2 Approaches to Hierarchical Graph Transformation

In this section, we will consider approaches to hierarchical graph transformation, try to identify common features that we would like to support in our framework, and decide which other features are specific to particular approaches and should remain so.

As a first step, we provide a classification of (flat) graph transformation approaches, and then we try to fit existing approaches to hierarchical graph transformation into this classification. As we will see, there exist deep differences between different approaches, both in the flat and the hierarchical case. This justifies our choice of a framework that leaves the choice of a specific approach open.

This section is structured as follows. In Subsection 4.2.1, we introduce a classification of flat-graph transformation approaches. In Subsection 4.2.2, we apply this classification to existing hierarchical graph transformation approaches. In Subsection 4.2.3, we make some concluding remarks.

4.2.1 Approaches to graph transformation

The aim of this subsection is twofold. Firstly, we want to give the reader a feeling of the different possibilities to define graph transformation. To this purpose, we sketch and classify the most important approaches known in the literature. Secondly, by illustrating the—often deep—differences between various approaches, we would like to convince the reader that it is difficult to pick one for the definition of a generic notion of hierarchical graph transformation. This is a further motivation for the alternative choice of a generic framework where one can plug in his or her own favorite graph transformation approach.

As a first step, we introduce a classification of graph transformation approaches into two broad categories or paradigms, according to the interpretation of a graph that they adopt. In the *gluing* paradigm, a graph is considered a set of (hyper-)edges, glued together by nodes, and graph transformation is essentially based on the substitution of one or more edges. In the *connecting* paradigm, a graph is considered as a set of nodes connected by edges, and graph transformation is based on the substitution of one or more nodes.

We can characterize the two paradigms w.r.t. several aspects, as illustrated below (cf. Table 4.1, which has been adapted from [BJ96]):

- **Substitution.**
  In the gluing paradigm we speak about *edge replacement* (or *hyperedge replacement* if we use hypergraphs): An edge in the host graph is chosen, and it is replaced with a new graph, the daughter graph. In more complex situations, the mother graph consists of a set of edges (together with their attachment nodes) rather than of a single edge.

  In the connecting paradigm we speak about *node replacement*: One node is chosen and replaced with the daughter graph. In some approaches the mother graph can be more complex than a single node and contain a set of nodes with edges between them.

- **Interface.**
  During graph transformation, the host graph is split into a context graph and a mother graph. The mother graph is removed and replaced by the daughter graph (compare Subsection 4.1.1). In the gluing paradigm, the context and the mother graph are glued together by *interface nodes*, which are preserved when the mother graph is removed. In the connecting paradigm, the context and the mother graph are connected by *interface edges*, which are deleted when the mother graph is removed.

- **Connection.**
  After removing the mother graph from the host graph and inserting the daugh-

ter graph, we need to *connect* the daughter graph to the context graph. Gluing approaches use the preserved interface nodes to glue the daughter graph to the context graph, while connecting approaches establish new edges between the daughter graph and the context graph.

Most approaches found in the literature fall in either the gluing or the connecting paradigm. Here are some examples of gluing approaches (see again Table 4.1):

- *Hyperedge-replacement (HR) graph grammars*, based on the substitution of a hyperedge with a graph in a host graph. The reader can find a presentation of these grammars e.g. in [DHK97]and in [Hab92].

- *The categorical approaches* to graph grammars, named according to the formal framework used to define graph transformation. The most important categorical approaches are *double-pushout* and *single-pushout grammars* (see [CMR$^+$97], [EHK$^+$97]). In Chapter 6, we will use the double-pushout approach for defining an approach to hierarchical graph transformation by instantiating our framework.

Examples of connecting approaches are:

- *Node replacement graph grammars*, among these *node label controlled (NLC) grammars*, *neighborhood controlled embedding (NCE) grammars*, and variations of these two approaches. A thorough presentation of node replacement is given in [ER97]. A proposal for a categorical framework for node rewriting based on pullback rewriting can be found in [BJ96].

- PROGRES, a language and a system for the specification and prototyping of software systems, also uses the connecting approach, but it allows more complex mother graphs (more than one node). This approach is described e.g. in [Sch97]and in [SWZ99].

These examples are summarized in Table 4.1. The approach that we have sketched and used in Subsection 4.1.1 integrates aspects of both the gluing and connecting paradigms, since the replacement process involves both gluing nodes and connecting (embedding) edges.

This brief overview gives an impression of the variety of existing approaches to graph transformation. This variety concerns first of all the way in which one thinks about graph transformation. For example, node replacement may turn out to be more concise in certain applications, as it is the case in the web example of Subsection 4.1.1, which can be considered a pure node-substitution approach if we suppose that anchor nodes are deleted and re-created, together with their incident edges. In other applications, for example in picture generation (see [DK99]), (hyper-)edge replacement may be better suited. Secondly, different approaches have different expressive power, in terms of the operations on graphs that they allow to express, or of their generative power in the

	Hyperedge replacement	*Node replacement*
Graph	Set of edges glued by nodes	Set of nodes linked by edges
Substitution	A (hyper-)edge pattern	A node pattern
Interface	A set of nodes	A set of edges
Connection	Glues interface nodes	Creates interface edges
Examples	HR [DHK97] DPO [CMR$^+$97] SPO [EHK$^+$97]	NCE [ER97] pullback rewr. [BJ96] PROGRES [SWZ99]

Table 4.1: Hyperedge versus node replacement

case of graph grammars. A lot of literature has been written comparing the generative power of different classes of graph grammars, and the reader can find good introductory texts in [ER97], [DHK97], [Cou97].

Due to this variety, it is difficult to pick one approach as *the* canonical way of doing graph transformation. In the next subsection, we show how this observation can also be applied to the case of hierarchical graphs.

4.2.2 A classification of hierarchical graph transformation approaches

In this subsection, we try to apply the criteria used in Subsection 4.2.1 to the classification of existing hierarchical graph transformation approaches. Note that while the classification of traditional graph transformation approaches is well-known and established, what we present here is, up to our knowledge, new. The purpose of this subsection is to show that:

- Even though hierarchical graphs are more complex structures than graphs, the principle of graph rewriting, consisting in substituting a substructure with some new structure, can still be applied to them. Furthermore, similar problems of finding a suitable embedding arise.

- The diversity of approaches suggests that it is difficult to choose one as the canonical way to transform hierarchical graphs. This is in favor of our generic framework, although a possible alternative would be to integrate all approaches in one—analogously to what has been done for flat graphs in [AEH$^+$99] (see again Section 4.1 and compare [KR90]).

We will first discuss the single approaches one by one, and then summarize our discussion in Table 4.2. Next to the classification criteria identified in Subsection 4.2.1, we also point out whether a given approach is coupling or de-coupling in our sense. (Compare Subsection 2.3.4.)

H-graphs

In [Pra79], hierarchical graphs are used to model the runtime data structures inside a software system. Hierarchical graph transformation is used as a syntax-directed method to generate the initial state of such a system: A (context free) string grammar describing the syntax of a language is paired with a hierarchical graph grammar, such that to each construct of the language it is possible to associate a hierarchical graph corresponding to the runtime structures needed to support that construct.

Hierarchical graphs are directed, node and edge labelled and the hierarchy is built by allowing node labels to contain nested hierarchical graphs, implying that this approach is coupling. Hierarchical-graph transformation is based on node replacement: A node in a hierarchical graph is rewritten to a new hierarchical graph, which is embedded in the context graph by establishing new edges to special nodes of the right-hand side of a rule, called the *input* and the *output* nodes. We conclude that this is a connecting, coupling approach.

The AGG approach

In AGG, a tool developed at the Technical University of Berlin (see e.g. [LB93] and [AEH+99, Subsection 3.2.1]) for creating and animating graph transformation systems in the DPO or SPO approaches, graphs are represented as collections of *atoms* (which can model nodes or edges), together with three kinds of edges: *source* and *target edges* for linking an edge to its ends, and *abstraction edges* to link an atom to its parent atom in the abstraction hierarchy. The abstraction hierarchy allows to define a hierarchical structuring of a graph, grouping subgraphs into nodes, and bundles of parallel edges into edges.

The transformations on such flat graphs are defined by means of SPO rules. Therefore, in this approach hierarchical graphs have a flat (coupled) representation, and the transformation uses a gluing approach.

Distributed systems

Distributed systems have been modeled using graph transformation in [Tae96]. A distributed system is seen as a network whose nodes contain an internal graph describing its state, while network edges model the connections/communications between the different nodes of the network. This approach can be seen as a primitive kind of hierarchical graph, where components of the network are linked to each other by interfaces, but do not form a strictly hierarchical structure (compare the H-graph approach mentioned above). This approach is decoupling, since the local graphs and the network graph are distinct structures.

This approach is based on the DPO approach and therefore it is a gluing approach. Transformations are defined by combining network rules, which transform topological

structure of a distributed system, with local rules, which transform the local graphs.

Multilevel graph grammars

Multilevel graph grammars (see [PP95]) are a model of hierarchical graph and hierarchical graph transformation based on the double-pushout approach to graph transformation. In this approach, a hierarchical graph is obtained by folding a subgraph of a given graph into a new (smaller) graph by applying a graph transformation rule. This process can be applied several times in order to obtain a hierarchical structure. By applying rules in the reverse order we can restore the original graph. Double-pushout rewriting rules are extended in order to be compatible with the folding.

In this approach, the hierarchy is coded in a sequence of productions that must be applied in order to obtain a folded graph from a flat graph, and therefore we can consider it a decoupled approach. Hierarchical graph transformation uses a variation of the double-pushout approach and therefore belongs to the gluing paradigm.

Graph package systems

The model introduced in [BEMW00] uses a decoupled representation of hierarchical graphs (it was a first attempt in the direction of the framework presented in this thesis). The underlying graph is directed and the hierarchy is a tree of packages. That model also introduces a framework for defining hierarchical graph transformation starting from a given graph transformation approach. This framework is not instantiated to any particular graph transformation approach, and therefore it is not possible to fit it in the classification introduced in Section 4.1.

Hierarchical hypergraph transformation

Another DPO-based approach to hierarchical graph transformation can be found in [DHP02]. In this paper, hierarchical hypergraphs are defined recursively, by using special hyperedges called *frames* which in turn contain hierarchical graphs. This is therefore a coupled approach.

The DPO approach is extended to the category of hierarchical graph, where morphisms map local graphs of one hierarchical graph to local graphs of another hierarchical graph in a way that it is compatible with the two hierarchies. This approach is therefore, again, a gluing approach.

Aggregated graphs

In [EH00], a DPO-based hierarchical graph model for software system modeling is presented. In this model, the hierarchy information is obtained by considering certain

Approach	*Graph*	*Substitution*	*Interface/connection*	*Coupling*
H-graphs [Pra79]	Labelled-directed nesting by labels	Single node	Creates edges to I/O nodes	YES
AGG [AEH+99]	Atoms, *src*, *tgt*, *abstraction* links	Edges	Glues intf. nodes	YES
Distr. sys. [Tae96]	Network of local graphs	Local and netw. edges DPO based	Glues intf. nodes	NO
ML gramm. [PP95]	Global graph folding rules	DPO based	Glues intf. nodes	NO
GPS's [BEMW00]	Global graph pack. hierarchy	—	—	NO
H. hyperg. [DHP02]	Hypergraph hierarchy in hyperedges	DPO based	Glues intf. nodes	YES
Agg. graphs [EH00]	Directed graph aggregation edges	Edges	Glues intf. nodes	YES

Table 4.2: Classification of HG transformation approaches

edges as *aggregation edges*. Apart from this, double-pushout typed graph transformation (as described e.g. in [CEMP96]) is used. Thus this is a coupled approach, which uses a gluing approach for both the hierarchy and the underlying graph.

4.2.3 Summary

In this section, we have described a well-known classification of graph transformation approaches, which we have subsequently applied to hierarchical graph transformation approaches. We can now make the following considerations:

- Hierarchical-graph transformation, as known from the literature, is based on similar concepts as (flat) graph transformation.
- Graph and hierarchical graph transformation approaches can differ w.r.t. their generative power, and—even more important from a modeling point of view—for the way they allow one to think about and to work with graphs and their transformation.

Again, these considerations are in favor of our choice for a framework where hierarchical graph transformation is defined in a flexible, approach-independent way.

As announced in the introduction to this chapter, Chapter 7 is dedicated to comparing—by means of our framework—different approaches to hierarchical graph

transformation. There the reader can find a more in-depth discussion of the approaches mentioned above.

4.3 A Generic Model of Hierarchical Graph Transformation

In this section, we will outline our model of hierarchical graph transformation in an informal way, i.e. stressing mostly the ideas that motivate it and the intuition behind the introduced concepts. We postpone formal definitions to Chapter 5. As already discussed in the introduction to this chapter, our hierarchical graph model aims at abstracting from approach-specific aspects and at identifying common concepts of existing (and possibly of future) hierarchical graph transformation approaches. We want to specify hierarchical graph transformation as a combination of several graph transformations. This has the following consequences:

- One can reuse an existing *flat graph* transformation approach—possibly as it is—to define transformations of the underlying graph, of the hierarchy, and of the coupling graph. In other words, one need not invent a new approach to graph transformation when dealing with hierarchical graphs.

- One can use different approaches to graph transformation for the different components of a hierarchical graph. For example, when modeling hypertext structures, transformations on the underlying graph can be better modeled with some connecting graph transformation approach (see e.g. the remark at the end of Subsection 4.2.1), while the hierarchical structure may be appropriately transformed using some gluing approach.

An important ingredient for our framework is the concept of a *graph transformation approach*, as described in [KK96]. This notion considers the common features of many different graph transformation approaches, and allows to speak about graphs, rules, and graph transformation at a high level of abstraction. Hierarchical graph transformation rules are then triples of graph transformation rules, and their semantics is derived from that of the component rules using a standard construction.

In order to coordinate the application of such triples rules, we introduce the notions of a *tracking graph transformation approach*, of *coordinated rules* and *coordinated hierarchical graph transformation* (see further in this section). If we compare our model with other existing approaches, we see that coupled ones (e.g. hierarchical hypergraph transformation) do not need coordination (it is built-in), while decoupled ones provide it through some ad-hoc mechanism (e.g. in distributed systems). The additional complexity of our model is compensated by its generality and approach-independence.

4.3.1 A framework for hierarchical graph transformation

Since we want to develop a concept of hierarchical graph transformation where we allow to reuse several existing graph transformation techniques, we look for some abstract notion of graph transformation that allows us to take existing graph transformation approaches, and use them as "black boxes" for defining hierarchical graph transformation.

The notion of a *graph transformation approach*, as formalized in [KK96], serves this purpose. A graph transformation approach is any device $\mathcal{A}$ that provides a class of graphs $\mathcal{G}$, a class of rules $\mathcal{R}$, a mechanism to associate to each rule a derivation relation on graphs, i.e. an operator $\Rightarrow$ that associates to each rule $r \in \mathcal{R}$ a relation $\Rightarrow^r \subseteq \mathcal{G} \times \mathcal{G}$, a mechanism to control rule application $\mathcal{C}$, and a class $\mathcal{E}$ of graph class expressions, that can be used to specify classes of graphs. A particular use of graph class expressions is to specify classes of initial and terminal graphs for a transformation unit, see again [KK96].

As an example, the reader can consider a graph transformation approach $\mathcal{A}_{\mathcal{W}} = (\mathcal{G}_{\mathcal{W}}, \mathcal{R}_{\mathcal{W}}, \Rightarrow_{\mathcal{W}}, \mathcal{C}_{\mathcal{W}}, \mathcal{E}_{\mathcal{W}})$, based on our example of Section 4.1.1, where graphs model the structure of a hypertext, and graph transformation rules specify hypertext transformations. In this case, $\mathcal{G}_{\mathcal{W}}$ is the class of all directed graphs typed by the graph of Figure 2.3 in Chapter 2. The class of rules $\mathcal{R}_{\mathcal{W}}$ can contain rules like `add-page` shown in Figure 4.1. Furthermore, let $G \in \mathcal{G}_{\mathcal{W}}$ be the host graph depicted in Figure 4.2.a and $G' \in \mathcal{G}_{\mathcal{W}}$ be the new host graph in Figure 4.3.d. Then the pair (G, G') belongs to $\Rightarrow_{\mathcal{W}}^{\texttt{add-page}}$. We do not illustrate control conditions and expressions, since in this thesis we are mainly interested in how a single derivation step is defined. They will be considered, however, in the formal definitions of Chapter 5. The interested reader can find more examples of graph transformation approaches in [Kus99, Section 2.1].

Let us now suppose that we have three graph transformation approaches, $\mathcal{A}_x = (x, \mathcal{R}_x, \Rightarrow_x, \mathcal{C}_x, \mathcal{E}_x)$ $(x \in \{\mathcal{G}, \mathcal{D}, \mathcal{B}\})$, where $\mathcal{G}$ is a generic class of graphs, $\mathcal{D}$ is a class of rooted dags, and $\mathcal{B}$ is a class of coupling graphs. Suppose also that $H = (G, D, B) \in \mathcal{G} \times \mathcal{D} \times \mathcal{B}$ is a hierarchical graph. Then, a derivation step on H can be obtained in the following way:

1. Split H into its three component graphs G, D and B.
2. Apply a rule $\gamma \in \mathcal{R}_{\mathcal{G}}$ to G yielding a graph G', a rule $\delta \in \mathcal{R}_{\mathcal{D}}$ to D yielding a new hierarchy D', a rule $\beta \in \mathcal{R}_{\mathcal{B}}$ to B yielding a new coupling graph B'.
3. Join G', D', and B' together and check whether the result is in turn a hierarchical graph.

This procedure is illustrated in Figure 4.4. In this transformation, we add a new page and the corresponding anchor to a web graph, and we add a new page package to the hierarchy graph. Finally, the added nodes and package must be inserted into the coupling graph as well, together with the appropriate edges between them. We

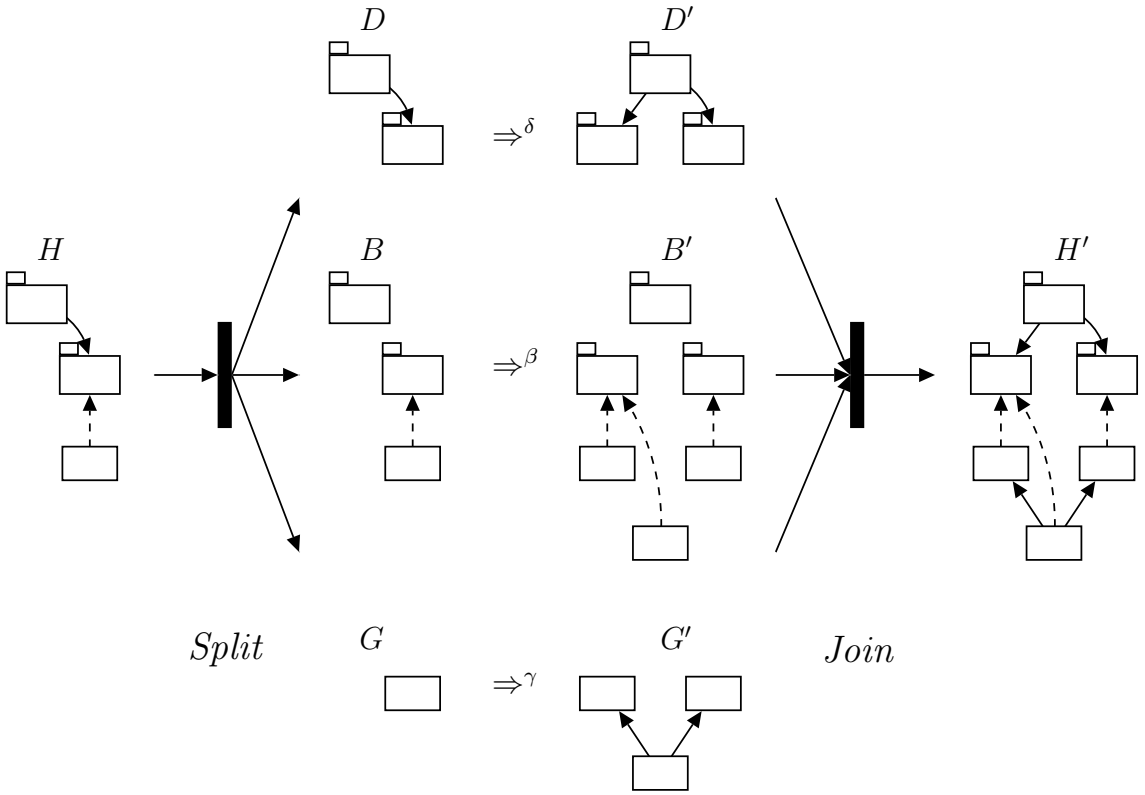


Figure 4.4: A hierarchical graph transformation step.

suppose that we have the appropriate rules in the three transformation approaches, and we use them to obtain the wanted transformation. (We will better illustrate this example in Chapter 5, where we formally define our framework.) Notice that the three transformation approaches can be seen as black boxes into which a graph is fed and a result graph is obtained. Thus, the concept of a graph transformation approach serves as an interface between the overall hierarchical graph transformation mechanism and the existing flat graph transformation machinery. In fact, we do not show the three component rules explicitly: the notion of a transformation approach hides the details of rules and rule application.

Notice that the three component graphs are transformed independently: Only at the end do we combine them and check that the resulting triple be a hierarchical graph. This means that

- we require that the set of nodes $N_{G'}$ be equal to the set of atoms $A_{B'}$ (or that the set of atoms $A_{G'} = N_{G'} \cup E_{G'}$ be equal to $A_{B'}$, in case we transform full hierarchical graphs), and that the set of packages $P_{D'}$ be equal to the set of packages $P_{B'}$ (compare Definition 3.7 on page 47),

- if we are transforming full hierarchical graphs, we require that the additional condition on edges (see again Definition 3.7) be satisfied.

The semantics $\Rightarrow^r$ of a rule r is a relation on graphs and, in several concrete graph transformation approaches, graph transformation is defined up to isomorphism. This means that, if G, G', G'' are graphs, $G \Rightarrow^r G'$ and G' is isomorphic to G'', then we have $G \Rightarrow^r G''$ as well. As a consequence, we can have the case that $H = (G, D, B)$ is a hierarchical graph, G', G'' are isomorphic graphs, D', D'' are isomorphic hierarchy graphs, B', B'' are isomorphic coupling graphs, such that $G \Rightarrow^\gamma G', G''$, $D \Rightarrow^\delta D', D''$, $B \Rightarrow^\beta B', B''$, but *while*, say, $H' = (G', D', B')$ *is* a hierarchical graph, $H'' = (G'', D'', B'')$ *is not* a hierarchical graph (Definition 3.7 requires that the *same* nodes and edges are used to glue B'' with G'' and D''). In this case, we have that $(G, D, B) \Rightarrow^{(\gamma,\delta,\beta)} (G', D', B')$, while $(G, D, B) \not\Rightarrow^{(\gamma,\delta,\beta)} (G'', D'', B'')$.

This approach is not sufficient yet, since

1. we rely on the identities of nodes and edges in the resulting component graphs to determine whether they form a valid derived hierarchical graph or not, i.e. we are not able to specify which nodes/edges should be glued after applying a triple rule,

2. we are not able to ensure that the components of a triple rule perform the same operations on common nodes/edges stemming from different components of a hierarchical graph.

In Subsection 4.3.2, we illustrate the disadvantages of this "lack of coordination" between rules, and introduce the notion of coordinated rules.

4.3.2 Coordinating the components of hierarchical graph rules

Up to now, we have modeled hierarchical graphs as triples of graphs that share common elements and defined rules that operate on the component graphs without taking this sharing into account. For example, an underlying graph rule γ does not know what a coupling rule β is doing with the shared nodes and edges of the hierarchical graph being transformed. Only after the three component rules have been applied do we look again at the identity of nodes and edges in order to glue the resulting component graphs to a hierarchical graph.

As an example of inconsistencies that can arise because of the lack of coordination between rules, we consider again the transformation step depicted in Figure 4.4. This transformation (specified by some rule γ) adds a new page and a new anchor node to the underlying graph, together with a structural edge from the anchor to the page, and a hyperlink edge to an already existing page. A rule δ specifies the insertion of a new package in the hierarchy graph. A rule β specifies the insertion of the new package and

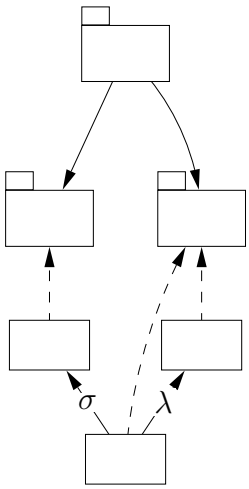


Figure 4.5: Unwanted result.

of the two new nodes, and of the appropriate containment edges between the former and the latter.

Our problems start when we take (three graphs isomorphic to) the three result graphs, and we glue them together. In our example, the new anchor should be put in the *same package* as its *owning page*, but this information is spread over two different rules—as a coupling edge between the anchor node and the package containing the owner page in the coupling rule, and as a σ-labelled edge from the anchor node to the page node in the graph rule.

Then, it can happen that the anchor is inserted in the package containing the page node that is the target of the hyperlink edge, as illustrated in Figure 4.5. The problem lies in the fact that the coupling graph is isomorphic to the coupling graph B' in Figure 4.4, but the two page nodes are swapped. By coordinating the rule application, i.e. by specifying the correct gluing of nodes and edges in the result graph explicitly, we will be able to filter out the wrong transformations.

We better illustrate this problem and the ideas for its solution by comparing hierarchical graph transformation to the transformation of distributed graphs. A distributed graph (see [Tae96], [CMR$^+$97]Subsection 3.2.4, 3.6.2) consists of a collection of *local graphs* connected through *interfaces*. An interface is modeled as a pair of graph morphisms (recall the definition of graph morphisms in Section 3.18) $DG = (G_1 \leftarrow G_0 \rightarrow G_2)$, where G_1 and G_2 are the local graphs, and G_0 is the interface graph. Intuitively, local graphs represent the state of a distributed system, while interfaces represent common elements of two local states. In this approach, the term *graph* stands for node and edge labelled, directed graph.

Roughly speaking, a hierarchical graph $H = (G, D, B)$ can be seen as a distributed graph with three local graphs and two interfaces between them. The interface between G and B is a discrete graph—i.e. a graph without edges—GB with $N_{GB} = N_G$. The morphisms $\varphi_G : GB \to G$ and $\varphi_B : GB \to B$ are defined in an obvious way. Similarly, we can define the interface between D and B as a discrete graph DB with $N_{DB} = P_D$, and the morphisms $\psi_D : DB \to D$ and $\psi_B : DB \to B$. This construction is illustrated in Figure 4.6.

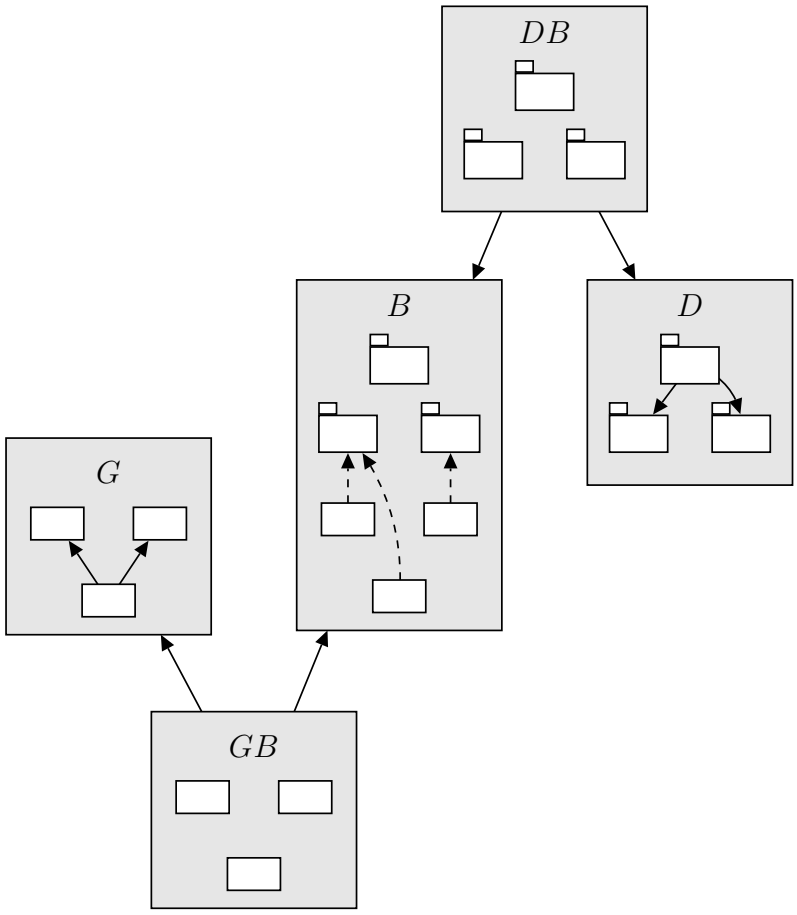


Figure 4.6: Hierarchical graphs as distributed graphs.

This comparison is quite rough, since the underlying graph G can be any graph (that provides a skeleton) and, in order to be precise, we would need a more general notion of distributed graph where the local graphs may be non-homogeneous. The case of a full hierarchical graph, where certain nodes of the coupling graph must be identified with edges of the underlying graph, would also require a special treatment. Although we cannot model hierarchical graphs as distributed graphs directly, we use this parallel

to better understand what are the missing features of our approach.

When manipulating a distributed state, we need to synchronize the transformations on the local states in order to maintain consistency. This means that we must perform the same operations (delete/update/create) on shared elements at the same time, otherwise the local images of shared data become inconsistent. In the case of hierarchical graph transformation, even though component rules are synchronous—they are applied at the same time—they are applied independently to the components of a hierarchical graph, without any constraints on operations performed on shared nodes or edges.

To solve this problem, in distributed graph transformation—see again [Roz97]—the notion of *synchronization* is used: The transformation rules that are applied to the local graphs are synchronized by means of subrules that are applied to the interface graph. This provides both synchronization in the actual meaning of the term (we transform the local graphs and the interface graph at the *same time*) and coordination in our sense (the application of local rules to the local graphs of a distributed graph has the *same effect* on the common interface graph).

In order to add coordination to hierarchical graph transformation, we first look for a way to "glue" component rules. Unfortunately, our framework does not use a specific kind of rules and can even allow different kinds of rules in different components, therefore the gluing cannot be defined by using subrules and rule morphisms. Alternatively, we will specify nodes and edges that must be identified by means of relations between items of different component rules. In order to do this, we need to extend the current notion of a transformation approach, otherwise we cannot say anything about the rules' left- or right-hand sides, or about nodes and edges inside them.

Second, coordination involves that a transformation step on the local graphs performs the *same operations* on the shared subgraph, which means that the *same nodes and edges* are deleted (resp. preserved, created). Thus we need to *map* items in the left-hand sides of rules to items in the right-hand sides, as well items inside a graph before rule application to items inside the corresponding graph after rule application, otherwise we cannot speak about deleted, preserved or created nodes or edges at all. Such mapping can be modeled using special *track* functions, or using some notion of node and edge identity. (We do not consider the case where a preserved item can be split into several items, in which case we should use a tracking relation rather that a function.) Finally, we must be able to relate elements of the component rules with "matched" elements of the component graphs, and to ensure that this matching be compatible with the gluing and the tracking information.

We would like to extend the notion of a graph transformation approach—where graphs and rules are currently atomic entities—so that it provides such additional information in an abstract, approach independent way. Before proceeding, we refine the above discussion by identifying a list of reasonable additional requirements for our framework. We need the following:

1. A way to glue the component graphs into a hierarchical graph. This ingredient is already in the basic notion of hierarchical graph, but we think it is useful to recall it here.

2. A way to speak about a left- and right-hand side of a rule.

3. A way to speak about nodes and edges inside the left- and right-hand side of a rule.

4. A way to speak about deleted (resp. preserved, created) nodes of a rule, so that we can say whether two rules specify the same operation.

5. A way to speak about nodes and edges inside a graph, and about deleted (resp. preserved, created) nodes and edges upon graph transformation.

6. A way to ensure that deleted (resp. preserved, created) nodes and edges during a transformation step are consistent with what is specified in the applied rule.

7. A way to glue two rules along common nodes/edges in the left- and right-hand side.

8. A way to coordinate the application of the components of a hierarchical rule to the components of a hierarchical graph according to the gluing of the rules.

All these requirements imply that we have access to some internal elements of a rule: left- and right-hand side graph, nodes and edges inside these graphs, and so on. On the other hand, the notion of a graph transformation approach only provides a class of rule names, and hides all internal details of a rule. We therefore need to extend the notion of a graph transformation approach.

Requirements 1 through 6 are provided by the notion of a *tracking graph transformation approach*, discussed in 4.3.3, while requirements 7 and 8 are provided by *coordinated hierarchical graph transformation approaches*, discussed in 4.3.4. Both concepts will be formally defined in Chapter 5.

4.3.3 Tracking graph transformation

In this subsection, we informally discuss how the basic notion of graph transformation, presented in Subsection 4.3.1, is extended to that of a tracking graph transformation approach. To this purpose, we assume that we have a given graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, as in Subsection 4.3.1.

Points 2 and 3 above can be addressed by associating to each rule $r \in \mathcal{R}$ two skeletons (see Definition 3.1 on page 43) LS_r and RS_r, containing the nodes and edges of the actual left- and right-hand side graphs of r, which are hidden by the existing notion of a graph transformation approach.

Borrowing a well-known idea from single-pushout graph transformation (see Subsection 4.2.1, [CMR+97], [EHK+97]), point 4 is addressed by defining a partial skeleton morphism (see Definition 3.20) $tr_r : LS_r \to RS_r$ between the left- and the right-hand side skeleton. If we consider tr_r as a function from A_{LS_r} to A_{RS_r} (recall that A_S denotes the set $N_S \cup E_S$ of atoms of a skeleton), this means that the set of preserved items of a rule is $\mathbf{dom}(tr_r)$, i.e. the preserved items are the ones that are mapped from the left- to the right-hand side; the set of deleted items is $A_{LS_r} \setminus \mathbf{dom}(tr_r)$, i.e. the deleted items are the ones that are not mapped, and the set of newly created items is $A_{RS_r} \setminus tr_r(A_{LS_r})$ are in the right-hand side but have no preimage in the left-hand side.

Such a morphism can easily be defined for many concrete approaches to graph transformation, in particular for the double-pushout approach, for the single-pushout approach (!), for the approach sketched in Section 4.1, for the PROGRES approach, etc. In case there is no natural way to define such a morphism, we can always take the empty one. The triple (LS_r, RS_r, tr_r) is called the *skeleton* of r.

A similar idea is used to address point 5 above. In fact, we can see a rule skeleton as the skeleton of a direct derivation in the following way. Let r be some rule in a graph transformation approach, and $G \Rightarrow^r G'$ a direct derivation step. We can model the deleted (resp. preserved, created) nodes and edges corresponding to this step as a partial skeleton morphism $\varphi : S_G \to S_{G'}$. Thus, the structure $(S_G, S_{G'}, \varphi)$, which is formally the same as a rule skeleton, models, at an abstract level, how nodes and edges are transformed during a derivation step. We then call such a structure a *direct derivation skeleton*. Again, while we can always define the skeleton of a graph, it depends on the chosen graph transformation approach whether it is possible—or whether it makes sense—to define a tracking morphism as done above.

Once we have a rule skeleton $S_r = (LS_r, RS_r, tr_r)$ and a derivation skeleton $(S_G, S_{G'}, \varphi)$, we want to ensure that they are consistent with each other, i.e. that during the transformation step $G \Rightarrow^r G'$, nodes and edges are deleted (resp. preserved, created) as specified in S_r, thus addressing point 6 above. Again, using an idea very close to the single-pushout approach, we do this by mapping the nodes and edges of the left-hand side skeleton to the *matched* nodes and edges in the skeleton S_G and, similarly, the nodes and edges of the right-hand side skeleton to nodes and edges of $S_{G'}$, i.e. we provide two total skeleton morphisms $m : LS_r \to S_G$ and $m' : RS_r \to S_{G'}$, and require that $\varphi \circ m = m' \circ tr_r$.

We call such a derivation step—i.e. one that provides a rule skeleton, a derivation skeleton, and two skeleton morphisms m, m' as above—a *tracking derivation step*. An example of a tracking derivation step is depicted in Figure 4.7. Notice that both nodes and edges of the four skeletons are depicted as rectangles (the edges having a thicker border) and the incidence relations are depicted as undirected edges. Notice also that the horizontal morphisms are partial (some elements are deleted and therefore not mapped), while the vertical morphisms are total (all elements of a rule must have an image in the transformed graph). Finally, the reader can easily verify that $\varphi \circ m =$

$m' \circ tr_r$.

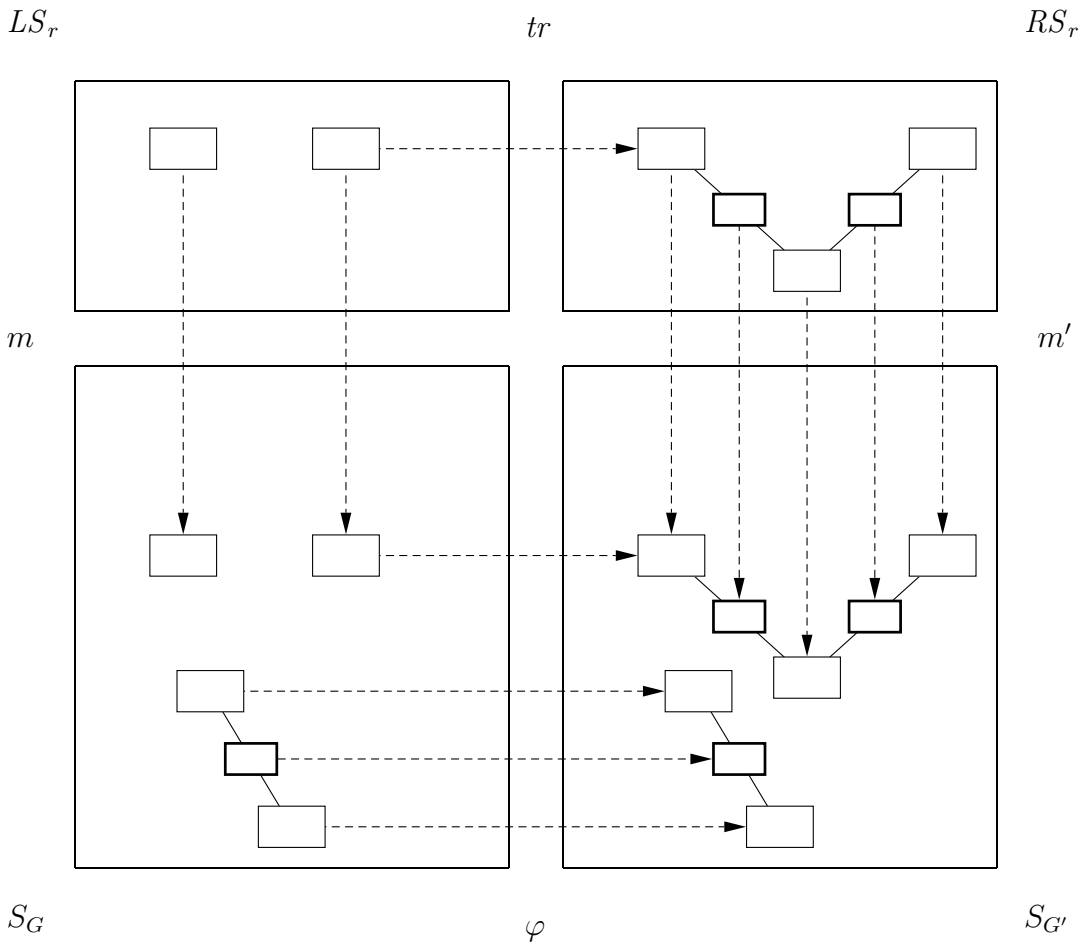


Figure 4.7: A tracking derivation step.

Summarizing, in order to fulfill the requirements 2, 3, 4, 5, 6 above, we use the notion of a tracking derivation step. A graph transformation approach where all rules provide a skeleton and where for all direct derivation steps there exists at least a derivation skeleton is called a *tracking graph transformation approach*. Some well-known approaches—like DPO, SPO, NLC, as illustrated in Appendix A—can be easily interpreted as tracking approaches.

4.3.4 Coordinated hierarchical graph transformation

Now that we have a means to speak about what happens to nodes and edges during a derivation step in one graph transformation approach, we can consider the problem

of coordinating the effect of component rules in a hierarchical graph transformation approach. In this subsection, we will further extend our notion of hierarchical graph transformation to that of coordinated hierarchical graph transformation, thus addressing points 7 (coordinated rules) and 8 (coordinated rule application) above.

In order to address point 7, we must define correspondences between atoms in different rules—namely between atoms of a coupling rule β and atoms of an underlying graph rule γ (of a hierarchy rule δ, respectively). Since rules can belong to different graph transformation approaches, and therefore be non-homogeneous, we need some kind of approach-independent interface between them. If we assume that we are using tracking graph transformation approaches, for each rule we have a rule skeleton, which will provide such an interface. With these assumptions, the gluing of two rules r and r' is represented by a pair of relations $\simeq^L \subseteq A_{LS_r} \times A_{LS_{r'}}$ and $\simeq^R \subseteq A_{RS_r} \times A_{RS_{r'}}$, specifying correspondences between atoms of the left-hand sides (resp. atoms of the right-hand sides) of r and r'. Such correspondences must be compatible with the tracking morphism of rule skeletons, namely if an atom x of LS_r corresponds to an atom x' of $LS_{r'}$ ($x \simeq^L x'$) then

- either both x and x' are preserved atoms in the respective rules, i.e. $x \in \mathbf{dom}(tr_r)$, $x' \in \mathbf{dom}(tr_{r'})$, and $tr_r(x) \simeq^R tr_{r'}(x')$,
- or both x and x' are deleted atoms, i.e. $x \notin \mathbf{dom}(tr_r)$ and $x' \notin \mathbf{dom}(tr_{r'})$,

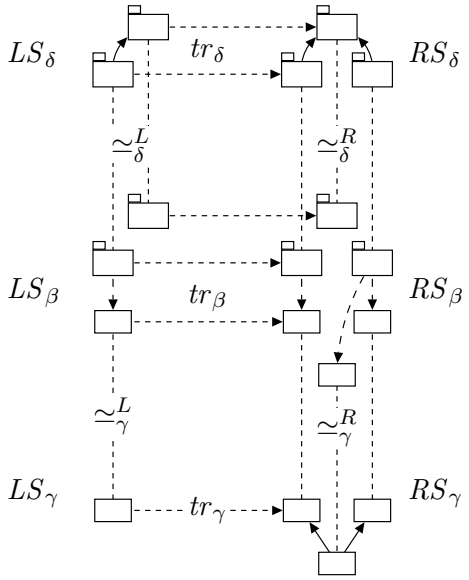


Figure 4.8: A coordinated rule.

This situation is depicted in Figure 4.8. Here we show three rules γ, β, δ, together with their tracking morphisms, the correspondence relations $\simeq^L_\gamma$ and $\simeq^R_\gamma$ between γ and β, and the correspondence relations $\simeq^L_\delta$ and $\simeq^R_\delta$ between δ and β. Notice that, in order to help the reader's intuition, we have depicted left- and right-hand sides of rules as concrete directed graphs rather than skeletons. Also, we have not depicted the correspondence relations and the morphism arrows for edges.

Summarizing, *coordinated hierarchical graph transformation rules* will be triples of rules γ, δ, β, together with four correspondence relations $\simeq^L_\gamma \subseteq A_{LS_\gamma} \times A_{LS_\beta}$, $\simeq^R_\gamma \subseteq A_{RS_\gamma} \times A_{RS_\beta}$, $\simeq^L_\delta \subseteq A_{LS_\delta} \times A_{LS_\beta}$, $\simeq^R_\delta \subseteq A_{RS_\delta} \times A_{RS_\beta}$. Notice that we do not need any gluing between γ and δ.

As a last task, we must define a coordinated hierarchical graph transformation step. This will be a hierarchical graph transformation step, satisfying additional constraints due to the coordination information. In the light of what has been discussed up to now, these constraints can be made explicit as follows. Let $H = (G, D, B)$ and $H' = (G', D', B')$ be two hierarchical graphs, and $r = (\gamma, \delta, \beta, \simeq^L_\gamma, \simeq^R_\gamma, \simeq^L_\delta, \simeq^R_\delta)$ be a coordinated rule. Then H' is derived from H with a coordinated derivation step using rule r, in symbols $H \Rrightarrow^r H'$, if:

1. The hierarchical graph H' can be derived from H using $r' = (\gamma, \delta, \beta)$ in a plain hierarchical graph transformation step.

2. All transformation steps on the components of H are tracking, and, in particular, they provide the matching morphisms $m_\gamma : LS_\gamma \to S_G$, $m_\delta : LS_\delta \to S_D$, $m_\beta : LS_\beta \to S_B$, $m'_\gamma : RS_\gamma \to S_{G'}$, $m'_\delta : RS_\delta \to S_{D'}$, $m'_\beta : RS_\beta \to S_{B'}$, as well as the tracking morphisms $tr_\gamma : LS_\gamma \to RS_\gamma$, $tr_\delta : LS_\delta \to RS_\delta$, $tr_\beta : LS_\beta \to RS_\beta$, $\varphi_\gamma : S_G \to S_{G'}$, $\varphi_\delta : S_D \to S_{D'}$, $\varphi_\beta : S_B \to S_{B'}$.

3. The gluing between the component graphs of the hierarchical graphs H and H' is compatible

   (a) with the gluing on the rules and with the matching morphisms. For example, if an atom of LS_γ is glued to an atom of LS_β, then their images w.r.t. m_γ and m_β are the same atom in G and B. In other words, glued atoms in the rules match the same atoms in the graphs.

   (b) with the tracking morphisms provided by the tracking derivation steps on the components. For example, if x is an atom of both G and B, then it is either deleted in both graphs, or it is mapped to the same atom $\varphi_\gamma(x) = \varphi_\beta(x)$ in G' and B'. In other words, the rules must delete/preserve the same atoms in the component graphs.

If we interpret the sharing of nodes and edges between the component graphs of a hierarchical graph as correspondence relations $\simeq^G \subseteq A_G \times A_B$, $\simeq^D \subseteq A_D \times A_B$,

$\simeq^{G'} \subseteq A_{G'} \times A_{B'}$, $\simeq^{D'} \subseteq A_{D'} \times A_{B'}$, we can summarize the above conditions in the diagram of Figure 4.9. Here the side and middle vertical squares exist thanks to the fact that the three component transformation approaches are tracking. The two back squares exist and the morphisms are compatible with the relations since the applied rule is coordinated. The existence of the top, bottom (see 3.a above), and front squares (see 3.b above), and the compatibility of morphisms and relations in them ensure that the derivation step is coordinated.

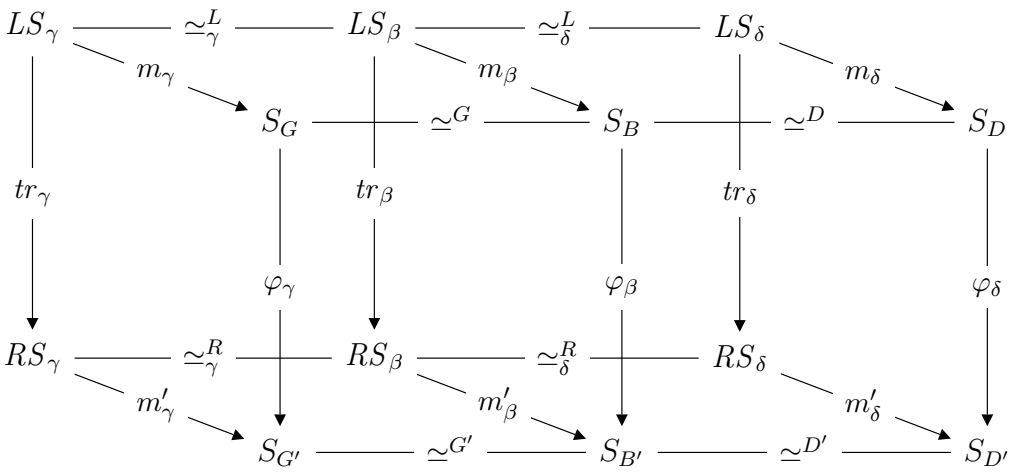


Figure 4.9: Coordinated derivation step.

With the discussion of a coordinated derivation step, we have concluded our informal presentation of coordinated hierarchical graph transformation. This concept will be presented formally in Chapter 5. An example can be found in Chapter 6.

4.4 Summary

In the present chapter, we have motivated and informally described a notion of hierarchical graph transformation, which further develops the idea of hierarchical structuring as an orthogonal concept to the underlying graph to be structured.

We have first reviewed the general notions of rule-based graph transformation and of a transformation system. We have then considered and classified different approaches to graph transformation and shown how this classification can be applied to hierarchical graph transformation.

Since different approaches to graph and hierarchical graph transformation exist, we have proposed a generic framework, which can be instantiated in several ways by choosing different graph transformation approaches. Like a hierarchical graph is mod-

eled as a triple of graphs, hierarchical graph transformation rules are triples of graph transformation rules.

The application of such triples to the components of a hierarchical graph is coordinated by gluing items in the different rules, and by defining corresponding constraints on each transformation step. Thanks to the notions of a *tracking graph transformation approach* and to that of a *coordinated hierarchical graph rule*, the gluing and its derived constraints are specified independently from the kinds of graphs and graph transformation approaches used for the component rules.

The framework allows to combine three arbitrary (tracking) graph transformation approaches and produce a hierarchical graph transformation approach.

Chapter 5

A Framework for Hierarchical Graph Transformation

Following the discussion of Chapter 4, in this chapter we formally define a framework for rule-based hierarchical graph transformation. Its main ingredients are:

1. The notion of a *graph transformation approach*—see e.g. [KK96], [Kus99]—which captures the common elements that allow different approaches to graph transformation to be used in our framework. This notion allows us to speak about graphs, rules and graph transformation at a very abstract level, i.e. avoiding all the details of specific approaches.

2. The use of a standard construction to combine three existing graph transformation approaches into a hierarchical graph transformation approach. In this construction, hierarchical graph transformation rules are triples of graph transformation rules taken from the three original approaches.

3. The use of *coordinated rules*, i.e. triple rules where the components are glued along common nodes and edges.

In order to define coordinated rules and transformation, we have extended the original notion of a graph transformation approach to that of a *tracking graph transformation approach*, which allows to track preserved nodes and edges both within rules and within transformation steps.

In Section 5.1, we recall the concept of a graph transformation approach, we consider its extension to tracking graph transformation approaches, and we introduce the notation necessary for the following subsections. In Section 5.2, we introduce the concept of a hierarchical graph transformation approach, and a standard construction for building such approaches from given graph transformation approaches. In Section 5.3, we discuss the problem of adapting graph transformation approaches that cannot be used directly in our framework. In Section 5.4, we give a summary of the chapter.

5.1 Graph Transformation Approaches

In this section, we discuss graph transformation approaches. We introduce the basic notion of a graph transformation approach in Subsection 5.1.1, while *tracking* graph transformation approaches are dealt with in Subsection 5.1.2.

5.1.1 Basic notions

The notion of a graph transformation approach (see e.g. [KK96], [Kus99], [KK99b], [KKS97], [KK99a]) 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 [KK96]), 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 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 5.1 (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 6.1. The interested reader can find more examples in [Kus99, Section 2.1].

5.1.2 Tracking graph transformation approaches

In this subsection, we formally define tracking graph transformation approaches, which we have already discussed informally in Section 4.3.3. We start by introducing rule skeletons. A rule skeleton specifies a left- and right-hand side skeleton, and a partial morphism between the two. In this way, we can speak about nodes and edges inside a rule, and namely in its left- and right-hand side. The partial morphism from the left- to the right-hand side skeleton allows us to track preserved elements of the rule.

The notion of a rule skeleton relies on that of a partial skeleton morphism. (See also *total* skeletons morphisms, introduced in Definition 3.20.) We provide a separate definition, since (partial) skeleton morphisms will occur several times in this chapter.

Definition 5.2 (Partial skeleton morphisms)
Given two graph skeletons $S_1 = (N_1, E_1, I_1)$, $S_2 = (N_2, E_2, I_2)$, 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 $I_1(e, n)$, then $n \in \mathbf{dom}(\mu_{\mathrm{N}})$ and $I_2(\mu_{\mathrm{E}}(e), \mu_{\mathrm{N}}(n))$.

A (total) skeleton morphism is a partial morphism where the component functions are total. 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 certain contexts, skeleton morphisms play a particular role, therefore we introduce the terms *rule* (resp. *direct derivation*) *skeleton*.

Definition 5.3 (Rule 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. In some contexts, a rule skeleton will be called *direct derivation skeleton*.

We will illustrate rule skeletons in Section 6.1. A graph transformation approach where rules provide rule skeletons and where direct derivations provide direct derivation skeletons is called a tracking graph transformation approach.

Definition 5.4 (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),

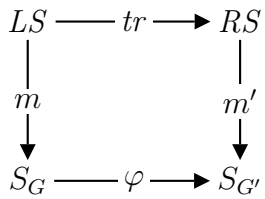


Figure 5.1: Tracking derivation.

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 $\langle m, m', \varphi \rangle$, where $m : LS \to S_G$ and $m' : RS \to S_{G'}$ are skeleton morphisms, and $(S_G, S_{G'}, \varphi)$ is a direct derivation skeleton such that

$$m' \circ tr = \varphi \circ m$$

i.e. such that the diagram of Figure 5.1 commutes.

While rule skeletons and direct derivation skeletons allow to track preserved graph elements between two graphs—namely the left- and right-hand side of a rule, or the graph before and the graph after a derivation step—tracking derivation adds consistency constraints between a rule skeleton and a direct derivation skeleton by means of matching morphisms.

We postpone the illustration of a concrete tracking derivation step to Section 6.1. In Appendix A, we will consider more examples of tracking graph transformation approaches.

5.2 Hierarchical graph transformation

In this section, we introduce a framework that allows to define hierarchical graph transformation approaches as a combination of existing graph transformation approaches. If we consider again our model of a hierarchical graph, namely a tuple $H = (G, D, B)$, where G is a graph, D is a rooted dag, and B is a coupling graph, it is a natural idea to apply graph transformation to G, D, and B. Therefore, we suppose that we have three transformation approaches, one for each component of a hierarchical graph, and then show how they can be combined together. The component rules of a hierarchical graph rule are glued by identifying common nodes and edges. Such rules are called coordinated rules, and are defined assuming that the used graph transformation approaches are tracking.

The framework is presented in two steps. In the first step (Subsection 5.2.1), we introduce hierarchical graph transformation *without* rule coordination, where the basic mechanism for combining graph transformation approaches is described. In the second step (Subsection 5.2.2), coordinated rules and transformation are presented.

5.2.1 Combining graph transformation approaches

In this subsection, we will introduce a standard construction that allows to define hierarchical graph transformation if three appropriate graph transformation approaches are available. To begin with, we need special names for transformation approaches where the class of graphs contains hierarchical graphs, hierarchy graphs, and coupling graphs. These are introduced in the following definition.

Definition 5.5 (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 5.1, 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.

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

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 5.6 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 hierarchical graph transformation approach $\mathcal{A}_{\mathcal{H}}(\mathcal{A}_{\mathcal{G}}, \mathcal{A}_{\mathcal{D}}, \mathcal{A}_{\mathcal{B}}) = (\mathcal{H}, \mathcal{R}_{\mathcal{H}}, \Rightarrow_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ 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
$$\begin{aligned} SEM_\mathcal{H}(C) := \{((G, D, B), (G', D', B')) \in \mathcal{H} \times \mathcal{H} \mid \\ (G, G') \in SEM_\mathcal{G}(C_\mathcal{G}) \wedge \\ (D, D') \in SEM_\mathcal{D}(C_\mathcal{D}) \wedge \\ (B, B') \in SEM_\mathcal{B}(C_\mathcal{B})\} \end{aligned}$$
- $\mathcal{E}_\mathcal{H} := \mathcal{E}_\mathcal{G} \times \mathcal{E}_\mathcal{D} \times \mathcal{E}_\mathcal{B}$, and, for $X = (X_\mathcal{G}, X_\mathcal{D}, X_\mathcal{B}) \in \mathcal{E}_\mathcal{H}$, the semantics is defined by
$$SEM_\mathcal{H}(X) := SEM_\mathcal{G}(X_\mathcal{G}) \times SEM_\mathcal{D}(X_\mathcal{D}) \times SEM_\mathcal{B}(X_\mathcal{B}) \cap \mathcal{H}$$

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 *hierarchical graph transformation rule*.

In a similar fashion, we construct a class of full hierarchical graphs $\mathcal{F} := \mathcal{F}(\mathcal{G}, \mathcal{D}, \mathcal{B})$, and a full hierarchical graph transformation approach $\mathcal{A}_\mathcal{F}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B}) = (\mathcal{F}, \mathcal{R}_\mathcal{F}, \Rightarrow_\mathcal{F}, \mathcal{C}_\mathcal{F}, \mathcal{E}_\mathcal{F})$ as above, replacing $\mathcal{H}$ with $\mathcal{F}$ everywhere.

It can be useful to assume that the three 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.

5.2.2 Coordinated rules and derivations

Coordinated rules are triples of graph transformation rules—i.e. hierarchical graph transformation rules—with additional information about correspondences between elements of the coupling graph rule and elements of the other rules. For example, we may 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 (see below).

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.

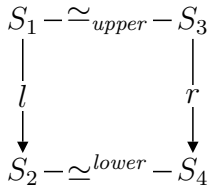


Figure 5.2: Gluing rule skeletons.

Definition 5.7 (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 $f : A_{S_1} \to A_{S_2}$ 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, $\simeq_{upper} \subseteq A_{S_1} \times A_{S_3}$ and $\simeq_{lower} \subseteq A_{S_2} \times A_{S_4}$ be two relations. We say that the quadruple $\langle l, r, \simeq_{upper}, \simeq_{lower} \rangle$ *commutes* if, for all $a \in A_{S_1}$, for all $c \in A_{S_3}$, we have that if $a \simeq_{upper} c$, 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 $l(a) \simeq_{lower} r(c)$.

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 $\simeq_{upper}$, $\simeq_{lower}$.

Consider a diagram like that of Figure 5.2. Intuitively, the notion of commutativity introduced in Definition 5.7 says that if two elements of the upper skeletons are related through $\simeq_{upper}$, then either they are both mapped to two elements of the lower skeletons that are related through $\simeq_{lower}$, or they are both outside the domain of the two partial morphisms in the diagram.

Definition 5.8 (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

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

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

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

such that all relations are injective[1], and $\langle tr_\gamma, tr_\beta, \simeq_\gamma^L, \simeq_\gamma^R \rangle$ and $\langle tr_\delta, tr_\beta, \simeq_\delta^L, \simeq_\delta^R \rangle$ commute. We indicate with $\mathcal{CO}(\mathcal{R})$ the class of all coordinated rules over $\mathcal{R}$.

A *full coordinated rule* over $\mathcal{R}$ is a system

$$r = (\gamma, \delta, \beta, \simeq_\gamma^L, \simeq_\gamma^R, \simeq_\delta^L, \simeq_\delta^R)$$

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

$$\begin{array}{rcl} \simeq_\gamma^L & \subseteq & \left(N_{LS_\gamma} \cup E_{LS_\gamma}\right) \times N_{LS_\beta} \\ \simeq_\gamma^R & \subseteq & \left(N_{RS_\gamma} \cup E_{RS_\gamma}\right) \times N_{RS_\beta} \\ \simeq_\delta^L & \subseteq & N_{LS_\delta} \times N_{LS_\beta} \\ \simeq_\delta^R & \subseteq & N_{RS_\delta} \times N_{RS_\beta} \end{array}$$

such that all relations are injective, and $\langle tr_\gamma, tr_\beta, \simeq_\gamma^L, \simeq_\gamma^R \rangle$ and $\langle tr_\delta, tr_\beta, \simeq_\delta^L, \simeq_\delta^R \rangle$ commute. We indicate with $\mathcal{FCO}(\mathcal{R})$ the class of all full coordinated rules over $\mathcal{R}$.

In Figure 5.3, we depict the relations and morphisms of a coordinated hierarchical graph rule $r = (\gamma, \delta, \beta, \simeq_\gamma^L, \simeq_\gamma^R, \simeq_\delta^L, \simeq_\delta^R)$

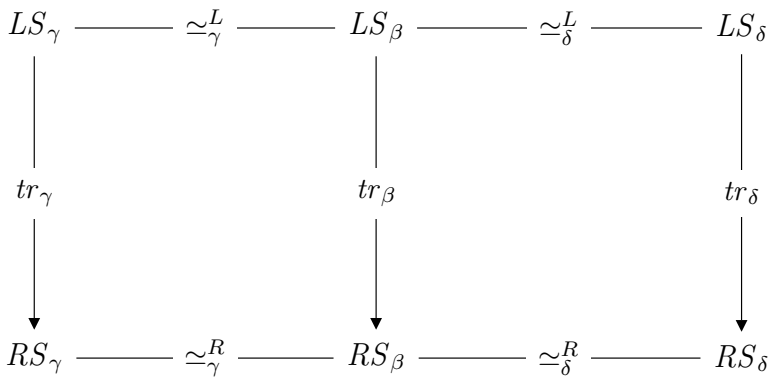


Figure 5.3: Coordinated rule diagram.

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

[1]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 $(a, b), (a, b') \in R$ holds.

Definition 5.9 (Coordinated HG transformation approach)
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 $\mathcal{A}_\mathcal{H}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B}) = (\mathcal{H}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be the corresponding tracking hierarchical graph transformation approach. Then, a *coordinated hierarchical graph transformation approach* over $\mathcal{A}_\mathcal{H}$ is a hierarchical graph transformation approach

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

where, for all $H, H' \in \mathcal{G}$, $H = (G, D, B)$, $H' = (G', D', B')$, for all rules $r \in \mathcal{CO}(\mathcal{R})$, $r = (\gamma, \delta, \beta, \simeq^L_\gamma, \simeq^R_\gamma, \simeq^L_\delta, \simeq^R_\delta)$, 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 5.4, such that all squares in the diagram commute, where
   - the back squares exist and commute since r is a coordinated rule,
   - if $G \Rightarrow^\gamma_\mathcal{G} G'$, $D \Rightarrow^\delta_\mathcal{D} D'$, $B \Rightarrow^\beta_\mathcal{B} B'$, the vertical side and middle squares exist and commute since the three component approaches are tracking,
   - the front horizontal relations are defined as $\simeq^G := \{(a, a) \mid a \in A_B\}$, $\simeq^{G'} := \{(a, a) \mid a \in A_{B'}\}$, $\simeq^D := \{(p, p) \mid p \in P_B\}$, $\simeq^{D'} := \{(p, p) \mid p \in P_{B'}\}$[2],

We call a derivation step $H \Rrightarrow^r H'$ like above a *coordinated derivation step.*

For the tracking full hierarchical graph transformation approach $\mathcal{A}_\mathcal{F}(\mathcal{A}_\mathcal{G}, \mathcal{A}_\mathcal{D}, \mathcal{A}_\mathcal{B})$, we define a *coordinated full hierarchical graph transformation approach* over $\mathcal{A}_\mathcal{F}$ as a hierarchical graph transformation approach $\mathcal{FCO}(\mathcal{A}_\mathcal{H})$ in the same way as above. Notice that the relations $\simeq^G$, $\simeq^{G'}$, $\simeq^D$, $\simeq^{D'}$, will not be the same as in the (plain) hierarchical graph case since, for $F = (G, D, B) \in \mathcal{F}$, we have $A_B = A_G$, $A_{B'} = A_{G'}$.

5.3 Using the framework

In this section, we discuss how one can adapt graph transformation approaches that cannot be used directly in the framework presented in Subsection 5.2.1. In this framework we need a graph transformation approach that handles hierarchy graphs, and a graph transformation approach that handles coupling graphs. An approach cannot be used directly if:

[2]Remember that $A_B = N_G$, $A_{B'} = N_{G'}$, $P_B = P_D$, $P_{B'} = P_{D'}$.

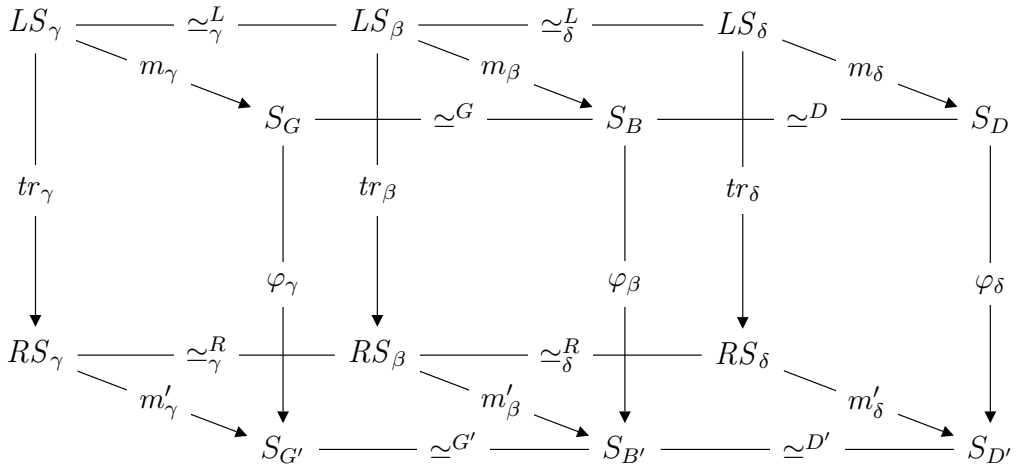


Figure 5.4: Coordinated derivation step.

1. It uses a different graph representation[3]. For example, it considers edges as entities with their own identities, and allows parallel edges, in contrast with the requirements of Definition 3.7 on page 47.

2. It contains graphs that are not valid hierarchy or coupling graphs, for example cyclic graphs, or graphs that are not bipartite, respectively.

3. It contains rules that, when applied to valid hierarchy or coupling graphs, produce invalid ones. For example, a rule may transform a hierarchy dag into a cyclic graph.

The issue considered in point 1 is a technical, but important one. Hierarchy and coupling graph transformation approaches must use simple graphs, otherwise they do not comply with the definition of a hierarchical graph and cannot be used in Construction 5.6 and Definition 5.9. In Subsection 5.3.1, we describe the use of different kinds of graphs for encoding hierarchy and coupling graphs, thus providing a general method for adapting graph transformation approaches that we want to use in the mentioned construction.

In Subsection 5.3.2 we address points 2 and 3 above, namely the need for restricting the class of graphs and the class of rules of a transformation approach, in order to produce a hierarchy or a coupling graph transformation approach without unwanted graphs or rules.

[3]Of course, different representations will influence the classes of graphs that we can represent (see also point 2).

5.3.1 Coding graphs

In this subsection, the whole discussion is applied to hierarchy graphs, the case of coupling graphs being analogous. Given a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, it may be that, although $\mathcal{G}$ is not a class of hierarchy graphs, its graphs provide a *reasonable* encoding for hierarchy graphs. The problem of deciding whether some graph is a "reasonable" encoding of another graph is left to the user of the framework: our task only goes as far as capturing this encoding is concerned.

The fact that graphs in $\mathcal{G}$ *encode* hierarchy graphs in a class $\mathcal{D}$ is captured by a partial mapping $dec : \mathcal{G} \to \mathcal{D}$. We do not require that the mapping be total: some graphs in $\mathcal{G}$ may not be correct encodings. We do not require that the mapping be injective either: there may be more than one representation in $\mathcal{G}$ of a graph in $\mathcal{D}$. In this way we allow such situations as

1. two isomorphic graphs $G, G' \in \mathcal{G}$ encoding the same graph in $\mathcal{D}$
2. redundant representation of elements of $D \in \mathcal{D}$ in a graph $G \in \mathcal{G}$, for example, parallel edges in G representing a single edge in D.

On the other hand, we require that every graph in $\mathcal{D}$ have a canonical representation (encoding) in $\mathcal{G}$, i.e. that a total injective mapping $enc : \mathcal{D} \to \mathcal{G}$ exists. Furthermore, if we decode the canonical encoding of a graph D, we must obtain back the same graph D.

The notion of encoding is defined formally as follows.

Definition 5.10 (Graph encoding)
Given a class of graphs $\mathcal{D}$, an *encoding* of $\mathcal{D}$ is a triple $\bigstar = \langle \mathcal{G}, enc, dec \rangle$, where $\mathcal{G}$ is a class of graphs, $enc : \mathcal{D} \to \mathcal{G}$ is a total injective mapping, and $dec : \mathcal{G} \to \mathcal{D}$ is a partial mapping, such that $dec \circ enc = \mathbf{id}_{\mathcal{D}}$.

Given a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, and an encoding $\bigstar = \langle \mathcal{G}, enc, dec \rangle$ of a class of graphs $\mathcal{D}$, we can induce a new graph transformation approach, where the semantics of rule application, of control conditions, and of graph expressions, is derived from the corresponding semantics in $\mathcal{A}$. This is described in the following definition.

Definition 5.11 (Induced graph transformation approach)
Given a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow_{\mathcal{A}}, \mathcal{C}, \mathcal{E})$, and an encoding $\bigstar = \langle \mathcal{G}, enc, dec \rangle$ of a class of graphs $\mathcal{D}$, we say that a graph transformation approach $\mathcal{I}$ is *induced* from $\mathcal{A}$ and $\bigstar$ if

$$\mathcal{I} = \mathcal{I}_{\mathcal{A},\bigstar} = (\mathcal{D}, \mathcal{R}, \Rightarrow_{\mathcal{I}}, \mathcal{C}, \mathcal{E})$$

and the semantics of $\mathcal{I}$ is defined as follows:

- For all $r \in \mathcal{R}$, for all $D, D' \in \mathcal{D}$, $D \Rightarrow^r_{\mathcal{I}} D'$ iff there exist $G, G' \in \mathcal{G}$ such that $G \Rightarrow^r_{\mathcal{A}} G'$, $dec(G) = D$, and $dec(G') = D'$, i.e. a derivation step in $\mathcal{I}$ is induced by a derivation step in $\mathcal{A}$ via the decoding function.

- For all $C \in \mathcal{C}$, we have

$$SEM_{\mathcal{I}}(C) := \{(dec(G), dec(G')) \mid (G, G') \in SEM_{\mathcal{A}}(C) \cap \mathbf{dom}(dec) \times \mathbf{dom}(dec)\}$$

  i.e. the semantics of a control condition C contains all the pairs that are decodings of pairs in the semantics of C in the original approach $\mathcal{A}$.

- For all $E \in \mathcal{E}$, we have

$$SEM_{\mathcal{I}}(E) := \{dec(G) \mid G \in SEM_{\mathcal{A}}(E) \cap \mathbf{dom}(dec)\}$$

  i.e. the semantics of an expression E contains all graphs that are decodings of graphs in the semantics of E in the original approach $\mathcal{A}$.

Using the above notions, we can construct a hierarchy transformation approach from an existing graph transformation approach $\mathcal{A}$ and an encoding $\bigstar$ of hierarchy graphs. For example, we can use the double-pushout approach (with node- and edge-labelled directed graphs where loops and parallel edges are allowed) to induce a hierarchy transformation approach (with simple unlabelled directed graphs) if we forget labels, forbid loops, and collapse parallel edges. Encoding and induced graph transformation approaches can be used for defining coupling graph transformation analogously. These ideas are illustrated with an example in Section 6.1.

5.3.2 Restricting graphs and rules

As a result of the discussion of Subsection 5.3.1, we can now assume that we have a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, where the graphs in $\mathcal{G}$ are in the proper format for representing hierarchies or coupling graphs. Before we can use $\mathcal{A}$, we may still need to restrict the class of graphs to be a class of rooted dags, and the class of rules to those rules that produce only rooted dags when applied to rooted dags. The situation regarding coupling graphs is similar.

We start by considering the notion of hierarchy- and coupling-preserving rules, which is formalized in the following definition.

Definition 5.12 (Hierarchy- and coupling-preserving rule)
Let $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be a graph transformation approach, and let $\mathcal{G}' \subseteq \mathcal{G}$ be a class of rooted dags. A rule $r \in \mathcal{R}$ is *hierarchy-preserving* if $\Rightarrow^r$ is closed w.r.t. $\mathcal{G}'$, i.e. if for all graphs $G \in \mathcal{G}'$, $H \in \mathcal{G}$, $G \Rightarrow^r H$ implies that $H \in \mathcal{G}'$. Similarly, let $\mathcal{G}'' \subseteq \mathcal{G}$ be a class of coupling graphs. A rule $r \in \mathcal{R}$ is *coupling-preserving* if $\Rightarrow^r$ is closed w.r.t. $\mathcal{G}''$, i.e. if for all graphs $G \in \mathcal{G}''$, $H \in \mathcal{G}$, $G \Rightarrow^r H$ implies that $H \in \mathcal{G}''$.

In some cases, we may consider different kinds of hierarchy graphs than rooted dags, and use a variation of Definition 5.12 accordingly. For example, in [Pra79] arbitrary hierarchies are used (see Chapter 7).

We are now able to express formally the restriction of an arbitrary graph transformation approach to a hierarchy or a coupling transformation approach. Suppose that for all $x \in \{\mathcal{G}, \mathcal{D}, \mathcal{B}\}$, $\mathcal{A}_x = (x, \mathcal{R}_x, \Rightarrow_x, \mathcal{C}_x, \mathcal{E}_x)$ is a graph transformation approach. Let $\overline{\mathcal{D}} \subseteq \mathcal{D}$ be a class of rooted dags, and $\overline{\mathcal{B}} \subseteq \mathcal{B}$ be a class of coupling graphs. Finally, let $\overline{\mathcal{R}}_{\mathcal{D}} \subseteq \mathcal{R}_{\mathcal{D}}$ be a class of hierarchy-preserving rules, and $\overline{\mathcal{R}}_{\mathcal{B}} \subseteq \mathcal{R}_{\mathcal{B}}$ be a class of coupling-preserving rules. Then we can apply Construction 5.6 to the approaches $\mathcal{A}_{\mathcal{G}}$, $\overline{\mathcal{A}}_{\mathcal{D}} = (\overline{\mathcal{D}}, \overline{\mathcal{R}}_D, \Rightarrow_{\mathcal{D}}, \mathcal{C}_{\mathcal{D}}, \mathcal{E}_{\mathcal{D}})$ and $\overline{\mathcal{A}}_{\mathcal{B}} = (\overline{\mathcal{B}}, \overline{\mathcal{R}}_{\mathcal{B}}, \Rightarrow_{\mathcal{B}}, \mathcal{C}_{\mathcal{B}}, \mathcal{E}_{\mathcal{B}})$.

When using a particular approach for hierarchy graph or coupling graph transformation, it is important that we can restrict the class of rules in an effective way. In Chapter 6, we study this problem when our framework is instantiated with the double-pushout approach, and we provide (decidable) conditions that ensure that a DPO rule be hierarchy-preserving (resp. coupling-preserving).

A last issue concerns the use of the obtained induced transformation approaches for defining coordinated transformation (see Definition 5.9). To this purpose, we must ensure that the induced hierarchy or coupling transformation approach be tracking. In many cases, it is easy to map nodes and edges of an encoding graph to nodes and edges of the encoded hierarchy or coupling graph in such a way that the resulting induced transformation approach is tracking if the encoding approach is. Further illustration of these ideas can be found in the example of Section 6.1, in the subsection dealing with tracking.

5.4 Summary

In this chapter, we have introduced a framework that allows to define hierarchical graph transformation. Here are the most important aspects of this framework:

- Our decoupled model of a hierarchical graph, consisting of a triple of graphs, suggests that we can define hierarchical graph transformation as a combination of transformations of the three component graphs.

- The concept of a graph transformation approach allows us to choose different methods for specifying such transformations on the component graphs.

- The notion of coordinated rules allows to define constraints on the actions that the components of a triple rule perform on the components of a hierarchical graph.

- Encoding allows to adapt existing graph transformation approaches that cannot be used directly in the framework.

- For each instantiation of the framework, rules that may produce inconsistent hierarchical graphs must be identified and eliminated.

The resulting notion provides a flexible, standard method for defining hierarchical graphs and their transformations: several kinds of graph can be structured in a hierarchical way, and several different approaches can be used to define transformations on them. In Chapter 6, we instantiate this framework using the double-pushout approach, we provide an illustrating example, and we investigate the problem of detecting forbidden rules.

Chapter 6

Hierarchical Graph Transformation Using the Double-pushout Approach

In this chapter, we illustrate the instantiation of the framework presented in Chapter 5 to a specific hierarchical graph transformation approach. This example is based on the double-pushout (DPO) approach to graph transformation (see [CMR+97]): the transformation rules for the three components of a hierarchical graphs are DPO rules. This approach can also be interpreted as an extension of the traditional DPO approach on flat graphs (the underlying graphs) where two new graphs (the hierarchy and the coupling graph) and corresponding transformation rules are added.

In the first part of the chapter (Section 6.1), we provide an informal presentation of the DPO approach, we show how it can be adapted to define hierarchy and coupling transformation, and provide an example of hierarchical graph transformation in our world-wide web scenario. Instantiating the framework also involves identifying hierarchy- and coupling-preserving rules (see the discussion in Section 5.3), which is the topic of the second part of the chapter. After recalling the formal definitions for the DPO approach (Section 6.2), we study the restrictions that we have to impose on DPO rules in order to obtain hierarchy- and coupling-preserving rules in Section 6.3 and Section 6.4. It turns out that we can characterize hierarchy- or coupling-preserving rules as rules that satisfy appropriate (decidable) conditions, as stated in Proposition 6.12 and in Proposition 6.20.

6.1 An Example

We now illustrate hierarchical graph transformation by defining some transformations for our web example (see Section 2.1). In this example, we use the *double-pushout* (DPO) approach to graph transformation, which we introduce here in an informal and intuitive way, postponing formal definitions to Section 6.2 (see also [CMR+97]). We do not outline the structure of this section here, since it should be read as a whole.

Encoding and restricting the component graphs

The DPO approach uses node and edge labelled directed graphs, described in Definition 3.3 on page 44. Since these are a suitable model for hypermedia networks, we use these graphs as they are in the underlying graph transformation approach. We have to choose an appropriate labeling for our specific example. 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 *hyperlink edges* originating from anchors (see again Figure 2.1 on page 10) and σ labels *structural edges*, i.e. all other edges.

As far as hierarchy and coupling graphs are concerned, we first need to find a suitable coding for them. (Since both graphs are simple graphs, they will have the same coding.) In hierarchy graphs, the labeling is irrelevant, and therefore we will use a trivial node and edge labeling alphabet $\Sigma_{\mathcal{D}} = \Delta_{\mathcal{D}} = \{\bot\}$. In coupling graphs, we will label atoms from the underlying graphs with their original label, and packages with $\bot$. In this way, if we suppose that $\bot \notin (\Sigma_{\mathcal{G}} \cup \Delta_{\mathcal{G}})$, we can use node labels to distinguish atoms from packages of a coupling graph. On the other hand, edge labels in a coupling graph are irrelevant. We therefore have $\Sigma_{\mathcal{B}} = \Sigma_{\mathcal{B}} \cup \Delta_{\mathcal{B}} \cup \{\bot\}$ and $\Delta_{\mathcal{B}} = \{\bot\}$.

Given a hierarchy graph $D = (N, E)$, with $E \subseteq N \times N \setminus \{(n, n) \mid n \in N\}$, we define $enc_{\mathcal{D}}(D)$ by letting $N_{enc_{\mathcal{D}}(D)} := N$, $E_{enc_{\mathcal{D}}(D)} := E$, for all $e = (u, v) \in E$ we let $s_{enc_{\mathcal{D}}(D)}(e) := u$, $t_{enc_{\mathcal{D}}(D)}(e) := v$, and $\lambda_{enc_{\mathcal{D}}(D)}(e) := \bot$. We also let, for all $u \in N$, $l_{enc_{\mathcal{D}}(D)}(u) := \bot$. The decoding function $dec_{\mathcal{D}}$ will remove node and edge labels and collapse parallel edges to a single edge. It will not be defined on graphs that contain loops. We allow graphs with parallel edges to be valid encodings, since it is too restrictive (see Section 6.3) to eliminate DPO rules that introduce parallel edges. We can imagine that a parallel edge represents a relation between two nodes in a redundant way.

For a coupling graph $B = (P, A, E)$, with $P \cap A = \emptyset$ and $E \subseteq (P \times A) \cup (A \times P)$, we define $enc_{\mathcal{B}}(B)$ by letting $N_{enc_{\mathcal{B}}(B)} := P \cup A$, $E_{enc_{\mathcal{B}}(B)} := E$, for all $e = (u, v) \in E$ we let $s_{enc_{\mathcal{B}}(B)}(e) := u$, $t_{enc_{\mathcal{B}}(B)}(e) := v$, and $\lambda_{enc_{\mathcal{B}}(B)}(e) := \bot$. We also let, for all $p \in P$, $l_{enc_{\mathcal{B}}(B)}(p) := \bot$. Given a fixed $b \in \Delta_{\mathcal{B}}$, we let, for all $x \in A$, $l_B(x) := b$. We allow other encodings of B that differ from $enc_{\mathcal{B}}(B)$ at most in the labels of its atoms, but all atoms will have labels in $\Delta_{\mathcal{B}}$. This is useful when we want to distinguish different

"types" of atoms in the coupling graph and rules (see the usage of labels in the next subsection). For all these encodings, the decoding function $dec_{\mathcal{B}}$ will remove node and edge labels and collapse parallel edges to a single edge (see above). It will sort nodes into sets P and A according to their labels. The decoding function will not be defined on graphs that contain loops.

The second step consists in restricting hierarchy graphs to be rooted dags, and coupling graphs to be bipartite graphs (compare the definitions in Chapter 3). This means that the coding of a hierarchy graph must be a rooted directed graph with no loops and no cycles, that possibly contains parallel edges. A coupling graph must be bipartite, with every atom connected to at least one package. Parallel edges are allowed, too.

We only consider loose coupling graphs in this example and in the rest of the chapter, i.e. coupling graphs where packages are *not* anchored to graph elements (see Definition 3.6). The reason for this choice is discussed at the end of Section 6.4.

Defining the rules

We now turn our attention back to our web example. Recall that in this 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.

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 triple rule (without coordination), depicted in Figure 6.1, illustrates the transformation "create project." 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 the 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, and that we do not display edge labels that are equal to $\perp$.

The second rule, depicted in Figure 6.2, specifies the transformation "add hyperlink." The graph part of the rule indicates that we should find two page nodes in

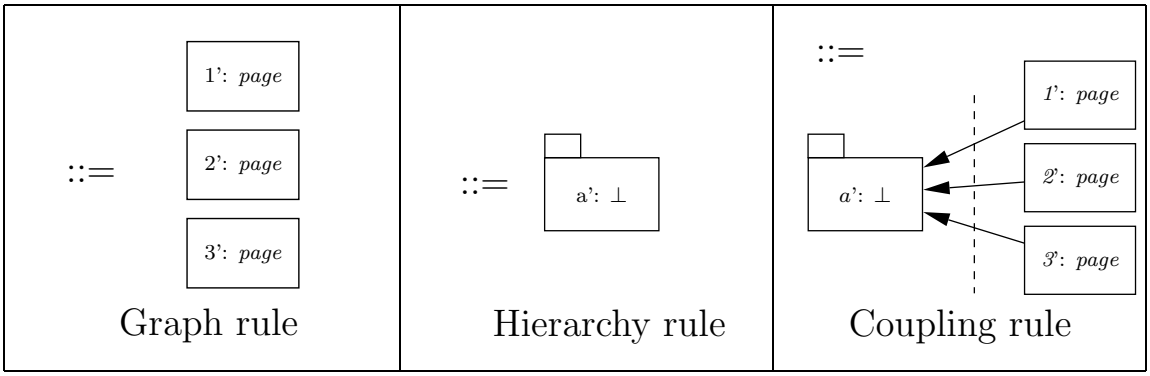


Figure 6.1: Creating a package and some internal nodes.

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 must be 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. Notice that this rule, although it does not change anything in the graph to which it is applied, is *not* the identity rule, since it cannot be applied to any graph.

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 same package as one of the two pages. Notice that we are using plain—rather than full—hierarchical graphs, since there is no edge containment.

As far as node identifiers are concerned, both rules use a notation similar to that used in Chapter 4, for example in Figure 4.1. More precisely, '*id* : *label* indicates a node in the left-hand side, *id*' = '*id* indicates a preserved node in the right-hand side (it has the same identifier as a node in the left-hand side), and *id*' : *label* indicates a new node in the right-hand side. We have used numbers as node identifiers and letters as package identifiers. Notice that all identifiers are *local* to the rule in which they appear, and this is reflected by the semantics of transformations: During a derivation step, we do not ensure that, say, node 1 in the left-hand side of the graph rule matches the same node in the graph as node 1 in the coupling rule[1]. In the continuation of this example we will remove this limitation, considering coordinated rules.

Notice that we not check that the hierarchy rule be hierarchy-preserving and that the coupling graph rule be coupling-preserving. This issue must be considered every time the framework is instantiated using a specific graph transformation approach. For the DPO case, it turns out that this property can be checked effectively (see further in

[1]Recall that the coupling graph shares its nodes with the underlying graph.

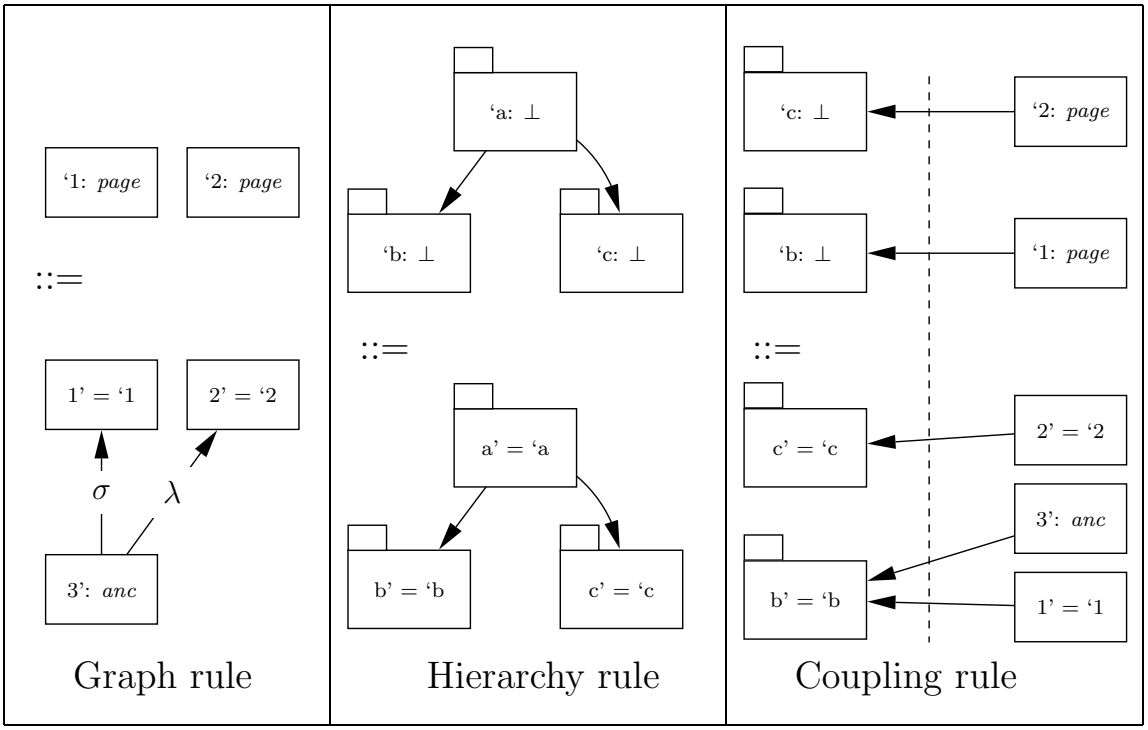


Figure 6.2: Adding a hyperlink.

this chapter).

Tracking graph transformation approaches

Before we illustrate coordinated rules, we open a parenthesis and show how the double-pushout can be naturally interpreted as a tracking graph transformation approach. Consider again the underlying graph rule γ on the left of Figure 6.2. Let us assign (arbitrarily) the names nl_1, and nl_2 to the left and the right node of the left-hand side respectively, and nr_1, nr_2, er_1, and er_2 to the left node, the right node, the σ-labelled edge and the λ-labeled edge of the right-hand side. Then, the skeleton of this rule is (SL, SR, tr), where

$$\begin{aligned} SL &= (\{nl_1, nl_2\}, \emptyset, \emptyset) \\ SR &= (\{nr_1, nr_2, nr_3\}, \{er_1, er_2\}, \\ &\quad \{(er_1, nr_3), (er_1, nr_1), (er_2, nr_3), (er_2, nr_2)\}) \\ tr &= \{(nl_1, nr_1), (nl_2, nr_2)\} \end{aligned}$$

In Figure 6.3, we depict a tracking derivation step based on this rule and on our running example. Notice that we represent a skeleton as a bipartite undirected graph:

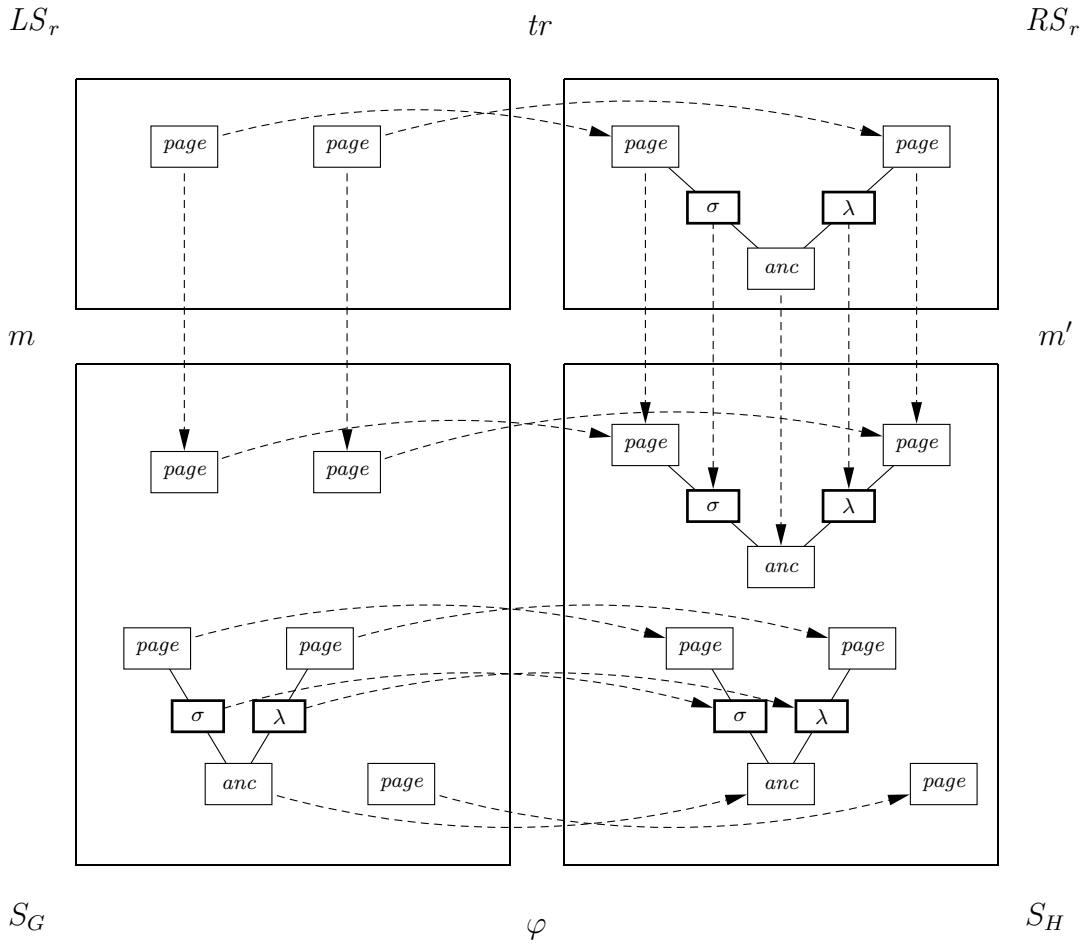


Figure 6.3: Tracking derivation.

Its nodes represent nodes and edges of the original graph—edges are depicted as squares with a thicker border. Undirected edges in this bipartite graph model the incidence relation. The matching and tracking morphisms are modeled as bundles of arrows.

In this tracking derivation step—based on the transformation step on the underlying graph depicted in Figure 6.4 and Figure 6.5 later in this section—we have two nodes in the left-hand side, which are mapped by m to matched nodes of the graph G, and by tr to their counterparts in the rule's right-hand side. The fact that m' is total and that the diagram commutes ensures that the two nodes are preserved. The nodes and edges in the right-hand side that have no preimage in the left-hand side are mapped to new nodes in H. All nodes and edges in G that are not matched by nodes and edges of the left-hand side are mapped to H by φ as they are. Notice that in this specific example there are no deleted nodes or edges.

As hinted at the end of Section 5.3, we also have to make sure that the induced hierarchy and coupling transformation approach are tracking, so that we can define coordinated rules. Since the DPO approach is tracking, and since nodes in a coding graph G are mapped injectively to packages of a decoded hierarchy $dec_{\mathcal{D}}(G)$, we can use the track functions provided by the DPO approach to define corresponding track functions for the induced hierarchy transformation approach. This track function will be undefined on (hierarchy) edges, since there may be (redundant) parallel edges that are collapsed during decoding. In general, we cannot state whether the unique edge obtained from the collapsing is preserved or not: one of the collapsed edges may be deleted while another is preserved. Having a track function that is undefined on hierarchy edges is no problem, since hierarchical rules are coordinated using packages and atoms only.

The situation for coupling graphs is similar: nodes in an encoding graph G are mapped injectively to packages or atoms of $dec_{\mathcal{B}}(G)$, while redundant coupling edges are collapsed. Again, since coordination affects only atoms and packages of a coupling graph, a track function that is undefined on coupling edges is appropriate for our needs.

While it can be interesting to understand better the relation between tracking and encoding, we will be content of these informal consideration here, leaving any further investigation as a topic for future work.

Coordination and direct derivation

In this subsection, we extend the rule depicted in Figure 6.2 to a coordinated rule. Let us call γ the underlying graph rule (left rule in Figure 6.2), δ the hierarchy graph rule (middle rule in Figure 6.2), β the coupling graph rule (right rule in Figure 6.2), and, for $x \in \{\gamma, \beta, \delta\}$, let us denote with $x.id$ a node in the left- or right-hand side skeleton of x.

We have already seen how to construct the skeleton of a DPO rule: tracking information (preserved nodes and edges) is indicated by node identifiers using a PROGRES-like notation (see also the examples in Chapter 4). For example, the left node of the left-hand side of the graph rule $\gamma.$‘1 is a preserved node and is mapped by tr_γ to the top left node $\gamma.1$’ in the right-hand side of the graph rule (see the notation 1’ = ‘1). We must now define the relations gluing the component rules γ, β, δ. For example, the top left node in the right-hand side of the graph rule (which has identifier 1) and the bottom right node in the right-hand side of the coupling rule must be in the $\simeq^R_\gamma$ relation. We have:

$$\begin{aligned} \simeq^L_\gamma &= \{(\gamma.\text{‘}1, \beta.\text{‘}1), (\gamma.\text{‘}2, \beta.\text{‘}2)\} \\ \simeq^R_\gamma &= \{(\gamma.1\text{’}, \beta.1\text{’}), (\gamma.2\text{’}, \beta.2\text{’}), (\gamma.3\text{’}, \beta.3\text{’})\} \end{aligned}$$

$$\begin{aligned} \simeq_\delta^L &= \{(\delta.\text{`b}, \beta.\text{`b}), (\delta.\text{`c}, \beta.\text{`c})\} \\ \simeq_\delta^R &= \{(\delta.\text{b'}, \beta.\text{b'}), (\delta.\text{c'}, \beta.\text{c'})\} \end{aligned}$$

Figure 6.4 and Figure 6.5 illustrate an application of the second rule with the coordination relations shown above. A hierarchical graph is depicted here with the packages and their edges at the top, the graph nodes and edges at the bottom, and the coupling edges (the dashed edges) in the middle. The "gluing" nodes and packages are overlapping. (Compare the analogous notation in Chapter 4.) The rectangles on the left of Figure 6.4 frame the matches of the left-hand sides of the graph rule γ, of the hierarchy rule δ, and of the coupling rule β. Notice that we are showing the concrete left-hand sides of rules, and not their skeletons.

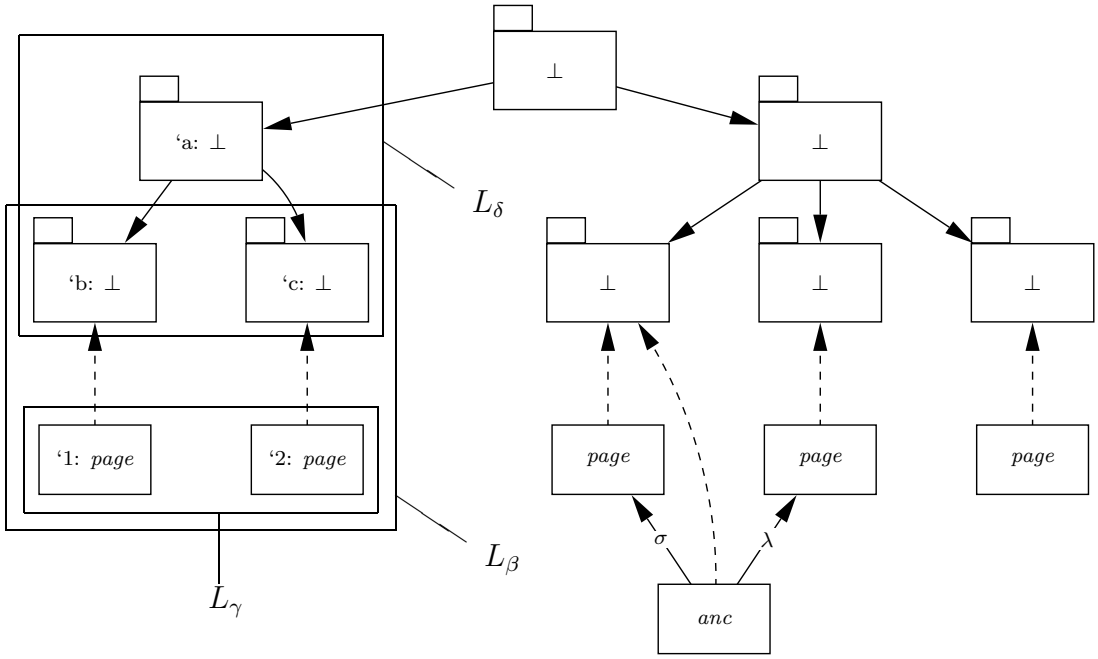


Figure 6.4: Before adding a hyperlink.

The result of the application of this rule is that a new anchor node is added, and it is placed in the appropriate package. This is ensured by the coordination between the three rules, namely, by the fact that $\gamma.3' \simeq_\gamma^R \beta.3'$ and $\beta.\text{b'} \simeq_\delta^R \delta.\text{b'}$. In Figure 6.5, the hierarchical graph after the transformation is depicted, with the matches of the right-hand sides of the rules again highlighted by rectangular frames. Also in this picture,

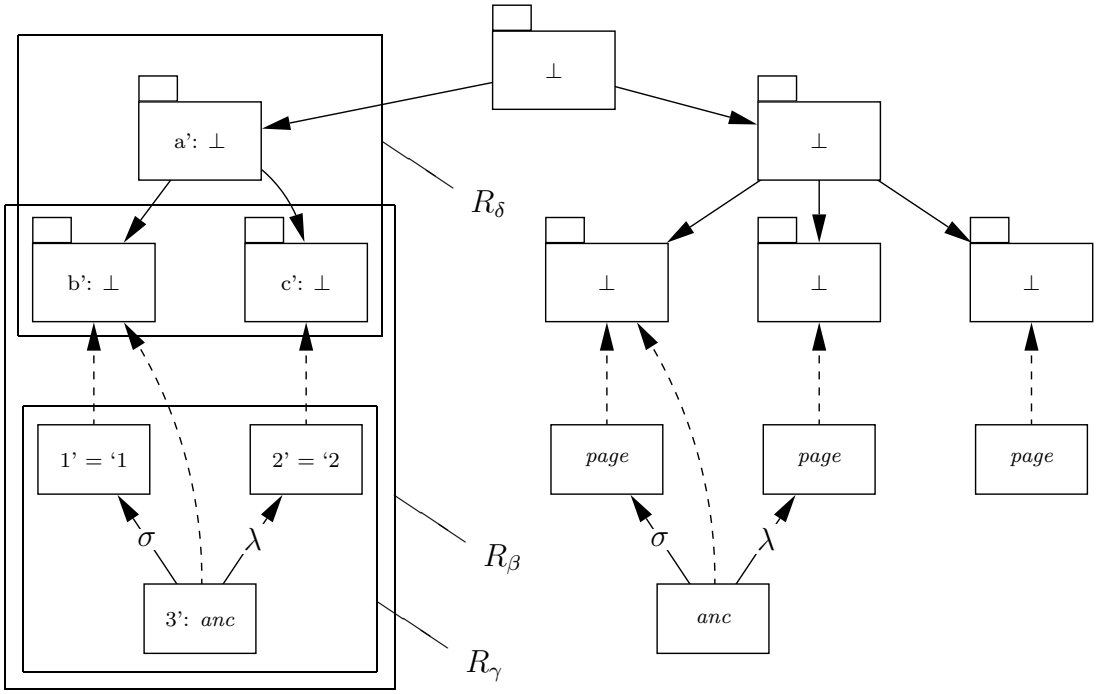


Figure 6.5: After adding a hyperlink.

the common nodes and packages in the component graphs, as well as the matching and gluing nodes in the rules are depicted as overlapping nodes and packages.

6.2 Double-pushout graph transformation

The double-pushout approach (DPO) to graph transformation was introduced at the beginning of the seventies (see [EPS73]) as a generalization of Chomsky grammars from strings to graphs (see also [Roz97, Chapter 3]). 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 rest of the section.

To begin with, we need the notion of a graph morphism, and of a category of graphs. (For a general book on category theory, the reader can refer to [Mac71].)

Definition 6.1 (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 $\text{л}_G = \text{л}_{G'} \circ \varphi_{\mathrm{E}}$. 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. 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 6.2 (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[2] of both L and R, we write $r = (L \supseteq K \subseteq R)$.

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 6.3 (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 $c' = h \circ c$.

[2] A graph A is a subgraph of a graph B if $N_A \subseteq N_B$, $E_A \subseteq E_B$, $s_A = s_B|_{N_A}$, $t_A = t_B|_{N_A}$, $l_A = l_B|_{N_A}$, $\text{л}_A = \text{л}_B|_{E_A}$. For a function f and a set $X \subseteq \mathbf{dom}(f)$, $f|_X$ denotes the restriction of f to X.

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 6.4
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 6.5 (Direct derivation)
Given a graph G and a rule $r = (L, i, K, j, R)$, if we can build a diagram as in Figure 6.6, i.e. if graphs D and H exist, together with morphism $m : L \to G$, $d : D \to G$, $k : K \to D$, $m' : R \to H$, $d' : D \to H$, 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 $G \Rightarrow_r H$. m is called an *occurrence* or *matching morphism*.

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 $G \Rightarrow_r H$.

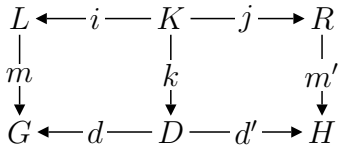


Figure 6.6: DPO derivation step.

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 $m : L \to G$;
2. check the *gluing condition* on K, and m (see e.g. [Roz97, p. 189] for the details), which ensures that we can
3. construct the context graph D by removing $m(L - K)$ from G, and morphisms $k : K \to D$, $d : D \to G$ such that

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

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

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 6.6, 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 $G \supseteq D \subseteq H$.

6.3 A Characterization of DPO Hierarchy Rules

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

To begin with, we observe once more that the DPO approach uses directed, node- and edge-labelled graphs. As already seen in Section 6.1, we encode a hierarchy graph with a directed graph without loops, where parallel edges are tolerated[3] (they just specify the child/parent relation on packages redundantly, and are eliminated during decoding), and where we use a trivial node and edge labeling by means of the alphabets $\Sigma = \Delta = \{\bot\}$. Our task is to characterize 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.

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 [HMP01] 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 6.6 (Dag preserving rules)
A DPO rule $r = (L, i, K, j, R)$ is *dag preserving* iff, for all graphs G, H, if G is a dag and $G \Rightarrow_r H$, then H is a dag as well.

Before proceeding, we recall some useful terms and notation concerning paths (see also Definition 3.4 on page 44). 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[4]:

[3] It is easy to check that, in order to avoid the introduction of parallel edges, we should forbid all rules that introduce an edge between two preserved nodes, which is a very restrictive condition.

[4] The first relation is the usual representation of edges as the Cartesian product of the set of nodes

- $u \succ_G v$ 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 6.8).

Definition 6.7 (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 characterization of dag preserving rules, which is expressed by the following proposition.

Proposition 6.8
A rule (L, i, K, j, R), where L, K and R are dags, is dag preserving iff it satisfies the path-checking condition.

Proof
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 6.6, 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 consequently 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 $G = (N_G, E_G, s_G, t_G, l_G, \text{ɲ}_G)$ be a dag, such that $N_G = N_L$ and $E_G = E_L \cup \{e\}$ (with $e \notin E_L$), $s_G(e) = v$, $t_G(e) = u$, $l_G = l_L$, $\text{ɲ}_G|L = \text{ɲ}_G$ and $\text{ɲ}_G(e)$ 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 6.6). 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 characterization of dag preserving rules.

Remark 6.9
In [HKP91], *jungles*[5] are used for representing terms over a certain signature, and (DPO)

[5] Jungles are acyclic hypergraphs that respect some special conditions on the degree of their nodes and on their labeling.

jungle rewriting for specifying term evaluation. In that paper a characterization 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 characterization 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 [HKP91, 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 6.10, rooted-dag preserving rules and, in Definition 6.11, 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 6.12. The following definition instantiates the general notion of a hierarchy-preserving rule (see Definition 5.12) to the DPO approach.

Definition 6.10 (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 6.11 (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 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 6.12 (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 B. The definition deals with graphs whose paths are oriented w.r.t. a given subgraph.

Definition 6.13 (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 6.14
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 6.15
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 6.16
Given a pushout $\langle i : K \to L, k : K \to D, d : D \to G, m : L \to G \rangle$ in the category of directed graphs, where G is a rooted dag and all morphisms are injective, we have the following:

1. *If $\rho_G \in m(N_L)$ 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 6.16.

Remark 6.17
Given a pushout $\langle i : K \to L, k : K \to D, d : D \to G, m : L \to G \rangle$ 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 6.14 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 6.12.

We are now ready to prove Proposition 6.12.

Proof
We already know from Proposition 6.8 that a rule preserves dags if and only if it satisfies the path-checking condition. We must show that, provided that a rule satisfies the path-checking condition, it preserves rooted-dags if and only if 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 $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, and $d' : D \to H$ be graph morphisms and graphs as in Definition 6.5 (see also Figure 6.6). We distinguish the following four cases:

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

   (a) If $v = m'(u)$ 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 $m'(\rho_R)$ 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 $\rho_G = m(\rho_L) = mi(\rho)$, 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 $d'k(w) = m'j(w) = m'j(\rho) = m'(\rho_R)$. Otherwise, if $w \neq \rho$, we have $j(w) \succ_R^+ \rho_R$, which implies $d'k(w) = m'j(w) \succ_H^+ m'(\rho_R)$ and, by concatenating the paths in H, we obtain $v \succ_H^+ m'(\rho_R)$ as required.

2. ρ_L is preserved and $m(\rho_L) \neq \rho_G$. In this case, if there existed $x \in N_L$ such that $\rho_G = m(x)$, we would have $x \succ_L^+ \rho_L$, $m(x) \succ_G^+ m(\rho_L) \succ_G^+ \rho_G = m(x)$, 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 $v = m'(u)$ for some $u \in N_R$, then $u \succ_R^+ \rho_R$. This implies that $v = m'(u) \succ_H^+ m'(\rho_R)$. On the other hand, we have $m(\rho_L) \succ_G^+ \rho_G$ and, since L is a dag, all nodes following $m(\rho_L)$ in this path have no preimage in L, otherwise we would have a cycle going through $m(\rho_L)$ in G. Let $\rho \in N_K$ such that $\rho_L = i(\rho)$ and $\rho_R = j(\rho)$. The path from $m(\rho_L)$ 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 $v \succ_H^+ m'(\rho_R) = d'k(\rho) \succ_H^+ d'(\overline{\rho})$, 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 $d'k(w) = m'j(w) \succ_H^+ d'(\overline{\rho})$ (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 $m(\rho_L) = \rho_G$. In this case we prove that $\rho_H = m'(\rho_R)$ (note that the proof is very similar to case 1 above). Suppose that $v \in N_H$.

   (a) If $v = m'(u)$ for some $u \in N_R$ there exists a path $v \succ_H^+ m'(\rho_R)$ 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 $\rho_G = m(\rho_L)$, 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 $d'k(w) = m'j(w) = m'(\rho_R)$. Otherwise, if $j(w) \neq \rho_R$, we have $j(w) \succ_R^+ \rho_R$, which implies $d'k(w) = m'j(w) \succ_H^+ m'(\rho_R)$ and, by concatenating the paths in H, we have $v \succ_H^+ m'(\rho_R)$ as required.

4. ρ_L is not preserved and $m(\rho_L) \neq \rho_G$. We prove that this case never occurs. In fact, if $m(\rho_L) \neq \rho_G$, G must contain at least two nodes and at least one edge. Furthermore $m(\rho_L) \succ_G^+ \rho_G$. By Lemma 6.15, there exists $w \in i(N_K)$ such that $m(w)$ lies on this path; as an extreme case, we can have $m(w) = \rho_G$, but we always have $w \neq \rho_L$, because ρ_L is not preserved. This implies that $w \succ_L^+ \rho_L$, and therefore $m(w) \succ_G^+ m(\rho_L) \succ_G^+ m(w)$, and G would not be a dag[6].

Only-if) From Proposition 6.8, 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 $G = (N, E, s, t, l, m)$ be the rooted dag with $N = N_L \cup \{\gamma\}$, $E = E_L \cup \{e\}$, $s(e) = \rho_L$, $t(e) = \rho$[7]. 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 $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, and $d' : D \to H$ be graph morphisms and graphs as in Definition 6.5, with m 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 6.17 and from the fact that $D \prec k(K)$ and $R \prec j(K)$ we conclude that H is not a rooted dag.

□

[6]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 $m(\rho_L) \succ_G^+ \rho_G$ with no intermediate node from L.

[7]The labeling functions are irrelevant.

6.4 A Characterization of DPO Coupling Rules

In this section, we will study which conditions must be satisfied by a DPO rule so that it transforms coupling graphs into coupling graphs. We will only consider *loose* coupling graphs here. We discuss *tight* coupling graphs at the end of the section.

In Section 6.1, we have already presented a way to code coupling graphs as directed, node and edge labelled graphs. In this coding, we distinguish between the "package nodes" and the "atom nodes" of the coupling graph by labelling the former with a special label $\bot$, and the latter with labels from a set of atom labels Σ_{A}, with $\bot \notin \Sigma_{\mathrm{A}}$. In other words, for the coupling graph we use a node alphabet $\Sigma = \{\bot\} \cup \Sigma_{\mathrm{A}}$, and the edge alphabet $\Delta = \{\bot\}$ (edge labels in the coupling graph are irrelevant). A graph B over Σ and Δ encodes a loose coupling graph iff, for all $n \in N_B$, if $l_B(n) \in \Sigma_{\mathrm{A}}$, then there exists an edge $e \in E_B$ with $t_B(e) = n$ and $l_B(s_B(e)) = \bot$. We now want to characterize DPO rules that transform such a graph to a graph with the same properties, i.e. find a condition that can decide, for each DPO rule r, whether r is coupling-preserving. We first formalize the notion of a coupling-preserving rule in the DPO approach in the following definition (see also Definition 5.12).

Definition 6.18 (Coupling-preserving rules)
A rule $r = (L \supseteq K \subseteq R)$, where L, K and R are directed graphs over $\Sigma = \{\bot\} \cup \Sigma_{\mathrm{A}}$ and $\Delta = \{\bot\}$, is *coupling-preserving* iff, for all directed graphs B, B' over Σ and Δ, if $B \in \mathcal{BT}$ and $B \Rightarrow_r B'$, then $B' \in \mathcal{BT}$.

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 $\emptyset \neq M \subseteq \Sigma$, we define $N_G^M := \bigcup_{\lambda \in M} N_G^\lambda$.

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

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

i.e. the graph containing the newly created nodes of rule r, the graph of deleted nodes, and the graph of preserved nodes.

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

Definition 6.19 (Coupling-graph preservation condition)
A rule $r = (L \supseteq K \subseteq R)$, where L, K and R are directed graphs over $\{\bot\} \cup \Sigma_{\mathrm{A}}$ and $\{\bot\}$, satisfies the *coupling-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 $n \in N_{PRES(r)} \cap N_R^{\Sigma_A}$, we have that $\mathbf{indeg}_R(n) = 0$ implies $\mathbf{indeg}_L(n) = 0$.

This condition says that a new atom node must be connected to at least one package node[8], 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.

We can now prove that this condition is necessary and sufficient.

Proposition 6.20
A rule r is coupling-preserving iff it satisfies the coupling-graph preservation condition.

Proof
If) Suppose that $r = (L \supseteq K \subseteq R)$ is a rule satisfying the coupling-graph preservation condition, and suppose that G is a coupling graph, H is a graph such that $G \Rightarrow_r H$. Then there exists a graph D and morphisms $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, 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 $n \in N_H^{\Sigma_A}$.

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 coupling 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 coupling 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.

[8] We call *atom node* a node with label in Σ_A and *package node* a node with label $\perp$.

Only if) Suppose that $r = (L \supseteq K \subseteq R)$ is a rule which does not satisfy the coupling-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 $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, 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 $m'(n)$ 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 coupling 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 coupling graphs, no matter to which coupling 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 $G = (N_G, E_G, s_G, t_G, l_G, \text{л}_G)$, 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 $\text{л}_G(e) := \perp$.

   G is a coupling 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 $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, and $d' : D \to H$, such that m is the inclusion of L in G and the two resulting squares are pushouts.

Now, let us consider again node n which violates the coupling-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 coupling graph.

□

With Proposition 6.20 we have provided a method for ensuring that a DPO rule preserves the structure of coupling graphs to which it is applied. Combined with Proposition 6.12, we obtain a characterization of DPO hierarchical graph transformation rules, in the sense that we are able to check statically whether triples[9] of DPO rules transform hierarchical graphs into hierarchical graphs.

An observation on tight coupling graphs Before we conclude this section, we want to make some observations on tight coupling graphs, which are used in hierarchical graphs where the packages are anchored to specific nodes and edges. In this case, we must ensure that each package is anchored to exactly one node. It is easy to verify that, each time a DPO rule on coupling graphs adds a new edge from a preserved package to some (preserved or new) atom, it is impossible to ensure that this rule produces correct coupling graphs. In fact, it is impossible to check that the preserved package had no outgoing edges, and therefore the new edge can easily produce an unwanted configuration: a package that is anchored to more than one node.

This problem can be addressed by using a graph transformation approach with *negative application conditions*, i.e. an approach where we can specify a forbidden context for the left-hand side of a rule. (See e.g. [Sch97, Page 526].)

6.5 Summary

In this chapter we have instantiated the framework introduced in Chapter 5 to the double-pushout graph transformation approach. In this case, the three components of a hierarchical graph are all transformed using rules in the double-pushout approach. We have illustrated the instantiated framework by means of an example in our world-wide web scenario.

Since we cannot apply DPO rules freely—because they could introduce inconsistencies in the hierarchy graphs or in the coupling graphs—we have provided conditions that characterize hierarchy- and coupling-preserving rules respectively. In this way, we

[9] Notice that the underlying graph rule need not be checked.

have shown all the necessary steps that allow to define a new hierarchical graph transformation approach within our framework. Further applications of DPO hierarchical graph transformation can be found in Chapter 8.

Chapter 7

Simulating Other Approaches to Hierarchical Graphs

While in Chapter 6 we showed how to use our framework to define a new hierarchical graph transformation approach, in this chapter we consider the opposite direction: given an existing approach, we try to simulate it by instantiating our framework. In this way we can compare the expressive power of our framework to that of other approaches, and gain a better understanding of these approaches using a common reference model (see also [BH01]). In this chapter we deal with approaches within the graph transformation field while other areas are considered in Section 9.2.

Our comparison focuses on the following aspects:

Statics: Whether our hierarchical graph model allows to represent faithfully the information that can be modeled in other approaches.

Dynamics: Whether our model can simulate the operations on hierarchical graphs that can be specified in other approaches.

In the static part, we need to model hierarchical graphs from one approach using our representation, i.e. to choose an appropriate notion of graph for the underlying graph, an appropriate coding for the hierarchy and the coupling graph, and possible constraints on these graphs. This translation tends to be easier for decoupled approaches, because the underlying graph and the hierarchy are already modeled as different structures (see e.g. distributed systems in Subsection 7.2.2), while coupled approaches require additional effort, since the role of certain items as components of an underlying graph must be separated from their role as components of the hierarchy (see e.g. AGG in Subsection 7.2.1). In the dynamic part, the task of translating rules to our approach can be rather complex, but it allows to have a better understanding of the manipulations performed by a specific approach (see Section 7.1, Subsection 7.2.6), and to interpret different approaches using a common framework. We believe that these advantages are

worth the price in terms of complexity (decoupling, tracking, coordination) that has to be paid in order to keep our model general enough.

Most of this chapter (Section 7.1) is dedicated to the H-graph grammar approach, as used in [Pra79] (see also [Pra83]). This approach is interesting because of its *connecting* embedding mechanism (see Table 4.2 on page 79) which is more difficult to model than other *gluing* approaches, and suggests directions for possible improvements in our framework (see Subsection 7.1.6). For H-graph grammars we present a thorough discussion, recalling its main concepts, and proposing a translation to our framework based on NLC graph grammars, for which we provide a completeness/correctness result (Proposition 7.8). Other approaches to hierarchical graph transformation are considered at an informal level in Section 7.2. In Section 7.3, we give a summary of the chapter.

7.1 H-Graph Grammars à la Pratt

In [Pra79], *H-graphs* are used to model runtime data structures inside a software system. This approach is used for the definition of programming language semantics. Hierarchical graph transformation is used as a syntax-directed method to generate the initial state of such a system: a (context free) string grammar describing the syntax of a language is coupled with a hierarchical graph grammar, such that it is possible to associate to each construct of the string language a hierarchical graph corresponding to the runtime structures needed to support that construct. This approach is one of the first proposals for hierarchical graphs in the graph-transformation literature.

H-graph transformation is based on node replacement: A node in a hierarchical graph is rewritten to a new hierarchical graph, which is embedded in the context graph by establishing new edges to special nodes of the right-hand side of a rule, called the *input* and the *output* nodes. This approach is interesting because it is a connecting transformation approach, whereas all other approaches to hierarchical graph transformation that we know of are gluing approaches (see 7.2).

In this section, we show how an H-graph can be represented using our hierarchical graph model, and how an H-graph grammar can be translated to a corresponding hierarchical graph grammar based on our framework. The hierarchical graph grammars used here are obtained by instantiating our framework to a variety of NLC graph replacement grammars. This means that we can simulate the embedding mechanism of H-graph grammars through instructions using the labels of nodes in the context of the replaced node. In turns out that we can model H-graph grammars faithfully, with the limitation that we cannot represent the containment of edges in the derived hierarchical graphs. This points out an interesting problem, which we discuss in the summary to this section (see Subsection 7.1.6). A short version of this work has appeared in [BH01].

This section is organized as follows. In Subsection 7.1.1, we recall and illustrate the notion of an H-graph. In Subsection 7.1.2, we show how we can model H-graphs using

our hierarchical graph model. In Subsection 7.1.3, we recall the notion of an H-graph grammar, and in Subsection 7.1.5 we illustrate how these grammars can be simulated using our framework to hierarchical graph transformation. In the intermediate Subsection 7.1.4, we recall NLC rewriting and we define a variety of NLC grammars that we use in Subsection 7.1.5.

7.1.1 H-Graphs

In [Pra79] and [Pra83], hierarchical graphs (called H-graphs) are used to model complex runtime data structures in the definition of programming language semantics. In this subsection, we summarize this notion and illustrate it with an example.

H-graphs are directed, node- and edge-labelled hierarchical graphs where the hierarchy is built by allowing node labels to contain nested graphs. Therefore, this approach is coupling (compare Subsection 2.3.4). Cycles are allowed in the hierarchy structure, thus this approach is one of the few that does not restrict to dag-like hierarchical structures (compare the distributed systems approach, in Subsection 7.2.2).

Before recalling the definition of H-graphs, as it is found in [Pra79], we make a few remarks on notation. In order to facilitate cross reference with the original papers on H-graph, we try to keep the same notation and terminology, although some adaptations are needed to avoid clashes with the notation and terminology used for our hierarchical graphs. In particular, we have:

- In [Pra79], the terms *atom*, *terminal atom* and *non-terminal atom* are used. Instead, to avoid confusion with a set of atoms in a hierarchical graph, we will use the terms *symbol*, *terminal symbol*, *non-terminal symbol*, respectively.

- $\mathbf{N}$ will denote a global set of nodes, over which H-graphs are defined, whereas N is used in [Pra79]. Similarly, $\mathbf{A}$ and $\mathbf{B}$ are used instead of A and B, to indicate sets of symbols (alphabets).

Notice that in [Pra79] *extended directed graphs* are used, which are replaced by *directed graphs* here. The only difference is that an extended directed graphs distinguishes one of its nodes as the *initial node* of the graph. This feature is not used in [Pra79], but see e.g. [Pra83]. Also, since in this section we find it more convenient to represent labelled edges as triples (u, λ, v) where u, v are nodes and λ is a label, we provide a new definition for directed graphs here. (Compare Definition 3.3.)

We now define H-graphs.

Definition 7.1 (H-graphs)
Let $\mathbf{N}$ be a predefined set of *nodes* and $\mathbf{A}$ be a predefined set of *symbols* (an *alphabet*).

An *edge-labelled directed graph* X over $\mathbf{N}$ and $\mathbf{A}$ is a pair $X = (N, E)$, where $N \subseteq \mathbf{N}$ is a finite, non-empty *node set*, $E \subseteq N \times \mathbf{A} \times N$ is a finite *arc set*. Given an extended

directed graph X, we indicate its two components as N_X and E_X, respectively. We indicate the set of all directed graphs over $\mathbf{N}$ and $\mathbf{A}$ with $DG(\mathbf{N}, \mathbf{A})$. For our convenience, we also define the functions $s_X, t_X : E_X \to N_X$ and $\imath_X : E_X \to \mathbf{A}$ by letting, for all $e = (u, a, v) \in E_X$, $s_X(e) = u$, $t_X(e) = v$, $\imath_X(e) = a$.

An *H-graph* H over $\mathbf{N}$ and $\mathbf{A}$ is a pair $H = (N, V)$, where the *node set* N is a finite subset of $\mathbf{N}$, and $V : N \to \mathbf{A} \cup DG(\mathbf{N}, \mathbf{A})$ is the content function, mapping each node either to a label or to an internal directed graph.

Again, given an H-graph H, we indicate its components with N_H and V_H respectively. We indicate the set of all H-graphs over a set of nodes $\mathbf{N}$ and the alphabet $\mathbf{A}$ with $HG(\mathbf{N}, \mathbf{A})$.

In Figure 7.1, we show an H-graph over the alphabet

$$\mathbf{A} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{car}, \mathtt{cdr}, \mathtt{nil}, \#\}$$

modeling the list `(B (C A))`.

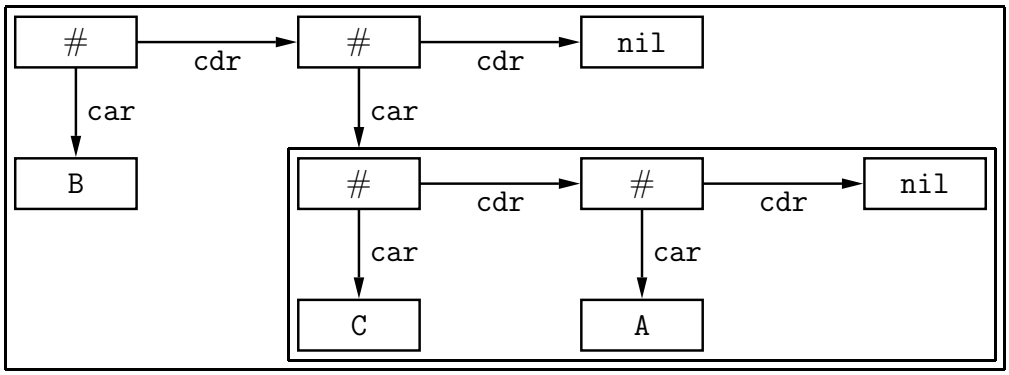


Figure 7.1: An H-graph modeling the list `(B (C A))`.

7.1.2 Modeling H-graphs

In this subsection, we show how an H-graph can be represented using a hierarchical graph obtained by instantiating our framework. We will use directed, node and edge labelled graphs for the three components of a hierarchical graph. Loops are not allowed, while multiple edges are allowed as long as they carry different labels. Given a node alphabet Σ and an edge alphabet Δ, we find it technically more convenient to represent such graphs as triples $G = (N_G, E_G, l_G)$, where $E_G \subseteq N_G \times \Delta \times N_G$, and $l_G : N_G \to \Sigma$. We will still use the notation s_G, t_G, and $\imath_G$ when we want to indicate the source, the target, or the label of an edge explicitly. (Compare Definition 3.3 and Definition 7.1.)

In order to have a uniform notion of a graph, we use the same node- and edge-labelled graphs for underlying graphs and for encoding hierarchy and coupling graphs. The encoding is similar to that used in the example of Chapter 6, since we add labels that are forgotten during decoding. As in Chapter 6, labels in the encoding of coupling graphs are used to distinguish between packages and atoms. We also assume that the set of symbols over which H-graphs are built is partitioned into a set of *terminal* and a set of *non-terminal symbols*. This feature will be used in H-graph grammars for distinguishing between terminal and non-terminal nodes (see Definition 7.3 and Construction 7.5).

When defining transformations on these graphs, we will need to distinguish special nodes, in order to embed the right-hand side of a rule in the host graph. This is done by storing special symbols—called tags—in node labels. Therefore, if a node has label a in an H-graph, it will have label (a, t), where t is an appropriate tag, in the corresponding hierarchical graph. We use the following tags:

— **n** tags *normal* nodes, opposed to *input/output* nodes (see below) inside underlying and the coupling graphs.

— **p** tags *normal* packages inside hierarchy and coupling graphs.

— **e** tags edges.

— **i** tags *input* nodes inside underlying graphs, and *input* packages inside hierarchy graphs.

— **o** tags *output* nodes inside the underlying graph and coupling graphs.

— **io** tags *input/output* nodes inside underlying graphs.

Tags are introduced in Construction 7.2 and their use will be further explained in Subsection 7.1.5.

The translation of H-graphs to hierarchical graphs consists in splitting the information contained in an H-graph into three graphs. The nodes and the edges of an H-graph will be stored in the corresponding underlying graph. For each non-atomic node, a package will be created, and the nesting between packages will be defined according to the nesting of their corresponding nodes. In the coupling graph, packages are anchored to their corresponding nodes, while nodes and edges will be contained in a package, if they are contained in that package's anchor node in the original H-graph. Node and edge labels are carried over from the H-graph to the underlying graph, with tags added to them, as already mentioned. Packages in the hierarchy graph will carry the same labels (but possibly different tags) as their anchor nodes in the underlying graph. Nodes, edges and packages are represented as nodes in the coupling graph, and they keep their labels there as well (again, their tag can be different). Edges in the hierarchy and in the coupling have all the same $(\perp, \mathbf{e})$ label.

Construction 7.2 (Translation of H-graphs) Let $\mathbf{N}$ be a set of nodes, $\mathbf{A}$ a set of *terminal symbols* and $\mathbf{B}$ a set of *non-terminal symbols* with $\mathbf{A} \cap \mathbf{B} = \emptyset$. Let $\mathbf{tags} = \{\mathbf{n}, \mathbf{p}, \mathbf{e}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}$ be a fixed *tag alphabet*.

Given the set $\overline{\Sigma} = \mathbf{A} \cup \mathbf{B}$, we define the node alphabets $\Sigma := \left(\overline{\Sigma} \cup \{\bot\}\right) \times \mathbf{tags}$ (where $\bot \notin \overline{\Sigma}$), and the edge alphabet $\Delta := (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$. Given $(x, t) \in \Sigma$ (resp. $(x, t) \in \Delta$), we call t a *node tag* (resp. *edge tag*). A $(\bot, t)$ label on a node n will indicate that the node is not atomic inside its H-graph H, i.e. $V_H(n)$ is a graph. We restrict the edge labels to $\mathbf{A}$, because edges are not rewritten and therefore carry only terminal labels (see also Definition 7.3 and Construction 7.5). The symbol $\bot$ is used in edge labels in the hierarchy and in the coupling graph. Tags will store embedding information that is used when hierarchical graphs are rewritten.

Given such an alphabet Σ and $s \in \Sigma$ (Δ and $d \in \Delta$, respectively), we indicate with s^1 the first component (with s^2 the second component) of s. Given a set X and a labeling function $l : X \to \Sigma$ ($l : X \to \Delta$, respectively), we indicate with l^1 the function associating $(l(x))^1$ to each $x \in X$, and with l^2 the function associating $(l(x))^2$ to each $x \in X$.

Given an H-graph $H = (N, V)$ over $\mathbf{N}$, we construct the underlying directed graph $G(H) = (N_{G(H)}, E_{G(H)}, l_{G(H)})$ of H, defined as follows:

- We let $N_{G(H)} := N$.
- We let

$$E_{G(H)} := \left\{ (u, (a, \mathbf{e}), v) \mid (u, a, v) \in \bigcup_{n \in N} E_{V(n)} \right\}$$

  where, for $V(n) \in \mathbf{A} \cup \mathbf{B}$, $E_{V(n)} := \emptyset$. Intuitively, we collect all edges from the graphs contained in the nodes of H, and we add an $\mathbf{e}$ tag to them.

- For all $u \in N$, we let

$$l_{G(H)}(u) := \begin{cases} (V(u), \mathbf{n}) & \text{if } V(u) \in \mathbf{A} \cup \mathbf{B} \\ (\bot, \mathbf{n}) & \text{otherwise} \end{cases}$$

  where $\bot$ in the first component marks all nodes in G that are not labelled with a symbol of $\mathbf{A}$ or $\mathbf{B}$, and $\mathbf{n}$ in the second component marks the fact that this is a plain node (as opposed to input/output nodes, see Subsection 7.1.5).

From $G(H)$ and the content function V it is easy to build a hierarchical graph in our sense. We let $\mathcal{P}$ be a predefined denumerable set of packages, and we take any injective partial function $\Pi : N \to \mathcal{P}$, defined for all and only the $u \in N$ such that $V(u) \notin \mathbf{A}$. Thus Π associates a distinct package to every node that has a graph as its content

(value), or that has a non-terminal label[1] . For all $u \in \mathbf{dom}(\Pi) = \{u \in M \mid V(u) \notin \mathbf{A}\}$ we will write p_u instead of $\Pi(u)$.

With this notation, we define the full hierarchical graph

$$\mathbf{FHG}(H) = (G(H), D(H), FB(H))$$

in the following way:

- $G(H)$ has been defined above.
- $P_{D(H)} = P_{B(H)} := \{p_u \mid u \in N \wedge V(u) \notin \mathbf{A}\}$.
- $E_{D(H)} := \{(p, (\bot, \mathbf{e}), p') \mid \exists u, v \in N : V(u), V(v) \notin \mathbf{A} \cup \mathbf{B} \wedge p = p_u \wedge p' = p_v \wedge v \in N_{V(u)}\}$. Intuitively, a package p' is nested in a package p—$(p, (\bot, \mathbf{e}), p') \in E_{D(H)}$— if and only if p and p' are associated to nodes u and v and v is in the graph $V(u)$.
- For all $p_u \in P_{D(H)}$, we let

$$l_{D(H)}(p_u) := \begin{cases} (V(u), \mathbf{p}) & \text{if } V(u) \in \mathbf{B} \\ (\bot, \mathbf{p}) & \text{otherwise} \end{cases}$$

  while, from the definition of $E_{D(H)}$, we have, for all $e \in E_{D(H)}$, $\lambda_{D(H)}(e) = (\bot, \mathbf{e})$.
- $A_{FB(H)} := N_{G(H)} \cup E_{G(H)}$.
- $E_{FB(H)} := \{(x, (\bot, \mathbf{e}), p_u) \mid x \in A_{FB(H)} \wedge p_u \in P_{FB(H)} \wedge x \in V(u)\} \cup \{(p_u, (\bot, \mathbf{e}), u) \mid u \in N_{G(H)}, V(u) \notin \mathbf{A}\}$, i.e. an atom (node or edge) is contained in a package p (this is modeled by a node-to-package edge) if and only if p is associated to a node u and the considered atom is in the graph $V(u)$, and every package p_u is anchored to node u (this is modeled by a package-to-node edge).
- For all $p \in P_{D(H)}$, we let $l_{FB(H)}(p) = l_{D(H)}(p)$, and for all $u \in N_{G(H)}$, we let $l_{FB(H)}(u) := l_{G(H)}(u)$. For all $e \in E_{G(H)}$, we let $l_{FB(H)}(e) := \lambda_{G(H)}(e)$. Notice that, for $x \in N_{FB(H)}$, we can distinguish packages of $D(H)$ from nodes and edges of $G(H)$ by their tags, $\mathbf{p}$, $\mathbf{n}$, and $\mathbf{e}$, respectively.

Given a H-graph H, we also define a *weak* translation to a plain hierarchical graph $\mathbf{HG}(H) = (G(H), D(H), B(H))$, where $G(H)$ and $D(H)$ are the same as for $\mathbf{FHG}(H)$, while

$$\begin{aligned} A_{B(H)} &= N_{G(H)} \\ P_{B(H)} &= P_{FB(H)} \\ E_{B(H)} &= E_{FB(H)} \cap \left(A_{B(H)} \times \Delta \times P_{B(H)} \cup P_{B(H)} \times \Delta \times A_{B(H)}\right) \\ l_{B(H)} &= l_{FB(H)}|_{N_{B(H)}} \end{aligned}$$

[1]Notice that we associate a package to every node labelled with a non-terminal symbol $b \in \mathbf{B}$. The reason for this will be clear when translating H-graph grammars in Subsection 7.1.5.

Therefore, the weak translation of H to $\mathbf{HG}(H)$ "forgets" the placement of edges in the hierarchy, while all the other information contained in H is translated as in $\mathbf{FHG}(H)$.

With respect to the conditions on hierarchical graphs stated in Definition 3.7 (page 47), we observe that the definition of an H-graph does not require that hierarchies be rooted dags (the function V is not restricted in any way), and therefore the corresponding hierarchical graphs may not satisfy this restriction as well. This does not motivate dropping the corresponding conditions in Definition 3.7 since

- most approaches require that hierarchies be acyclic, and we believe this is a reasonable choice in general;
- all H-graphs generated by a particular H-graph grammar *are* indeed rooted (see Subsection 7.1.3).

Another small problem in our translation is that all root packages are anchored to a node that is not contained in any package. This would be a problem and would require the introduction of an additional "dummy" root package, containing everything. This would also solve the problem of multiple roots. We avoid such a solution because it would not bring about any substantial information and would add useless complexity to our translation.

In Figure 7.2, we show the full hierarchical graph $\mathbf{FHG}(H)$ corresponding to the H-graph H of Figure 7.1. Notice that:

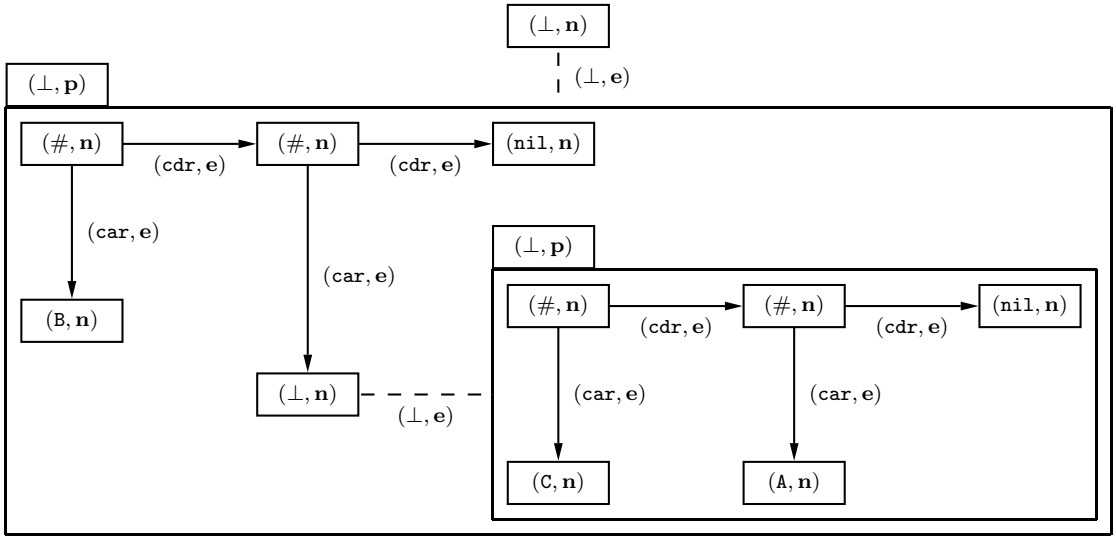


Figure 7.2: Translation of the H-graph of Figure 7.1.

- All nodes from N_H are in $N_{G(H)}$, and complex nodes have become normal nodes with label $(\perp, \mathbf{n})$.
- Two packages have been created, corresponding to the two complex nodes in H. These packages are labelled with $(\perp, \mathbf{p})$.
- The packages are qualified by the nodes they derive from, and this is indicated by $(\perp, \mathbf{e})$-labelled, dashed edges. These edges are in $FB(H)$.
- The $(\perp, \mathbf{e})$-labelled package nesting edges contained in $E_{D(H)}$ are not depicted explicitly: the nesting is indicated by drawing one package inside the other.
- The $(\perp, \mathbf{e})$-labelled edges of the coupling graph $FB(H)$, indicating to which package each node or edge of the underlying graph is associated, are not depicted either: they are indicated by drawing nodes and edges inside packages.

7.1.3 H-graph grammars

After considering H-graphs and their representation as hierarchical graphs in our sense, we turn our attention to H-graph transformation. We first recall the notion of a H-graph grammar, as defined in [Pra79].

Definition 7.3 (H-graph grammar and direct derivation)
An *H-graph grammar* is a system $F = (\mathbf{B}, \mathbf{A}, S, R)$, where $\mathbf{B}$ is a finite alphabet of *non-terminal symbols*, $\mathbf{A}$ is an alphabet of *terminal symbols*, $S \in \mathbf{B}$ is the *start symbol*, and R is a finite set of *productions*, each member of which is a quintuple (C, K, G, I, O), where

- $C \in \mathbf{B}$ is the *left-hand non-terminal*,
- $K = (N_K, V_K)$ is an H-graph (the *right-hand side*) with node labels in $\mathbf{B} \cup \mathbf{A}$, and arc labels in $\mathbf{A}$,
- $G = (N_G, E_G)$ is an extended directed graph over N_K and $\mathbf{A}$, and
- I and O, the input and output connections, are nodes in the node set of G.

Let $F = (\mathbf{B}, \mathbf{A}, S, R)$ be an H-graph grammar defined as above and H be an H-graph containing a node n with a non-terminal label $V(n) = C \in \mathbf{B}$. Let $p = (C, K, G, I, O) \in R$ be a production of F. Then an H-graph H' is *directly derived* from H by applying production p to node n if and only if H' is obtained from H by:

1. replacing n with a copy of G and a copy of K (i.e. the content function of the nodes of G is provided by K),

2. the arcs entering node n in H are replaced by arcs entering node I of G,

3. the arcs leaving node n in H are replaced by arcs leaving node O of G.

The *H-graph language* of a grammar $F = (\mathbf{B}, \mathbf{A}, S, R)$ is the set $\mathcal{L}(F)$ of all H-graphs without non-terminal symbols that can be derived by subsequent applications of rules in R to the H-graph consisting of one node m which contains one node n with label S. In the sequel, we will abuse notation and indicate this graph with S as well.

In what follows, we provide a construction where we show how H-graph grammars can be interpreted in the generic hierarchical graph transformation framework. For it, we need to formalize the derivation of H-graphs through H-graph grammars more precisely.

Let $p = (C, K, G, I, O)$ be a production as above, with $K = (N_K, V_K)$, and let $H = (N_H, V_H)$ be an H-graph where we suppose, without loss of generality, that $N_H \cap N_K = \emptyset$. Then the H-graph $H' = (N_{H'}, V_{H'})$, derived from H by applying p to a node n of H (with $V_H(n) = C$) is built as follows:

- let $N_{H'} := N_H \setminus \{n\} \cup N_K$,
- let $M_n := \{m \in N_H \mid n \in N_{V(m)}\}$,
- let $V_n : M_n \to EDG(N_H, \mathbf{A})$ be defined as follows: for all $m \in M_n$, let

$$\begin{array}{rcl} N_{V_n(m)} & := & N_{V(m)} \setminus \{n\} \cup N_G \\ E_{V_n(m)} & := & E_{V(m)} \setminus \{(n, a, v), (v, a, n) \mid v \in N_H, a \in \mathbf{A}\} \cup \\ & & \{(O, a, v) \mid (n, a, v) \in E_{V(m)}\} \cup \\ & & \{(v, a, I) \mid (v, a, n) \in E_{V(m)}\} \cup E_G \end{array}$$

- let $V_{H'} := V_H|_{(N_H \setminus (M_n \cup \{n\}))} \cup V_n \cup V_K$,

In the following example (adapted from [Pra79]), we consider some H-graph grammar rules that can be used for deriving list-graphs like the one depicted in Figure 7.1. The necessary rules are depicted in Figure 7.3. Rule r_2 serves to extend a given list with a new element, while rule r_1 terminates a list. Rules r_3 and r_4 expand a list element to an atom or to a sublist, respectively. Rules r_5 and r_6 assign an atom node its terminal label.

We look closer at rule r_4, because it is the only one to have a non-flat right-hand side. In this rule the left-hand non-terminal is $C = \mathit{list\_element}$; the graph K consists of three nodes, n_1 labelled with #, n_3 with *list*, and n_2 containing n_3; the graph G (the root graph of the right-hand side) contains n_1 and n_2, together with the edge $(n_1, \texttt{car}, n_2)$; the input and output nodes coincide, namely we have $I = O = n_1$.

In Figure 7.4, we depict the first steps in the derivation of the H-graph corresponding to the list `(B (C A))`. We consider in particular the last step, in which r_4 is applied.

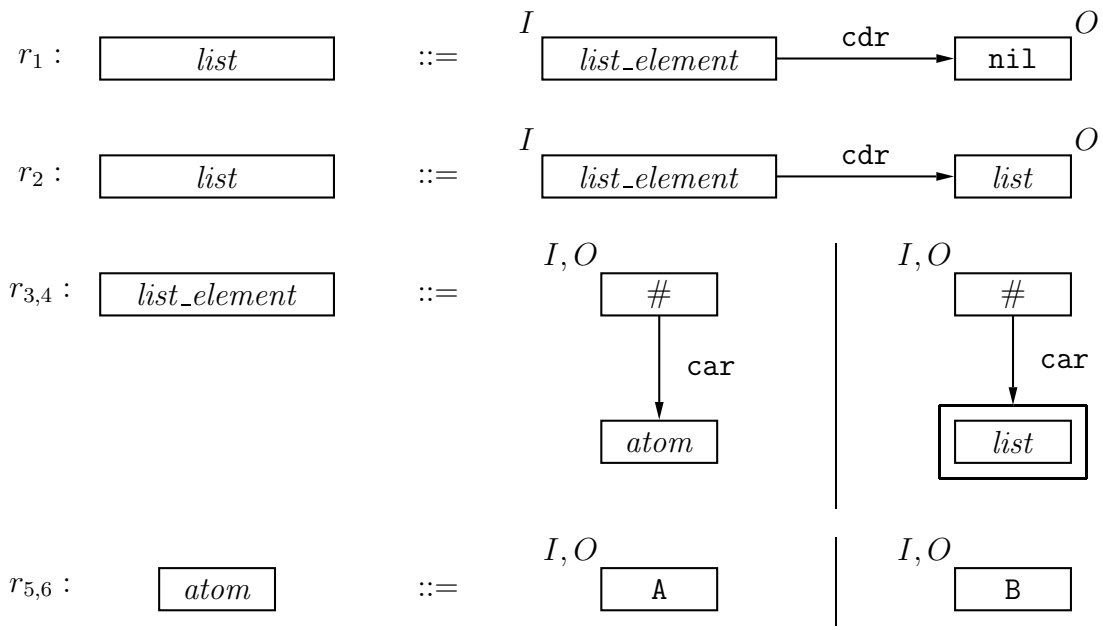


Figure 7.3: Some rules of an H-graph grammar.

The node labelled with *list_element* is replaced with the graph K in the right-hand side of r_4, i.e. the three nodes n_1, n_2, n_3 are added, and n_3 is a node in the label (value) of n_2. The elements of G, i.e. n_1, n_2 and the edge between them, are in the same level of the as the replaced node was, i.e. directly inside the root node. The incoming and outgoing edges are redirected to $n_1 = I = O$.

7.1.4 NLC grammars

In order to model H-graph grammars in our framework, we need to provide three graph transformation approaches with which Construction 5.6 of Chapter 5 can be instantiated. To this purpose, we have chosen NLC rewriting for all the three approaches. In Subsection 7.1.5, we will motivate this choice, while in this subsection we describe NLC rewriting in more detail.

In the *node-label controlled* (NLC) approach to graph rewriting (see also Subsection 4.2.1), an induced subgraph (the *mother graph*) of the host graph is replaced with a new graph (the *daughter graph*). Much research on NLC rewriting deals with the restricted case where the left-hand side graph only contains one node (see e.g. [Jan83], see also [ER97] for an introduction to node replacement in general, and to NLC node-replacement in particular). Rules are of type $L \to R$ (where L and R are graphs), or

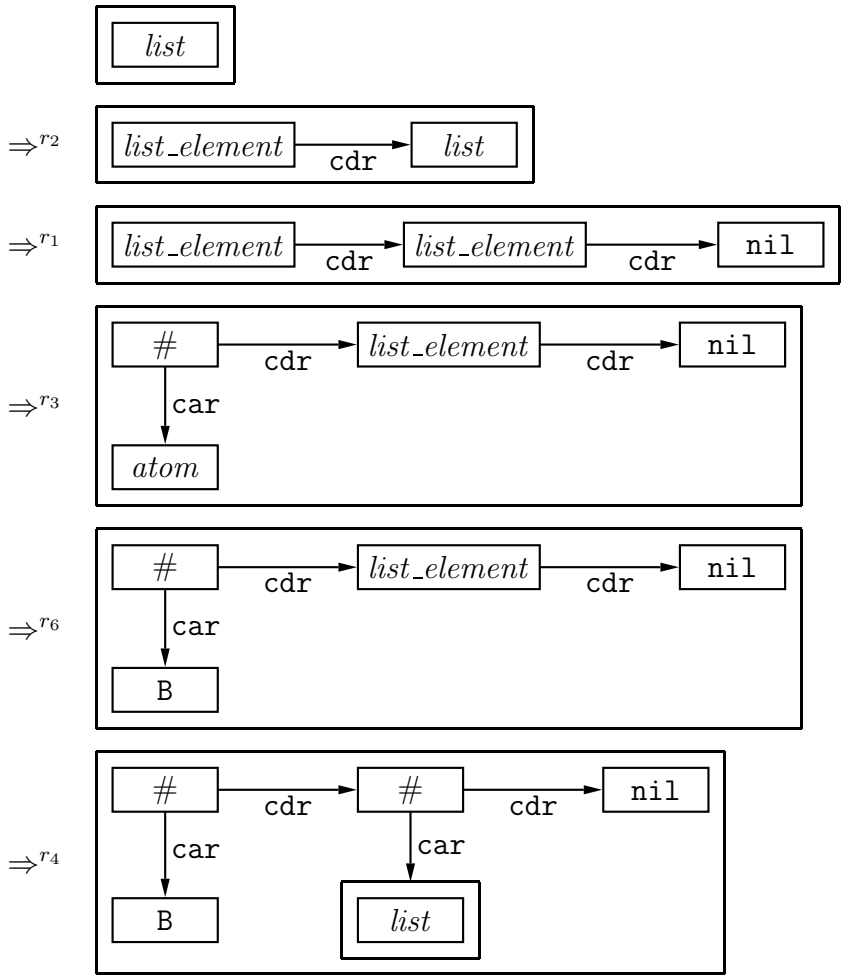


Figure 7.4: Deriving the list `(B (C A))`

$X \rightarrow R$ (where X is a node label) in the case of node replacement.

During an NLC rewriting step, the daughter graph is connected to the *context* (*neighborhood*) of the mother graph, i.e. to the nodes that are adjacent to the mother graph, by establishing new edges. In the restricted case of node replacement, the creation of these edges is controlled by the labels of the neighbor nodes, and by the labels of nodes in the daughter graph. This is specified by *connection instructions* that are part of the grammar. Connection instructions are tuples of the type (κ, δ), where κ and δ are node labels. Such an instruction specifies that we should create an edge linking every κ-labelled neighbor (context node) of the replaced node with all δ-labelled nodes in the daughter graph.

An extension to the basic NLC rewriting mechanism uses directed graphs. In this case, connection instructions contain an extra component $d \in \{in, out\}$, which allows

to distinguish between incoming and outgoing edges w.r.t. the mother graph. We will need this feature, since the three components of $\mathbf{HG}(H)$, for an H-graph H, are all directed graphs. When using edge-labelled graphs, we can further refine connection instructions so that they handle these labels as well. The resulting instructions for directed, node and edge-labelled graphs, are tuples of the type $(\kappa, p/q, \delta, d)$, where p and q are edge labels, meaning that we can apply the instruction to a p-labelled edge, and create corresponding q-labelled connecting edges. (See again [ER97].)

Although we use edge-labelled graphs, in our case these labels do not influence the embedding process: There is no control on the label of connecting edges, and the labels of new connecting edges are inherited from the labels of the corresponding edges that connected the context graph to the mother graph. This choice is suitable both for transformations of the underlying graph, where embedding simulates the embedding mechanism of H-grammars based on I and O nodes, and for transformations of the hierarchy and coupling graphs, whose edges are in any case unlabelled. Therefore, we will not need the p/q-component in connection instructions for translating H-graph grammars.

As far as the choice between node-replacement and the more general graph-replacement, we observe the following. Node-replacement is sufficient for the specification of transformations on underlying and hierarchy graphs:

- For every rewriting step of an H-graph grammar, exactly one node is replaced in the underlying graph.

- If during node replacement, new packages are added to the hierarchy, this operation on the hierarchy graph can be simulated by replacing a "leaf" package in the hierarchy graph with the wanted subhierarchy.

On the other hand, modeling the transformation of the coupling graph, can possibly require that, for each transformation step, a node and a package be replaced (see Subsection 7.1.5 for more details). Therefore we looked for a suitable extension of NLC node-replacement to the more general case of graph replacement.

In [JR82], grammars with *neighborhood-controlled embedding* (NCE) are investigated. These allow arbitrary graphs as left-hand sides of rules. The embedding mechanism is very close to that of NLC, the key differences being that the embedding mechanism of NCE can distinguish single nodes inside the right-hand sides of rules, whereas NLC can only distinguish nodes with different labels. NCE grammars are more powerful than NLC ones, since we can always simulate NLC connection instructions with NCE connection instructions (see again [ER97]). Although these grammars would be a candidate for our purposes, we prefer to look for a formalism using the less powerful NLC embedding mechanism, thus showing that the "very basic" embedding provided by H-graph grammars can be simulated without using the full power of NCE rewriting.

A generalization of NLC to graph replacement can be found in [GJRT84]. In this paper, *generalized handle grammars* (GH grammars) are used, which use parallel substitution of graphs with a particular structure called *generalized handles*. These grammars are still not optimal for our purpose—for example, we do not need parallel application of rules, with the correspondingly complex embedding mechanism—so we decided to define our own notion of NLC rewriting. The main differences between this notion of NLC grammars and GH grammars concern

- the application of rules: GH grammars allow parallel application of rules, while our grammars are strictly sequential,
- and the kind of connection instructions: local to rules in GH grammars, global and with a finer control on node labels in our case (GH grammar do not distinguish the labels of nodes inside the mother graphs).

Distinguishing between labels of nodes inside mother graphs will allow to distinguish nodes from packages when transforming coupling graphs (see Construction 7.5).

Summarizing, our variety of NLC graph grammars use directed, node- and edge-labelled graphs, graph replacement rules, and a global set of embedding instruction of type (κ, μ, δ, d), whose components indicate a label in the context graph, a label in the mother graph, a label in the daughter graph, and an edge direction out of the set $\{in, out\}$, respectively. Edges labels are not checked during embedding, but they are carried over to the new edges from the corresponding deleted edges.

In the rest of this thesis, the terms *NLC rewriting*, *NLC graph grammar*, and the like, indicate the NLC grammars defined here, unless different indications are provided. Throughout this section, the term *directed node- and edge-labelled graph* refers to the same notion used in Subsection 7.1.2.

Definition 7.4 (NLC grammar)
A *node-label controlled* (*NLC*) *graph grammar* is a system

$$\Gamma = (\Sigma, T, \Delta, P, C, S)$$

where

- Σ is an alphabet of *node labels*, which is partitioned into two non-empty sets $T \subset \Sigma$ of *terminals* and $\Sigma \setminus T$ of *non-terminals*;
- Δ is a set of *edge labels*;
- P is a set of *productions* of the form $p : L \to R$, where L and R are graphs over Σ and Δ;
- $C \subseteq \Sigma \times \Sigma \times \Sigma \times \{in, out\}$ is a global *connection relation*, where every tuple $(\kappa, \mu, \delta, d) \in C$ is a *connection instruction*;

- S is a directed, node- and edge-labelled graph over Σ and Δ called the *start graph*.

Given a production $p : L \to R$, with L containing exactly one node with label $X \in \Sigma$, we will often use the alternate notation $p : X \to R$.

Given a graph G over Σ and Δ, and a production $p : L \to R \in P$, we can apply p to G in the following way:

- find two graphs L' and R' that are isomorphic to L and R, respectively, i.e. the isomorphisms[2] $g : L \to L'$ and $h : R \to R'$, where L' is an induced subgraph of G, and R' is disjoint from G;
- let D be the graph:
  - $N_D := N_G \setminus N_{L'}$,
  - $E_D := \{(u, \lambda, v) \in E_G \mid u, v \in N_D\}$,
  - $l_D := l_G|_{N_D}$;
- let E be the graph $D \cup R'$;
- let H be the graph:
  - $N_H := N_E$, $l_H := l_E$;
  - $E_H := E_E \cup$
    $\{(z, \lambda, y), z \in N_D, y \in N_{R'} \mid \exists x \in N_{L'} : (z, \lambda, x) \in E_G,$
    $(l_G(z), l_{L'}(x), l_{R'}(y), in) \in C\} \cup$
    $\{(y, \lambda, z), z \in N_D, y \in N_{R'} \mid \exists x \in N_{L'} : (x, \lambda, z) \in E_G,$
    $(l_G(z), l_{L'}(x), l_{R'}(y), out) \in C\}$

    i.e. for every edge (z, λ, x) going from a node z of D to a node x of L', we create an edge (z, λ, y) from z to every node $y \in R'$ such that C contains the instruction $(l_G(z), l_{L'}(x), l_{R'}(y), in)$. Analogously, for every edge (x, λ, z) going from a node x of L' to a node z of D, we create an edge (y, λ, z) to z from every node $y \in R'$ such that C contains the instruction $(l_G(z), l_{L'}(x), l_{R'}(y), out)$.

Given a graph G, a rule p as above, and a graph H obtained by applying p to G, we say that H is *directly derived* from G via p, and we write $G \Rightarrow_p H$. For a graph $\overline{G}$ isomorphic to G and a graph $\overline{H}$ isomorphic to H ($\overline{G} \cong G$, $\overline{H} \cong H$), we say that $\overline{H}$ is directly derived from $\overline{G}$ as well (direct derivation is defined up to isomorphism). If, for $k > 1$, $G = G_0, G_1, \ldots, G_{k-1}$, $G_k = H$ are graphs, $p_1, \ldots, p_k \in P$ are productions, and $G_{i-1} \Rightarrow_{p_i} G_i$ for all i, $1 \leq i \leq k$, we write $G \Rightarrow_P^+ H$. We write $G \Rightarrow_P^* H$ if $G \Rightarrow_P^+ H$ or $G \cong H$. The language of Γ is the set of graphs $\mathcal{L}(\Gamma) := \{G \mid \text{“}G \text{ is a graph over } T \text{ and } \Delta\text{”}, S \Rightarrow_P^* G\}$.

[2] An *isomorphism* $f : G \to H$ between two graphs G and H, is a homomorphism $f = \langle f_{\mathrm{N}}, f_{\mathrm{E}} \rangle$, such that $f_{\mathrm{N}} : N_G \to N_H$ and $f_{\mathrm{E}} : E_G \to E_H$ are both injective and surjective functions.

In Subsection 7.1.5, we will use NLC graph grammars to simulate the transformations defined by H-graph grammars.

7.1.5 Translating H-grammars

In this subsection, we show how H-graph grammar rules can be simulated by instantiating our framework for hierarchical graph transformation appropriately. H-graph grammars use node replacement with some kind of embedding (I/O nodes). If we model the hierarchy as a graph, we see that hierarchy transformation uses a sort of node (package) replacement as well: when a node in the original H-graph is replaced, a new subhierarchy (the graph K) is inserted at the place where the node was. Similarly, the coupling graph is transformed by removing the replaced node and package, inserting new nodes and packages, and the appropriate links between them[3].

It turns out that the NLC embedding mechanism is appropriate for modeling the required embedding in the three component transformation rules. In fact, during each derivation step we need to distinguish between at most four different categories of nodes in the right-hand side of a rule, namely: 1) nodes to which incoming edges must be redirected, 2) nodes to which outgoing edges must be redirected, 3) nodes to which both incoming and outgoing nodes must be redirected, 4) all other nodes. The rules for the underlying graph and for the hierarchy are node replacement rules, while the left-hand sides of coupling graph rules contain two nodes and one edge between them. (See also the discussion in 7.1.4.)

As hinted in footnote 3, some further remarks concerning the transformation of coupling graphs are needed. As shown in in Chapter 4 and Chapter 5 when speaking of coordinated rules, coupling graphs must be transformed in such a way as to reflect the changes to the underlying and the hierarchy graphs. In the case of our translation of H-graph grammars, where H-graphs are coded as full hierarchical graphs, this would mean that deleted/preserved/added nodes and edges of the underlying graph must correspond to deleted/preserved/added atoms in the coupling graph, and, likewise, deleted/preserved/added packages of the hierarchy graph must correspond to deleted/preserved/added packages in the coupling graph.

Notice that underlying graph transformations are specified using NLC rewriting, which can delete and/or add a variable number of edges (the connecting edges) during each derivation step. A coordinated rule on the coupling graph should specify a similar transformation on the set of atoms (nodes), which is not possible using NLC rewriting, where a fixed number of nodes is deleted (the left-hand side of a rule) and added (the right-hand side) during a direct derivation.

This suggests that a faithful translation of H-graph grammars requires a different transformation approach for the coupling graph. For example, the PROGRES approach

[3]The situation with coupling graphs is slightly more complicated, as discussed below.

(see [SWZ99]) allows to match and delete a variable number of nodes by means of *node set patterns*. However, PROGRES does not allow the creation of such a set of nodes. As another example, substitution-based graph rewriting (see [HP96]) allows to match an a-priori unbounded graph with a variable, and either delete it or copy it an arbitrary number of times. Still, by using this approach we would not be able to access single nodes in the copied graphs and appropriately specify their containment relation w.r.t. preserved packages in the coupling graph. The addition of a variable set of nodes (atoms) together with the appropriate connections to packages of the coupling graph is *not* supported by any graph transformation approach up to our knowledge.

Summarizing:

1. Using NLC rewriting as a coupling graph transformation is insufficient for a faithful translation of H-graph grammars.

2. H-graph grammars are an example that motivates the use of different graph transformation approaches for the components of a hierarchical graph transformation approach.

3. There does not exist—up to our knowledge—an appropriate transformation approach for coupling graphs in the translation of H-graph grammars.

While we think that looking for such an approach can be an interesting development of this research, in this work we follow a simpler solution. If we consider the weaker coding of H-graphs, where we use plain hierarchical graphs and we ignore the fact that edges are local to specific component of the hierarchy (see the difference between $\mathbf{FHG}(H)$ and $\mathbf{HG}(H)$ in Construction 7.2), then it turns out that we can use NLC rewriting for coupling graph transformation too. Therefore, we consider this "weak" translation here, leaving a full translation as a topic for future research.

For the translation of each H-graph grammar rule, we need several component rules. This is due to the fact that we use node-label tags (e.g. $\mathbf{n}$, $\mathbf{i}$, $\mathbf{o}$, and so on) to specify the input and output nodes of an H-grammar rule. These tags, once they have been introduced in the host graph, must not have any influence on the following derivation. Therefore, for each H-graph grammar rule π with a non-terminal symbol C on its left-hand side, we provide several underlying graph rules $\gamma_\pi^{\mathbf{n}}$, $\gamma_\pi^{\mathbf{i}}$, $\gamma_\pi^{\mathbf{o}}$, and so on. All these rules have the same right-hand side, while their left-hand side node is labelled with $(C, \mathbf{n})$, $(C, \mathbf{i})$, $(C, \mathbf{o})$, and so on. In this way, there exists always one rule γ_π^t that is applicable, independently of the garbage tag t that is attached to the node to be rewritten. A similar mechanism is used for hierarchy and coupling graph rules.

In Construction 7.5, we show how to translate H-graph grammar rules to hierarchical graph rules made of triples of NLC rules. Formal details are freely interspersed with informal illustration in order to provide the intuition behind the definitions. Notice that, as pointed out in the remarks to Definition 7.7, the translated rules can be considered coordinated.

Before proceeding, we introduce some useful notation related to functions. Given two sets A, B, a function $f : A \to B$, and two elements $a \in A$, $b \in B$, the notation $f[a \to b]$ denotes the function $g : A \to B$ defined, for all $x \in \mathbf{dom}(f) \cup \{a\}$, as

$$g(x) := \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$

while the notation $[a \to b]$ is equivalent to $\emptyset[a \to b]$.

Construction 7.5 (Translation of H-graph grammar rules)
Let $F = (\mathbf{B}, \mathbf{A}, S, R)$ be a given H-graph grammar, and let $\pi = (C, K, G, I, O) \in R$ be an H-graph grammar rule. We want to translate such a rule to a triple of rules $(\gamma_\pi, \delta_\pi, \beta_\pi)$ in our framework. All three rules will be NLC rules (see 7.1.4).

We first consider the underlying graph rules $\gamma^x_\pi = (L^x_{\gamma,\pi}, R_{\gamma,\pi})$ $(x \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\})$. The left-hand side graphs are, for $x \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}$,

$$L^x_{\gamma,\pi} = (\{n\}, \emptyset, \emptyset, \emptyset, [n \to (C, x)], \emptyset)$$

i.e. graphs with one node, labelled with the non-terminal C and tagged with all possible tags. The right-hand side is the same graph $R_{\gamma,\pi}$ for all rules, and it is defined as:

- $N_{R_{\gamma,\pi}} := N_K$,
- $E_{R_{\gamma,\pi}} := E_{G(K)} \cup E_G$,
- for all $n \in N_{R_{\gamma,\pi}}$, we let

$$l_{R_{\gamma,\pi}} := \begin{cases} l_{G(K)}(n) & \text{if } n \notin \{I, O\} \\ (l^1_{G(K)}(n), \mathbf{i}) & \text{if } n = I \neq O \\ (l^1_{G(K)}(n), \mathbf{o}) & \text{if } n = O \neq I \\ (l^1_{G(K)}(n), \mathbf{io}) & \text{if } n = O = I \end{cases}$$

i.e. $R_{\gamma,\pi}$ is the underlying graph of K, with the edges of G added to it, and with the input and output nodes tagged by the appropriate label. Notice that the node label of $R_{\gamma,\pi}$ codes the original label in K in the first component (with $\perp$ indicating that the node has a non-atomic label) while in the second component it stores information about the node being the input node I, the output node O, both the input and output node, or any other node.

There are four underlying-graph rules corresponding to π, one for each possible node label containing C:

$$\begin{aligned} \gamma^{\mathbf{n}}_\pi &: L^{\mathbf{n}}_{\gamma,\pi} \to R_{\gamma,\pi} \\ \gamma^{\mathbf{i}}_\pi &: L^{\mathbf{i}}_{\gamma,\pi} \to R_{\gamma,\pi} \\ \gamma^{\mathbf{o}}_\pi &: L^{\mathbf{o}}_{\gamma,\pi} \to R_{\gamma,\pi} \\ \gamma^{\mathbf{io}}_\pi &: L^{\mathbf{io}}_{\gamma,\pi} \to R_{\gamma,\pi} \end{aligned}$$

and the connection relation associated to π is

$$\begin{aligned}\mathcal{C}_{\gamma,\pi} \;:=\; & \{((x_1,y_1),(C,y_2),(x_3,\mathbf{i}),in) \mid \\ & x_1,x_3 \in B \cup A \cup \{\bot\}, y_1,y_2 \in \{\mathbf{n},\mathbf{i},\mathbf{o},\mathbf{io}\}\} \cup \\ & \{((x_1,y_1),(C,y_2),(x_3,\mathbf{o}),out) \mid \\ & x_1,x_3 \in B \cup A \cup \{\bot\}, y_1,y_2 \in \{\mathbf{n},\mathbf{i},\mathbf{o},\mathbf{io}\}\}\end{aligned}$$

while the global connection relation for the underlying graph grammar is

$$\mathcal{C}_\gamma := \bigcup_{\pi \in R} \mathcal{C}_{\gamma,\pi}$$

Notice again that the tags contained in node labels are relevant in the right-hand side of a rule, while they are irrelevant in the left-hand side and in the derived graph. In fact, while the symbols **n**, **i**, **o**, **io** in a right-hand side graph are used by the connection relation to redirect edges during a derivation step, these symbols are just garbage afterwards. This is why we need several NLC rules for each H-grammar rule: We must be able to apply a rule to a node with label (C,t), no matter what the value of the tag t is. The situation for hierarchy and for connection rules is similar.

As an example, consider the rule $\pi = r_4$ of Figure 7.3. In Figure 7.5, we depict the corresponding underlying graph rule $\gamma^{\mathbf{n}}_{r_4}$. The left-hand side has an **n** tag, while rules

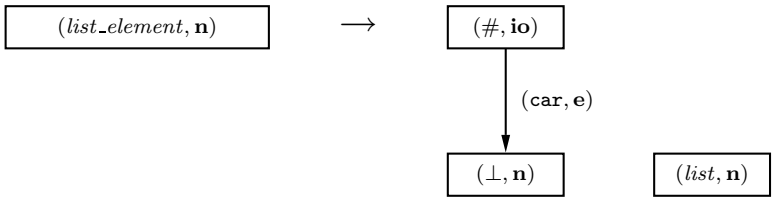


Figure 7.5: The rule $\gamma^{\mathbf{n}}_{r_4}$.

γ^x_π, for $x \in \{\mathbf{i},\mathbf{o},\mathbf{io}\}$ use the other tags. The # node is tagged with **io**, since in the original H-graph rule it was both input and output node of the production. All other nodes are tagged with **n**.

Given an H-graph grammar $F = (\mathbf{B},\mathbf{A},S,R)$, the underlying graph NLC grammar is $F_\gamma = (\Sigma_\gamma, T_\gamma, \Delta_\gamma, P_\gamma, C_\gamma, S_\gamma)$, where

- $\Sigma_\gamma = (\mathbf{A} \cup \mathbf{B} \cup \{\bot\}) \times \mathbf{tags}$, as defined above,
- $T_\gamma = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags} \subseteq \Sigma_\gamma$,
- $\Delta_\gamma = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags} = T_\gamma$, as defined above,

- $P_\gamma = \{\gamma_\pi^t \mid t \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}, \pi \in R\}$,
- C_γ has been defined above,
- $S_\gamma = G(S)$.

As a second step, we define the two hierarchy rules corresponding to π. In hierarchy rules, tags are used to connect the newly added packages to the proper ancestors. These ancestor packages are exactly the ones that contained the replaced node n in the underlying graph. Since n is a non-terminal node, it has a corresponding non-terminal package in the hierarchy graph, which is anchored to it in the coupling graph. Tags are used in hierarchy graph rules to distinguish between normal (internal) packages (tagged with **p**) and root packages of the right-hand side (tagged with **i**), which must be embedded in the hierarchy graph by redirecting hierarchy edges. The **i** tag indicates that incoming hierarchy edges that were incident in the replaced package must be redirected to these packages, which become then children of the packages that are the sources of these edges.

The left-hand side graphs are, for $x \in \{\mathbf{p}, \mathbf{i}\}$,

$$L_{\delta_\pi^x} = (\{q\}, \emptyset, \emptyset, \emptyset, [q \to (C, x)], \emptyset)$$

while the right-hand side is the graph $R_{\delta,\pi}$, defined as:

- $N_{R_{\delta,\pi}} := N_{D(K)}$,
- $E_{R_{\delta,\pi}} := E_{D(K)}$,
- for all $q \in N_{R_{\delta,\pi}}$, we let

$$l_{R_{\delta,\pi}}(q) := \begin{cases} (V_K(u), \mathbf{i}) & \text{if } u \in N_G,\ q = p_u,\ V_K(u) \in \mathbf{B} \\ (V_K(u), \mathbf{p}) & \text{if } u \notin N_G,\ q = p_u,\ V_K(u) \in \mathbf{B} \\ (\bot, \mathbf{i}) & \text{if } u \in N_G,\ q = p_u,\ V_K(u) \notin \mathbf{B} \\ (\bot, \mathbf{p}) & \text{if } u \notin N_G,\ q = p_u,\ V_K(u) \notin \mathbf{B} \end{cases}$$

  i.e. all packages have the same label as in $D(K)$, excepting the ones that are anchored to a node in N_G, which are tagged as "input" packages by an **i** in the second component of their label.

Then we define the hierarchy rewrite rules as

$$\begin{aligned} \delta_\pi^{\mathbf{p}} &: (C, \mathbf{p}) \to R_{\delta,\pi} \\ \delta_\pi^{\mathbf{i}} &: (C, \mathbf{i}) \to R_{\delta,\pi} \end{aligned}$$

with the connection relation associated to π being

$$\mathcal{C}_{\delta,\pi} := \{((x_1, y_1), (C, y_2), (x_3, \mathbf{i}), in) \mid x_1, x_3 \in (\mathbf{B} \cup \{\bot\}), y_1, y_2 \in \{\mathbf{p}, \mathbf{i}\}\}$$

and the global hierarchy connection relation being

$$\mathcal{C}_\delta := \bigcup_{\pi \in R} \mathcal{C}_{\delta,\pi}$$

As an example, consider again the rule $\pi = r_4$ of Figure 7.3. In Figure 7.6, we depict the corresponding hierarchy graph rule $\delta^{\mathbf{p}}_{r_4}$. In this rule, the left-hand side contains one

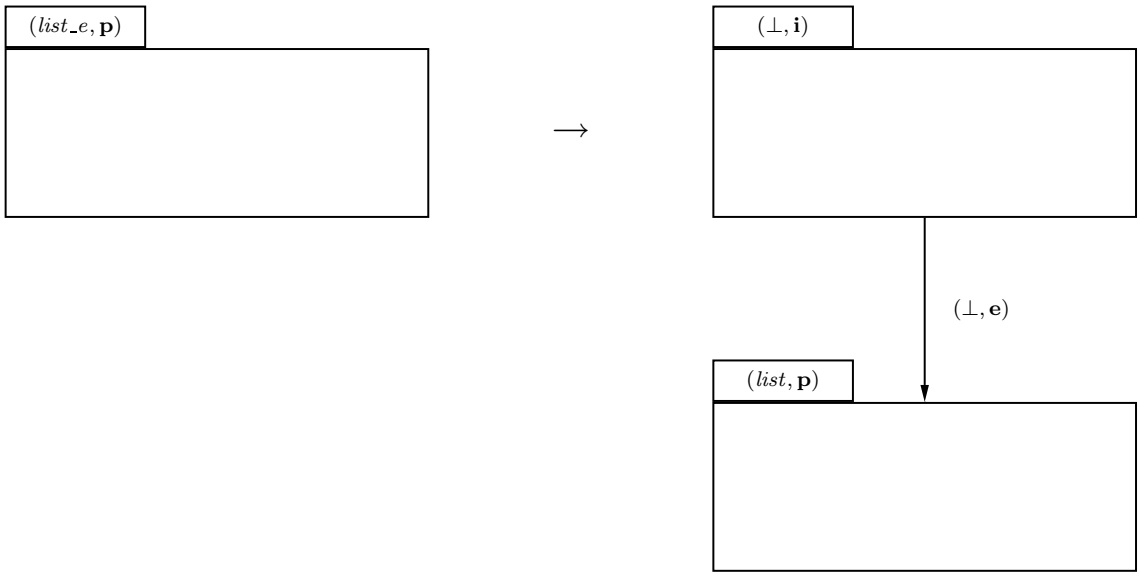


Figure 7.6: The rule $\delta^{\mathbf{p}}_{r_4}$.

package with label $(\mathit{list\_element}, \mathbf{p})$. This will match a "dummy" package p in the graph to be rewritten: p should contain no nodes and works as a place-holder, which indicates that we can add a new subhierarchy at that point in the hierarchy. Dummy packages are labelled with non-terminal symbols and will disappear when the end of a derivation is reached.

The dummy package matched by the left-hand side is replaced by the subhierarchy depicted in the right-hand side of the rule. A "real" package is labelled with $(\bot, \mathbf{i})$, where the $\mathbf{i}$ tag ensures that all hierarchy edges incoming into the replaced package p are now incoming into this new package. As a result the new subhierarchy is inserted exactly where the replaced package p lay.

The right-hand side also contains a new dummy package, labelled with $(\mathit{list}, \mathbf{p})$. This ensures that we can expand the hierarchy inside the $(\bot, \mathbf{i})$-package whenever the $(\mathit{list}, \mathbf{n})$-node in the underlying graph (see Figure 7.5) is further rewritten.

Given an H-graph grammar $F = (\mathbf{B}, \mathbf{A}, S, R)$, the hierarchy graph NLC grammar is $F_\delta = (\Sigma_\delta, T_\delta, \Delta_\delta, P_\delta, C_\delta, S_\delta)$, where

- $\Sigma_\delta = (\mathbf{A} \cup \mathbf{B} \cup \{\bot\}) \times \mathbf{tags}$, as defined above,
- $T_\delta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags} \subseteq \Sigma_\delta$,
- $\Delta_\delta = T_\delta$, as defined above,
- $P_\delta = \{\delta^t_\pi \mid t \in \{\mathbf{p}, \mathbf{i}\}, \pi \in R\}$,
- C_δ has been defined above,
- $S_\delta = D(S)$.

We now consider the transformation of the coupling graph. In this case, we need to replace a non-terminal node together with its qualified package and the edge between them. Of the two nodes in the left-hand side, the "underlying graph" node can have one out of the two tags **n**, **o** for the graph node, while the "package" node has a fixed tag **p**. The usage of these tags will be explained shortly. As a result of these different tags, we need two rules for the coupling graph, all with the same right-hand side.

Given the H-grammar rule π as above, we define, for $x \in \{\mathbf{n}, \mathbf{o}\}$, the following left-hand-side graphs $L_{\beta^x_\pi}$:

- $A_{L_{\beta^x_\pi}} := \{u\}$,
- $P_{L_{\beta^x_\pi}} := \{q\}$,
- $E_{L_{\beta^x_\pi}} := \{(q, (\bot, \mathbf{e}), u)\}$,
- $l_{L_{\beta^x_\pi}}(u) := (C, x)$, $l_{L_{\beta^x_\pi}}(q) := (C, \mathbf{p})$, i.e. both the node and the package are labelled with the left-hand-side non-terminal of π, and the node can take the tags **n**, **o**,

and the following right-hand-side graph R_{β_π}:

- $A_{R_{\beta_\pi}} := N_K$,
- $P_{R_{\beta_\pi}} := N_{R_{\delta,\pi}}$,
- $E_{R_{\beta_\pi}} := E_{B(K)}$,
- for all $x \in N_{R_{\beta_\pi}} = A_{R_{\beta_\pi}} \cup P_{R_{\beta_\pi}}$, we let

$$l_{R_{\beta_\pi}}(x) := \begin{cases} (V_K(x), \mathbf{o}) & \text{if } x \in N_G,\ V_K(x) \in \mathbf{B} \cup \mathbf{A} \\ (V_K(x), \mathbf{n}) & \text{if } x \in N_K \setminus N_G,\ V_K(x) \in \mathbf{B} \cup \mathbf{A} \\ (\bot, \mathbf{o}) & \text{if } x \in N_G,\ V_K(x) \notin \mathbf{B} \cup \mathbf{A} \\ (\bot, \mathbf{n}) & \text{if } x \in N_K \setminus N_G,\ V_K(x) \notin \mathbf{B} \cup \mathbf{A} \\ (V_K(u), \mathbf{p}) & \text{if } x = p_u,\ u \in N_K,\ V_K(u) \in \mathbf{B} \\ (\bot, \mathbf{p}) & \text{if } x = p_u,\ u \in N_K,\ V_K(u) \notin \mathbf{B} \end{cases}$$

i.e. all nodes and packages of R_{β_π} have the same label as they would have in $B(K)$, excepting the nodes that belong to N_G, which are tagged as "output" nodes. This tagging ensures that these nodes are contained in (connected to) the same packages as the replaced node in the left-hand side of the rule. Notice that the tagging allows to distinguish between atoms and packages of R_{β_π}: the former have tags in $\{\mathbf{n}, \mathbf{o}\}$, while the latter have tag $\mathbf{p}$.

For $x \in \{\mathbf{n}, \mathbf{o}\}$, we have the two coupling rules corresponding to π

$$\beta_\pi^x : L_{\beta_\pi^x} \to R_{\beta_\pi}$$

The connection relation corresponding to a rule π is

$$\begin{aligned} \mathcal{C}_{\beta,\pi} \quad := \quad & \{((x_1, \mathbf{p}), (x_2, y_2), (x_3, \mathbf{o}), \mathit{out}) \mid \\ & x_1, x_2, x_3 \in (\{\bot\} \cup \mathbf{B} \cup \mathbf{A}), y_2 \in \{\mathbf{n}, \mathbf{o}\}\} \end{aligned}$$

and the global connection relation is

$$\mathcal{C}_\beta := \bigcup_{\pi \in R} \mathcal{C}_{\beta,\pi}$$

As a last example, consider again the rule $\pi = r_4$ of Figure 7.3. In Figure 7.7, we depict the corresponding coupling graph rule $\beta_{r_4}^{\mathbf{p}}$. This rule has been obtained from r_4 as follows:

- The replaced node in the original H-graph grammar, with label *list_element*, has a corresponding node n in the left-hand side of β_{r_4}, with label $(\mathit{list\_element}, \mathbf{n})$. (Compare rule γ_{r_4}.)
- A corresponding package with label $(\mathit{list\_element}, \mathbf{p})$ is also in the left-hand side of β_{r_4}. (Compare rule γ_{r_4}.) This package is anchored to node n, as indicated by the dashed arrow. This ensures that a node is substituted together with its package. Recall also that non-terminal nodes have always a non-terminal "dummy" package anchored to them. All non-terminal "dummy" packages eventually disappear when their qualifying nodes are replaced with a graph containing only terminal nodes.
- The right-hand side contains the three nodes that are inserted during the rewrite step. These are the nodes in N_K, taken from the rule $\pi = (C, K, G, I, O)$, and exactly the same nodes added by γ_π. The nodes that are in the top level inside the right-hand side of π—i.e. the ones that are in N_G—are tagged with $\mathbf{o}$. In this way (see also the connection instructions in $\mathcal{C}_{\beta,\pi}$), we ensure that the top level nodes are inserted in the same packages that contained the replaced node n: the coupling edges are redirected from this node to the $\mathbf{o}$-tagged ones.

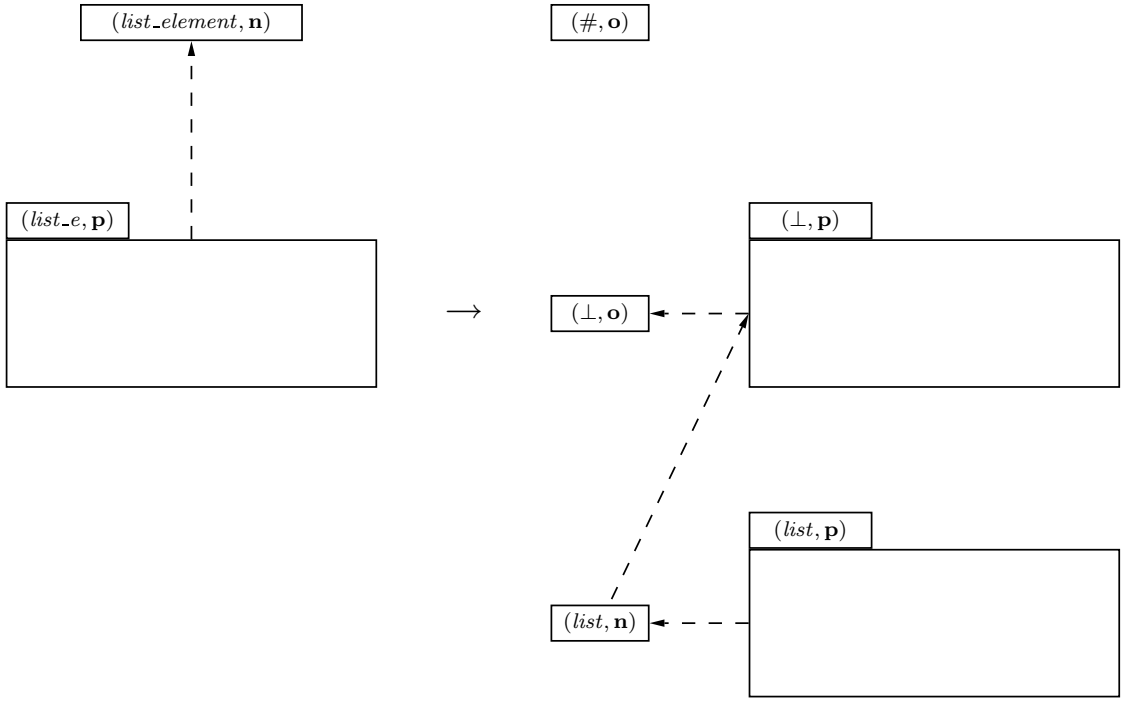


Figure 7.7: The rule $\beta^{\mathbf{n}}_{r_4}$.

- The right-hand side also contains two packages, one for the complex (terminal) $(\perp, \mathbf{o})$-labelled node, and one for the non-terminal $(\mathit{list}, \mathbf{n})$-labelled node in the right-hand side. These packages are exactly the same that are added by δ_π.
- The right-hand side packages are anchored to the corresponding nodes, as indicated by the dashed edges running from packages to nodes. (Edge labels are omitted, since they are all $(\perp, \mathbf{e})$.) This defines the anchoring of newly added packages.
- The $(\mathit{list}, \mathbf{n})$-labelled node is contained in the $(\perp, \mathbf{p})$-labelled package, as indicated by the dashed edge from the node to the package. In this way, the right-hand side specifies the containment of nodes w.r.t. packages that are created by rule β_π itself. It can be, although this is not the case in the specific example of Figure 7.7, that a node with tag $\mathbf{o}$ is also contained in a package in the right-hand side. This means that the new node is added to the new package, as well as to all the packages that contained the node replaced during the derivation step.

It is clear that this rule specifies all the wanted transformation in the coupling graph. This will be proved in Proposition 7.8.

Given an H-graph grammar $F = (\mathbf{B}, \mathbf{A}, S, R)$, the coupling graph NLC grammar is $F_\beta = (\Sigma_\beta, T_\beta, \Delta_\beta, P_\beta, C_\beta, S_\beta)$, where

- $\Sigma_\beta = (\mathbf{A} \cup \mathbf{B} \cup \{\bot\}) \times \mathbf{tags}$, as defined above,
- $T_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags} \subseteq \Sigma_\beta$,
- $\Delta_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags} = T_\beta$, as defined above,
- $P_\beta = \{\beta_\pi^t \mid t \in \{\mathbf{n}, \mathbf{o}\}, \pi \in R\}$,
- C_β has been defined above,
- $S_\beta = B(S)$.

The construction presented up to now allows to translate every H-graph grammar rule to triples of NLC rules forming hierarchical graph rules in our sense. We are left with the task of combining such rules, and defining a hierarchical graph grammar. Given an H-graph grammar $F = (\mathbf{B}, \mathbf{A}, S, R)$, the corresponding hierarchical graph grammar $F_{\gamma,\delta,\beta}$ is a system $(S_{\gamma,\delta,\beta}, R_{\gamma,\delta,\beta})$, where

- $S_{\gamma,\delta,\beta} = (S_\gamma, S_\delta, S_\beta)$,
- $R_{\gamma,\delta,\beta} = \{r_\pi \mid \pi \in R\}$, where $r_\pi = \{(\gamma_\pi^{t_1}, \delta_\pi^{t_2}, \beta_\pi^{t_3}) \mid t_1 \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}, t_2 \in \{\mathbf{p}, \mathbf{i}\}, t_3 \in \{\mathbf{n}, \mathbf{o}\}\}$

It easy to see that a rule $(\gamma, \delta, \beta) \in r_\pi$ as above is coordinated in the sense of Subsection 4.3.4 and Subsection 5.2.2. In fact, there is an obvious correspondence between nodes of the left-hand sides, and between nodes of the right-hand sides, respectively.

After showing how to translate H-graph grammar rules to triples of NLC rules, we need to provide the semantics of the translated rules, i.e. to say how they can be applied to hierarchical graphs. Before proceeding, we need a notion of equivalence between the translated graphs. In fact, since some tags are only needed to drive the derivation and we can throw them away when a derivation is complete, we consider hierarchical graphs that only differ for their tagging as equivalent because they code the same H-graph.

We formalize this equivalence in the following definition.

Definition 7.6 (Structural equivalence of translated H-graphs)
Let $\mathcal{H}$ be a set of hierarchical graphs defined as in Construction 7.5, and let $HG = (G, D, B) \in \mathcal{H}$. We define the *untagged* hierarchical graph $\mathbf{ut}(HG) = (\mathbf{ut}(G), \mathbf{ut}(D), \mathbf{ut}(B))$ by letting

- $N_{\mathbf{ut}(G)} := N_G$, $E_{\mathbf{ut}(G)} := E_G$, and, for all $n \in N_G$, $l_{\mathbf{ut}(G)}(n) := (l_G^1(n), \mathbf{n})$,
- $N_{\mathbf{ut}(D)} := N_D$, $E_{\mathbf{ut}(D)} := E_D$, and, for all $n \in N_D$, $l_{\mathbf{ut}(D)}(n) := (l_D^1(n), \mathbf{p})$,

- $N_{\mathbf{ut}(B)} := N_B$, $E_{\mathbf{ut}(B)} := E_B$, and, for all $x \in N_B = A_B \cup P_B$, we let

$$l_{\mathbf{ut}(B)}(x) := \begin{cases} l_B(x) & \text{if } x \in P_B \\ (l^1_B(x), \mathbf{n}) & \text{if } x \in A_B \end{cases}$$

We also define the set $\mathbf{ut}(\mathcal{H}) := \{\mathbf{ut}(HG) \mid HG \in \mathcal{H}\}$.

We say that two hierarchical graphs HG, HG', defined as in Construction 7.2, are *structurally equivalent* if $\mathbf{ut}(HG)$ is isomorphic to $\mathbf{ut}(HG')$. We indicate that two hierarchical graphs HG, HG' are structurally equivalent with the notation $HG \sim HG'$. Notice that, for an H-graph H, $\mathbf{ut}(\mathbf{HG}(H)) = \mathbf{HG}(H)$.

We now look at the way the translated rules are applied to translated hierarchical graphs. The translated rules $r = (\gamma, \delta, \beta)$ and the corresponding transformation steps are coordinated, since

- the NLC approach can be seen as a (trivially) tracking graph transformation approach (see Appendix A),
- the coordination relations $\simeq^L_\gamma$, $\simeq^R_\gamma$, $\simeq^L_\delta$, and $\simeq^R_\delta$ are provided by the labels in the left- and right-hand sides of rules,
- the coordination relations $\simeq^G$, $\simeq^{G'}$, $\simeq^D$, and $\simeq^{D'}$ (see the diagram of Figure 4.9, page 92) are defined by the identity of nodes in the transformed component graphs.

We find it convenient to introduce a special definition to illustrate the instantiation of coordinated derivation in the particular case of translated H-graph grammar rules. In this definition, we avoid mentioning the coordination relations explicitly.

Definition 7.7 (Application of translated rules)
Let H be an H-graph over the alphabets $\mathbf{A}$ and $\mathbf{B}$, and $(G, D, B) = \mathbf{HG}(H)$ be its corresponding hierarchical graph, defined as in Construction 7.2. Let π be an H-graph grammar production, and $r = (\gamma_\pi, \delta_\pi, \beta_\pi) \in r_\pi$.

Then a hierarchical graph (G', D', B') is obtained by a coordinated application of r to (G, D, B)—in symbols $(G, D, B) \Rrightarrow_r (G', D', B')$—if

- there exists $X \in \Sigma$, $t_\gamma \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}$, $t_\delta \in \{\mathbf{p}, \mathbf{i}\}$, $t_\beta \in \{\mathbf{n}, \mathbf{o}\}$ such that
  - $\gamma_\pi = (X, t_\gamma) \to R_\gamma$, where R_γ is a graph over Σ and Δ
  - $\delta_\pi = (X, t_\delta) \to R_\delta$, where R_δ is a graph over Σ and Δ,
  - $\beta_\pi = L_\beta \to R_\beta$, where L_β and R_β are graphs over Σ and Δ, with $L_\beta = (\{n', q'\}, (q', (e, \mathbf{e}), n'), l_{L_\beta})$, with $l_{L_\beta}(n') = (X, t_\beta)$, $l_{L_\beta}(q') = (X, \mathbf{p})$,

- there exist a node $n \in N_G$ and a package $q \in P_D$ with $l_G(n) = (X, t_\gamma)$, $l_D(q) = (X, t_\delta)$, $l_B(n) = (X, t_\beta)$, $l_B(q) = (X, \mathbf{p})$,
- G' is derived from G by applying γ to n,
- D' is derived from D by applying δ to q,
- B' is derived from B by matching L_β with the subgraph of B induced by n and q (notice that there must be an edge from q to n in B, which is matched by the edge between q' and n' in N_{L_β}).

Let R be a set of H-graph grammar rules, and let $R_{\gamma,\delta,\beta} = \{r \in r_\pi \mid \pi \in R\}$ be the corresponding set of hierarchical graph rules. Then, for two hierarchical graphs HG, HG', we write $HG \Rrightarrow_{R_{\gamma,\delta,\beta}} HG'$ if and only if there exists $r \in R_{\gamma,\delta,\beta}$ such that $HG \Rrightarrow_r HG'$. The transitive closure $\Rrightarrow^+_{R_{\gamma,\delta,\beta}}$ and reflexive-transitive closure $\Rrightarrow^*_{R_{\gamma,\delta,\beta}}$ of these relations are defined in an obvious way.

Let $F = (\mathbf{B}, \mathbf{A}, S, R)$ be a given H-graph grammar, and let $F_{\gamma,\delta,\beta} = (S_{\gamma,\delta,\beta}, R_{\gamma,\delta,\beta})$ be the corresponding hierarchical graph grammar, as defined at the end of Construction 7.5. Let $\Sigma = \Sigma_\gamma = \Sigma_\delta = \Sigma_\beta = (\mathbf{B} \cup \mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, $T = T_\gamma = T_\delta = T_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, and $\Delta = \Delta_\gamma = \Delta_\delta = \Delta_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, be the alphabets defined as in Construction 7.5.

We define the language generated by $F_{\gamma,\delta,\beta}$ as the set of hierarchical graphs

$$\begin{aligned}\mathcal{L}(F_{\gamma,\delta,\beta}) \;:=\; &\{(G, D, B) \mid S_{\gamma,\delta,\beta} \Rrightarrow^*_{R_{\gamma,\delta,\beta}} (G, D, B) \\ &\text{“}G,\ D,\ B \text{ are labelled over } T,\ \Delta\text{”}\}\end{aligned}$$

We have already pointed out (after Construction 7.5) that, given an H-graph grammar rule π, the rules in r_π are coordinated. Since the three component rules of such a rule $(\gamma, \delta, \beta) \in r_\pi$ are applied to corresponding nodes and packages in the three components of (G, D, B), it is easy to verify that the notion of coordinated derivation presented here is a particular case of the general notion presented in Subsection 4.3.4 and Subsection 5.2.2.

We are now ready to introduce the main result of this section, which states that our translated grammars can mimic the behaviour of the original H-graph grammars, up to the loss of information about the location of edges in the hierarchy.

Proposition 7.8 *Let H and H' be two H-graphs, π be an H-graph grammar rule, and let $HG \sim \mathbf{HG}(H)$ be a hierarchical graph. Then $H \Rightarrow_\pi H'$ iff there exists a rule $r = (\gamma, \delta, \beta) \in r_\pi$ and a hierarchical graph $HG' \sim \mathbf{HG}(H')$ such that $HG \Rrightarrow_r HG'$.*

The proof of Proposition 7.8 can be found in Appendix B. From Proposition 7.8, we obtain the following corollary.

Corollary 7.9 *Let $F = (\mathbf{B}, \mathbf{A}, S, R)$ be an H-graph grammar, and $F_{\gamma,\delta,\beta}$ be the corresponding hierarchical graph grammar defined as in Construction 7.5. If H and H' are two H-graphs, and $HG \sim \mathbf{HG}(H)$ is a given hierarchical graph, then there exists a derivation*

$$H = H_0 \Rightarrow_{\pi_1} H_1 \ldots H_{k-1} \Rightarrow_{\pi_k} H_k = H'$$

with $\pi_i \in R$ (for $i \in \{1, \ldots, k\}$) iff there exists a derivation

$$HG = HG_0 \Rrightarrow_{r_1} HG_1 \ldots HG_{k-1} \Rrightarrow_{r_k} HG_k = HG'$$

where, for all $i = 0, \ldots, k$, $HG_i \sim \mathbf{HG}(H_i)$ and, for all $i = 1, \ldots, k$, $r_i \in r_{\pi_i}$.

Proof Since, by Proposition 7.8, every step $H \Rightarrow_\pi H'$, with $\pi \in R$, exists if and only if a step $\mathbf{HG}(H) \sim HG \Rrightarrow_r HG' \sim \mathbf{HG}(H')$ with $r \in r_\pi$ exists, the thesis follows immediately.

□

We also have the following.

Corollary 7.10 *Given an H-graph grammar F and the corresponding hierarchical graph grammar $F_{\gamma,\delta,\beta}$, we have $\mathbf{ut}(\mathcal{L}(F_{\gamma,\delta,\beta})) = \{\mathbf{HG}(H) \mid H \in \mathcal{L}(F)\}$.*

Proof Let $F = (\mathbf{B}, \mathbf{A}, S, R)$ be a given H-graph grammar, and let $F_{\gamma,\delta,\beta} = (S_{\gamma,\delta,\beta}, R_{\gamma,\delta,\beta})$, $\Sigma = \Sigma_\gamma = \Sigma_\delta = \Sigma_\beta = (\mathbf{B} \cup \mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, $T = T_\gamma = T_\delta = T_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, and $\Delta = \Delta_\gamma = \Delta_\delta = \Delta_\beta = (\mathbf{A} \cup \{\bot\}) \times \mathbf{tags}$, be defined as in Construction 7.5.

If $H \in \mathcal{L}(F)$, then $S \Rightarrow_R^* H$ and H is labelled over $\mathbf{A}$ only. Since $S_{\gamma,\delta,\beta} = (S_\gamma, S_\delta, S_\beta) = (G(S), D(S), B(S)) = \mathbf{HG}(S)$, we have that, by Corollary 7.9, there exists $HG \sim \mathbf{HG}(H)$ such that $S_{\gamma,\delta,\beta} \Rrightarrow_{R_{\gamma,\delta,\beta}}^* HG$. Since HG is labelled over T and Δ, it belongs to $\mathcal{L}(F_{\gamma,\delta,\beta})$, and therefore $\mathbf{HG}(H) = \mathbf{ut}(HG) \in \mathbf{ut}(\mathcal{L}(F_{\gamma,\delta,\beta}))$. This proves that $\{\mathbf{HG}(H) \mid H \in \mathcal{L}(F)\} \subseteq \mathbf{ut}(\mathcal{L}(F_{\gamma,\delta,\beta}))$.

In the other direction, if $HG \in \mathcal{L}(F_{\gamma,\delta,\beta})$, then $S_{\gamma,\delta,\beta} \Rrightarrow_{R_{\gamma,\delta,\beta}}^* HG$ and HG is labelled over T and Δ. By Corollary 7.9, we have that there exists an H-graph H, such that $S \Rightarrow_R^* H$ and $\mathbf{HG}(H) \sim HG$ ($\mathbf{HG}(H) = \mathbf{ut}(HG)$). Since H is labelled over $\mathbf{A}$, it belongs to $\mathcal{L}(F)$, and therefore $\mathbf{ut}(HG) = \mathbf{HG}(H) \in \{\mathbf{HG}(H) \mid H \in \mathcal{L}(F)\}$. This proves that $\mathbf{ut}(\mathcal{L}(F_{\gamma,\delta,\beta})) \subseteq \{\mathbf{HG}(H) \mid H \in \mathcal{L}(F)\}$.

□

7.1.6 Summary

In this section, we have analyzed a particular approach to hierarchical graph transformation, called H-graphs and used in several papers (e.g. in [Pra79], [Pra83]) by Pratt for modeling data structures inside running software systems. Pratt introduces H-graph grammars as a means to generate the H-graphs representing the initial states of such systems in a rule-based way. This data model is the basis for an approach to the definition of programming language semantics.

The comparison with our framework shows that H-graphs are a coupled approach to hierarchical graphs, where complex nodes are used to represent the hierarchy information. The basic graph model is that of directed, node- and edge-labelled graphs. The hierarchy of an H-graph has a free structure, allowing cycles and not requiring the existence of a root component. Edges can be local to components of the hierarchy. H-graphs can be faithfully translated to hierarchical graphs in our sense, by instantiating our framework using triples of directed, node- and edge-labelled graphs.

An H-graph grammar allows to generate a language of H-graphs starting from and initial H-graph. The rewrite mechanism is node replacement, with a primitive embedding mechanism based on a pair of input and output nodes in the right-hand sides of rules. Since the hierarchy is stored in the nodes of an H-graph, node replacement also defines changes to the hierarchy. The embedding of the new hierarchy consists in inserting some selected *root* nodes from the right-hand side in the place where the rewritten node was.

This rewrite mechanism can be simulated by instantiating our framework using NLC rewriting for all three components of a hierarchical graph. It turns out that derivation steps on H-graphs correspond to derivation steps in the translated grammars, as expressed by Proposition 7.8. This translation shows that H-graph grammars provide a fairly simple rewrite mechanism for the underlying graph and for the hierarchy. On the other hand, the transformation of the coupling information can be simulated at the cost of loosing some information about the original H-graphs: they "forget"—by using plain instead of full hierarchical graphs—the placement of edges in the hierarchy. (See the discussion in Subsection 7.1.4.)

Thus, the translation of H-graph grammars illustrates possible problems that one may encounter when using full hierarchical graphs: in certain cases it is difficult to coordinate the transformation of edges in the underlying graph with the transformation of the corresponding atoms in the coupling graph. Future work on our framework should aim at a better understanding of these limitations, and consider possible solutions. These may involve defining a new mechanism for the transformation of coupling graphs, or finding an alternative representation hierarchical for the coupling graph.

7.2 Other Approaches to Hierarchical Graphs

In this section, we consider other approaches to hierarchical graphs and their transformation within the graph transformation community. We will provide a short summary of each approach, and then sketch how it can be modeled within our framework.

7.2.1 The AGG approach

AGG is a tool developed at the Technical University of Berlin (see e.g. [LB93], [AEH+99], [ERT99], [Rud97]) for creating and animating graph transformation systems in the DPO or SPO approaches. In AGG, a graph H is internally represented as a collection O_H of *objects*—which can model nodes or edges—provided with three partial mappings, $s_H : O \to O$ and $t_H : O \to O$, which link each edge-like object to its *source* and *target*, and $a_H : O \to O$, the *abstraction* mapping, which defines a hierarchical structuring that groups subgraphs into nodes, and bundles of parallel edges into edges. The hierarchy is a tree, since there exists a unique *top object*, and the hierarchy is cycle-free. Other elements of these graphs, which can be overlooked as far as hierarchical structuring is concerned, are object labels and attributes. This representation can be interpreted as a graph where the objects are the nodes, and the three partial functions define three sets of directed edges. Transformations of these flat graphs are defined by means of SPO rules. Therefore, in this approach hierarchical graphs have a flat (coupled) representation, where the hierarchy is modeled by special nodes and edges. The transformation uses a gluing approach.

A translation of these graphs to our approach can be obtained by translating an AGG graph H into a (plain) hierarchical graph $\mathbf{HG}(H) = (G(H), D(H), B(H))$, where the three components are directed graphs. In $\mathbf{HG}(H)$, each object o that is in the range of the abstraction mapping a_H in H is split to a node n_o of $G(H)$ and a package p_o of $D(H)$ anchored to n_o in $B(H)$. Then, if o_1 and o_2 are objects and $o_2 = a_H(o_1)$, we add a containment edge from n_{o_1} to p_{o_2} in $B(H)$. If, in addition, o_1 is in the range of a_H too (i.e. it is a component of the hierarchy), we add a hierarchy edge in $D(H)$ from p_{o_2} to p_{o_1}. The source and target mappings s_H and t_H are modeled as directed edges in the underlying graph.

A single-pushout production $\pi = m : L \to R$ can also be split to a coordinated triple of single-pushout rules $r_\pi = (\gamma, \beta, \delta, \simeq_\gamma^L, \simeq_\gamma^R, \simeq_\delta^L, \simeq_\delta^R)$, where $\gamma = m_\gamma : G(L) \to G(R)$, $\delta = m_\delta : D(L) \to D(R)$, $\beta = m_\beta : B(L) \to B(R)$, such that

- for all $o \in O_L$, $o' \in O_R$, $m_\gamma(n_o) = n_{o'}$ if and only if $m(o) = o'$,
- for all $o \in O_L$, $o' \in O_R$, $m_\delta(p_o) = p_{o'}$ if and only if $m(o) = o'$,
- the coordination relations $\simeq_\gamma^L$, $\simeq_\gamma^R$, $\simeq_\delta^L$, $\simeq_\delta^R$ are defined in an obvious way since, for all $o \in O_L$, $o' \in O_R$, $n_o \in N_{G(L)} \cap A_{B(L)}$, $n_{o'} \in N_{G(R)} \cap A_{B(R)}$, $p_o \in N_{D(L)} \cap A_{B(L)}$, $p_{o'} \in N_{D(R)} \cap A_{B(R)}$.

It seems reasonable to conjecture that $H \Rightarrow_\pi H'$ if and only if $\mathbf{HG}(H) \Rrightarrow_{r_\pi} \mathbf{HG}(H')$. Investigating this conjecture can be the topic of future work.

7.2.2 Distributed systems

Distributed systems have been modeled using graph transformation in [Tae96]. A distributed system is seen as a network whose nodes contain internal graphs describing local states, while network edges model connections between different nodes. The internal graphs of adjacent network nodes communicate through shared subgraphs. (See also Section 2.2, Section 4.2, and Subsection 4.3.2.)

We can see this approach as a decoupled hierarchical graph, since the local graphs form the underlying graph, and the network graph corresponds to the hierarchy. Notice, however, that there is no restriction on the structure of the hierarchy (compare H-graphs in Section 7.1). Also, the relation between different network nodes does not impose a layered structuring, thus, strictly speaking, distributed graphs cannot be considered an approach to hierarchical graphs.

In order to translate such a structure to a hierarchical graph, we imagine that the network nodes correspond to the packages, and we take the gluing of all local graphs along their interfaces as the underlying graph. The coupling graph can be defined in an obvious way by assigning all nodes and edges of the underlying graph to the appropriate component of the network graph.

Transformations of distributed systems are defined using the DPO approach to graph transformation, and therefore this is a gluing approach. Transformations are defined by combining network rules, which transform the topological structure of a distributed system, with local rules, which transform the local graphs. Local rules are applied only to the local graphs of network components that are preserved by the network rules.

A translation of such rules to our framework can be sketched as follows.

- Network rules are mapped to DPO hierarchy rules as they are.
- Local rules are glued together to form DPO rules on the underlying graph.
- Every coupling graph rule must specify
  1. the addition/removal of new/deleted items to/from preserved packages, i.e. to/from preserved network components,
  2. the deletion of items local to deleted packages,
  3. the addition of the content of new network components to the corresponding new packages.

  Since transformations of the underlying and hierarchy graphs are specified using the DPO approach, at every step a predefined number of nodes/edges must be

added to/removed from the coupling graph. Thus these transformations can be specified using DPO rules as well, provided that they are coordinated with the corresponding underlying and hierarchy graph rules.

Another possible interpretation sees distributed systems as hierarchies with exactly two layers. The corresponding translation to our model only differs in that we have to add a new *root* package that contains all the other packages, and corresponding hierarchy edges that must be handled during transformation. It can be easily verified that the resulting (trivial) hierarchy transformations can be defined using DPO rules.

7.2.3 Encapsulated hierarchical graphs

Encapsulated hierarchical graphs were introduced in [ES95], and besides addressing the general need for a hierarchical graph data model, they focused on issues like *boundary crossing edges*, *encapsulation*, *classification* (*typing*), and *refinement* (*inheritance*). This model is therefore strongly influenced by software engineering and object-oriented modeling concepts.

An *encapsulated graph* is a node- and edge-labelled directed graph, together with import/export interfaces that allow to distinguish between graph elements that are *owned* by that graph, elements that are *exported* to or *hidden* from the external world, and elements that are *imported* from the external world. Thus, every encapsulated graph provides a "view" of the data to be modeled, where certain elements are internal to (owned by) the view, and others are imported from somewhere else. An *encapsulated hierarchical graph* is an encapsulated graph where nodes can contain an internal (encapsulated hierarchical) graph. The nesting between these *complex nodes* must be a tree-like structure, which implies that the graphs at the leaves only contain atomic nodes. It is clear that this approach is coupled in our sense (see Section 2.3.4).

This construction can be simulated by an encapsulated hierarchical graph in our sense (see Section 3.3). The translation consists in decoupling complex nodes into an ordinary node and a package anchored to it. Packages form a tree-like structure, as prescribed by the nesting of the nodes to which the packages are anchored. All owned and imported nodes and edges of a complex node are contained in the corresponding package. Import/export between the levels of the hierarchy satisfies slightly different constraints: in the model of [ES95], an exported node or edge is automatically an own node of the upper level, while in our approach, we still need to explicitly import such element into the upper level. Since our model is more general, we can easily restrict it by adding a further constraint.

The model of [ES95] also provides typing (hierarchical graph schemata) and inheritance. Typing can be modeled in our framework, but the possible additional constraints on package interfaces—that are considered in [ES95]—should be considered in addition. Inheritance has not been considered in our model, and it can be the topic of future research.

Finally, the model of [ES95] does not consider any operation on hierarchical graph, which is considered the main drawback of this model.

7.2.4 Multilevel graph grammars

Multilevel graph grammars (see [PP95]) are motivated by applications where a visual representation of graphs needs to be structured by *hiding* certain details (nodes and edges) of these graphs (see also Subsection 4.2.2). In this approach, the structuring of graphs in various levels is represented as a sequence of hiding productions that must be applied in order to obtain a folded graph from a flat graph. Since the representation of the underlying graph and of the additional structuring is stored in a folded graph and in a sequence of rules, we can consider multilevel graphs as a highly coupled approach. Multilevel graph transformation uses a variation of the double-pushout approach, and therefore belongs to the gluing paradigm. In this subsection, we sketch the main aspects of this approach, and provide some hints on how to simulate it within our framework.

A multilevel graph is represented by a triple $M = (G, RP, HP)$, where G is a node- and edge-labelled graph, RP is an ordered set of *restoring productions*, and HP is an ordered set of *hiding productions*.[4] The intuition is that hiding production can be applied to G to increase its level of folding, while restoring productions are applied to unfold a folded graph. When a hiding production p is applied to G giving a graph G', its inverse p^{-1}—a production that restores G from G'—is added to the sequence of restoring productions. A multilevel graph M is in *normal form* if it has no hiding productions.

A multilevel graph $M = (G, RP, \emptyset)$ can be translated to a hierarchical graph $\mathbf{HG}(M) = (G(H), D(H), B(H), n)$. The index $n = k$, whose usage will be clear shortly, is equal to the number of productions in the sequence RP. The underlying graph $G(H)$ is the graph derived from G by applying the restoring productions $RP = \{p_1, \ldots, p_k\}$ in the specified order, i.e. the underlying graph is the original, unfolded graph. The hierarchy graph contains a root package ρ, and a package p_i ($i \in \{1, \ldots, k\}$) for each production (we use the same names for packages and production). The nesting of packages defines a linear[5] hierarchy graph where $\rho \succ p_k \succ p_{k-1} \succ \ldots \succ p_1$ (newly applied productions "hide" the older ones and push them towards the bottom of the hierarchy). In the coupling graph, no anchoring of packages is defined (we are modeling a decoupled approach), whereas a node or edge x belongs to a package p_i ($i \in \{1, \ldots, k\}$) if x belongs to the production's right-hand side, since this means that it is *restored* by that productions. The ρ package will contain the graph G. In the more general case of a multilevel graph

[4] We assume that G and the graphs contained in the rules are defined over a common set of nodes and edges, so that the occurrence morphisms to be used for these rules are implicitly and uniquely determined.

[5] In [PP95], we found no indications about more refined ways to order the levels of a multilevel graph.

$M = (G, \{p_1, \ldots, p_{k'}\}, \{p_{k'+1}^{-1}, \ldots, p_k^{-1}\})$, we let $\mathbf{HG}(M) := (G(M), D(M), B(M), n)$, where the three graphs are built as above, and $n := k'$. Thus, n indicates the level of folding by pointing at the package corresponding to last restoring production. If $RP = \emptyset$, then $n = 0$, thus pointing to the root package. (Notice that n could be coded in the coupling graph by letting the selected package be qualified by a special dummy node.)

Multilevel graph transformation involve two kinds of basic operations. The first consists in applying hiding (resp. restoring productions) to a multilevel graph in order to increase (resp. decrease) its level of hiding. The applied production p is removed from the sequence of hiding (resp. restoring) productions, and its inverse p^{-1} is added to the sequence of restoring (resp. hiding) productions. These operations can be easily modeled by decreasing or decreasing the value of the index n.

The second kind of operation consists in applying a production of type $q = (q_1, q_2)$, where q_1 is an ordinary double-pushout production, while q_2 is a hiding production and is a subproduction of q_1. In this way, q specifies both an ordinary transformation of a graph, and adds a further level of hiding to it through q_2. Given a multilevel graph $M = (G, \{p_1, \ldots, p_{k'}\}, \{p_{k'+1}, \ldots, p_k\})$, we apply q to it and derive

$$M' = (G', \{p_1, \ldots, p_{k'}, q_2^{-1}\}, \{p'_{k'+1}, \ldots, p'_k\})$$

where $G \Rightarrow_{q_1} G'$, $p'_{k'+1}, \ldots, p'_k$ are productions derived from $p_{k'+1}$, ... , p_k, in such such a way as to be compatible with the transformed graph G' (see again [PP95] for more details).

Such a transformation step can be simulated by a triple rule that

1. Transforms the underlying graph using q_1: if q_1 can be applied to the folded graph G in $M = (G, RP, HP)$, it can be applied also to its unfolded counterpart, which is the underlying graph in $\mathbf{HG}(M)$.

2. Inserts a new package q_2^{-1} between $p_{k'}$ and $p_{k'+1}$.

3. Assigns the occurrence of the left-hand side of q_2 in G to the content of the corresponding package.

4. Puts in ρ the nodes added by q_1 but not hidden by q_2. Removes from ρ the preserved nodes that are hidden by q_2. (See item 3 above.)

5. Although the packages $p_{k'+1}, \ldots, p_k$ are preserved (or rather, renamed to $p'_{k'+1}$, ... , p'_k), their content is rearranged by removing from these packages the nodes and edges that are deleted by q_1, and therefore cannot be hidden in these packages anymore.

6. The index n is incremented by one, in order to point to the new restoring production q_2^{-1}.

Notice that the transformation on the underlying graph can be specified by means of double-pushout rules, since q_1 can be applied. The transformation on the hierarchy graph (see point 2 above) can also be specified by means of a double-pushout rule. Finally, since the transformation on the coupling graph involves mimicking additions/deletions to the underlying graph and to the hierarchy graphs, and changing the containment relation as specified in 3, 4, 5 above, we think that this transformation can be simulated using a single-pushout rule. In fact, while 3, 4 require the addition/deletion of a known number of nodes and edges, point 5 requires the removal of nodes and edges from all the last $k - k' + 1$ packages, where k and k' depend on the graph to which the transformation is applied. This result can however be obtained by using the single-pushout approach, where dangling edges are automatically eliminated. As hinted above, the index n can be simulated by qualifying the indexed package by a dummy node. In this way, modifying the index corresponds to changing the qualified package, an operation that can also be easily specified using the single-pushout approach on the coupling graph.

Multilevel graph grammars are a very peculiar model of hierarchical graph transformation. Since the hierarchy is defined implicitly by means of operations, it is a complex task to describe the implicit hierarchical structure and the operations that are performed on it. Due to these limitations we believe that, although this model is useful in applications like graph visualisation (see Section 2.2.1), it can hardly serve as a general model of hierarchical graphs. We do not look at further details of these transformations: a thorough comparison of our framework with multilevel graph grammars can be the topic of future research.

7.2.5 Graph package systems

The model introduced in [BEMW00] uses a decoupled representation of hierarchical graphs based on packages. In a graph package system, the underlying graph is a node- and edge-labelled, directed graph, and the hierarchy graph is a tree. The coupling graph is restricted by means of import/export interfaces, i.e. the model provides encapsulation (see Section 3.3).

Graph package systems also provide hierarchical graph types. It is easy to see that these are a particular case of our notion of typing: the underlying graph has a type, the hierarchy graph has a type, and the association relation has restrictions that can be modeled by typing the coupling graph as well.

Graph package systems also provide a primitive mechanism for defining hierarchical graph transformation. Given a graph transformation approach $\mathcal{A} = (\mathcal{G}, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, a class of hierarchical graphs $\mathcal{H}$, and a mapping $f : \mathcal{G} \to \mathcal{H}$, the framework allows to define a hierarchical graph transformation approach $\mathcal{A}_{\mathcal{H}} = (\mathcal{H}, \mathcal{R}, \Rightarrow_{\mathcal{H}}, \mathcal{C}, \mathcal{E})$, where the semantics of rules, control conditions and expressions is transfered from $\mathcal{A}$ to $\mathcal{A}_{\mathcal{H}}$ by means of f. In [BEMW00], a particular mechanism for defining f is described.

An interesting idea is that, when using encapsulated hierarchical graphs, we should define local transformations of hierarchical graphs by requiring that the occurrence of the left-hand side of a rule be matched by a subgraph of the underlying graph that is local to a package. This constraint is easy to ensure in our framework, since in a tracking graph transformation approach an occurrence of the left-hand side is represented in an abstract way using skeleton morphisms, and we can specify that the elements of an occurrence are all contained in the same package. (See Subsection 5.1.2.)

Thus, although graph package systems provide a model that is less powerful than that presented in this thesis, it suggests possible developments in the direction of encapsulated hierarchical graph transformation.

7.2.6 Hierarchical hypergraph transformation

Another DPO-based approach to hierarchical graph transformation can be found in [DHP02]. In this paper, hierarchical hypergraphs are defined recursively, by using special hyperedges called *frames* which in turn contain a hierarchical graph. This is therefore a coupled approach.

A morphism $f : G \to H$ between two hierarchical hypergraphs G and H is defined by mapping the top level hypergraph of G to the top level hypergraph of H and by recursively defining a morphism f_g, for each frame $g \in E_G$, mapping the content of frame g to the content of frame $f(g)$. Using these morphisms, double-pushout rules and derivations on hierarchical hypergraphs are defined.

In [BH01], a comparison between hierarchical hypergraph transformation and our framework is provided. We can translate a hierarchical hypergraph H to a triple $\mathbf{HG}(H) = (G(H), D(H), B(H))$ as follows. $G(H)$ is obtained by taking the disjoint union of the top level hypergraph and of all the hypergraphs contained inside frames, recursively. The graph $D(H)$ has a root package and one package for each frame of H. The nesting of packages corresponds to the nesting of frames, the packages corresponding to top-level frames being nested in the root package. $B(H)$ anchors each packages different from the root package to its frame, and each graph element of $G(H)$ to its package. Notice that $\mathbf{HG}(H)$ is strict, in the sense of Definition 3.13.

If we code $D(H)$ and $B(H)$ as node- and edge-labelled directed graphs in the usual way—see e.g. Chapter 6, Section 6.3 and Section 6.4—we can define morphisms and double-pushout transformation on the three components of $\mathbf{HG}(H)$. It turns out that a hierarchical morphism $f : H \to H'$ between two hierarchical hypergraphs can be translated into a triple of morphisms $f_G : G(H) \to G(H')$, $f_D : D(H) \to D(H')$, $f_B : B(H) \to B(H')$, and DPO derivations on hierarchical hypergraphs can be simulated by coordinated DPO derivations of translated hierarchical graphs. Thus, if π is a hierarchical hypergraph rule, and r_π is the corresponding generic hierarchical rule, we get a similar result for hierarchical hypergraph as for H-graphs (see Section 7.1, Proposition 7.8), as stated in Proposition 7.11 (see also [BH01]).

Proposition 7.11 *There exists a hierarchical hypergraph transformation step* $H \Rightarrow_\pi H'$ *if and only if there exists a coordinated generic hierarchical graph transformation step* $\mathbf{HG}(H) \Rrightarrow_{r_\pi} \mathbf{HG}(H')$.

Since here we consider a gluing approach, this result is the counterpart of Proposition 7.8, where we deal with a connecting approach (H-graph grammars). This shows that our abstract approach allows to model operations from both the paradigms discussed in Section 4.2.

7.2.7 Hierarchical graphs for modeling software system evolution

In [EH00], a model of hierarchical graph transformation is introduced with the purpose of modeling software systems architectures and their evolution. The hierarchical graph model is coupled, since the hierarchy is built around "aggregate nodes." Hierarchical graphs are typed. Hierarchical graph transformation is based on the double-pushout approach. In the rest of this subsection, we will call these hierarchical graphs *aggregated hierarchical graphs*.

The translation of an aggregated hierarchical graph H to a (plain) hierarchical graph $\mathbf{HG}(H) = (G(H), D(H), B(H))$ is straightforward (compare Subsection 7.2.1). $G(H)$ will contain all the nodes and edges of H, except the aggregation edges. For each "aggregate node" $n \in N_H$, we create a package p_n in $D(H)$. If two nodes m and n are aggregated and m contains n in H, then we let p_n be a subpackage of p_m in $D(H)$. In $B(H)$, every package p_n is anchored to the node n, and every node n is contained in p_m if and only if n is a component of the aggregate node m in H. Typing can be easily defined on the components of $\mathbf{HG}(H)$ in such a way as to be compatible with the typing defined on H.

Without considering the details, each DPO rule $\pi = (L, i, K, j, R)$ on aggregated graphs can be split into a coordinated rule $r_\pi = (\gamma, \delta, \beta, \simeq_L^\gamma, \simeq_R^\gamma, \simeq_L^\delta, \simeq_R^\delta)$. In this rule, the DPO rules γ, δ, β are obtained by splitting L, K, and R into $\mathbf{HG}(L)$, $\mathbf{HG}(K)$, and $\mathbf{HG}(R)$, respectively, and the morphisms i and j in an appropriate way. For example, $i_\gamma : G(K) \to G(L)$ is defined as $i_\gamma := i|_{G(K)}$. The coordination relations $\simeq_L^\gamma$, $\simeq_R^\gamma$, $\simeq_L^\delta$, $\simeq_R^\delta$ are defined by considering that the pairs of graphs $G(L)/B(L)$, $G(R)/B(R)$, $D(L)/B(L)$, $D(R)/B(R)$ share common nodes and edges. These split rules can then be applied to generic hierarchical graphs in order to simulate derivations on aggregated hierarchical graphs. We believe it is possible to prove the conjecture that $H \Rightarrow_\pi H'$ if and only if $\mathbf{HG}(H) \Rrightarrow_{r_\pi} \mathbf{HG}(H')$. We leave this as a topic for future research.

The fact that we can find a reasonable simulation of the aggregated hierarchical graph approach shows once more the generality of our framework. Our approach has also the advantage that it provides a criterion to check whether DPO rules transform

hierarchies consistently (see Proposition 6.12). Such a criterion can easily be applied to aggregated hierarchical graph transformation.

7.3 Summary

In this chapter, we have investigated the issue of translating existing approaches to hierarchical graphs and hierarchical graph transformation to our framework. The considered approaches all stem from the graph grammar field. Our task consists in finding a suitable representation for hierarchical graphs in the original approach, and in trying to define hierarchical graph transformation rules in our framework that simulate rules in the original hierarchical graph transformation approach.

As far as the static aspects are concerned, our model is general enough to represent all kinds of considered hierarchical graphs. In particular, the decoupled hierarchy graph can represent any kind of hierarchy, from the most restrictive tree-like hierarchy to the most liberal "any-graph" hierarchy. Decoupling provides a flexible representation that does not rely on some *aggregating item* in the graph. The price for this flexibility is that our approach requires in general a heavier representation. (Compare for example Section 7.1).

For the dynamic aspects, translating one approach to another can be a complex task but it provides more insight in the kind of manipulations that can be performed within a specific approach, and helps to compare different approaches with each other. Decoupled approaches tend to be easier to translate, since they often transform the underlying graph and the hierarchy as separate structures (see e.g. distributed systems, in Subsection 7.2.2). On the other hand, coupled approaches require that we understand the manipulations on the three components of a hierarchical graph separately, while these manipulations are specified together in the original rules.

In general, we conclude that the "essential aspects" and concepts that we have identified and developed from Chapter 2 to Chapter 5 seem to be widely applicable to existing approaches to hierarchical graphs. Among open issues, an important one concerns the coordination of transformations on the underlying graph with transformations of the coupling graph. As we have seen when translating H-graph grammars, (Section 7.1) this task can be a difficult one and suggests that new transformation approaches or new representations for coupling graphs should be investigated. Other ideas for future research on *encapsulated* hierarchical graph transformation are suggested by the comparison with the approaches of [ES95] and [BEMW00].

Chapter 8

Modeling Hypermedia with Hierarchical Graphs

In the previous chapters, we have developed a generic model of hierarchical graphs (from Chapter 2 to Chapter 5) and then used it to simulate other existing approaches in Chapter 7. In Chapter 6, we have illustrated the instantiation of our model using the DPO approach to graph transformation, and considered technical issues like the identification of consistent hierarchical graph transformation rules. Throughout this thesis, we have used hypermedia as a running example for illustrating issues and concepts related to hierarchical graphs. In this chapter, we investigate the application of hierarchical graphs—in particular of the model presented in Chapter 6—for modeling hypermedia structures and their transformations.

The purpose of this investigation is twofold. First, hypermedia illustrates very well the problems that can occur when using large, graph-like structures, and which can motivate the introduction of hierarchical structuring. While up to now we have used a toy example, in this chapter we consider a concrete case study. Second, we want to show that the concepts of hierarchical graphs and hierarchical graph transformation developed so far have meaningful applications in a real hypermedia environment.

Our case study considers the Hyper-G hypermedia system (see [Mau96]), which was designed and developed at the Graz University of Technology (Austria) during the first half of the nineties. It then became a commercial product and is presently known under the name HyperWave^{tm}[1]. HyperWave extends the data model of the world-wide web by providing, among other features, a hierarchical hypermedia model, the separation of hyperlinks from documents, and document search capabilities. This advanced model is an answer to the "lost in hyperspace syndrome"—the confusion one experiences when browsing a large hypertext—and similar problems of users and maintainers of large hypertexts.

The concept of a hierarchical hypertext is not new (see e.g. the Dexter model in

[1] Throughout this chapter, we will use the names Hyper-G and HyperWave interchangeably.

[HS94]). The underlying idea is that hyperlinks in a flat hypertext are used in two different roles, which should be clearly separated: *referential hyperlinks* are used to define cross references between related documents/topics, while *organizational hyperlinks* are used to define the structure of a hypertext as well as possible navigation paths through this structure (see also [BRS92]). The parallel with object-orientation, where a particular kind of association—aggregation—is chosen to represent a part-of relationship, is quite evident. (See also the discussion in Subsection 2.3.3.)

The second aim of this chapter is to investigate the applicability of the concepts developed in this thesis in the context of hypermedia. From a static point of view, our framework allows to classify the HyperWave data model as a coupled hierarchical graph model. The other interesting area is that of operations: A hypermedia collection is a complex graph-like structure that needs to be updated very often, while being kept consistent. Although some models of hypermedia systems have been proposed (see e.g. [SDW95]), up to our knowledge none of these models deals with the transformations of hypermedia networks inside such systems.

Applying hierarchical graph transformation to HyperWave has several advantages:

- It is a first step towards filling the gap between the high-level static data model and the low-level client/server protocol, by means of which HyperWave provides hyperweb operations.
- It provides an intuitive and concise way of specifying even complex hyperweb manipulations.
- Using appropriate transformation approaches (e.g. the one developed in Chapter 6), it allows to check statically that the specified operations preserve consistency inside a hyperweb. This can help to write correct applications and to reduce the amount of consistency checks that must be performed at runtime.

This chapter has the following structure. In Section 8.1, we provide more detailed motivations for applying hierarchical graph transformation in the hypermedia field. In Section 8.2, we recall the main concepts of hypermedia and give an introduction to HyperWave. In Section 8.3, we model hypermedia structures in HyperWave using hierarchical graphs. In Section 8.4, we investigate how hierarchical graph transformation can be used to specify hypermedia operations. In Section 8.5, we provide a summary of the chapter.

8.1 Motivation and Conceptual Framework

The distinguishing feature of hypermedia applications is that they provide information by means of a navigable structure—called a *hyperweb*—which consists of a collection of documents with hyperlinks between them. Besides this basic node/link structure, it

is a common practice to design hypermedia applications using higher-level navigation primitives, like *indices*, *guided tours*, and so on. (See e.g. [ISB95], [SRB96], [DL98].) This makes it easier for developers to manage the navigation structures of their applications, and helps users to find meaningful navigation paths through these structures. (See Section 8.2 for more details.)

The world-wide web, probably the best known existing hypermedia system, supports the implementation of applications by providing basic functionalities like document storage and hyperlinking, while the realization of higher-level navigation structures is left to the specific application running on top of them, or to the authors of WWW pages, if these are created manually. Other systems—among them HyperWave—provide higher-level navigation structures as primitive constructs. The consistency of these navigation structures is maintained automatically by the system itself. In spite of the higher costs in terms of computational resources, these systems offer a better support for developing large hypermedia applications. (See again Section 8.2.)

With the increasing complexity of hypermedia systems, it is more and more important that their behaviour be specified in a precise way. Formal methods have been proposed for the specification of different aspects of hypermedia systems. For example, in [SDW95] Petri nets are used to model synchronization constraints within hypermedia documents (see also the Amsterdam model [HBvR94]). However, formal models of hypermedia network transformations are still missing (see also the discussion in [BtH00]).

Such transformations are an important aspect of hypermedia systems, since hypermedia networks are highly dynamic structures. As already said, HyperWave (and other modern systems) provides a very advanced data model, with high-level primitives for structuring the content of a hyperweb. Compared to this, the model of operations is very poor, and does not allow one to reason in terms of the underlying navigation paths or hierarchical structures. Instead, hyperweb transformations are currently specified through elementary operations, that can be issued by a client written in C or using some scripting language. It is difficult to trace the evolution of a hyperweb during the execution of such a program, and to ensure that the hyperweb is transformed consistently. In fact, consistency checks are performed by the server after every transaction.

In order to tackle these problems, we propose a specification technique based on hierarchical graph transformation. The first step is to look at static aspects of hypermedia. While the basic data model of the WWW can be naturally represented as a flat graph, higher-level navigation structures are more appropriately modeled as hierarchical graphs—and in particular using our hierarchical graph model.

Hierarchical graph transformation is then applied to this static model to specify hypermedia manipulations. It is not our intention to define a complete specification language within this chapter, but to illustrate the use of hierarchical graph transformation and its benefits. The possibility to define complex operations in a concise and intuitive way is a well-known advantage of graph transformation, which is inherited in the hierarchical case. The formal and declarative nature of graph transformation rules

allows one to prove properties about them. In this chapter we will reuse the results of Chapter 6 and show that, by using appropriate rules, we can ensure that the internal consistency constraints of a HyperWave repository are preserved.

By applying our data model for modeling hypermedia structures and transformations in a real system, in this chapter we bring together and illustrate several ideas that have been considered in this thesis, namely:

1. The idea that graphs are a natural model for many different kinds of information structures and, in particular, of hypermedia structures.

2. The idea that hierarchical structuring emerges naturally in many areas, when the size of the modeled data or the need for abstraction suggest that some grouping mechanism should be introduced.

3. The idea that rules are a suitable mechanism for specifying complex transformations of (hierarchical) graph-like structures like hypermedia structures.

From a practical point of view, this chapter shows that hierarchical graph transformation is an appropriate formalism for specifying hypermedia structures and operations.

8.2 Hypermedia in HyperWave

In this section, we consider the data model of HyperWave, and we discuss the problems that motivated its introduction. This discussion considers the issues presented in Chapter 1 in more detail and in a more concrete setting.

8.2.1 Motivations for hypermedia structuring

The term *hypermedia* (see e.g. [MSHR98]) combines the terms *hyper-text* and *multimedia*. A hypertext is a collection of *documents* (*nodes*) with references (*hyperlinks* or simply *links*) between them that is navigable by means of appropriate piece of software called a *browser*. A multimedia document can contain, besides text, images, sound, movies, and so on. We call such collections *hyperwebs* or *hypermedia networks*. A *hypermedia application* is a piece of software that provides information in the form of a hyperweb. A web site is an example of a hypermedia application. We will occasionally use the term hypertext instead of hyperweb, when the content of the documents to be navigated is irrelevant. In Figure 8.1, we depict an example hyperweb containing plain text documents, pdf documents, and mpeg movies.

A *hypermedia system* (see e.g. [ISB95, page 35]) is a piece of software that allows to create, store and manage hypermedia information, thus supporting the creation of hypermedia applications. The World-Wide Web (together with the authoring and

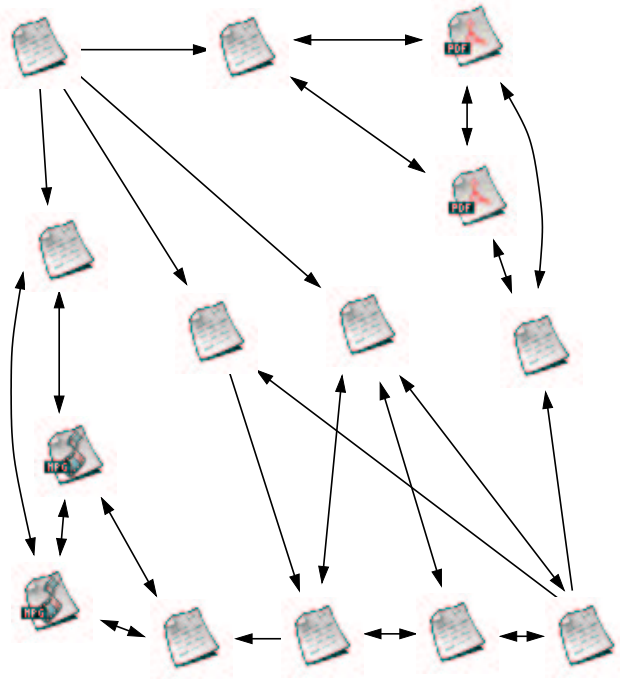


Figure 8.1: A hyperweb.

management tools available for it) can be considered as a big (distributed) hypermedia system. HyperWave is another example.

An important issue within the hypermedia field can be illustrated by looking at the WWW data model. Hyperwebs normally contain a large number of documents and links, and often become hard to use and to manage. On one hand, a user can easily get "lost in hyperspace", i.e. be incapable of orientation in the hypertext he is browsing, because he does not have an overview of its structure. On the other hand, authors have problems developing hypermedia structures that are understandable, and to keep them consistent: *broken links* (hyperlinks pointing nowhere) and *orphan documents* (documents that are unreachable) are among the best known problems.

To overcome these problems, many hypermedia applications provide additional structural information: we can have hierarchies of *composite* documents (compare also the Dexter model [HS94]), *guided tours* that can be navigated forwards and backwards, *tables of contents*, and so on. On the WWW, these structures are realized by means of special hyperlinks, called *structural* (or *organizational*), while the remaining hyperlinks are called *referential*. (When they are used to model document hierarchies, structural links can be easily compared with the notion of aggregation in object-oriented modeling, see also Subsections 2.2.3, 2.3.3, and 7.2.7.) This distinction is however only part of an application's design, and is not supported by any primitive in the WWW data

model.

Among the disadvantages of organizational links we have the following:

- The same construct (hyperlinks) is used for modeling two conceptually different types of information: the organization of a collection of documents, and the references between related documents. Thus a correct use of such links relies on the experience and discipline of the authors.

- A hyperweb is overloaded with a large number of structural links that need to be kept consistent every time the structure of the web is modified, for example when inserting a new element in a hierarchy.

In practice, structural links are often considered a low-level construct and are used to implement higher-level structures (hierarchies, guided tours, and so on): many WWW applications rely on tools that manage such structural links automatically. Other models—like HyperWave—incorporate high-level structures as basic constructs of their hypermedia model.

8.2.2 The HyperWave data model

For our example, we analyze the model used by HyperWave$^{\text{tm}}$ system (see e.g. [Mau96]). This model assumes that a hypermedia *repository*—i.e. a hyperweb—provides documents (nodes) with *referential links* (edges) between them. Referential links are actually attached to documents through *anchors*. Each link originates from a *source anchor* that is attached to the source document, while the target can be a a *destination anchor*, a document, or a *container* (see below). Containers model higher-level (organizational) structures of a hyperweb. They can be of four types:

1. *Collections*, i.e. bags where documents and (recursively) other containers can be put.

2. *Sequences*, i.e. special collections with predefined sequential order between the internal elements.

3. *Multiclusters*, i.e. containers whose internal documents must be displayed at the same time. For example: a text can be displayed and a video clip of someone reading the text can be played.

4. *Alternative clusters*, i.e. containers where one of the contained documents must be displayed, possibly depending on the user browsing the hypertext. For example, a document can be stored in different languages, and the system can choose the right version from the user's options.

Documents and containers can be shared between different collections. All documents must be contained in at least one container. All containers must be contained in at least one collection, except the *root collection*, which must always exist (no empty repositories are allowed) and is not contained in any other collection. The hierarchy of collections must be cycle free, i.e. it must form a rooted dag. Orphans, i.e. documents that are in no container, are forbidden.

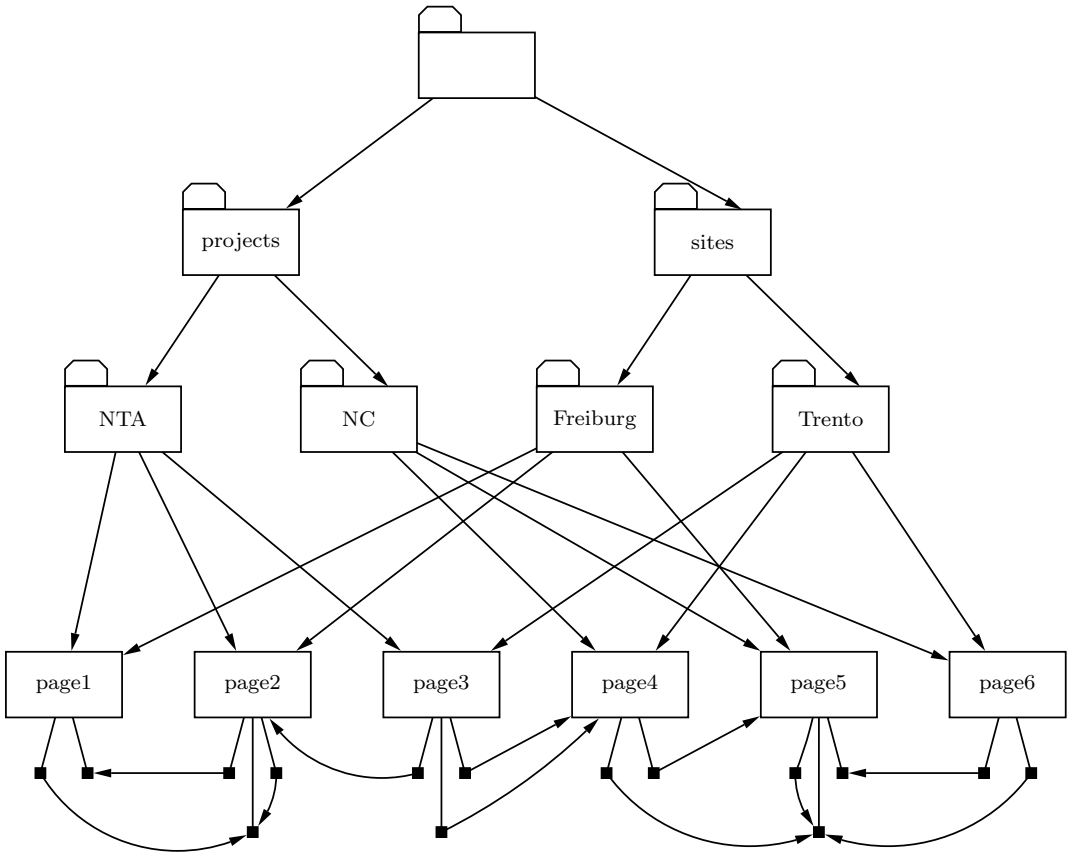


Figure 8.2: The running example as a HyperWave repository.

In Figure 8.2, we model our running example using the HyperWave primitives: projects and sites are modeled as collections (rectangles with tabs), while pages are documents (rectangles) and anchors correspond to HyperWave anchors (small black boxes). Arrows between collections indicate their nesting, while arrows from collections to documents indicate that a collection contains that document. Anchors are attached to pages using undirected lines, while hyperlinks are drawn as arrows originating from

anchors. Notice that collections and packages have slightly different tabs in order to distinguish them from each other.

A HyperWave server contains a database, where containers, documents and anchors are stored as objects, whereas anchor attachment and document/container location in the collection hierarchy are stored as relations.

8.2.3 Dynamic aspects in HyperWave

Several kinds of multimedia information, like videos, sounds, complex animations, are inherently dynamic. It often happens that several pieces of multimedia information must be displayed at the same time (they must be synchronized with each other) or react to external events like user input. These aspects concern the behaviour of a given hypermedia content and have been investigated by several authors (e.g. [HBvR94], [SDW95]). (For example, in the cited [SDW95], a model based on Petri nets is proposed.)

On the other hand, the hypermedia content of a repository—the structure of a hypermedia network—can also change: Many applications modify the content of a hypermedia repository very often, creating or deleting documents, and changing their interconnections. This is another kind of dynamics, that has not been given much attention up to now (see also [BtH00]), and that we consider w.r.t. the HyperWave case study.

Changes to a repository can be performed by a client by connecting to a HyperWave server through tcp/ip and exchange messages with it, using a protocol called HG-CSP. Typical commands include querying the server for certain objects, modifying or deleting existing objects, creating new objects, and so on. (See e.g. [Mau96, Appendix F].) Such clients provide a higher-level interface to the server. For example, with Harmony ([Mau96, Chapter 15]) one can edit a repository's content using a graphical user interface.

As an example, we consider the following operations:

1. *Insert document.* Inserts a document in the repository and puts it in some existing collection.

2. *Add hyperlink pointing to document.* A hyperlink from a document to a destination document is created. A source anchor is also created.

In order to perform operation 1 above, a client must first query the Hypermedia database to obtain the identifier of the collection in which the document must be inserted, and then issue a command to insert the new document in the database in the wanted collection. Alternatively, one can use the command-line tool `hginsdoc` to perform this operation in one step.

In order to perform operation 2 above, one has to retrieve the identifiers of the source and the target document, create a new source anchor, attach it to the source document, and let the destination (`dest`) attribute of the anchor point to the destination document. We know of no command-line tools that can perform this task in one step.

These examples show that, in spite of HyperWave's advanced data structuring, the specification of manipulations on a hypermedia repository can only be done using very low-level primitives. While these low-level primitives are suitable for the client-server protocol, a higher-level language would allow a more concise and intuitive specification of complex hyperweb manipulations. These higher-level specifications could be translated to the lower-level protocol during implementation.

In Section 8.4, we show how hierarchical graph transformation can be used to specify hypermedia manipulations at a higher-level of abstraction, and discuss the benefits of such an approach.

8.3 Hypermedia as Hierarchical Graphs

In this section, we show that the data model of HyperWave is a particular instance of our hierarchical graph data model. We will review the elements of the HyperWave model one by one and see how they are mapped to elements of a hierarchical graph. HyperWave repositories will be translated to hierarchical graphs using directed graphs for all three components. (Compare Chapter 6 and the translation of the H-graph model in Chapter 7.)

As already illustrated in the running example used throughout this work, a hyperweb can naturally be seen as a graph, where nodes represent documents, and edges represent hyperlinks. Additional information can be stored by using attributed graphs, where attributes of the appropriate type can contain images, text, sound, and so on. Here we are mainly interested in the navigation structure of a hypermedia application, rather than in the internal content of single documents, thus we will consider a hyperweb as a network of nodes and edges. For the same reason and for reasons of space, we consider only the first type of container listed in Section 8.2, namely collections: sequences are just collections ordered by means of some attribute of the documents contained in them, while clusters are related to the internal structure of a document. Collections are sufficient to illustrate the main ideas of our model, and providing a model of all Hypermedia containers can be a task for future work. Summarizing, a Hypermedia repository can be easily interpreted as a hierarchical graph in our sense, where the underlying graph represents the navigation structure, and the hierarchy the organization structure.

We now consider this interpretation in more detail. To begin with, we consider collections. Collections have two roles in a HyperWave repository: they are both containers of documents, and special kind of documents one can navigate to, with associated at-

tributes, and so on. Therefore, collections are both part of the navigable structure (nodes of the underlying graph) and of the organizational structure (packages in the associated hierarchy graph) of a repository. Let us suppose that a repository R contains a set of collections $\mathbf{coll}(R)$ and, for each $c \in \mathbf{coll}(R)$ that is not the root collection, let n_c be a node and p_c be a package. For each $c, c' \in \mathbf{coll}(R)$, if c is contained in c' we let n_c be contained in $p_{c'}$ and p_c be a child of $p_{c'}$ in the hierarchy. For the root collection $\rho \in \mathbf{coll}(R)$, we only define a package p_ρ, and we let it be the root of the package hierarchy. By requiring that a hierarchy be a rooted dag we satisfy the constraints on HyperWave collection hierarchies.

As a second step, we consider documents and anchors. Let $\mathbf{doc}(R)$ be the set of documents contained in a repository R. Since these are navigable objects, we let them be nodes of the underlying graph. Anchors are also part of the navigation structure. Therefore the elements of the set of anchors $\mathbf{anc}(R)$ stored in R will also be nodes of the underlying graph. Anchors are attached to documents, and this is modeled by edges between document nodes and anchor nodes. Hyperlinks, connecting source anchors to destination anchors, destination documents, or destination collections, are also modeled as edges. The three types of nodes (anchors, documents, collections) and the two types of edges (attachment and hyperlinks) can be distinguished by means of labels. This model reflects faithfully the representation of data in HyperWave, where anchors, documents and collections are objects, and hyperlinks and anchor/document attachment are modeled as *relations*.

Finally, we have to consider the coupling graph. Obviously, every package p_c is anchored to the node n_c. Every document node $d \in \mathbf{doc}(R)$ is contained by all packages p_c, such that d is contained by c in R. Every anchor $a \in \mathbf{anc}(R)$ is contained in the same package p that contains the document to which the anchor is attached to.

Summarizing, a HyperWave repository R is modeled by a (plain) hierarchical graph $\mathbf{HG}(R) = (G(R), D(R), B(R)) = (G, D, B)$, where:

- $G = (N_G, E_G, s_G, t_G, l_G, \text{л}_G)$ is a directed graph, where
  - $N_G = \mathbf{anc}(R) \cup \mathbf{doc}(R) \cup \{n_c \mid c \in \mathbf{coll}(R) \setminus \{\rho\}\}$,
  - E_G, s_G and t_G are defined as described above,
  - $l_G : N_G \to \{coll, doc, anc\}$ and $\text{л}_G : E_G \to \{\sigma, \lambda\}$ are defined in an obvious way.
- $D = (N_D, E_D, s_D, t_D, l_D, \text{л}_D)$ is a directed graph, where $P_D = \{p_c \mid c \in \mathbf{coll}(R)\}$ and the edges representing the hierarchy relation have been defined above. Since we want to use a uniform graph model for the three components of the hierarchical graph, nodes and edges carry a trivial labeling (e.g. $\perp$ everywhere).
- $B = (N_B, E_B, s_B, t_B, l_B, \text{л}_B)$ is a directed graph, where $N_B = A_B \cup P_B = N_G \cup P_D$. The anchoring and containment relations, coded by the edges, have been described

above. Following Section 6.4, we let the nodes of B have the same label as in G or D, according to the graph they come from. All edges will be labelled with $\perp$.

Notice that the condition on hierarchical graph, that requires that all graph elements must be at least in one package, ensures that there are no *orphans*.

8.4 Modeling Hypermedia Transformations

In this section, we will use the model of HyperWave repositories that we have described in Section 8.3, and we will apply hierarchical graph transformation to specify manipulations on such repositories.

As already pointed out, in HyperWave changes to a hypermedia repository are specified by means of transactions that are requested by a client application while communicating with a server. Even though there exist clients—like Harmony—that provide a graphical user interface and support authoring, most transformations are performed by modifying one object at a time: Complex manipulations of interrelated objects (documents, anchors, collections) must be specified as a sequence of smaller operations (transactions).

Compared to this approach, hierarchical graph transformation allows to specify complex hypermedia manipulations in an intuitive and concise way. Furthermore, since graph transformation has a formally defined semantics, we can prove properties of the specified manipulations. For example, we will show that in some cases we can statically check whether the specified transformations preserve the consistency of the hypermedia database to which they are applied. The rest of this section is dedicated to illustrating these ideas.

It is outside the scope of this thesis to define a complete specification language for hypermedia transformation, based on graph transformation. Instead, we illustrate our points by means of example transformations. In these examples we use the double-pushout approach to hierarchical graph transformation (see Chapter 6). Although other graph transformation approaches can be used for hypermedia (see the example in Subsection 4.1.1), discussing other approaches to hypermedia based on graph transformation can be a topic for future research.

Since we use hierarchical graphs to model the content of HyperWave repositories, we will depict them using the same notation as in Figure 8.2. A similar notation will be used as a shorthand in graph transformation rules.

This section has the following structure. In Subsection 8.4.1 we will show how the simple operations described in Subsection 8.2.3 can be specified using hierarchical graph transformation. In Subsection 8.4.2, we consider a more complex example operation. In Subsection 8.4.3, we discuss the benefits of our approach w.r.t. the existing one.

8.4.1 Basic operations

We first define a hierarchical rule that specifies operation 1 of Subsection 8.2.3 (creation of a new document and insertion in an existing collection). This rule looks for the parent collection, and inserts the document into it. It is depicted in Figure 8.3.

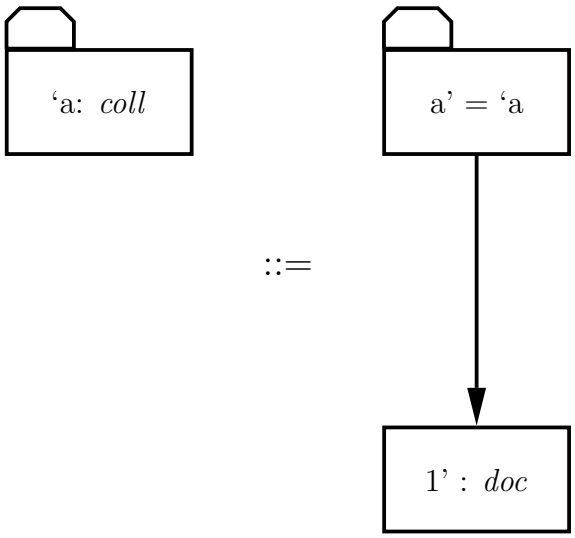


Figure 8.3: Inserting a document into a collection.

As announced, in Figure 8.3 we use the same notation of Figure 8.2. This means that, in order to reconstruct the intended DPO hierarchical rule we have to:

- Split every collection into a package and a node in the underlying graph, which have an anchoring edge in the coupling graph. These nodes, packages and edges are mapped from the left to the right-hand side in an obvious way.

- Consider any collection-to-document edge as coding a containment edge from the document to the corresponding collection package in the coupling graph.

Notice that the graphical notation that we use is unambiguous (see also the next examples).

As far as operation 2 (add a hyperlink pointing to document) is concerned, we have already seen a similar rule in Section 6.1. Here we provide a new, slightly different rule since we need not require that the source and target page of the hyperlink be in two packages that have a common ancestor, i.e. that they be in the same collection. The new version of the rule is shown in Figure 8.4.

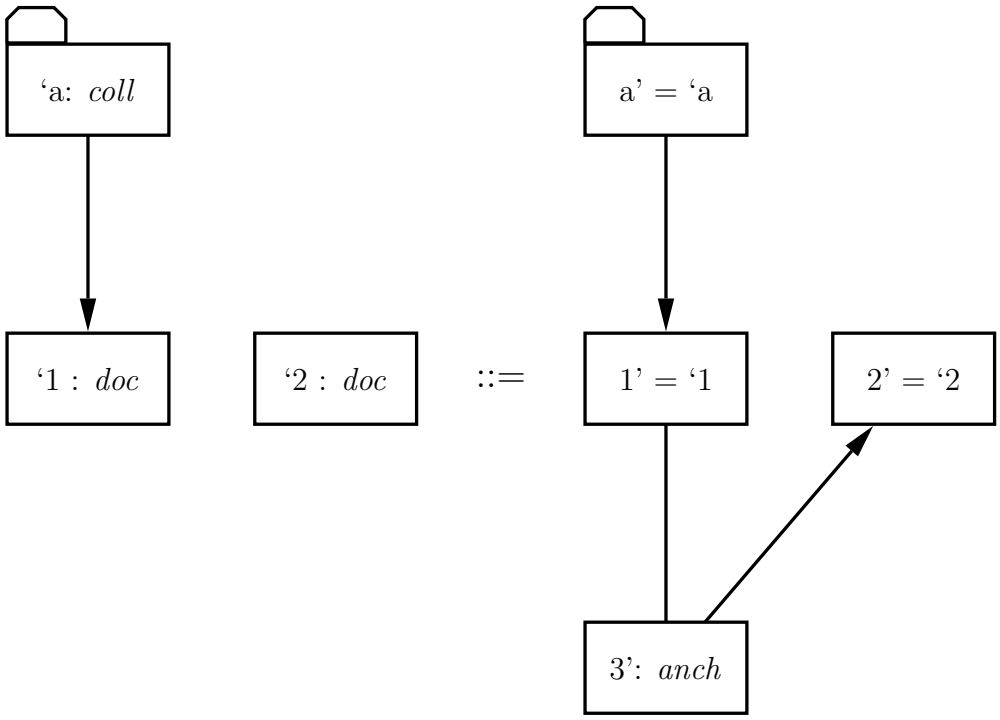


Figure 8.4: Adding a hyperlink.

Again, we use a shorthand notation for hierarchical rules, in order to make the modeled hypermedia structure easier to perceive. In addition to the previous rule, we must notice that:

- Anchors are attached to documents by means of σ-labelled edges in our model. These are depicted as undirected edges here.
- A containment edge from an anchor to the collection package containing the document is also present in the coupling graph, although not explicitly depicted.
- Hyperlinks are λ-labelled edges in the hierarchical graph representation, while here we omit the label.

Once more, the reader should notice that the DPO hierarchical graph rule underlying this shorthand notation is well-defined.

8.4.2 Complex operations

We now consider a more complex transformation, in a real world scenario. We suppose that a software company uses HyperWave for storing all documents of its employees

in a centralized repository. Every user has his or her own home collection, where documents related to his or her work are stored, as well as some personal documents. For example, a project manager will have a collection where documents about managed projects are stored, a phone book document, a collection with personal stuff, and an archive collection with old documents, e.g. documents related to completed projects. The content of the home collection of a project manager can look like in Figure 8.5.

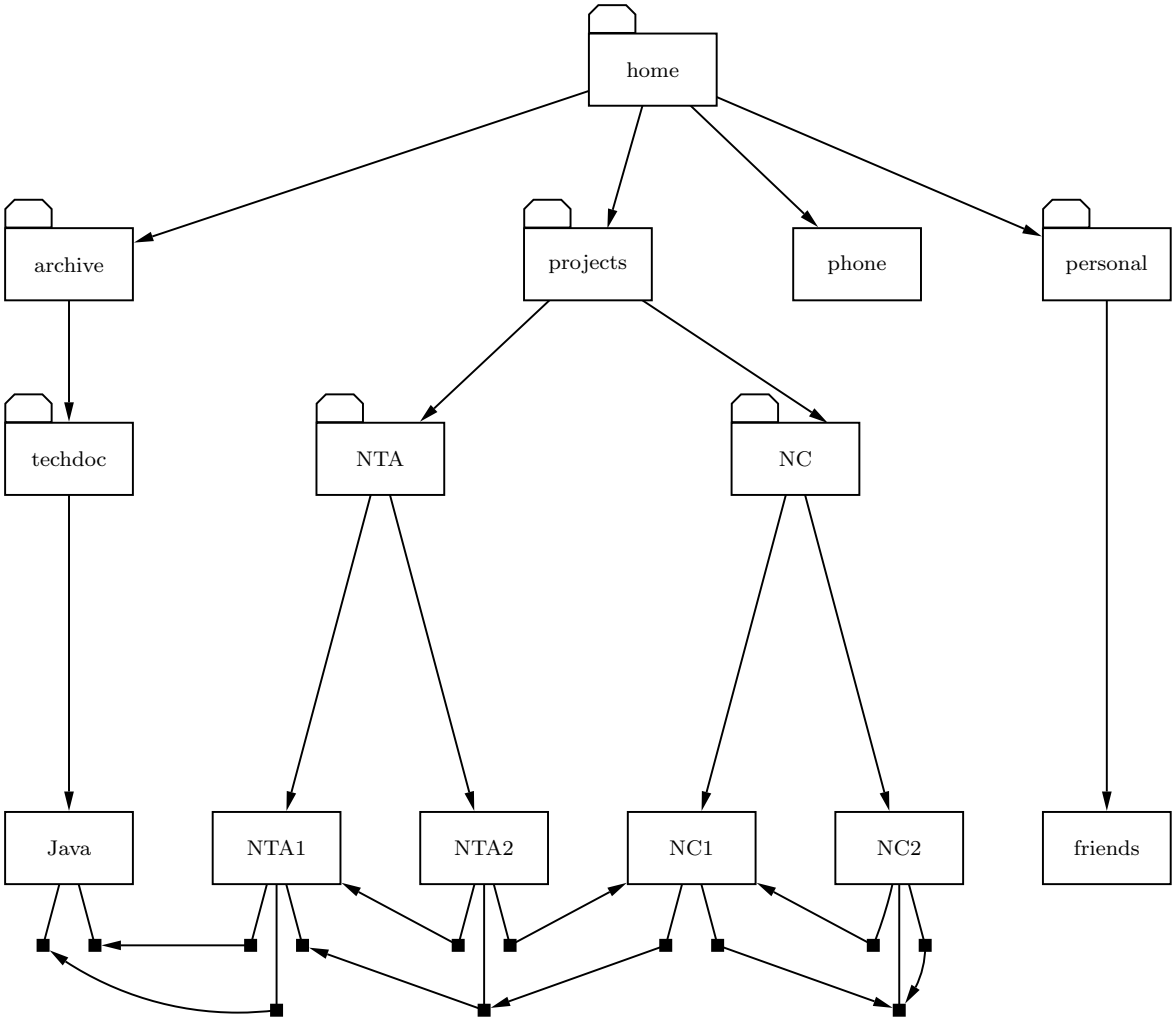


Figure 8.5: An employee's home collection.

Next, we suppose that the software company has several types of users, each one with a corresponding structure for their home collection. For example, while a project manager has the task to coordinate software projects and has technical and organizational tasks, a project administrator has mainly staff and budget administration tasks.

Therefore, an administrator's home collection will contain a staff and a financial report document. Furthermore, it will not contain a "projects" collection in the top level of its hierarchy.

This scenario can now help us to illustrate the complex transformations that can occur in a hypermedia structure. From time to time, our software company is reorganized, and employees are assigned to different tasks. For example, a manager can become an administrator. Their information structure also needs to be reorganized accordingly, and this is taken care of by a software application.

In Figure 8.6, we show a hierarchical graph rule that serves our purpose, i.e. that allows to specify the required transformation from a manager to an administrator home collection. This rules hides the project collection inside the archive collection, and adds

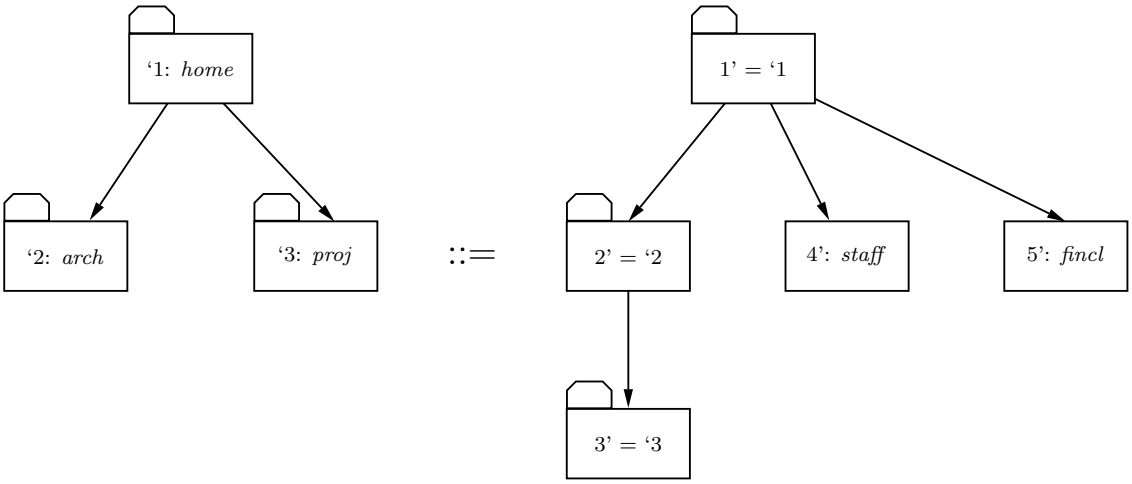


Figure 8.6: Reorganizing a collection.

the two new documents staff and financial report.

By applying this rule to the hyperweb of Figure 8.5, we obtain the hyperweb depicted in Figure 8.7.

8.4.3 Advantages of hierarchical graph transformation

In this subsection, we summarize and complete our discussion on the application of hierarchical graph transformation for the specification of hyperweb transformation.

Size of transactions First of all, we have seen that traditional approaches, like the HyperWave approach, rely on simple primitives for the specification of hyperweb transformation, while hierarchical graph transformation allows to specify transactions of arbitrary complexity.

An advantage of allowing bigger transactions is that we need not ensure consistency at each intermediate step of a large transformation: consistency can be checked only when the transaction is committed (see below the paragraph about consistency). HyperWave supports a similar but more restricted mechanism, by allowing *open links*, i.e. hyperlinks where the destination has not been inserted into a repository yet.

Style of specification In HyperWave hyperweb transformations are specified in a textual way (although single objects can be edited using a graphical interface e.g. using Harmony), while hierarchical graph transformation—like graph transformation in general—allows for visual specification. The latter approach exploits the visual character of the underlying data model, thus making the specification of operations a more intuitive task. Furthermore, the visual style together with the a-priori unlimited size of specified transactions provides for much more concise specification techniques.

Consistency Upon every transaction, a HyperWave server checks that the new hyperweb be in a consistent state, i.e. that there are no broken links, no orphan documents or collections, and that the collection hierarchy is still a rooted dag. As shown in Chapter 6, when using double-pushout for hierarchical graph transformation it is possible to check statically whether a rule will preserve these properties in a repository. In particular

1. link consistency is always ensured by DPO rules (no dangling edges are allowed),
2. the preservation of a consistent hierarchy can be ensured using Proposition 6.12,
3. the absence of orphan documents can be ensured using Proposition 6.20.

This is a first result showing that using hierarchical graph transformation can help write specifications that preserve consistency constraints. Besides the advantage of writing correct applications, this can be useful in case of big transactions (think of the installation of an electronic encyclopedia), where a lot of runtime checking can be saved.

The formal character of the graph transformation approach suggests that, if other approaches should be adopted for hyperweb transformation, similar results about consistency can be investigated.

Efficiency A hyperweb is stored in HyperWave as an object-oriented database, where all objects and relations are indexed, so that they can be accessed in a very efficient way. Although the operation of a HyperWave server is heavier w.r.t. that of a traditional

	HyperWave	*HG transformation*
Operations	Simple transformations	Complex transformations
Style	Textual, verbose	Visual, concise
Consistency	Checked at runtime	Runtime or static check
Efficiency	Good	Low

Table 8.1: HyperWave vs hierarchical graph transformation.

web server, the performance of such a system is quite good (but update operations tend to be slow, due to the required update of indices, see e.g. [Mau96, Section 10.7]).

On the other hand, executing (hierarchical) graph transformation rules is a very inefficient task. The first source of inefficiency is the fact that finding an occurrence of the left-hand side of a rule is an NP-complete problem (see e.g. [Meh84]). The second is the non-deterministic application of rules—every rule can have more than one occurrence, and more than one rule can be applicable at each derivation step—which can require (a lot) of backtracking when a derivation reaches a dead end.

This situation can be improved in several ways. First, by using attribute values, node types, by providing nodes to be matched as parameters to the rule, or by means of so called *partial matches*, the number of possible matches for a rule can be reduced. Second, although finding matches is an NP-problem, it is possible to provide algorithms that have a good performance on average (see e.g. [ERT99]). A last source of efficiency can be the use of a control mechanism that drives the application of rules. In this respect, our hierarchical graph transformation framework, where the notion of a graph transformation approach is used, can be easily extended by introducing transformation units (see [Kus99]), thus adding such a control mechanism.

The above discussion is summarized in Table 8.1.

8.5 Summary

In this chapter, we have considered the application of our hierarchical graph model in the hypermedia field for the specification of hyperweb manipulations. Our discussion considers a concrete case study, based on the Hyper-G/HyperWave$^{\text{tm}}$ system.

First of all, this investigation has shown what concrete problems can require the introduction of a hierarchical graph data model. This has been done by looking at the problems that motivated the introduction of the HyperWave data model, and by showing that this is just a particular kind of hierarchical graph model.

As a second step, we have pointed out that, in spite of its advanced model of hypermedia, HyperWave does not provide any high-level primitives for specifying hyperweb

manipulations. We have proposed an approach, based on hierarchical graph transformation, that allows to address this problem. By means of examples, we have shown that we can specify complex hyperweb transformations in an intuitive, concise, and precise way. Furthermore, by applying the double-pushout approach to hierarchical graph transformation, we were able to use the results of Chapter 6 in order to determine whether a given rule preserves the consistency of a hyperweb.

In spite of these encouraging results, our investigation is still at a preliminary stage in many respects. First of all, we have not provided a complete specification language, but rather presented the main ideas that can be the basis for defining such a language.

Secondly, we have used the double-pushout approach, while it is an open issue which transformation approach is better suited for specifying hypermedia transformation. This issue has been scarcely investigated (but see for example [GM00]—an approach to hypertext authoring support—where graphs and graph transformation are used for modeling hypermedia documents and operations), and more practical experience is needed. For example, when we want to substitute a document with a new version, we need to replace it and embed the new document in an unknown context. In this case, some kind of node replacement looks more suitable than double-pushout. (Compare also a similar example in Subsection 4.1.1.)

Another important issue concerning the practical applicability of our concepts is related to efficiency. This is a more general problem that affects graph transformation. A (partial) solution to this problem can be the introduction of some control mechanism on the application of rules, for example by means of transformation units (see [Kus99]), leading to a notion of programmed (hierarchical) graph transformation system (see also [Sch97]). Since transformation units rely on the notion of a graph transformation approach, which is provided by our framework as a result of Construction 5.6 (see also Definition 5.9), the extension of our framework to allow programmed transformation is straightforward.

A final remark concerns visibility/encapsulation. In fact, a feature of HyperWave, that we have not considered here, is the possibility to manage different users with different access permissions to collections, documents, or links. This suggests an extension of our model of hypermedia, using encapsulated hierarchical graphs, and introduces to an interesting direction of research, that of encapsulated hierarchical graph transformation. The idea is that a transformation rule should be applied "from the point of view of a particular user", i.e. that it should be allowed to access and modify only the portion of a hyperweb (of a hierarchical graph) that is visible to a specific users. First ideas in this direction have been considered in [BEMW00].

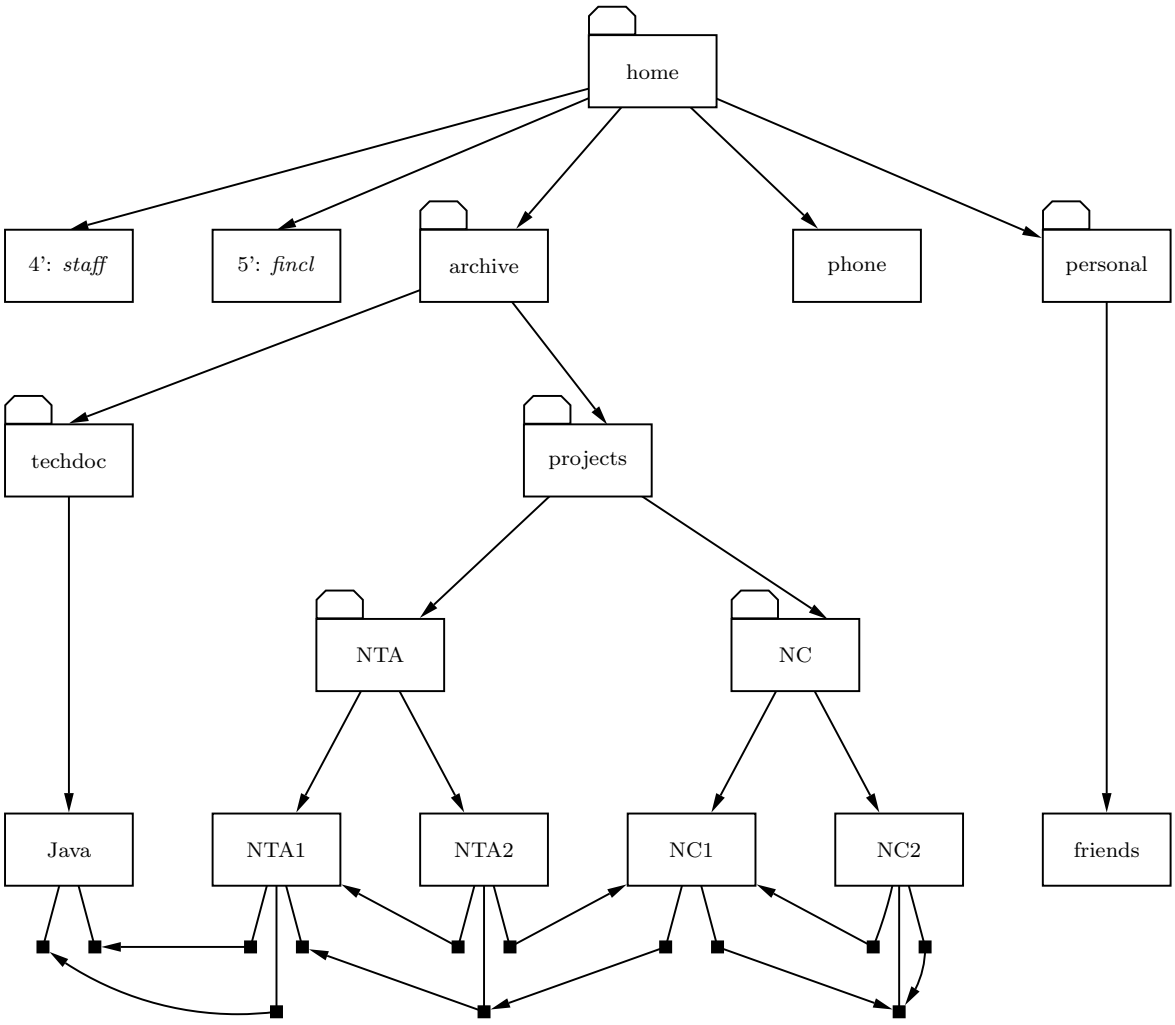


Figure 8.7: A home collection after reorganization.

Chapter 9

Conclusions

In this chapter, we summarize the results contained in the present thesis, compare them with related work, and suggest future developments of our hierarchical graph model.

9.1 Results

In this thesis we propose an abstract model of hierarchical graphs and their transformations. This model allows to add a hierarchy of packages to many different graph-like structures. It also supports encapsulation and typing, as well as transformation rules for defining operations. The features of this model have been chosen having in mind applications like object-oriented modeling, database and hypermedia modeling. Operations are based on the graph transformation paradigm (see [Roz97]). In this section, we summarize the results contained in this thesis following the conceptual organization outlined in Chapter 1.

Static aspects of hierarchical graphs

In Chapter 2, we have studied various notions of hierarchical graphs existing in the literature, and we have informally discussed the static aspects of our hierarchical graph model. In Chapter 3, we have provided the corresponding formal definitions.

Particular attention has been paid to hierarchical graphs as a data model in the areas of object-orientation, of databases, of hypermedia, and of graph grammars and graph transformation. The result of this investigation was that the notion of hierarchical structuring occurs in all the considered areas, although the terminology and the concrete formalizations may vary considerably: A graph may be a network of objects and links, or the content of a database, or a hypermedia network, while a hierarchy can be modeled using aggregate objects, views, containers, and so on.

As a consequence, our first task was to identify the common aspects of different approaches, and to distinguish these aspects from those that are tied to a particular

approach or application, and that should not be considered essential for hierarchical graphs. In this respect, also the very notion of a graph was a delicate issue: there are many different kinds of graphs that can be hierarchically structured. We chose an abstract notion of a graph, which we called a *graph skeleton*.

Another important decision was that the graph to be structured and the hierarchy should be kept separate. This *decoupled* approach (see also [BEMW00]) allows for more flexibility in the definition of hierarchical graphs and hierarchical graph transformation rules. On the other hand, in a *coupled* approach, the hierarchy is built on graph elements, i.e. nodes and/or edges contain nested graphs. In our model a hierarchical graph is represented as a triple consisting of an *underlying graph* (the graph to be structured), a *hierarchy graph* whose components are called *graph packages* and are generic containers of nodes and edges, and a *coupling graph* that relates the first two graphs. Although packages are distinct entities, they can still be anchored to nodes or edges. Thus, if necessary, it is possible to recover the information that is implicitly stored in a coupled approach.

Other important features, which can be added to the core model, are *typing* and *encapsulation*. Hierarchical graph typing is an extension of (flat) graph typing, and allows to classify hierarchical graphs according to common structural properties. Encapsulation allows to define import/export interfaces between packages that are adjacent in the hierarchy, and to use them to control which packages have access to each graph element, i.e. to *hide* certain nodes or edges from certain packages. Hierarchical graphs with encapsulation are called *encapsulated hierarchical graphs*.

Operations on hierarchical graphs

As a second topic, we have considered an abstract model of hierarchical graph transformation in Chapter 4 (motivation and informal presentation) and in Chapter 5 (formal definitions).

The main idea is that we can specify hierarchical graph transformation by extending the notion of (flat) graph transformation (see [Roz97]) to the hierarchical case. Our decoupled approach, i.e. the use of three flat graphs to model one hierarchical graph, suggests that a hierarchical graph transformation rule can be modeled as the combination of three flat graph transformation rules.

As with the static aspects, we faced the problem of what graph transformation approach should be used in our model. Instead of choosing a particular one, we have used an abstract notion of graph transformation (extending the notion of a *graph transformation approach* used in [Kus99]), and provided a standard construction that combines three graph transformation approaches into a hierarchical graph transformation approach. (See Construction 5.6.)

An important issue concerns the need for *coordination* between the rules that are applied in parallel to the three components of a hierarchical graph. This problem is

addressed by extending graph transformation approaches to *tracking graph transformation approaches*, i.e. approaches where all transformation steps provide a way to track preserved nodes and edges of the transformed graph. This extension is then used to define *coordinated rules* and *coordinated hierarchical graph transformation*. (See Definition 5.9.)

A more technical issue concerns the problem of using a particular notion of graph with different transformation approaches. For example, since hierarchy graphs are always unlabelled directed graphs, we cannot apply the double-pushout approach (see e.g. [CMR+97]) as it is to transform these graphs: this approach uses labelled graphs. To address this problem, we propose a standard way to code graphs with other graphs, described in Section 5.3.

Defining concrete models

After describing a generic framework for hierarchical graphs and their transformation, we have considered ways to instantiate it and define concrete approaches.

The main problem when instantiating the framework is that we have to ensure that hierarchy graphs are transformed into correct hierarchy graphs and that coupling graphs are transformed into correct coupling graphs as well. In Chapter 6 we have instantiated our framework using the double-pushout approach for all the three components of a hierarchical graph, and provided some decidable conditions on rules that ensure this kind of correctness (see Proposition 6.12 and Proposition 6.20). In Chapter 8 we have considered an application of this new model (see also below).

Besides using the framework for defining new approaches, we exploit it for studying existing ones (see Chapter 7). By translating several existing hierarchical graph transformation approaches to our model we can classify them and compare them with each other. This comparison shows that our model covers the static aspects of hierarchical graphs quite easily, whereas providing a complete simulation of dynamic aspects is a much more complex task.

Among the considered approaches, we have studied H-graph grammars (see e.g. [Pra79]) in full detail. This approach is interesting because it follows the node-replacement/connecting paradigm (see Section 4.2), whereas most other approaches follow the (hyper)edge-replacement/gluing paradigm. For H-graphs we have provided a detailed translation of graphs and rules, and proved the equivalence of translated rules with the original ones. The translated rules are a variety of NLC rules (see [Roz97]). This approach points out a limitation in our framework, namely the difficulty to model certain embedding mechanisms in the coupling graphs (see Subsection 7.1.5 for more details). This suggests possible future developments, where more powerful transformation mechanisms are used for coupling graphs, or a different representation of hierarchical graphs is introduced. This example also shows that certain applications may require the use of different graph transformation approaches in the three components of hierarchical

graph rules.

An application

The definition of our model started by looking at application areas, and identifying common aspects of hierarchical structuring in these areas. As a conclusion of our work, we went back to the specific area of hypermedia systems, and tried to apply the concepts that we have developed.

The considered application concerns Hyper-G/HyperWave, a hypermedia system developed in the early nineties at the university of Graz (Austria). The advanced data model of HyperWave can be easily interpreted as a hierarchical graph data model by instantiating our framework.

As far as operations are concerned, we have described several examples showing that hierarchical graph transformation provides a concise and intuitive approach for specifying (complex) hypermedia manipulations. Another advantage is that, if we use the double-pushout approach, we can use the results from Chapter 6 to ensure that the specified manipulations preserve the consistency of the hypermedia structures to which they are applied.

This approach, however, still needs to be improved as far as efficiency is concerned. Future work also concerns the use of encapsulation: HyperWave defines visibilities (access permissions) on hypermedia documents, which suggests the extension of the hierarchical graph transformation model in order to handle encapsulated hierarchical graph transformation.

9.2 Related Work

In this section, we provide an overview on related approaches to hierarchical graphs and their transformations. We do not elaborate on approaches that have already been discussed in Chapter 1, 4 or 7, excepting the approaches related to graph transformation, for which we provide a brief summary here. The remaining approaches stem mostly from areas like database and knowledge modelling. Some of them have been conceived as general-purpose models (e.g. [Har88]), while others are targeted to specific applications (e.g. [PL94]). For each approach we provide a short summary and we outline its relation with our model.

From a static point of view, it turns out that our model covers the considered approaches quite well. An interesting exception, which can be a starting point for future extensions to our model, is the use of the Cartesian product as introduced in higraphs (see below).

As far as operations are concerned, a thorough comparison with other approaches requires a considerable effort, but the first results presented in Chapter 7 and [BH01]

are encouraging. In this respect, another interesting topic for future research is the comparison with the Hyperlog model (see below), which provides a rule-based transformation approach whose relation to known approaches in the graph transformation field should be investigated.

Approaches in the graph transformation field

Our hierarchical graph model relies on rules for the definition of operations, and it belongs to the area of graph transformation ([Roz97], [RE+99a], [RE+99b]). We have already analyzed related approaches within this field in detail in Subsection 2.2.6, Subsection 4.2.2, and in Chapter 7. We make some final considerations here.

Rule-based hierarchical graph transformation has been considered by several authors, motivated by different requirements and application areas. Although appropriate for particular purposes, none of the existing approaches can be considered abstract enough to serve as a general hierarchical graph model. In fact, each approach enforces restrictions on the available notion of graph, on the kind of hierarchy, and on the provided operations, so that it cannot capture the variety of the others in a natural way.

In contrast, our model aims more at flexibility than at a particular application, and provides a generic framework which can be instantiated to obtain specific hierarchical graph transformation approaches. First results (see Chapter 7) indicate that our model can simulate faithfully the behaviour of existing approaches. However, a lot of work needs to be done for a precise classification of hierarchical graph transformation approaches.

Higraphs and hygraphs

The *higraph* formalism—a visual model for databases, knowledge representation, statecharts and similar applications—is introduced in [Har88]. This paper is interesting since it investigates the notion of a hierarchical graph from an abstract point of view. It identifies two orthogonal concepts, namely

1. the notion of a graph (see also Chapter 1) as a means to model a set of objects and a relation between them,

2. and the use of *blobs* to describe sets and relations between them (subsets, union, intersection, Cartesian product, and so on).

The main idea is that a visual data model requires both these concepts, and their combination leads to the definition of higraphs. A higraph then consists of a collection of *blobs*, which are linked through edges, and related to each other by the set-theoretic relations mentioned in point 2 above.

From a conceptual point of view, this formalism is very similar to ours, since we also consider relational information (the underlying graph) and groupings (the hierarchy of packages) that are combined together (see Chapter 2). However, higraphs are a coupled approach, where blobs play both the role of nodes and of grouping elements. Our approach is more general since it is decoupled: nodes and edges are distinct from packages. (See again Subsection 2.3.4.)

Another difference is that the relations between blobs are of a set-theoretic nature: a blob is either the union or the Cartesian product of its sub-blobs. On one hand packages are more flexible because they do not enforce that the content of a package be the union of the contents of its subpackages: a more flexible distribution of graph elements is allowed. On the other hand, building the Cartesian product of subpackages is not supported in our model. We believe that this operation is a less natural mechanism for building hierarchies of objects than grouping is. However, this is an interesting idea and could be the basis for future extensions of our model.

In [CM93] and [Con94], a variation of higraphs, called *hygraphs*, is presented. The main difference is that hygraphs are a decoupled approach since nodes are distinct from blobs. Also, more than one blob can be anchored to a single node. It is easy to see that this model can also be represented in our framework if we relax the injectivity condition in Definition 3.6 on page 46 (several packages can be anchored to one node).

Hyperlog

Hypernode/Hyperlog (see [PL94]) is a hierarchical graph data model aimed at the implementation of databases and, in particular, of hypertext databases. This model is coupled, since the hierarchy is based on complex nodes called *hypernodes*. A collection of hypernodes is called a *hypernode repository*. Hypernodes are typed. A repository can be easily represented as a hierarchical graph in our sense, provided that we split (as usual) hypernodes into pairs of nodes and packages anchored to them. The underlying graph can be obtained by composing all the local graphs of hypernodes. The hierarchy graph can be obtained by considering the nesting of hypernodes (notice that cyclic hierarchies are allowed).

Hyperlog also provides rule-based operations, where queries are used to specify patterns to be found in a repository, as well as updates (deletion/creation of hypernodes, atomic nodes and edges). Each rule is of the form $p \leftarrow p_1, \ldots, p_k$, where $p_1, \ldots, p_k$ are the patterns to be matched, and p specifies the updates. Matching and updates use variable substitution. Although these rules remind graph rewrite rules, the particular formalism used by the authors requires a deeper analysis—which is outside the scope of this work—in order to provide a satisfactory comparison with graph transformation.

9.3 Open Problems and Future Work

In this section, we consider open issues and outline possible future developments of our model.

A first issue concerns the representation of hierarchical graphs. Our experience with translating existing hierarchical graph transformation approaches showed that certain operations on the coupling graph are difficult or impossible to model in our framework. The main reason is that edges of the underlying graph are represented as nodes in the coupling graph. As a consequence, if a rule modifies the underlying graph using an embedding mechanism (for example NLC embedding) that introduces an unbounded number of edges into the underlying graph, a similar effect cannot be obtained on the corresponding nodes in the coupling graph. This limitation is currently overcome by "forgetting" the location of edges in a hierarchy, i.e. by removing them from the coupling graph. A better future solution can rely on a different—homogeneous—representation of edges in the coupling graph, or on the use of new, more powerful graph transformation approaches.

A second issue concerns typing. We support typing only partially in our model, since typed hierarchical graph transformation has not been considered, whereas some approaches do provide typing, e.g. [EH00]. In view of our decoupled approach, typed transformation should be easily obtained as a combination of three typed graph transformation approaches.

Similarly, encapsulation has been considered only from a static point of view. The transformation of encapsulated hierarchical graphs (EHG) has already been considered (although not in detail) in [BEMW00]. Here, an interesting idea is that operations should be associated to packages and be local to the portion of a hierarchical graph "visible" from a certain package. In this way, a transformation applied to a package should have "write/update permission" on nodes and edges owned by that package, and only "read/query permission" on imported items. Other operations could involve switching the public/private (exported/non-exported) state of a node or edge, and so on.

On the application side, in our hypermedia case study (see Chapter 8) encapsulation would allow to distinguish local documents of a collection from imported ones and associate operations to a particular collection. This could be a way to model access privileges of HyperWave. Besides supporting encapsulation, our case study could be further extended considering performance issues, and developing a complete language/specification method for hypermedia transformation.

Another interesting application area concerns the current effort for the development of GXL, a common graph exchange format based on XML (see [HWS00]). This format should allow different tools to exchange data in the form of graphs. Recent discussion considered the support of hierarchical graphs in GXL, and many different representations were proposed for them. We think that, thanks to its abstract nature, our abstract

model of a hierarchical graph can be a basis for improving hierarchical graph support in GXL.

Another possible extension concerns the way in which the content of subpackages is combined in order to form the content of a package. While up to now we have only considered mechanisms like sharing and import/export, certain approaches (see [Har88]) suggest the use of set-theoretic operations like the Cartesian product: the content of a package can be obtained as the Cartesian product of the content of its subpackages. We think that this can be an interesting topic for future investigation.

Appendix A

Tracking Graph Transformation Approaches

In this appendix, we show how some existing approaches to graph transformation can be seen as tracking approaches in a natural way. For reasons of space, we will only sketch these examples. Working out the details is an easy exercise. We assume that the reader is familiar with the definition of rules and direct transformation in the various approaches. The reader can also refer to the appropriate chapter of [Roz97].

The double-pushout approach

Given a double-pushout rule $r = (L, i, K, j, R)$[1]. Its skeleton is given by (S_L, S_R, tr), where $tr : S_L \to S_R$ is defined as follows:

- For all nodes $n \in N_L$ such that there exists $n' \in N_K : n = i_{\mathrm{N}}(n')$ let $tr_{\mathrm{N}}(n) := j_{\mathrm{N}}(n')$.
- For all edges $e \in E_K$ such that there exists $e' \in E_K : e = i_{\mathrm{E}}(e')$ let $tr_{\mathrm{E}}(n) := j_{\mathrm{E}}(e')$.

This construction is always possible, since i is injective.

Analogously, let us consider a direct derivation step using r with G being the start graph, D the intermediate graph, H the transformed graph, and $m : L \to G$, $k : K \to D$, $m' : R \to H$, $d : D \to G$, $d' : D \to G'$ the corresponding morphisms. Then the skeleton of the derivation step is (S_G, S_H, φ), where $\varphi : S_G \to S_H$ is defined as follows:

- For all nodes $n \in N_G$ such that there exists a node $n' \in N_D : n = d_{\mathrm{N}}(n')$ let $\varphi_{\mathrm{N}}(n) := d'_{\mathrm{N}}(n)$.

[1] If the reader is not familiar with the DPO approach, he or she can find the corresponding definitions in Chapter 6, Section 6.2. We think that introducing them here would make our presentation unnecessarily heavy.

- For all edges $e \in E_G$ such that there exists an edge $e' \in E_D : e = d_{\mathrm{E}}(e')$ let $\varphi_{\mathrm{E}}(e) := d'_{\mathrm{E}}(e)$.

Again, this construction is always possible since d is injective.

Finally, when considered as skeleton morphisms $m : S_L \to S_G$, $m' : S_R \to S_H$, m and m' are the morphisms required by the definition of a tracking graph transformation approach, since it is easy to verify that $\varphi \circ m = m' \circ tr$.

The single-pushout approach

Given a single-pushout rule $r = (L \xrightarrow{t} R)$, where $t : L \to R$ is a partial injective graph morphism, we can easily define its skeleton as (S_L, S_R, tr), where $tr : S_L \to S_R$ is defined by letting $tr_{\mathrm{N}} := t_{\mathrm{N}}$, and $tr_{\mathrm{E}} := t_{\mathrm{E}}$.

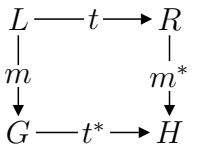


Figure A.1: SPO derivation.

Analogously, let us consider a direct derivation step using r with G being the start graph, H the transformed graph, and $m : L \to G$, $m^* : R \to H$, $t^* : G \to H$ the corresponding morphisms such that the resulting diagram (see Figure A.1) is a push-out. Then the skeleton of the derivation step is (S_G, S_H, φ), where $\varphi : S_G \to S_H$ is defined by letting $\varphi_{\mathrm{N}}(n) := t^*_{\mathrm{N}}(n)$ and $\varphi_{\mathrm{E}}(e) := t^*_{\mathrm{E}}(e)$.

Finally, m and m^* provide in a natural way the skeleton morphisms $m : S_L \to S_G$, $m^* : S_R \to S_H$ that are required by the definition of a tracking graph transformation approach, since it is easy to verify that $\varphi \circ m = m^* \circ tr$.

NLC rewriting

As an example of tracking transformation approaches based on node replacement, we consider node-label controlled (NLC) rewriting (see e.g. [ER97], see also Chapter 7). In NLC graph grammars, we have a given alphabet of terminal symbols Σ and an alphabet of non-terminals Δ, and a production is a pair $X \to D$, where X is some non-terminal symbol and D is an undirected graph with its nodes labelled over $\Sigma \cup \Delta$. A direct derivation from a graph G is specified as follows:

1. A node n of G with label X is chosen.

2. A new graph H is built with

   - $N_H := N_G \setminus \{n\} \cup N_D$, where we suppose, without loss of generality, that $N_G \cap N_D = \emptyset$.
   - E_H is obtained from E_G by removing all edges incident in n, adding all edges of D, and establishing new edges according to an embedding mechanism—whose details are irrelevant here—controlled by node labels.

The skeleton of a rule $X \to D$ is a triple (LS, RS, tr), where $LS := (\{\nu\}, \emptyset, \emptyset)$ (with ν some node), $RS := S_D$, $tr := \emptyset$.

Let E_G^n be the set of edges of G that are incident in n. The skeleton of a derivation step is built by taking a triple (S_G, S_H, φ) and two morphisms $m : LS \to S_G$, $m' : RS \to S_H$, where for all $x \in (N_G \cup E_G) \setminus (E_G^n \cup \{n\})$ we have $\varphi(x) = x$ (every element different from n and from its incident edges is preserved), $m(\nu) = n$ (see above), and m' maps all nodes and edges of D to their copy in H.

Appendix B

Proofs for Various Results

In this appendix, we provide the proofs that we have omitted in the chapters where they are presented. These results are to be found in Chapter 6 and in Chapter 7.

Lemma 6.15

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 6.16

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

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

Proof If $\rho_G \in m(N_L)$ then, for each $u \in N_D - k(N_K)$ we have $d(u) \succ_G^+ \rho_G$. By Lemma 6.15, 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 $\rho_G \in m(N_L)$ then $D \succ k(K)$. In a similar way, we can prove that if $\rho_G \in d(N_D)$ then $L \succ i(K)$.

□

Proposition 7.8

Let H and H' be two H-graphs, π be an H-grammar rule, and let $HG \sim \mathbf{HG}(H)$ be a hierarchical graph. Then $H \Rightarrow_\pi H'$ iff there exists a rule $r = (\gamma, \delta, \beta) \in r_\pi$ and a hierarchical graph $HG' \sim \mathbf{HG}(H')$ such that $HG \Rrightarrow_r HG'$.

Proof In what follows, let $\mathbf{A}$ be a terminal alphabet, $\mathbf{B}$ a non-terminal alphabet, and $\mathbf{N}$ a predefined set of nodes. Let $H = (N_H, V_H)$ be an H-graph over $\mathbf{N}$ and $\mathbf{A} \cup \mathbf{B}$. Let $\pi = (C, K, G, I, O)$ be an H-graph grammar rule over the same alphabets, and r_π the corresponding set of hierarchical graph rules built as in Construction 7.5. Let $HG = (\Gamma, Д, Б) \sim \mathbf{HG}(H)$ be a hierarchical graph.

We will first prove that π can be applied to H in a node n iff there exists $r = (\gamma, \delta, \beta) \in r_\pi$ such that γ can be applied to Γ in n, δ can be applied to $Д$ in p_n, and β can be applied to $Б$ by matching the subgraph induced by $\{n, p_n\}$.

If) It is easy to see that if π can be applied to H in n then there exists a coordinated application of some rule $(\gamma, \delta, \beta) \in r_\pi$ to $(\Gamma, Д, Б) = HG \sim \mathbf{HG}(H)$, where the same node $n \in N_\Gamma = N_{G(H)} = N_H$ and its qualified package $p_n \in P_Д$ are replaced. In fact, if $\pi = (C, K, G, I, O)$, we have $V_H(n) = C$, and therefore $l_\Gamma(n) = (C, t_\gamma)$, for some $t_\gamma \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}$, $l_Д(p_n) = (C, t_\delta)$, for some $t_\delta \in \{\mathbf{p}, \mathbf{i}\}$, $l_Б(n) = (C, t_\beta)$, for some $t_\beta \in \{\mathbf{n}, \mathbf{o}\}$, and $l_Б(p_n) = (C, \mathbf{p})$. This implies that we can apply $\gamma = \gamma_\pi^{t_\gamma}$ to Γ in n, $\delta = \delta_\pi^{t_\delta}$ to $Д$ in p_n, $\beta = \beta_\pi^{t_\beta}$ to $Б$ in n and p_n.

Only-if) Since $\gamma = \gamma_\pi^t = (C, t) \to R_\gamma$ for some $C \in \mathbf{B}$, some tag $t \in \{\mathbf{n}, \mathbf{i}, \mathbf{o}, \mathbf{io}\}$, some graph R_γ, the fact that γ is applicable to Γ in n, implies that $l_\Gamma(n) = (C, t)$. But then $l_{G(H)}(n) = (C, \mathbf{n})$ because $\mathbf{ut}(\Gamma) = \mathbf{HG}(H)$. This implies that $V_H(n) = C$ and therefore π can be applied to H in n.

From these observations, the proof boils down to showing that, if $H \Rightarrow_\pi H'$ by applying π in n, and $HG \Rrightarrow_r HG' = (\Gamma', Д', Б')$, for some $r = (\gamma, \delta, \beta) \in r_\pi$, by applying γ to n in Γ, δ to $q = p_n$ in $Д$, β to n and q in $Б$, then $HG' \sim \mathbf{HG}(H')$. We proceed by proving that the corresponding graphs in HG' and $\mathbf{HG}(H')$ may only differ for their node tags, i.e. that $\mathbf{ut}(HG') = \mathbf{ut}(\mathbf{HG}(H')) = \mathbf{HG}(H')$.

In what follows, we will abuse notation and indicate with n both the replaced node of H, Γ and $Б$, and the node in the left-hand sides of γ and β. The same holds for q,

which will be a replaced package of Д and Б, as well as a package in the left-hand sides of δ and β. Also, for a hierarchy graph D, we will use the notation N_D instead of P_D in order to have a uniform notation.

By looking at how the translated rules are defined, we can represent the component graphs of HG' in the following way:

1. $N_{\Gamma'} = N_\Gamma \setminus \{n\} \cup N_{R_\gamma}$
2. $E_{\Gamma'} = E_\Gamma \setminus E_\Gamma^{n-} \cup E_{\Gamma,\gamma}^{n+} \cup E_{R_\gamma}$
3. $l_{\Gamma'} = l_\Gamma|_{N_\Gamma \setminus \{n\}} \cup l_{R_\gamma}$
4. $N_{Д'} = N_Д \setminus \{q\} \cup N_{R_{\delta,p}}$
5. $E_{Д'} = E_Д \setminus E_Д^{q-} \cup E_{Д,\delta}^{q+} \cup E_{R_\delta}$
6. $l_{Д'} = l_Д|_{N_Д \setminus \{q\}} \cup l_{R_\delta}$
7. $N_{Б'} = N_Б \setminus \{n, q\} \cup N_{R_{\beta,p}}$
8. $E_{Б'} = E_Б \setminus E_Б^{n,q-} \cup E_{Б,\beta}^{n,q+} \cup E_{R_\beta}$
9. $l_{Б'} = l_Б|_{N_Б \setminus \{n,q\}} \cup l_{R_\beta}$

where

a. In equation 2, E_Γ^{n-} is the set $\{e \in E_\Gamma \mid n \in \mathbf{Ends}_\Gamma(e)\}$ (with $\mathbf{Ends}_\Gamma(e) = \{s_\Gamma(e), t_\Gamma(e)\}$) of edges that are removed when node n is replaced. The set $E_Д^{q-}$ in equation 5 and the set $E_Б^{n,q-}$ in equation 8, are defined in a similar way.

b. In equation 2, $E_{\Gamma,\gamma}^{n+}$ is the set of new connecting edges, defined according to Definition 7.4 as

$$\begin{aligned}
E_{\Gamma,\gamma}^{n+} &= \{(u,\lambda,v) \mid (u,\lambda,n) \in E_\Gamma, \\
&\quad (l_\Gamma(u), l_{L_\gamma}(n), l_{R_\gamma}(v), in) \in \mathcal{C}_\gamma\} \cup \\
&\quad \{(v,\lambda,u) \mid (n,\lambda,u) \in E_\Gamma, \\
&\quad (l_\Gamma(u), l_{L_\gamma}(n), l_{R_\gamma}(v), out) \in \mathcal{C}_\gamma\} \\
&= \{(u,\lambda,v) \mid (u,\lambda,n) \in E_\Gamma, \exists X_3 \in \overline{\Sigma} : l_{R_\gamma}(v) = (X_3, \mathbf{i}), \\
&\quad (l_\Gamma(u), (C,\mathbf{n}), (X_3,\mathbf{i}), in) \in \mathcal{C}_\gamma\} \cup \\
&\quad \{(v,\lambda,u) \mid (n,\lambda,u) \in E_\Gamma, \exists X_3 \in \overline{\Sigma} : l_{R_\gamma}(v) = (X_3, \mathbf{o}), \\
&\quad (l_\Gamma(u), (C,\mathbf{n}), (X_3,\mathbf{o}), out) \in \mathcal{C}_\gamma\} \\
&= \{(u,\lambda,I) \mid (u,\lambda,n) \in E_\Gamma\} \cup \{(O,\lambda,u) \mid (n,\lambda,u) \in E_\Gamma\}
\end{aligned}$$

where the last equation says that the set of new edges contains one edge (O,λ,u) for each outgoing edge (n,λ,u) starting in the removed node n, and one edge (u,λ,I) for each incoming edge (u,λ,n) ending in the removed node n.

c. In equation 5, the set of new connecting edges in the hierarchy graph is again defined, according to the NLC rewrite mechanism given in Definition 7.4, as

$$\begin{aligned} E^{q+}_{\text{Д},\delta} &= \{(q', \lambda, q'') \mid (q', \lambda, q) \in E_{\text{Д}}, \\ &\quad (l_{\text{Д}}(q'), l_{L_\delta}(q), l_{R_\delta}(q''), in) \in \mathcal{C}_\delta\} \cup \\ &\quad \{(q'', \lambda, q') \mid (q, \lambda, q') \in E_{\text{Д}}, \\ &\quad (l_{\text{Д}}(q'), l_{L_\delta}(q), l_{R_\delta}(q''), out) \in \mathcal{C}_\delta\} \end{aligned}$$

where, since $\mathcal{C}_\delta$ does not contain any instruction of type (x, y, z, out), the second set of edges is empty[1]. Furthermore,

- $l_{L_\delta}(q) = (C, t_\delta)$,
- and $l^2_{R_\delta}(q'') = \mathbf{i}$ if and only if $q'' = p_u$ for some $u \in N_{G_\pi}$, as can be seen from the definition of hierarchy rules in Construction 7.5,

and therefore we have

$$\begin{aligned} E^{q+}_{\text{Д},\delta} &= \{(q', \lambda, q'') \mid (q', \lambda, q) \in E_{\text{Д}}, \exists X_3 \in \overline{\Sigma} : l_{R_\delta}(q'') = (X_3, \mathbf{i}), \\ &\quad (l_{\text{Д}}(q'), (C, \mathbf{p}), (X_3, \mathbf{i}), in) \in \mathcal{C}_\delta\} \\ &= \{(q', (\bot, \mathbf{e}), p_u) \mid (q', (\bot, \mathbf{e}), q) \in E_{\text{Д}}, u \in N_G\} \end{aligned}$$

i.e. all packages p_u corresponding to "root" nodes of the right-hand side of π are nested in all packages in which the replaced package q was nested.

d. In equation 8, the set of new connecting edges in the coupling graph is

$$\begin{aligned} E^{n,q+}_{\text{Б},\beta} &= \{(q', \lambda, v) \mid (q', \lambda, n) \in E_{\text{Б}}, \\ &\quad (l_{\text{Б}}(q'), l_{L_\beta}(n), l_{R_\beta}(v), in) \in \mathcal{C}_\beta\} \cup \\ &\quad \{(v, \lambda, q') \mid (n, \lambda, q') \in E_{\text{Б}}, \\ &\quad (l_{\text{Б}}(q'), l_{L_\beta}(n), l_{R_\beta}(v), out) \in \mathcal{C}_\beta\} \cup \\ &\quad \{(v, \lambda, q') \mid (v, \lambda, q) \in E_{\text{Б}}, \\ &\quad (l_{\text{Б}}(v), l_{L_\beta}(q), l_{R_\beta}(q'), in) \in \mathcal{C}_\beta\} \cup \\ &\quad \{(q', \lambda, v) \mid (q, \lambda, v) \in E_{\text{Б}}, \\ &\quad (l_{\text{Б}}(v), l_{L_\beta}(q), l_{R_\beta}(q'), out) \in \mathcal{C}_\beta\} \end{aligned}$$

where, since the replaced package q is labelled with a non-terminal symbol $C \in \mathbf{B}$, it does not contain any nodes or edges (q is only used as a place holder for hierarchy transformation), and therefore q has no incoming edges. As a consequence, the

[1] Another reason why this set is empty, is that q is a leaf of the hierarchy tree, and therefore it has no outgoing edges.

third set is empty. Furthermore, the only outgoing edge of q in Б is the one linking it to n, which means that also the fourth set is empty. This edge is also the only incoming edge of n in Б, which means that the first set is empty too. We then have

$$\begin{aligned} E^{n,q+}_{\text{Б},\beta} &= \{(v,\lambda,q') \mid (n,\lambda,q') \in E_{\text{Б}}, \\ &\quad (l_{\text{Б}}(q'), l_{L_\beta}(n), l_{R_\beta}(v), out) \in \mathcal{C}_\beta\} \end{aligned}$$

Now, recall that

- $l_{L_\beta}(n) = (C, \mathbf{n})$,
- $\mathcal{C}_\beta$ only contains instructions of type (x, y, z, out) with $z^2 = \mathbf{o}$,
- if, for some node v, we have $l^2_{R_\beta}(v) = \mathbf{o}$, then (see the definition of l_{R_β}) v belongs to N_G,

and therefore

$$\begin{aligned} E^{n,q+}_{\text{Б},\beta} &= \{(v,\lambda,q') \mid (n,\lambda,q') \in E_{\text{Б}}, \exists X_3 \in \overline{\Sigma} : l_{R_\beta}(v) = (X_3, \mathbf{o}), \\ &\quad (l_{\text{Б}}(q'), (C, \mathbf{n}), (X_3, \mathbf{o}), out) \in \mathcal{C}_\beta\} \\ &= \{(v,\lambda,q') \mid (n,\lambda,q') \in E_{\text{Б}}, v \in N_G\} \end{aligned}$$

We are now ready to show that $\mathbf{HG}(H') \sim HG'$, i.e. $\mathbf{ut}(\mathbf{HG}(H'))$ is isomorphic to $\mathbf{ut}(HG')$. Recall that H' is constructed by letting $N_{H'} := N_H \setminus \{n\} \cup N_K$ and $V_{H'} =: V_H|_{(N_H \setminus (M_n \cup \{n\}))} \cup V_n \cup V_K$. as described in Definition 7.3. We compare the components of $\mathbf{HG}(H')$ and HG' one by one.

Underlying graphs

We first show that $\mathbf{ut}(G(H'))$ is isomorphic to $\mathbf{ut}(\Gamma')$ by comparing the components of $G(H')$ and Γ' one by one.

<u>*Set of nodes*</u>
It is easy to verify that

$$\begin{aligned} N_{G(H')} &= N_{H'} \\ &= N_H \setminus \{n\} \cup N_K \\ &= N_{G(H)} \setminus \{n\} \cup N_{R_\gamma} \\ &= N_\Gamma \setminus \{n\} \cup N_{R_\gamma} \\ &= N_{\Gamma'} \end{aligned}$$

Set of edges
We have the following equations

$$\begin{aligned} E_{G(H')} &= \bigcup_{m \in N_{H'}} E_{V_{H'}(m)} \\ &= \bigcup_{m \in (N_H \setminus (M_n \cup \{n\}))} E_{V_H(m)} \cup \bigcup_{m \in M_n} E_{V_n(m)} \cup \bigcup_{m \in N_K} E_{V_K(m)} \end{aligned}$$

and, by expanding the second term, we have

$$\begin{aligned} E_{G(H')} = & \bigcup_{m \in (N_H \setminus (M_n \cup \{n\}))} E_{V_H(m)} \cup \\ & \bigcup_{m \in M_n} \left(E_{V_H(m)} \setminus E^{n-}_{V_H(m)} \cup E^{n+}_{V_H(m),\pi} \cup E_G \right) \cup \\ & \bigcup_{m \in N_K} E_{V_K(m)} \end{aligned}$$

where $E^{n-}_{V_H(m)}$ indicates the set of edges of the graph $V_H(m)$ that are incident in n, and therefore need to be removed. The set

$$E^{n+}_{V_H(m),\pi} = \{(O, \lambda, u) \mid (n, \lambda, u) \in E_{V_H(m)}\} \cup \{(u, \lambda, I) \mid (u, \lambda, n) \in E_{V_H(m)}\}$$

contains the new connecting edges added to the same graph, by applying π to n. Proceeding, we get

$$\begin{aligned} E_{G(H')} = & \bigcup_{m \in (N_H \setminus (M_n \cup \{n\}))} E_{V_H(m)} \cup \bigcup_{m \in M_n} E_{V_H(m)} \setminus \bigcup_{m \in M_n} E^{n-}_{V_H(m)} \cup \\ & \bigcup_{m \in M_n} E^{n+}_{V_H(m),\pi} \cup E_G \cup \bigcup_{m \in N_K} E_{V_K(m)} \\ = & \left(\bigcup_{m \in (N_H \setminus (M_n \cup \{n\}))} E_{V_H(m)} \cup \bigcup_{m \in M_n} E_{V_H(m)} \right) \setminus \bigcup_{m \in M_n} E^{n-}_{V_H(m)} \cup \\ & \bigcup_{m \in M_n} E^{n+}_{V_H(m),\pi} \cup \left(\bigcup_{m \in N_K} E_{V_K(m)} \cup E_G \right) \\ = & \bigcup_{m \in N_H \setminus \{n\}} E_{V_H(m)} \setminus \bigcup_{m \in M_n} E^{n-}_{V_H(m)} \cup \\ & \bigcup_{m \in M_n} E^{n+}_{V_H(m),\pi} \cup \left(E_{G(K)} \cup E_G \right) \end{aligned}$$

We now observe that:

- Since n is a non-terminal node, we have $E_{V_H(n)} = \emptyset$ and therefore

$$\bigcup_{m \in N_H \setminus \{n\}} E_{V_H(m)} = \bigcup_{m \in N_H} E_{V_H(m)}$$

- The set of all edges incident in n in the various local graphs, is the set of edges incident in n in $G(H)$ and in Γ, formally:

$$\begin{aligned}
\bigcup_{m \in M_n} E^{n-}_{V_H(m)} &= \bigcup_{m \in M_n} \{e \in E_{V_H(m)} \mid n \in \mathbf{Ends}_{V_H(m)}(e)\} \\
&= \bigcup_{m \in M_n} \{e \in E_{V_H(m)} \mid n \in \mathbf{Ends}_{G(H)}(e)\} \\
&= \{e \in \bigcup_{m \in M_n} E_{V_H(m)} \mid n \in \mathbf{Ends}_{G(H)}(e)\} \\
&= \{e \in \bigcup_{m \in N_H} E_{V_H(m)} \mid n \in \mathbf{Ends}_{G(H)}(e)\} \\
&= \{e \in E_{G(H)} \mid n \in \mathbf{Ends}_{G(H)}(e)\} \\
&= \{e \in E_\Gamma \mid n \in \mathbf{Ends}_\Gamma(e)\} \\
&= E^{n-}_\Gamma
\end{aligned}$$

  where the second last equation holds because $G(H)$ and Γ can only differ for the tagging of their nodes.

- The set $\bigcup_{m \in M_n} E^{n+}_{V_H(m),\pi}$ of all new connecting edges of H' is equal to the set of new connecting edges of Γ'. Formally, we have:

$$\begin{aligned}
\bigcup_{m \in M_n} E^{n+}_{V_H(m),\pi} &= \bigcup_{m \in M_n} \{(O, \lambda, v) \mid (n, \lambda, v) \in E_{V_H(m)}\} \cup \\
&\quad \bigcup_{m \in M_n} \{(v, \lambda, I) \mid (v, \lambda, n) \in E_{V_H(m)}\} \\
&= \{(O, \lambda, v) \mid (n, \lambda, v) \in \bigcup_{m \in M_n} E_{V_H(m)}\} \cup \\
&\quad \{(v, \lambda, I) \mid (v, \lambda, n) \in \bigcup_{m \in M_n} E_{V_H(m)}\}
\end{aligned}$$

  and, since only the local graphs of nodes in M_n can contain edges incident in n,

we have

$$\begin{aligned}
\bigcup_{m\in M_n} E^{n+}_{V_H(m),\pi} &= \{(O,\lambda,v) \mid (n,\lambda,v) \in \bigcup_{m\in N_H} E_{V_H(m)}\} \cup \\
&\quad \{(v,\lambda,I) \mid (v,\lambda,n) \in \bigcup_{m\in N_H} E_{V_H(m)}\} \\
&= \{(O,\lambda,v) \mid (n,\lambda,v) \in E_{G(H)}\} \cup \\
&\quad \{(v,\lambda,I) \mid (v,\lambda,n) \in E_{G(H)}\} \\
&= \{(O,\lambda,v) \mid (n,\lambda,v) \in E_\Gamma\} \cup \\
&\quad \{(v,\lambda,I) \mid (v,\lambda,n) \in E_\Gamma\} \\
&= E^{n+}_{\Gamma,\gamma}
\end{aligned}$$

where the last equation holds in view of observation b on page 205.

We can now go back to our main line of reasoning, obtaining:

$$\begin{aligned}
E_{G(H')} &= \bigcup_{m\in N_H} E_{V_H(m)} \setminus E^{n-}_\Gamma \cup E^{n+}_{\Gamma,\gamma} \cup E_{R_\gamma} \\
&= E_{G(H)} \setminus E^{n-}_\Gamma \cup E^{n+}_{\Gamma,\gamma} \cup E_{R_\gamma} \\
&= E_\Gamma \setminus E^{n-}_\Gamma \cup E^{n+}_{\Gamma,\gamma} \cup E_{R_\gamma} \\
&= E_{\Gamma'}
\end{aligned}$$

Node labels

It is clear that $l_{\Gamma'}|_{N_{G(H)}\setminus\{n\}} = l_{G(H')}|_{N_{G(H)}\setminus\{n\}}$. If $u \in N_{R_\gamma}$ (recall that $N_{R_\gamma} \subseteq N_{G(H)}$ and $N_{R_\gamma} \subseteq N_\Gamma$), then $l^1_{G(H')}(u) = V_K(u) = l^1_{R_\gamma}(u) = l^1_{\Gamma'}(u)$, and then $l_{\mathbf{ut}(G(H'))}(u) = (l_{G(H')}(u), \mathbf{n}) = (l_{\Gamma'}(u), \mathbf{n}) = l_{\mathbf{ut}(\Gamma')}(u)$.

Hierarchy graphs

As a second step, we want to show that $\mathbf{ut}(Д')$ is isomorphic $\mathbf{ut}(D(H'))$.

Set of nodes

It is easy to verify that

$$\begin{aligned}
N_{D(H')} &= \{p_u \mid u \in N_{H'}, V_{H'}(u) \notin \mathbf{A}\} \\
&= \{p_u \mid u \in N_H \setminus \{n\}, V_{H'}(u) \notin \mathbf{A}\} \cup \{p_u \mid u \in N_K, V_{H'}(u) \notin \mathbf{A}\} \\
&= \{p_u \mid u \in N_H, V_{H'}(u) \notin \mathbf{A}\} \setminus \{p_n\} \cup \{p_u \mid u \in N_K, V_K(u) \notin \mathbf{A}\} \\
&= \{p_u \mid u \in N_H, V_H(u) \notin \mathbf{A}\} \setminus \{p_n\} \cup N_{D(K)} \\
&= N_{D(H)} \setminus \{q\} \cup N_{R_\delta} \\
&= N_Д \setminus \{q\} \cup N_{R_\delta} \\
&= N_{Д'}
\end{aligned}$$

Set of edges
We have the following equations

$$\begin{aligned}
E_{D(H')} &= \{(p_u,(\bot,\mathbf{e}),p_v) \mid u,v \in N_{H'} : \\
&\qquad V_{H'}(u), V_{H'}(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_{H'}(u)}\} \\
&= \{(p_u,(\bot,\mathbf{e}),p_v) \mid u,v \in N_H \setminus \{n\} : \\
&\qquad V_{H'}(u), V_{H'}(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_{H'}(u)}\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u \in N_K, v \in (N_H \setminus \{n\}) : \\
&\qquad V_{H'}(u), V_{H'}(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_{H'}(u)}\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u \in (N_H \setminus \{n\}), v \in N_K : \\
&\qquad V_{H'}(u), V_{H'}(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_{H'}(u)}\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u,v \in N_K : \\
&\qquad V_{H'}(u), V_{H'}(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_{H'}(u)}\}
\end{aligned}$$

where the second set is empty, since no new package p_u, $u \in N_K$ can be the parent of an old package p_v, $v \in N_{H'}$. In the third set we must have all edges $(p,(\bot,\mathbf{e}),p')$ such that p contained n and $p' \in N_G$, therefore we have

$$\begin{aligned}
E_{D(H')} &= \{(p_u,(\bot,\mathbf{e}),p_v) \mid u,v \in N_H \setminus \{n\} : \\
&\qquad V_H(u), V_H(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_H(u)}\} \cup \\
&\quad \emptyset \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u \in (N_H \setminus \{n\}), v \in N_G : \\
&\qquad n \in N_{V_H(u)}\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u,v \in N_K : \\
&\qquad V_K(u), V_K(v) \notin (\mathbf{A} \cup \mathbf{B}), v \in N_{V_K(u)}\} \\
&= E_{D(H)} \setminus E^{p_n-}_{D(H)} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u \in (N_H \setminus \{n\}), v \in N_G : n \in N_{V_H(u)}\} \cup \\
&\quad E_{D(K)} \\
&= E_{D(H)} \setminus E^{q-}_{D(H)} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid u \in (N_H \setminus \{n\}), v \in N_G, \\
&\qquad (p_u,(\bot,\mathbf{e}),p_n) \in E_H\} \cup \\
&\quad E_{D(K)} \\
&= E_{D(H)} \setminus E^{q-}_{D(H)} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),p_v) \mid (p_u,(\bot,\mathbf{e}),q) \in E_H, \\
&\qquad u \in (N_H \setminus \{n\}), v \in N_G\} \cup \\
&\quad E_{D(K)} \\
&= E_{D(H)} \setminus E^{q-}_{D(H)} \cup E^{q+}_{Д,\delta} \cup E_{D(K)}
\end{aligned}$$

$$\begin{aligned} &= E_{Д} \setminus E^{q-}_{Д} \cup E^{q+}_{Д,\delta} \cup E_{R_\delta} \\ &= E_{Д'} \end{aligned}$$

Node labels
For all $p_u \in N_{D(H)} \setminus \{q\} = N_{Д} \setminus \{q\} = \{p_u \mid u \in N_H \setminus \{n\}\}$, we have $l^1_{D(H')}(u) = V_{H'}(u) = V_H(u) = l^1_{\Gamma}(u) = l^1_{\Gamma'}(u)$.

For $p_u \in N_{R_\delta} = N_K$, then $l^1_{D(H')}(p_u) = V_K(u) = l^1_{Д'}(p)$.

Coupling graphs

Finally, we show that $\mathbf{ut}(Б')$ is isomorphic $\mathbf{ut}(B(H'))$.

Set of nodes
It is easy to verify that

$$\begin{aligned} N_{B(H')} &= A_{B(H')} \cup P_{B(H')} \\ &= N_{G(H')} \cup P_{D(H')} \\ &= N_{\Gamma'} \cup P_{Д'} \\ &= N_{Б'} \end{aligned}$$

Set of edges
We have the following equations

$$\begin{aligned} E_{B(H')} &= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in A_{B(H')}, p_v \in P_{B(H')}, u \in V_{H'}(v)\} \cup \\ &\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{G(H')}, V_{H'}(u) \notin \mathbf{A}\} \\ &= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(H')}, p_v \in P_{D(H')}, u \in V_{H'}(v)\} \cup \\ &\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{G(H)} \setminus \{n\}, V_{H'}(u) \notin \mathbf{A}\} \cup \\ &\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_K, V_{H'}(u) \notin \mathbf{A}\} \\ &= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(H)} \setminus \{n\}, p_v \in P_{D(H)} \setminus \{p_n\}, \\ &\qquad\qquad u \in V_{H'}(v)\} \cup \\ &\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(H)} \setminus \{n\}, p_v \in P_{D(K)}, u \in V_{H'}(v)\} \cup \\ &\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, p_v \in P_{D(H)} \setminus \{p_n\}, u \in V_{H'}(v)\} \cup \\ &\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, p_v \in P_{D(K)}, u \in V_{H'}(v)\} \cup \\ &\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{G(H)} \setminus \{n\}, V_H(u) \notin \mathbf{A}\} \cup \\ &\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_K, V_K(u) \notin \mathbf{A}\} \end{aligned}$$

and, by reordering the components of the union, we obtain

$$\begin{aligned}
E_{B(H')} &= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(H)} \setminus \{n\}, p_v \in P_{D(H)} \setminus \{p_n\}, \\
&\qquad\qquad u \in V_{H'}(v)\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{G(H)} \setminus \{n\}, V_H(u) \notin \mathbf{A}\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(H)} \setminus \{n\}, p_v \in P_{D(K)}, u \in V_{H'}(v)\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, p_v \in P_{D(H)} \setminus \{p_n\}, u \in V_{H'}(v)\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, p_v \in P_{D(K)}, u \in V_{H'}(v)\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_K, V_K(u) \notin \mathbf{A}\}
\end{aligned}$$

where the third set is empty because there are no nodes from the old graph H that are contained in packages that are inserted by replacing $q = p_n$. Proceeding, we obtain

$$\begin{aligned}
E_{B(H')} &= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{B(H)} \setminus \{n\}, p_v \in P_{B(H)} \setminus \{p_n\}, \\
&\qquad\qquad u \in V_H(v)\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{B(H)} \setminus \{n\}, V_H(u) \notin \mathbf{A}\} \cup \\
&\quad \emptyset \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, v \in M_n\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{G(K)}, p_v \in P_{D(K)}, u \in V_K(v)\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_K, V_K(u) \notin \mathbf{A}\} \\
&= \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{B(H)} \setminus \{n\}, p_v \in P_{B(H)} \setminus \{p_n\}\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_{B(H)} \setminus \{n\}\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_G, v \in M_n\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid u \in N_{B(K)}, p_v \in P_{B(K)}, u \in V_K(v)\} \cup \\
&\quad \{(p_u,(\bot,\mathbf{e}),u) \mid u \in N_K, V_K(u) \notin \mathbf{A}\} \\
&= E_{B(H)} \setminus \\
&\quad \{e \in E_{B(H)} \mid n \in \mathbf{Ends}_{B(H)}(e) \vee p_n = q \in \mathbf{Ends}_{B(H)}(e)\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid (n,(\bot,\mathbf{e}),p_v) \in E_{B(H)}, u \in N_G\} \cup \\
&\quad E_{B(K)}
\end{aligned}$$

where the edges from the removed node n to p_v are in $E_{B(H)} = E_{\text{Б}}$ since $v \in M_n$, i.e. v contained n in H. In view of observation d on page 206, we get

$$\begin{aligned}
E_{B(H')} &= E_{\text{Б}} \setminus \{e \in E_{\text{Б}} \mid n \in \mathbf{Ends}_{\text{Б}}(e) \vee p_n = q \in \mathbf{Ends}_{\text{Б}}(e)\} \cup \\
&\quad \{(u,(\bot,\mathbf{e}),p_v) \mid (n,(\bot,\mathbf{e}),p_v) \in E_{\text{Б}}, u \in N_G\} \cup \\
&\quad E_{R_\beta} \\
&= E_{\text{Б}} \setminus E_{\text{Б}}^{n,q-} \cup E_{\text{Б},\beta}^{n,q+} \cup E_{R_\beta} \\
&= E_{\text{Б}'}
\end{aligned}$$

Node labels

For all $u \in N_\Gamma \setminus \{n\}$, we have $l^1_{\text{Б}'}(u) = l^1_{\text{Б}}(u) = l^1_\Gamma(u) = l^1_{G(H)}(u) = l^1_{G(H')}(u) = l^1_{B(H')}(u)$.

For all $p \in N_\text{Д} \setminus \{p_n\}$, we have $l^1_{\text{Б}'}(p) = l^1_{\text{Б}}(p) = l^1_\text{Д}(p) = l^1_{D(H)}(p) = l^1_{D(H')}(p) = l^1_{B(H')}(p)$.

For all $u \in N_K$, we have $l^1_{\text{Б}'}(u) = l^1_{R_\beta}(u) = l^1_{B(K)}(u) = l^1_{B(H')}(u)$, where the last equality can be obtained by looking at the definition of the graphs $B(K)$ and $B(H')$ and recalling that H' contains K. Similarly, for all $u \in N_K$, we have $l^1_{\text{Б}'}(p_u) = l^1_{R_\beta}(p_u) = l^1_{B(K)}(u) = l^1_{B(H')}(u)$.

By proving that $\mathbf{ut}(G(H')) = \mathbf{ut}(\Gamma')$, $\mathbf{ut}(D(H')) = \mathbf{ut}(\text{Д}')$, and $\mathbf{ut}(B(H')) = \mathbf{ut}(\text{Б}')$ we have obtained our thesis. □

Bibliography

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, April 1999.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Java Series. Addison-Wesley, 1996.

[BBH94] W. Bachl, F.-J. Brandenburg, and T. Hickl. Hierarchical graph design using HiGraD. Technical report, Passau, 1994.

[BEMW00] Giorgio Busatto, Gregor Engels, Katharina Mehner, and Annika Wagner. A framework for adding packages to graph transformation systems. In Ehrig et al. [EEKR00], pages 352–367.

[Ber95] F. Bertault. Adocs: a drawing system for generic combinatorial structures. *Lecture Notes in Computer Science*, 1027:24–27, 1995.

[BH01] Giorgio Busatto and Berthold Hoffmann. Comparing notions of hierarchical graph transformation. In Taentzer et al. [TBP01], pages 312–319.

[BJ96] Michel Bauderon and Hélène Jacquet. Node rewriting in graph and hypergraphs: A categorical framework. Technical Report 1134-96, 1996.

[BKK01] Giorgio Busatto, Hans-Jörg Kreowski, and Sabine Kuske. Abstract hierarchical graph transformation. Technical Report 1, University of Bremen, Bremen, Germany, 2001.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings, 1993.

[Bra96] Franz-Josef Brandenburg, editor. *Graph Drawing, Symposium on Graph Drawing, GD'95*. Number 1027 in *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 20-22 September 1996.

[BRS92] R. A. Botafogo, E. Rivlin, and B. Shneiderman. Structural analysis of hypertext: identifying hierarchies and useful metrics. *ACM Transactions on Information Systems*, 10:142–180, Apr 1992.

[BtH00] Giorgio Busatto and Pieter Jan 't Hoen. A graph-grammar based approach for the specification of hypermedia application dynamics. In Corradini and Heckel [CH00], pages 403–409.

[CEER96] Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors. *Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*. Springer, 1996.

[CEMP96] Andrea Corradini, Hartmut Ehrig, Ugo Montanari, and Julia Padberg. The category of typed graph grammars and its adjunction with categories of derivations. In Cuny et al. [CEER96], pages 56–74.

[CH00] A. Corradini and R. Heckel, editors. *Workshop on Graph Transformation and Visual Modelling, Satellite of ICALP'2000*. Carleton Scientific, 2000.

[Che76] P. Chen. Database design based on entity and relationship. In S. Bing Yao, editor, *Principles of Database Design*, Englewood Cliffs, NJ, 1976. Prentice-Hall.

[CM93] Mariano P. Consens and Alberto O. Mendelzon. Hy$^+$: A Hygraph-based query and visualization system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):511–516, June 1993.

[CM95] Andrea Corradini and Ugo Montanari, editors. *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[CMR$^+$97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In Rozenberg [Roz97], chapter 3, pages 163–246.

[Con94] Mariano P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*. PhD thesis, Department of Computer Science, University of Toronto (Canada), 1994.

[Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [Roz97], chapter 5, pages 313–400.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [Roz97], chapter 2, pages 95–162.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, to appear, 2002.

[DK99] Frank Drewes and Hans-Jörg Kreowski. Picture generation by collage grammars. In Ehrig et al. [RE+99a], chapter 11, pages 397–458. With CD-ROM including a lot of Graph Transformation Software.

[DL98] O. M. F. De Troyer and C. J. Leune. WSDM: a user centered design method for Web sites. *Computer Networks and ISDN Systems*, 30(1–7):85–94, apr 1998.

[EEKR00] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*, number 1764 in *Lecture Notes in Computer Science*. Springer, 2000.

[EH00] G. Engels and R. Heckel. Graph transformation as unifying formal framework for system modeling and model evolution. In Welzl et al. [WMR00], pages 127–150.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In Rozenberg [Roz97], chapter 4, pages 247–312.

[EPS73] Hartmut Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conf. on Automata and Switching Theory*, pages 167–180, Iowa City, 1973.

[ER97] Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [Roz97], chapter 1, pages 1–94.

[ERT99] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In Ehrig et al. [RE+99a], chapter 13, pages 550–604. With CD-ROM including a lot of Graph Transformation Software.

[ES95] Gregor Engels and Andy Schürr. Encapsulated hierachical graphs, graph types, and meta types. In Corradini and Montanari [CM95].

[GJRT84] H.J. Genrich, D. Janssens, G. Rozenberg, and P.S. Thiagarajan. Generalized handle grammars and their relation to Petri Nets. *Elektronische Informations-Verarbeitung und Kybernetik*, 20(4):179–206, 1984.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Java Series. Addison-Wesley, 1996.

[GM00] Felix H. Gatzenmeier and Oliver Meyer. Improving the publication chain through high-level authoring support. In Nagl et al. [NSM00].

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*, pages 674–681. Addison-Wesley, 1983.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.

[Har88] David Harel. On visual formalisms. *Communications of the Association for Computing Machinery*, 31(5):514–530, 1988.

[HBvR94] Lynda Hardman, Dick Bulterman, and Guido van Rossum. The Amsterdam hypermedia model: adding time and context to the Dexter model. *Communications of the ACM*, 37(2):50–62, February 1994.

[Him93] Michael Himsolt. *Konzeption und Implementierung von Grapheditoren*. PhD thesis, Universität Passau, 1993.

[Him94] Michael Himsolt. Hierarchical graphs for graph grammars. In *Preproceedings of the Fifth International Workshop on Graph Grammars and Their Application to Computer Science*, 1994.

[HKP91] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. *Fundamenta Informaticae*, XV:37–60, 1991.

[HMP01] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout approach with injective matching. *Mathematical Structures in Computer Science*, pages 637–698, 2001.

[HP96] Annegret Habel and Detlef Plump. Graph unification and matching. In Cuny et al. [CEER96], pages 75–88.

[HS94] Frank Halasz and Mayer Schwartz. The Dexter hypertext reference model. *Communications of the Association for Computer Machinery*, 37(2):30–39, February 1994.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings WCRE'00*, nov 2000.

[ISB95] Tomás Isakowitz, Edward A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, August 1995.

[Jan83] Dirk Janssens. *Node Label Controlled Graph Grammars*. PhD thesis, Antwerp, 1983.

[JBR98] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison Wesley, 1998.

[JR82] Dirk Janssens and Grzegorz Rozenberg. Graph grammars with neighbourhood-controlled embedding. *Theoretical Computer Science*, 21:55–74, 1982.

[KA90] Setrag Khoshafian and Razmik Abnous. *Object Orientation, Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, 1990.

[Kel76] R. Keller. Formal verification of parallel programs. *Communications of the ACM*, 7:371–384, 1976.

[KK96] Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units—A step into GRACE. In Cuny et al. [CEER96], pages 89–106.

[KK99a] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [RE+99a], chapter 15, pages 607–638. With CD-ROM including a lot of Graph Transformation Software.

[KK99b] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[KKS97] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal of Software Engineering and Knowledge Engineering*, 7:479–502, 1997.

[KR90] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars, parts I and II. *Information Sciences*, 52:185–210 and 221–246, 1990.

[Kra99] Jan Kratochvil, editor. *Graph Drawing, 7th International Symposium, GD'99*. Number 1731 in LNCS, LNCS. Springer-Verlag, Berlin, Germany, September 1999.

[Kus99] Sabine Kuske. *Transformation Units – A Structuring Principle for Graph Transformation Systems*. PhD thesis, Fachbereich 3 (Mathematik & Informatik) der Universität Bremen, 1999.

[LB93] Michael Löwe and Martin Beyer. AGG — an implementation of algebraic graph rewriting. In *Rewriting Techniques and Applications*, volume 690 of *Lecture Notes in Computer Science*, pages 451–456. Springer-Verlag, 1993.

[Mac71] Saunders MacLane. *Categories for Working Mathematicians*. Springer-Verlag, New York, 1971.

[Mau96] Hermann Maurer. *Hyperwave, The Next Generation Web Solutions*. Addison Wesley Longman, Edinburgh gate, Harlow, Essex, England, 1996.

[Meh84] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer Verlag, 1984.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[MMPH96] B. Madden, P. Madden, S. Powers, and M. Himsolt. Portable graph layout and editing. *Lecture Notes in Computer Science*, 1027:385–395, 1996.

[MSHR98] Hermann Maurer, Nick Scherbakov, Zahran Halim, and Zaidah Razak. *From Databases to Hypermedia*. Springer, Berlin, 1998.

[NS95] Inc. NeXT Software. *Object Oriented Programming and the Objective-C Language*. 1995.

[NSM00] Manfred Nagl, Andreas Schürr, and Manfred Münch, editors. *Applications of Graph Transformations with Industrial Relevance*, number 1779 in *Lecture Notes in Computer Science*. Springer, 2000.

[NZ00] Jorg Niere and Albert Zundorf. Testing and simulating production control systems using the fujaba environment. In Nagl et al. [NSM00], pages 449–456.

[PL94] Alexandra Poulovassilis and Mark Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, 12(1):35–68, January 1994.

[PP95] Francesco Parisi-Presicce and Gabriele Piersanti. Multilevel graph grammars. In W. Mayr, Ernst, Gunter Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 51–64, 1995.

[Pra79] Terrence W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 389–400, 1979.

[Pra83] Terrence W. Pratt. Formal specification of software using h-graph semantics. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 314–332, 1983.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object Modelling and Design*. Prentice Hall, 1991.

[RE+99a] G. Rozenberg, H. Ehrig, et al., editors. *Handbook on Graph Grammars and Computing by Graph Transformation 2 (Specifications and Programming)*. World Scientific, Singapore, 1999. With CD-ROM including a lot of Graph Transformation Software.

[RE+99b] G. Rozenberg, H. Ehrig, et al., editors. *Handbook on Graph Grammars and Computing by Graph Transformation 3 (Concurrency)*. World Scientific, Singapore, 1999.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[Roz97] G. Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations)*. World Scientific, Singapore, 1997.

[RS92] G. Rozenberg and A. Salomaa. *Lindenmayer Systems*. Springer, Berlin, 1992.

[Rud97] Michael Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Master's thesis, Fachbereich Informatik, Technische Universität Berlin, December 1997.

[SB94] Daniel Schwabe and Simone D. J. Barbosa. Navigation modelling in hypermedia applications. Technical Report MCC 42/94, 1994.

[Sch97] Andy Schürr. Programmed graph replacement systems. In Rozenberg [Roz97], chapter 7, pages 479–546.

[SDW95] P. Senac, P. De Saqui-Sannes, and R. Willrich. Hierarchical time stream Petri net: A model for hypermedia systems. *Lecture Notes in Computer Science*, 935:451–470, 1995.

[SR95] Daniel Schwabe and Gustavo Rossi. The object-oriented hypermedia design model. *Communications of the ACM*, 38(8):45–46, August 1995.

[SRB96] D. Schwabe, G. Rossi, and S. D. J. Barbosa. Systematic hypermedia application design with OOHDM. In *Hypertext '96, Washington, DC, March 16–20, 1996: the Seventh ACM Conference on Hypertext: Proceedings*, pages 116–128, New York, NY, USA, 1996. ACM Press.

[Str91] Bjarne Stroustrup. *The C++ Programming Language: Second Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1991.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [RE+99a], chapter 13, pages 487–550. With CD-ROM including a lot of Graph Transformation Software.

[Tae96] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996. Shaker Verlag.

[TBP01] Gabriele Taentzer, Luciano Baresi, and Mauro Pezzè, editors. *Workshop on Graph Transformation and Visual Modelling, Satellite of ICALP'2001.* Elsevier, 2001.

[TT94] Roberto Tamassia and Ioannis G. Tollis, editors. *Proc. DIMACS Int. Work. Graph Drawing, GD'94*, number 894 in *Lecture Notes in Computer Science*, Berlin, Germany, 10–12 October 1994. Springer-Verlag.

[Wat90] David A. Watt. *Programming Languages Concepts and Paradigms.* International Series in Computing. Prentice Hall, Hemel Hempstaed, 1990.

[Whi98] Sue Whitesides, editor. *Graph Drawing,* 6^{th} *International Symposium, GD'98.* Number 1547 in *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Germany, August 1998.

[Wir85] Niklaus Wirth. *Programming in Modula2.* Springer, 1985.

[WMR00] Emo Welzl, Ugo Montanari, and Jose D. P. Rolim, editors. *Automata, languages and programming: 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9–15, 2000: proceedings*, volume 1853 of *Lecture Notes in Computer Science*, New York, NY, USA, 2000. Springer-Verlag Inc.

Index

E

F

G

H

I

S

T

U

V

W