Parallel Composition and Unfolding Semantics of Graph Grammars

vorgelegt von
Master in Science (M.Sc.)
Leila Ribeiro
aus Porto Alegre (Brasilien)

Vom Fachbereich 13 — Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades eines

Doktorin der Ingenieurswissenschaften
— Dr.-Ing. —

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr. Stefan Jähnichen, Technische Universität Berlin
Berichter:	Prof. Dr. Hartmut Ehrig, Technische Universität Berlin
Berichter:	Prof. Dr. Annegret Habel, Universität Hildesheim
Zusätzlicher Gutachter:	Dr. Andrea Corradini, Universtät Pisa

Tag der wissenschaftlichen Aussprache: 14. Juni 1996

Berlin 1996
D 83

Parallel Composition and Unfolding Semantics of Graph Grammars

vorgelegt von
Master in Science (M.Sc.)
Leila Ribeiro
aus Porto Alegre (Brasilien)

Vom Fachbereich 13 — Informatik —
der Technischen Universität Berlin
zur Erlangung des akademischen Grades eines
Doktorin der Ingenieurswissenschaften
— Dr.-Ing. —
genehmigte Dissertation

Promotionsausschuß:

Vorsitzender:	Prof. Dr. Stefan Jähnichen, Technische Universität Berlin
Berichter:	Prof. Dr. Hartmut Ehrig, Technische Universität Berlin
Berichter:	Prof. Dr. Annegret Habel, Universität Hildesheim
Zusätzlicher Gutachter:	Dr. Andrea Corradini, Universtät Pisa

Tag der wissenschaftlichen Aussprache: 14. Juni 1996

Berlin 1996
D 83

To My Family

Abstract

The main aims of this thesis are to provide an approach to the parallel composition of graph grammars and a semantics for graph grammars, called the unfolding semantics, in which the aspects of concurrency and compositionality with respect to the parallel composition play a central role.

The parallel composition of graph grammar allows the composition of grammars with respect to a shared part (that may be empty), and is based on parallel and amalgamated composition of the rules of the component grammars. Moreover, the result of the composition is suitably syntactically and semantically related to the component grammars.

The unfolding semantics of a graph grammar is a true concurrent, branching structure semantics in which states (graphs) as well as changes of states (derivations) are represented. The unfolding can be constructed incrementally, and we show that this yields the same result as a construction based on gluing of the deterministic computations of a grammar. Moreover, the unfolding of a graph grammar is itself a graph grammar that belong to a special class of graph grammars: the occurrence graph grammars. Here this class is defined axiomatically, and the members of this class can be seen as grammars that represent (deterministic and non-deterministic) computations of another grammars.

The semantics of a grammar obtained as the parallel composition of other grammars is isomorphic to the composition of the semantics of the component grammars. As the purpose of the parallel composition is to be a composition for concurrent and reactive systems, the fact that this composition is compatible with a true concurrency semantics is an attractive result.

Zusammenfassung

Das Hauptziel dieser Arbeit ist es, einen Ansatz für die parallele Komposition von Graph-Grammatiken und eine Unfolding-Semantik genannte Semantik für Graph-Grammatiken bereitzustellen, in der die Aspekte Nebenläufigkeit und Kompositionalität bzgl. der parallelen Komposition eine zentrale Rolle einnehmen.

Die parallele Komposition von Graph-Grammatiken erlaubt die Komposition von Grammatiken bzgl. eines gemeinsamen (möglicherweise leeren) Anteils und basiert auf der parallelen und amalgamierten Komposition von Regeln der komponierten Grammtiken. Darüber hinaus ist das Kompositionsergebnis syntaktisch und semantisch in geeigneter Weise mit den komponierten Grammatiken verknüpft.

Die Unfolding-Semantik einer Graph-Grammatik ist eine echt nebenläufige, verzweigende Semantik, in der sowohl Zustände (Graphen) als auch Zustandsänderungen (Ableitungen) repräsentiert sind. Das Unfolding kann inkrementell konstruiert werden und es wird gezeigt, daß dies das gleiche Result liefert wie die Verklebung der deterministischen Berechnungen einer Grammatik Darüberhinaus ist das Unfolding einer Graph-Grammatik selbst eine Graph-Grammatik, die einer speziellen Klasse von Graph-Grammatiken angehört: den Occurrence-Grammatiken. Hier wird diese Klasse axiomatisch definiert und die Elemente dieser Klasse können als Grammatiken gesehen werden, die (deterministische und nicht-deterministische) Berechnungen einer anderen Grammatik repräsentieren.

Die Semantik einer Grammatik, die aus der parallelen Komposition anderer Grammatiken entstanden ist, ist isomorph zur Komposition der Semantiken der komponierten Grammatiken. Dieses Kompatibilitätsresultat verbindet die parallele Komposition und die Unfolding Semantik in enger Weise. Da der Zweck der parallelen Komposition die Komposition nebenläufiger Systeme ist, stellt die Kompatibilität von Komposition und Nebenläufigkeitssemantik ein attraktives Ergebnis dar.

Foreword

Originally, my doctoral thesis was supposed to be on the area of Petri nets. Therefore, in the first years of my stay in Berlin I concentrated my studies on this area, more precisely on *algebraic high-level nets* [EPR94, PER95]. My interest in issues related to object-orientation made me take a closer look into graph grammars, where references (that are a very important subject for object-orientation) can be represented very naturally. The participation in various COMPUGRAPH workshops, in which I could learn a lot about the different approaches and application areas of graph grammars, and the discussions with Andrea Corradini and Martin Korff influenced my growing interest for this very exiting field. My first works on graph grammars were on the topic of relating graph grammars and Petri nets. These works (and also the many talks from Andrea Corradini on this topic) made clear to me that graph grammars can be seen as a generalization of Petri nets. However, the relationship between Petri nets and graph grammars with respect to practical use was not yet investigated deeply. Some efforts towards this relation were done within the project GRAPHIT (German/Brazilian Cooperation supported by CNPq and GMD), in which specifications of the same telephone system using graph grammars and Petri nets were developed and discussed with members of the Brazilian company Nutec. It turned out, within other results, that graph grammars are very useful to describe complex states in an natural way, whereas Petri nets are able to describe relationships between actions (transitions) more explicitly. The GRAPHIT project was of great importance for the development of the theory presented in this thesis because the concepts defined here were applied in the case study about the telephone system, and then I got a lot of feedback and new ideas from Nutec (specially from Paulo Castro) about using parallel composition for the specification of practical systems, and about the topics of research that still are needed and interesting from the industrial point of view.

Acknowledgments:

First of all, I would like to thank my referees:

Prof. Dr. Hartmut Ehrig (TU Berlin) not only gave me the chance to do my Ph.D. studies in his group in Berlin, but also supported me during my stay there. I am also thankful to his family, that was always so friendly to me.

Prof. Dr. Annegret Habel (University of Hildesheim) read this thesis very carefully, in spite of the considerably short period of time that she had to write the referee report, and

gave many valuable comments (and long lists of “open questions”) that helped a lot to improve previous versions of this thesis.

Dr. Andrea Corradini (University of Pisa) is an expert for semantics of concurrency, and therefore he was invited to be an external referee of my thesis. He was also one of the persons that influenced me to make research on the area of graph grammars. The discussions with Andrea about concurrency, Petri nets and graph grammars were always very fruitful and helped me to have a deeper understanding of these areas. Moreover, he also read and discussed with me previous versions of this thesis, and wrote a referee report in record time (as an external referee, his report had to be ready before the others).

Not only the referees were engaged in the (sometimes hard) job of reading and commenting my thesis. I am very thankful to *Dr. Martin Korff*, for reading and discussing with me (many times via oversee calls) about a big part of the contents of this thesis. *Reiko Heckel* had also read many parts of my thesis, mainly concerned with parallel composition of graph grammars.

Within the project GRAPHIT, *Paulo Castro*, from the company Nutec, gave interesting comments and ideas for the specification of practical problems using parallel composition of graph grammars.

There is one more person that contributed, although not directly, to this thesis: *Prof. Dr. Daltro José Nunes* (Federal University of Rio Grande do Sul, Brazil). He always supported me since the beginning of my studies in the university, and gave me valuable advices concerning my academical life (not only about technical but also about organizational issues).

There are a lot of persons here in Germany that made my stay here become a very nice experience to me. From the university, the *TFS-group* was always very friendly to me, in particular *Rosi Bardohl, Gabi Taentzer, Reiko Heckel, Alfio Martiti* and *Frau Barnewitz.* The *Korff family* had contributed a lot in making Germany a second home for me. In Italy, the *Corradini family* became very good friends of mine.

Last but not least, I would like to thank *my family* in Brazil, that supported me so much during these years in Berlin (and also during the years before). To express these thanks, I would like to switch to Portuguese... Muito obrigado por todo o apoio que vocês me deram sempre, apesar de muitas vezes a distância ter atrapalhado “um pouco”. Esta tese é dedicada a vocês que, mesmo estando longe, participaram ativamente de todas as etapas de sua elaboração e sempre torceram muito para o sucesso de meu Doutorado.

Contents

1

Introduction

In the last years, concurrent systems have gained more and more importance. This kind of systems usually consists of several autonomous components that run in parallel and interact with each other (for example, via messages). Although there are already many concurrent systems implemented and working, there is still a great amount of fear that there may be undiscovered bugs within each of these systems, that probably will only be discovered in critical situations. The complexity of a concurrent system is much bigger than the complexity of a sequential system because the interactions of independent components affects the behaviour of the whole system, such that it is not enough to know that each component works as expected to know that the whole system works as expected, but we also have to know how each component reacts to outside influences and how each component influences its outside while it is running. Many systems that work "correctly" when they run as stand-alone systems may generate many unexpected and unwanted results when combined with others. Therefore, the existence of formal methods that allow to prove that the system really works as expected is even more important than for the sequential case. Moreover, as concurrent systems are usually composed of smaller systems that work cooperatively together to reach some goal, ways of composing a concurrent system from smaller components are needed. This composition shall obviously be not only a syntactical way of composing systems, but shall assure that the system generated from the components by applying this composition operator does not behave in an unexpected (or unspecified) way.

Graph grammars [Ehr79] have originated from Chomsky grammars by substituting the replacement of strings through the replacement of graphs. Very soon it was noticed that graph grammars are very well suited to the specification of concurrent systems: a state of the system is represented by a graph in which different rules may be applied at the same time. To reach consistent results, not every set of rules may be applied at the same time to the graph representing the state. A great part of the theory of algebraic graph grammars deals with finding conditions under which the concurrent application of a set of rules leads to the same results as corresponding sequential applications of the same rules. By using different kinds of graphical representation for vertices and edges for different kinds of objects of a system, a graph becomes even more a powerful and expressive means of describing complex states. By representing states via graphs and changes of states via rules (whose left- and right-hand sides are graphs and are connected in some compatible way) one can achieve a quite good and understandable description of a complex system. Therefore, graph grammars seem to be a promising formalism for the description of concurrent and reactive systems.

Recently, graph grammars have started being used for the specification of bigger systems. This raised the question about compatible syntactical and semantical composition operators for graph grammars. Classically, a graph grammar is defined as a whole and not obtained by its component grammars. Correspondingly, the usual semantics of graph grammars (based on the application of rules) does not rely on the semantics of the possible "subgrammars" of a grammar, although the application of rules has a local character. In the last years, first attempts have been made towards a compositional semantics for graph grammars.Indeed, compositional semantics for graph transformation systems (graph grammars without initial graph) have been achieved in the framework of Double-Pushout graph grammars in [CH95, HCEL96] using an interleaving semantics that is compatible with a union operator, and in the framework of ESM (Extended Structure Morphism) systems in [Jan93, Jan96] using a process semantics that is compatible with the union of ESM systems. These two approaches are based on graph transformation systems, that is, graph grammars without an initial graph. The ability to specify an initial state for a system may have advantages for some applications. For specification purposes, the initial state has the role of restricting the computations that are allowed in the system and also the reachable states. Therefore, the specification of a initial state has a considerable effect on the language semantics of a graph grammar (reachable graphs) – because without an initial graph the language semantics would always consist of all graphs – and thus also on the analysis properties of a grammar, like deadlock, mutual exclusion, reachability and liveness properties. The same set of rules, with different initial states may exhibit very different behaviours.

> The main aims of this thesis are to provide an approach to the parallel composition of graph grammars and a semantics for graph grammars, called the unfolding semantics, in which the aspects of concurrency and compositionality with respect to the parallel composition play a central role.

According to this general aim, syntactical operators to compose grammars will be introduced, as well as a semantical model that is particularly well-suited for concurrent systems, and it will be shown that this semantics, called unfolding semantics, is compositional with respect to the syntactical composition operators. More concretely, the following results are achieved in this thesis:

Parallel Composition : A new concept of *parallel composition* of graph grammars is presented. The characteristics of this composition are:

- The *initial (start) graph* is taken into account.
- The composition of two grammars can be based on a shared part (*cooperative parallel composition*), or be a composition without any shared parts (*pure parallel composition*).
- The composition is based on *specialization morphisms*. These morphisms express the fact that both components to be composed are specializations of the shared parts.
- The result of the composition is suitably syntactically and semantically related to the component grammars.

Unfolding Semantics : A new semantical model, called *unfolding semantics*, for graph grammars in which the aspects of *concurrency* and *compositionality* play a central role is presented. The main characteristics of this semantics are:

- It is a *true concurrency* semantics.
- It is a *branching-structure* semantics: non-determinism is represented explicitly.
- It is *abstract:* results of derivations are described up to isomorphism.
- The *initial graph* is taken into account at the semantical level.
- Not only actions (and relationships among them) but also data (states) are represented at the semantical level.
- The unfolding is constructed *incrementally.*
- A number of important *relationships* that describe a system specified by a graph grammar, like the causal dependency, conflict and occurrence relations, are represented in the unfolding semantics. These relationships provide a good starting point to for analysis techniques for graph grammars.
- The unfolding semantics represents all sequential (and concurrent) derivations of a graph grammar.

Compositionality : The unfolding semantics is compositional with respect to the parallel composition of graph grammars.

Although parallel composition and unfolding semantics can be considered as stand-alone topics, to achieve the desired compositionality of the semantics many decisions taken in defining each of these constructions have influenced each other. These decisions will be discussed when the corresponding restrictions are made.

Many different ways of composing graph grammars have already been defined in the literature, for example the composition of classes in [Kor96] and the DIEGO approach [TS95]. There are also many different kinds of concurrent semantics for graph grammars, for example process semantics [Jan93, KR95, CMR96a] and event structures semantics [Sch94, CEL$^+$94a, Kor95, CEL$^+$96b]. The relationship between these approaches and the one introduced in this thesis will be discussed in more details in Chap. 7. The main advantage of our approach with respect to other existing ones is the compatibility of a true concurrency semantics with respect to the parallel composition operators within a framework of graph grammars. The compositionality of the unfolding with respect to the parallel composition gives an interesting aspect for the choice of type of composition and/or semantics to use. If one is interested in a concurrent semantics, the unfolding semantics is a good choice because, besides being a true-concurrent, branching structure semantics, it offers syntactical parallel composition operators that are compatible with it. If one is interested in composition operators for graph grammars, the parallel composition offers an unfolding semantics that is compatible with it.

Another contribution of this thesis is the definition of a class of graph grammars that can be considered as computations, in the sense that each rule of such a grammar represents a derivation step (of another grammar). These grammars, called *occurrence graph grammars*, can be used to represent deterministic and non-deterministic processes of graph grammars in the same way occurrence nets [NPW81] can be used to describe processes of Petri nets [GR83, BD87]. In the case of nets, occurrence nets are nets that are acyclic in which the relationships between transitions can be described by a causal and a conflict relationships.

In the case of graph grammars, these two relationships may not be enough to characterize the computation of a graph grammar. The fact that, especially in view of the preservation of items, the causality and conflict relationships may not be enough to describe suitably a concurrent system have been discussed in [JK93]. We will define concrete relationships that seem to capture the kernel of graph grammar computations. Moreover, it turns out that the unfolding of a graph grammar is an occurrence graph grammar. This fact means that the unfolding semantics enjoys one of the main advantages of an event structure semantics, that is the ability to reason about computations based on suitable relationships between the computation units (derivation steps) of a system. The unfolding also enjoys one of the advantages of the process semantics for graph grammars, that is the ability to describe states (as graphs). Therefore we believe that the unfolding semantics is a quite promising semantics for graph grammars, and that it can be used as a basis for analysis methods for graph grammars (as the unfolding semantics of Petri nets – see [McM92, McM95]).

The basic idea of parallel composition and unfolding presented in this thesis for graph grammars was inspired in corresponding ones for Petri nets as they can be found in [MMS94]. However, here there are two remarks that shall be made. The first one is that the most interesting form of parallel composition introduced here, namely the *cooperative parallel composition*, was not defined for Petri nets. This kind of composition allows to compose two components that share a common interface and we believe that this can be very useful also in the case of Petri nets (see Sect. 7.3 for more details). The other remark is that, although the idea of what is an unfolding is quite related to the corresponding idea of Petri nets, it is not an aim of this work to generalize the theory of Petri nets with respect to the adjunction between categories of Petri nets and occurrence nets, although this is a very interesting topic of research. Nevertheless, there are tight relationships between Petri nets and their unfoldings and the unfoldings of graph grammars and they will be discussed in Sect. 7.3.

The framework considered for the theoretical investigations developed in this thesis is the *Single-PushOut approach*, short SPO approach, to graph grammars [Löw90, Löw93]. The choice of this formalism was based on the fact that SPO graph grammars already provide a notion of a *concurrent derivation*, from which the unfolding semantics is a further development. Moreover, the absence of gluing conditions made the construction of the unfolding easier. We believe that it is possible to defined the concepts of unfolding semantics, as well as the parallel composition, for other kinds of graph grammars, like the DPO graph grammars (a discussion on this is made in Sect. 11). The formal investigations in this work will be done using category theory [AHS90, BW90]. Basic concepts are shortly reviewed in Appendix B.

Structure of the Thesis:

Chapter 2 : In this chapter an example of a specification of a telephone system using graph grammars is presented. The specification is done in 3 parts: an interface part, called global view, a phone component and a central component. This chapter serves as an informal introduction and motivation for the main concepts presented in this thesis.

Chapter 3 : This chapter reviews the basic notions of SPO graph grammars, including their sequential an concurrent semantics, using a kind of graphs called “typed graphs”.

Chapter 4 : In this chapter graph grammar morphisms and the parallel composition operators for graph grammars are introduced. It is shown that a composed graph grammar is suitably related to its components (by the existence of corresponding morphisms), and that the composition operators correspond to special categorical construction, namely product and pullbacks. This has the advantage that, from standard categorical results, we obtain that the parallel composition operators are compatible with each other, that they are unique (up to isomorphism) and that they are associative.

Chapter 5 : A semantical framework for graph grammars, namely occurrence graph grammars, is introduced in this chapter. An occurrence graph grammar is a grammar that represents (deterministic or non-deterministic) computations of some graph grammar. A relationship between an occurrence graph grammar and a grammar from which it represents some computations is provided by a folding of the occurrence graph grammar. This folding is formally described by a functor. It is shown that each rule of an occurrence graph grammar corresponds to an application of a corresponding rule in some derivation of its folded grammar. Moreover, it is shown that concurrent derivations (that can be considered as deterministic computations of a grammar) are indeed occurrence graph grammars.

Chapter 6 : This chapter introduces the unfoldings of graph grammars. The unfolding of a graph grammar is obtained inductively by applying the rules of the grammar starting from the initial graph of the grammar. An important result is that the unfolding is not only a graph grammar, but an occurrence graph grammar, and thus enjoys a lot of special properties that may be useful for analysis of the original graph grammar. It is also shown that the unfolding describes exactly all sequential and concurrent derivations of a graph grammar. Then a connection between parallel composition and unfoldings is established: we show that the unfolding semantics is compositional with respect to the parallel composition operators (formally this is expressed by the fact that the unfolding construction can be extended to a functor that preserves products and pullbacks).

Chap. 7 : This chapter contains relationships to other works in the areas of composition and concurrent semantics of graph grammars. Special attention is given to Petri nets and the relationships of Petri net concepts with the constructions developed in this thesis.

Appendix A : This appendix contains some mathematical conventions we use.

Appendix B : This appendix provides basic notions of category theory and set-theoretical characterizations of many of the categorical constructions in the categories we use.

Appendix C : This appendix contains some proofs and lemmas.

2

Parallel Composition and Unfolding of Graph Grammars: An Example

Graph grammars are well-suited as a specification formalism for parallel and concurrent systems (see, e.g., [Tae96]). In this chapter we will present (part of) the modeling of a Private Branching Exchange System, short PBX System, using graph grammars. This specification was developed within the project *"Graphical Support and Integration of Formal and Semi-Formal Methods for Software Specification and Development"*, short GRAPHIT. GRAPHIT is a project within the German/Brazilian cooperation *"Information and Technology"* supported by DFG and CNPq, and the main aim of the project is to integrate semi-formal and formal methods to provide a specification method meeting the requirements of industries. The partners of the GRAPHIT project are the Brazilian company Nutec, the German company MSB, the Brazilian Federal University of Rio Grande do Sul (UFRGS) and the German Technical University of Berlin (TU Berlin). The PBX system is an ongoing project of the company Nutec, that develops corresponding software and hardware. The main aim of this system is to control a private telephone net.

This chapter shall serve as an example and practical motivation for the theory that will be developed in the following chapters of this thesis. Although the theory will be developed for SPO graph grammars, as far as possible we will explain the constructions in this chapter independently of this concrete approach. Section 2.1 describes the telephone system to be modeled and Sect. 2.2 presents the main ideas of the modeling of the PBX system using graph grammars and how a specification of the whole system can be obtained from the specification of the components by using one of the parallel composition operators presented in this thesis (in Chap. 4). Section 2.2.5 contains some insights on three possible formal semantics for the system: one based on sequential computations, one based on concurrent computations and one based on unfoldings, where the latter semantics for graph grammars is newly introduced in Chap. 6 of this thesis.

Only a very small part of the specification is presented here. Besides the big reduction of the number of rules that describe the system, a further simplification made here is the absence of attributes used to specify the data-types involved in the system. We use a simple concept of typed graphs to specify different types. A full version of this case study using graphs with attributes can be found in [Rib96].

2.1 Telephone System

The kind of telephone system to be considered here is known as a Private Branch Exchange (PBX) system. A PBX provides an intelligent connection between a (small) telephone pool – as it can typically be found in companies – and several external lines giving access to an already existing (public) telephone net. The heart of such a system is a piece of hardware—often called a CENTRAL. The CENTRAL controls the (internal) communications between the PHONEs and manages the connection of PHONEs inside the system with PHONEs outside belonging to a second (external) telephone CENTRAL. Modern PBX systems additionally provide a number of features as, e.g., programmable keys, last number redialing, call-back, follow-me, or automatic-answering mechanisms. For simplicity, we will restrict to the internal side of such a telephone system, i.e., one CENTRAL connected to several standard PHONEs. Therefore, the main aim of a PBX system presented here is to control the calls between the telephones that are connected to it. The messages it receives from its phones are usually informations about the state of the hook of the phones and the digits dialed by the users of the phones. The reaction to these messages is to send appropriate tone/ring signals to the phones and establish a connection with the called phones. For a more complete description we refer to [Rib96].

2.2 Specification of the PBX System using Graph Grammars

A graph grammar specifies a system in terms of states and state changes, where the states are described by graphs and the state changes are described by rules having graphs at the left- and right-hand sides. The relationship between the left- and right-hand sides of a rule is given by mapping vertices and edges from the left- to the right-hand side in a compatible way. Compatible way means that is an edge $e1$ is mapped to an edge $e2$ then the source (target) vertex of $e1$ must be mapped to the source (target) vertex of $e2$. Such a compatible mapping between graphs is a called a *graph homomorphism*, or simply graph morphism. A rule of a graph grammar may express preservation, deletion and/or creation of vertices and edges. This is expressed by a graph morphism f in the following way:

- *Preservation:* An item belonging to the left-hand side of a rule and mapped via the morphism f to the right-hand side of this rule is *preserved* by this rule.
- *Deletion:* An item belonging to the left-hand side of a rule and not mapped via the morphism f to the right-hand side of this rule is *deleted* by this rule.
- *Creation:* An item belonging to the right-hand side of a rule that is not the image under the morphism f from any item of the left-hand side of the rule is *created* by this rule.

To allow the deletion of items by a rule the relationship between its left- and right-hand sides may be partial.

The operational behaviour of a system described by a graph grammar is described by applying the rules of the grammar to actual graphs representing the states of the system (starting from a given initial state). The application of a rule to an actual graph, called *derivation step*, is possible is there is an occurrence of the left-hand side of this rule into the actual graph. This occurrence, called *match*, is a total graph morphism because one intuitively expects that all elements of the left-hand side must be present at the actual graph to apply the rule. The result of the application of a rule is obtained by deleting from the actual graph all items that are deleted by the rule and adding to the actual graph the items that are created by the rule (in fact, in the approach to graph grammars used here other effects like identification of items and deletion of context are possible – these cases will be discussed in Chap. 3).

For specifying practical applications, there are usually a large number of vertices and edges in the graphs used to model the system. To make them easier to understand, a way to distinguish different kinds of vertices/edges is helpful. In the PBX example, we would like to distinguish vertices that correspond to telephones from vertices that represent PBX centrals, for example. A typing mechanism can be used to make this distinction. One can implement this typing mechanism in many different ways, for example using labeled graphs or graphs having different sets of vertices/edges. Here we will use a graph, called *type graph*, to specify the types of elements of the system: each vertex of the type graph represents one distinct type of vertex and each edge of the type graph represents one distinct type of edge. Each actual graph of the system must then have an interpretation in terms of this type graph. Formally, the concept of type graph was introduced in [Kor93, CMR96a].

A graph grammar consists of the following components:

- Type graph: specifying the different kinds of items that may occur in the system;
- Initial graph: specifying the initial state of the system;
- Rules: specifying the possible changes of state of the system. Each rule consists of a rule name and the rule itself (left- and right-hand sides connected by a morphism).

A telephone system is characterized by a high communication traffic and, in particular when more than one PHONE is involved, there is a high degree of desirable parallelism. Therefore specification methods that allow (at the syntactical and semantical levels) the modeling of parallelism are more suitable for this kind of system. Graph grammars is an example of such formalisms. Moreover, this system is a reactive system: the behaviour of each component is described by the reaction performed by the component to some message that was received by it. Reactive systems are very common in the area of concurrent systems where the communication is done through message passing (see [MP92] for more details).

We will present a way of specifying a system by composition of the specifications of the components. Obviously, the first step that has to be done towards such a specification is the identification of the components of the system. Moreover, we have to identify the relationships between the components and how these components cooperate to perform the tasks of a PBX system. The result of this specification task will be called *abstract global view* of the system. This view can be seen as an abstract interface for the development of the concrete components of the system. Then we may specify each component separately. These specifications can be seen as implementations (or specializations) of the abstract behaviour described by the abstract global view. Each of such specifications is called *local view* of the system. When all specifications of the components are ready, they can be suitably composed yielding a concrete specification for the whole system, called *concrete global view* of the system.

For the PBX system, we can recognize 3 components: CENTRAL, PHONE and ENVIRONMENT. The third component corresponds to the users. We included this component here for two reasons: i) the PBX system works as a reaction to the messages it gets from the users, and thus the inclusion of users is necessary to model suitably the whole system, ii) discussions with the company Nutec (to whom this specification was developed) led to the result that the specification of the user's behaviour can be very useful to produce a high-level user's manual for the system. The specification of the Environment component will not be further specialized here, it suffices to give an abstract view for it. Therefore we will have only two local components for the PBX system: Central and Phone. This idea is summarized in the picture below, where "AG" stands for abstract global and "CG" for concrete global. The arrows between the components mean *specialization* (or refinement) relationships. Both the Phone and the Central views are specializations of the abstract view, and the concrete global view can be seen as the smallest specialization of the abstract view including the specializations described by the Phone and the Central components.

$$\begin{tikzcd}
 & PBX - AGView & \\
\text{Phone } View \arrow[ur] & & \text{Central } View \arrow[ul] \\
 & PBX - CGView \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Each of these views will be described by a graph grammar, and the relationships among them will be described by graph grammar morphisms. The composition of the local views (taking into account their connections established in the abstract global view) to get the concrete global view of the system will be described by an operation on graph grammars called *cooperative parallel composition*, that will be defined in Chap. 4.

2.2.1 Abstract Global View of the PBX System

The idea of the specification of the abstract global view is to model the telephone system "in the large", i.e., to say which are the components of the system, which are the relationships between these components and what are the messages that are exchanged between them (in an action/reaction way). At this global level, the internal structure of each component is not considered. The abstract global view of the telephone system is described by a graph grammar called *AGV* consisting of the following components (type graph, initial graph and rules):

Type Graph: We may recognize two main types in the PBX system: the type Phone and the type Central. As there are many operations of phones and of centrals that are reactions to stimulus from their environments (users), we will also include an Environment type in the system. These types are drawn as ☎, ✺ and 🧍, respectively, in Figure 2.1, and the arrows between them correspond to "knows" relationships. For example, the arrow from the Phone to the Central means that the Phone knows this Central and thus can send messages to it. Phones may also send messages to users (for example, ringing signal). Therefore there is a knows relationship from Phone to Environment. In our specification of the PBX system, Centrals must never send messages to users, and therefore there is no arrow from Central to Environment.

Messages are also represented types. There is one message type for each kind of message that can be sent in the system. Messages are drawn as envelopes carrying a name (the

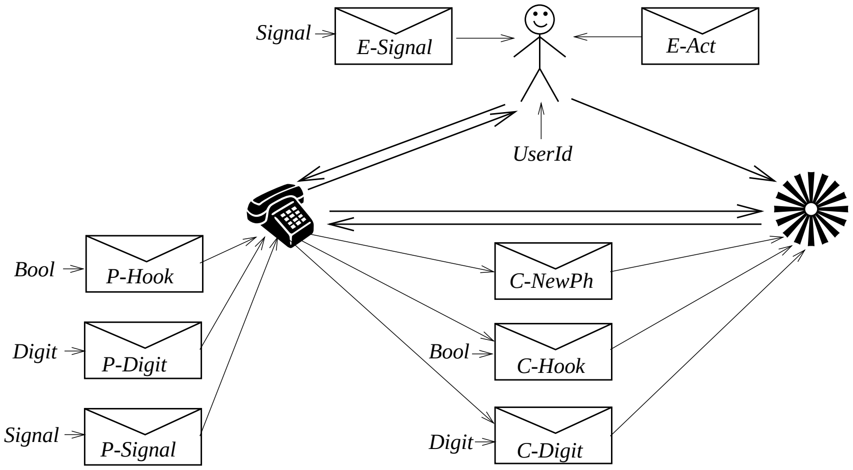


Figure 2.1: Type Graph *Type*

name of the message). Messages always have a target entity (in our case, a Phone, a Central or an Environment). This is represented by an arrow from the message to the corresponding target type. Messages may also have parameters, and these are obviously elements of some type. Parameter types may be references to some Phone or Central but also basic data types like natural numbers, booleans, digits, etc. Although basic data types are also modeled by vertices, they will be indicated just by their names in the graphical representation. Parameters are indicated by edges connecting the parameter type and the message. The data types used in this example are:

$$\begin{aligned}&Bool ::= On(T)|Off(F)\\&Signal ::= Mute|Free|Busy|Call|Carrier|Ring|Wrong\\&Digit ::= 0|1|2|3|4|5|6|7|8|9\\&UserId ::= \{user1, user2, user3, user4, user5\}\end{aligned}$$

The graphical notation used in Figure 2.1 is in fact a shorthand. For example, the vertex `P-Hook` describes two messages: `P-Hook(On)` and `P-Hook(Off)`. That is, there is one message for each possible data type carried by this message.[1] Messages of type `Hook` are used to inform Phones and Centrals about changes on the state of the hook of a telephone. Messages of type `Digit` are used to inform Phones and Centrals that a digit has been dialed (this digit is sent as a parameter of the message). Messages of kind `Signal` are used to send audio signals to Phones and Environments (ring, carrier-tone, busy-tone, mute, etc). The kind of signal is the parameter of the message.

Initial State: We start the system from a state in which 4 telephones are already connected to the PBX system. Thus, the initial state of the system consists of 4 Phones connected to a Central, together with the corresponding users (the user-ids are used for the case of checking the rights of some user to perform some activity, for example, a user must have special rights to connect a new telephone in the net). Moreover, the users are able to act (this is indicated by the `E-Act` messages connected to them). The initial state is depicted in graph *Ini* of Figure 2.2.

Rules: In a PBX system there are a number of actions such as sending dialed digits,

[1] In the model described in [Rib96] using graph grammars with attributes it is not necessary to have one message for each possible parameter it may carry because variables may be used.

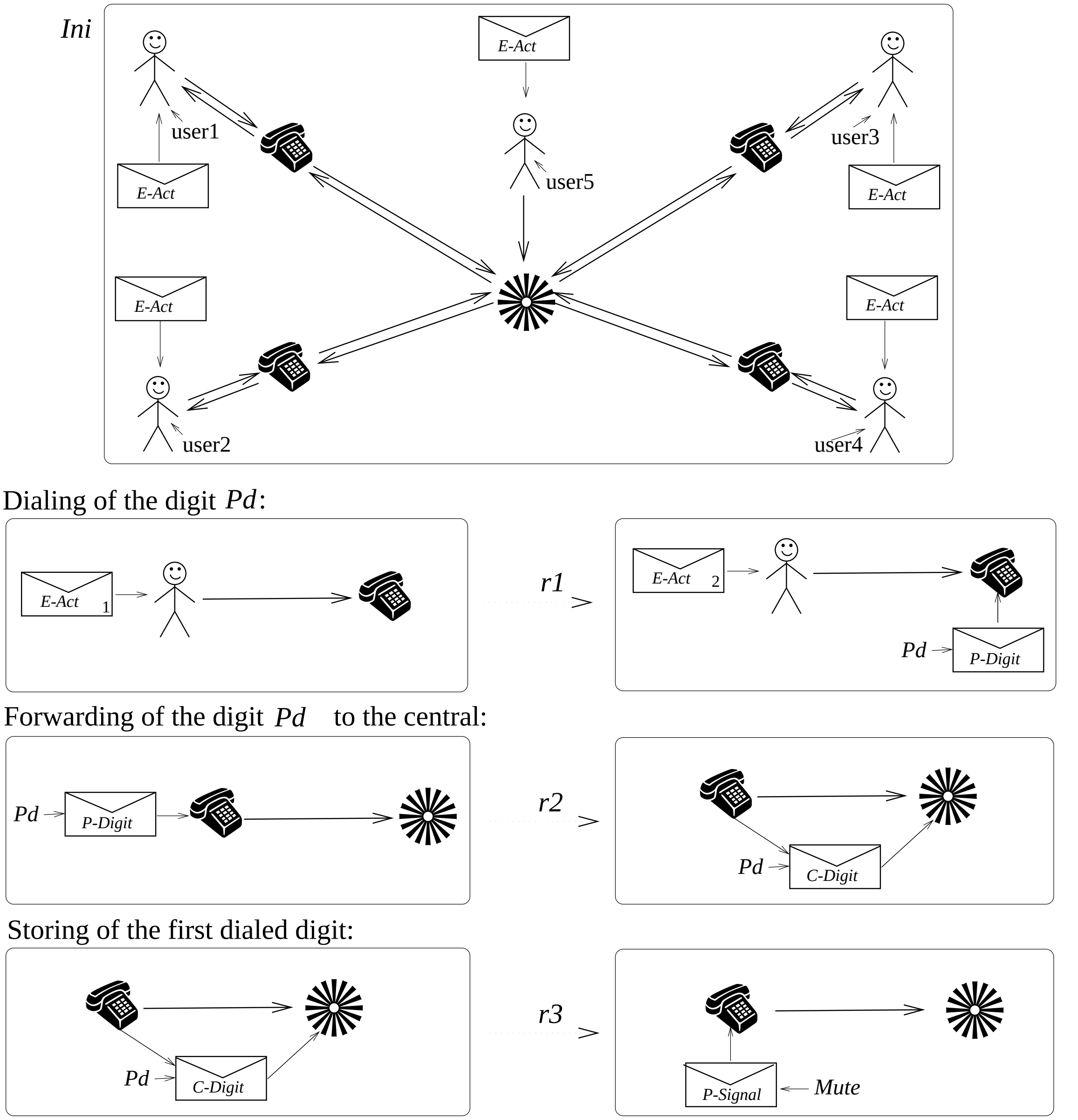


Figure 2.2: Abstract Global View: Initial State and Rules of *AGV*

establishing connections between phones, connecting new phones to the net, etc. These actions are modeled by rules. Three rules of these rules are shown in Figure 2.2 (the other can be found in [Rib96]). The first rule describes an action of the user, namely the dialing of the digit *Pd*. This is modeled by a rule that deletes the user's `E-Act` message and creates a message `P-Digit(Pd)` on its corresponding Phone. The purpose of the `E-Act` message is to model the fact that a user is able to act (therefore, a message `E-Act` is also created by the rule $r1$ indicated that the user is still able to act after dialing one digit). In the graphical notation, items that are in both left- and right-hand sides of rules are preserved. To be more precise, sometimes items are indexed (for example, the message `E-Act` of the left-hand side of rule $r1$ is different from the message `E-Act` of the right-hand side). The second rule ($r2$)

describes the forwarding of a digit *Pd* to the central, and the third rule ($r3$) describes the the behaviour of the central if this number is the first one dialed by this phone (in this case, the central just has to store this number and wait for a second one to establish a connection). At this level of abstraction – where we can not see the internal states of the components – it seems that the `C-Digit(Pd)` message was merely deleted. It is the task of the Central component to specify the storing of this digit in its internal variables.

2.2.2 Phone Local View of the PBX System: Grammar PLV

In this part the Phone component is further specified in grammar PLV. The internal state of a Phone is specified in the corresponding type graph *PType* (see Figure 2.3).This internal state consists of three vertices describing the hook status (*P.h*), the phone status (*P.st*) and if the phone has some digit to be sent to the central (*P.pd*). The identifiers of the edges will be used to describe the typing of the instances of this type graph. The rules of the grammar shown in Figure 2.2 are specialized to include the internal state of the Phone. For example, there are two rules that specialize rule $r2$, namely $Pr2.1$ and $Pr2.2$. The first rule sends the first dialed digit to the Central (when there is a carrier tone), and the second sends the second digit (when there is no tone). The initial graph $PIni$ is the initial graph Ini plus the internal structure of all phones, namely $P.h = On$ (the hook is on), $P.st = Free$ (the phone is free), $P.pd = F$ (no digit was dialed).

2.2.3 Central Local View of the PBX System: Grammar CLV

The specification of the Central component is shown in Figure 2.4 (grammar CLV). The type graph includes a component TAB that carries informations about each Phonethat is connected to the Central. These informations are the status of the Phone (*C.st*), the number of the Phone (*C.pnr*), the number dialed by a Phone (*C.dnr*), the information whether a Phone have dialed a digit (*C.pd*) and an established connection between two Phones (indicated by the *c*-edge). Rule $Cr2$ describes the situation in which a digit is sent from a Phone to the Central if the Phone status is $Carrier$, that is, it models the sending of the first digit of a telephone number to the Central. The component *C.pd* assures that the order of sending digits from Phones to Centrals is preserved. Rule $Cr3$ models the behaviour of the Central when it receives a digit message and the Phone status is $Carrier$: the Phone status is set to $Mute$, a corresponding signal message is sent to Phone and the dialed digit is stored in $C.dn$. In the initial graph $CIni$, we have the initial graph Ini plus the initial internal state of the Central: there is one TAB entry for each Phone, and the parameters are set correspondingly.

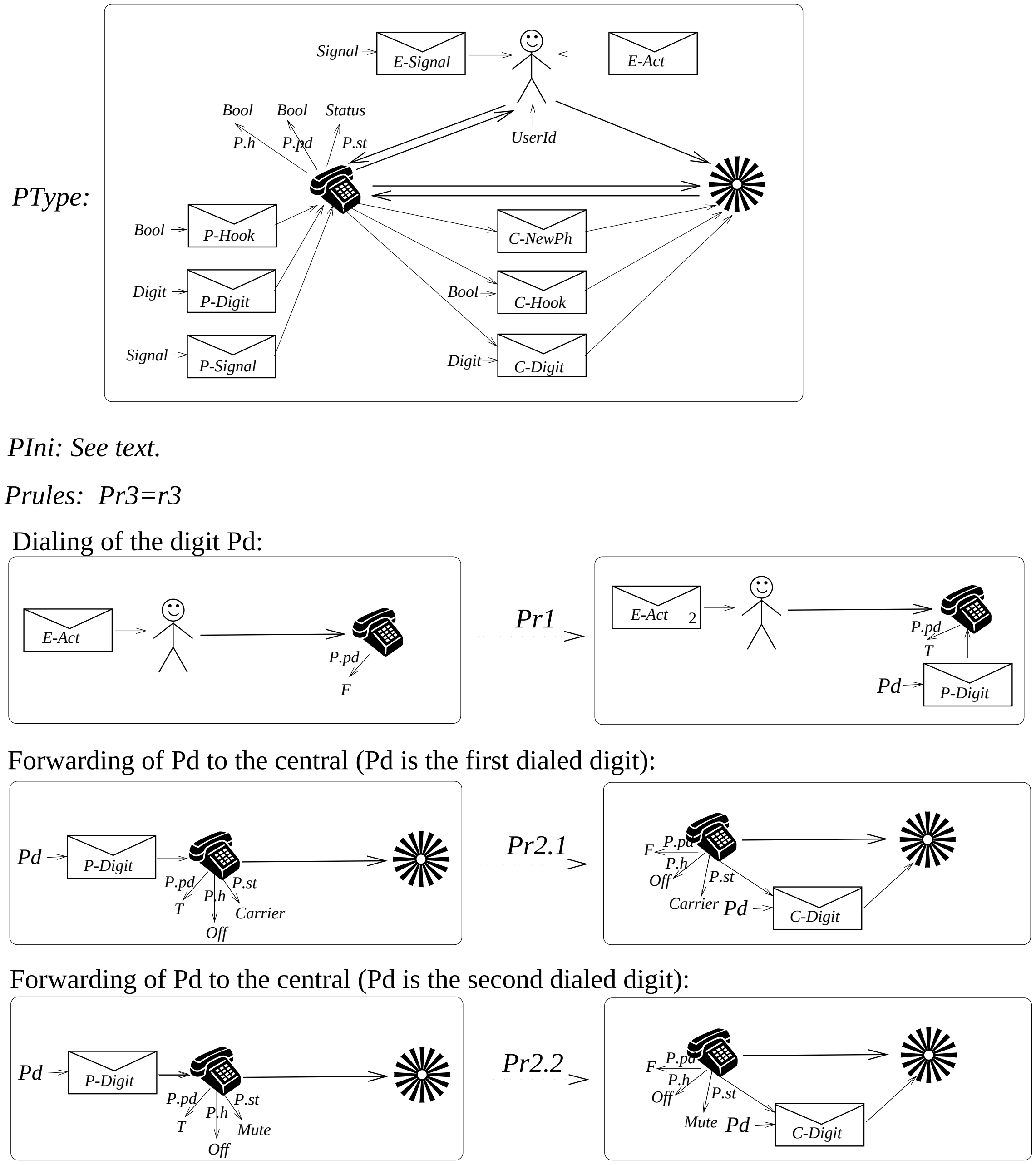


Figure 2.3: Phone Local View

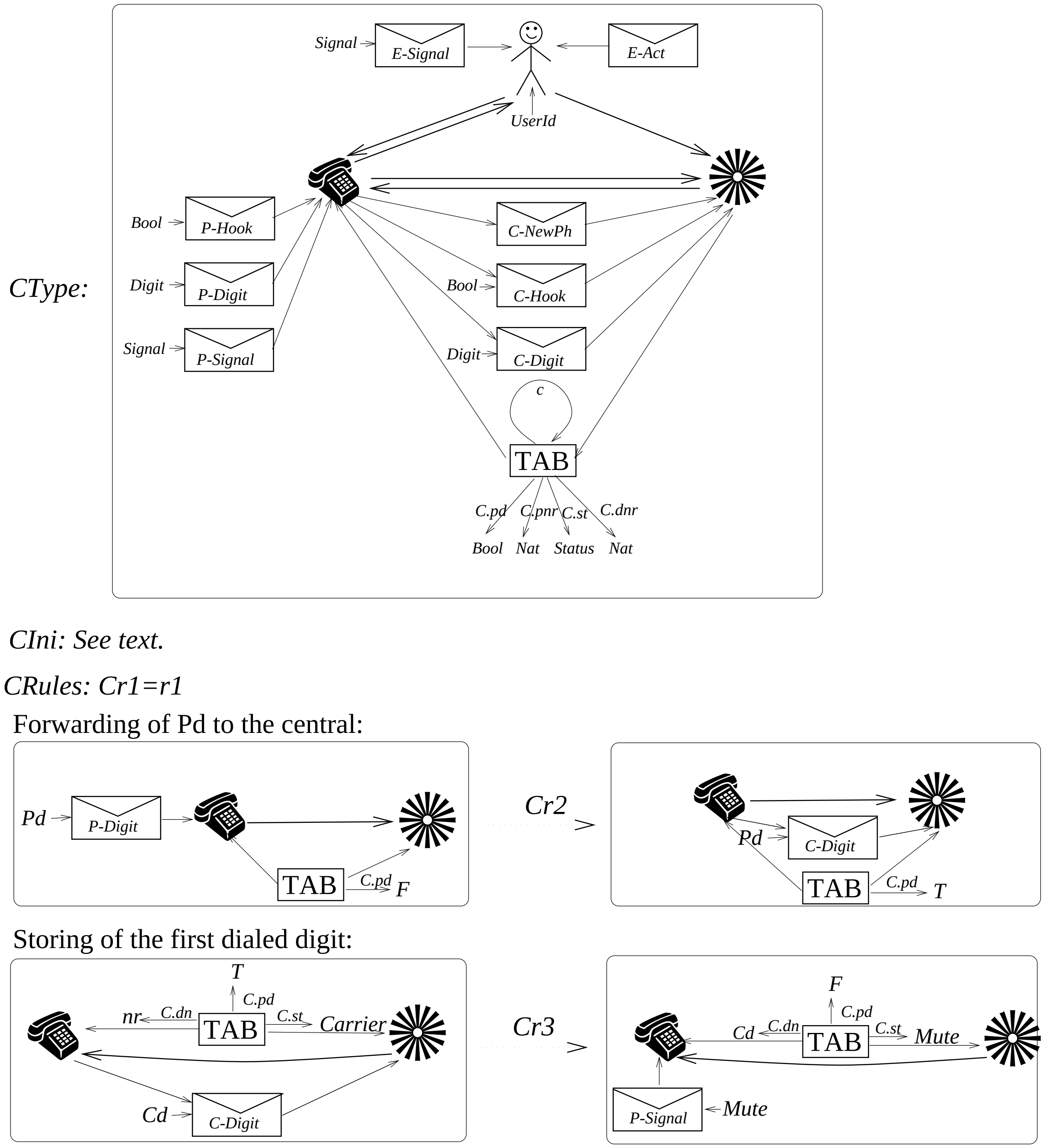


Figure 2.4: CENTRAL Local View

2.2.4 Concrete Global View of the PBX System: Grammar *CGV*

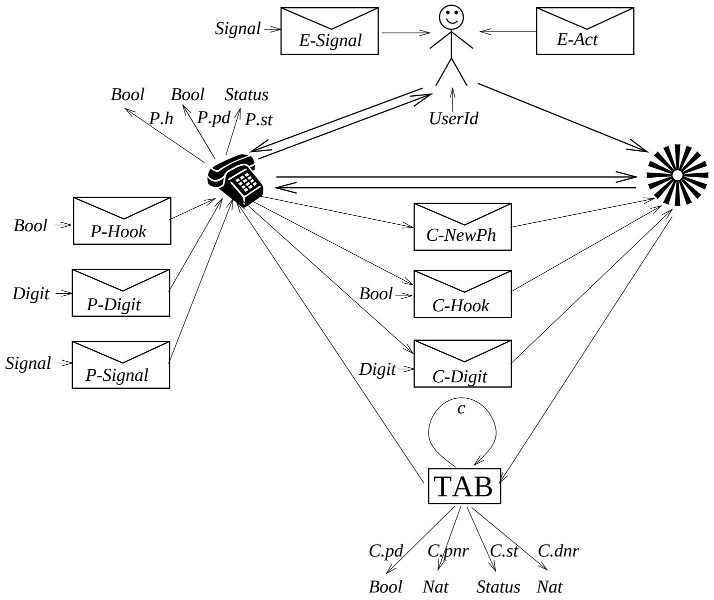


Figure 2.5: Concrete Global View: Type Graph

To get the concrete global view of the telephone system, we put both specializations (of Phone and Central components) together gluing them along the shared items. First, the type graph is constructed in this way, giving raise to the graph shown in Figure 2.5. Then the initial graph is constructed in the same way (see graph $PBXIni$ of Figure 2.6). The construction of the rules of the system is also done by gluing corresponding rules. According to the way in which the rules of the interface are specialized in the components, we may have the following situations:

1. *A rule of the interface is not specialized in any component.* In this case, the same rule will be part of the composed system (in our example, there is no rule in this situation).

2. *A rule of the interface is specialized by one rule of only one component.* This is the case of rules $r1$ (that is specialized only by the Phone local component) and $r3$ (that is specialized only by the Central local component). In this case, the specialized rule is part of the composed system, i.e., rules $Pr1 = PBXr1$ and $Cr3 = PBXr3$ are in the concrete global view of the telephone system.

3. *One rule of the interface is specialized in both components.* This is the case of rule $r2$. The result is that the resulting rule(s) of the composed system is(are) a gluing of the two specializations. This kind of composition models a synchronization between the components. If there are different specializations for the same interface rule within one component (for example $Pr2.1$ and $Pr2.2$), both these specializations are combined with the specialization of this rule done in the other component (in our example, giving raise to rules $PBXr2.1$ and $PBXr2.2$). Obviously, if there are more specializations for the same interface rule in both components, all combinations will become rules of the composed system. Rules $PBXr2.1$ and $PBXr2.2$ describe the fact that the Phones

can only send a digit to the Central when there is no pending digit (modeled by $C.pd = F$ in the left-hand sides of these rules).

4. *A rule of one component is not in the interface.* In this case, this rule shall be added to the composed system. If there are also rules in this situation in the other component, they will also be added to the composed system, and also their parallel compositions will be added. (In the example, there are no such rules.)

2.2.5 Semantics of the PBX System

Depending on the aspects of a system we are interested in, one semantical model may be more appropriate than others. For the telephone system, the main aspect we are interested in is concurrency. Therefore semantical models that describe concurrency seem to be more adequate in this case. This means that the reachable states are not so important but the way they are reached. A suitable semantics for concurrent systems shall provide means for reasoning about computations, for example, how they are obtained, which actions may happen in parallel, what are the relationships between different computations and between actions of the same computation, etc. To understand which kinds of relationships may occur between different actions of a system, we will give a small example. These relationships are described in different ways by different semantical models.

Example **2.1** The following actions are possible in the PBX system:

1. Phone 12 gets a `Digit(5)` message.
2. Phone 52 gets a `Digit(4)` message.
3. Phone 12 gets a `Digit(3)` message.
4. Phone 12 forwards the `Digit(3)` message (received in action 3.) to its central.

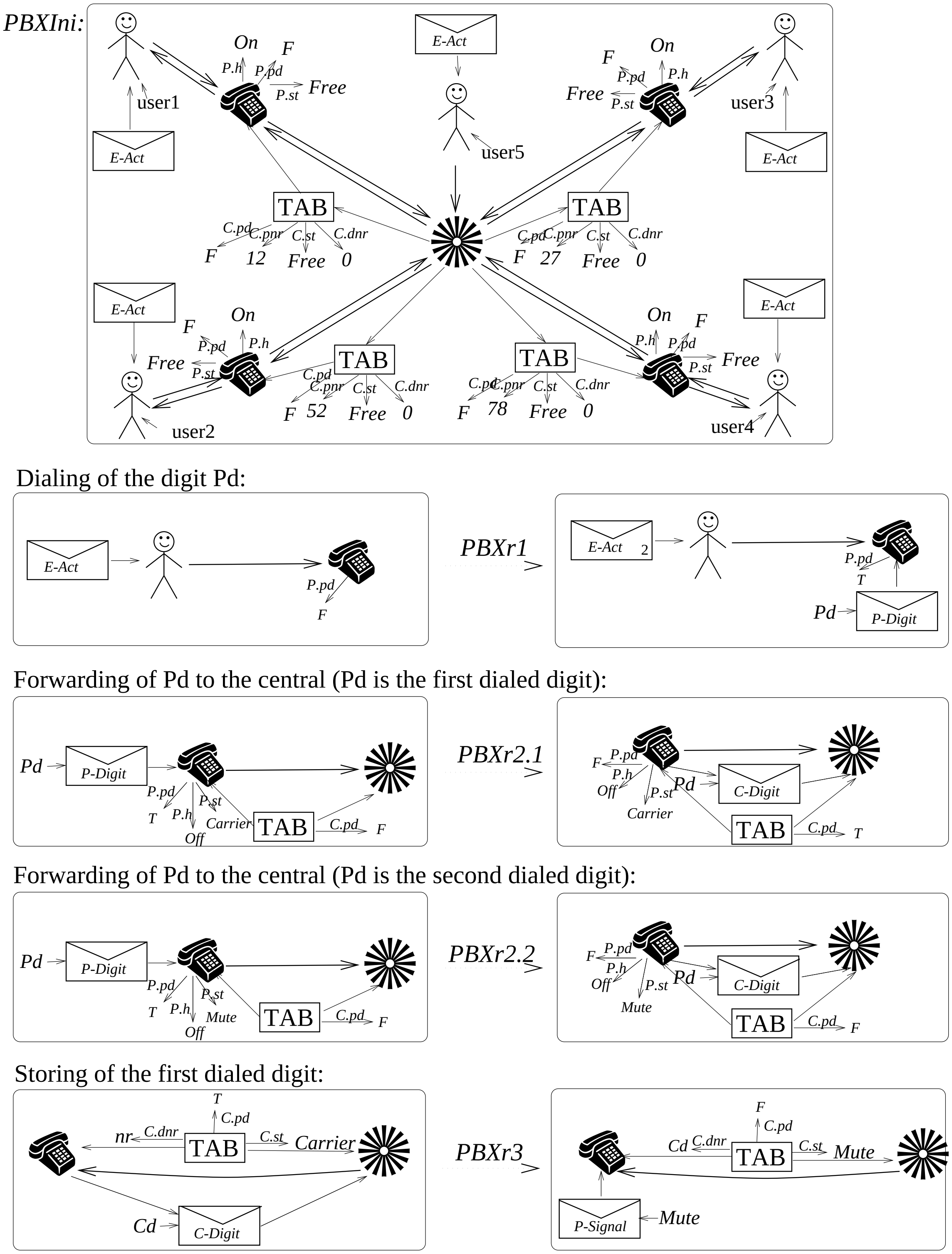
Obviously, actions 1 and 2 may occur in parallel because they involve different telephones. Actions 1 and 3 are in conflict because only one digit may be dialed at each time (phone numbers are *sequences* of digits). Action 4 depends on action 3 (Phone 12 can only send a digit that was dialed to the central). ☺

Next, we will present three kinds of semantics for the PBX system: one based on sequential computations, one based on concurrent computations and one based on unfoldings. In this thesis, we will newly introduce an unfolding semantics for (SPO) graph grammars. The aim of describing the already existing sequential and concurrent semantics of graph grammars here is to compare and motivate the unfolding semantics.

Sequential Semantics

A computation step of a system described by a graph grammar is modeled by a derivation step, i.e., an application of a rule to a graph with respect to a match. Thus the (operational) semantics of the system can be described by sequences of such derivation steps, called *sequential derivations*. This kind of semantics is called **sequential semantics**.

One derivation sequence of the PBX system, namely derivation $\sigma 4$, is shown in Figure 2.7. The matches used for the applications of the rules are indicated by corresponding indices.



Dialing of the digit Pd:

Forwarding of Pd to the central (Pd is the first dialed digit):

Forwarding of Pd to the central (Pd is the second dialed digit):

Storing of the first dialed digit:

Figure 2.6: Concrete Global View: Initial Graph and Rules

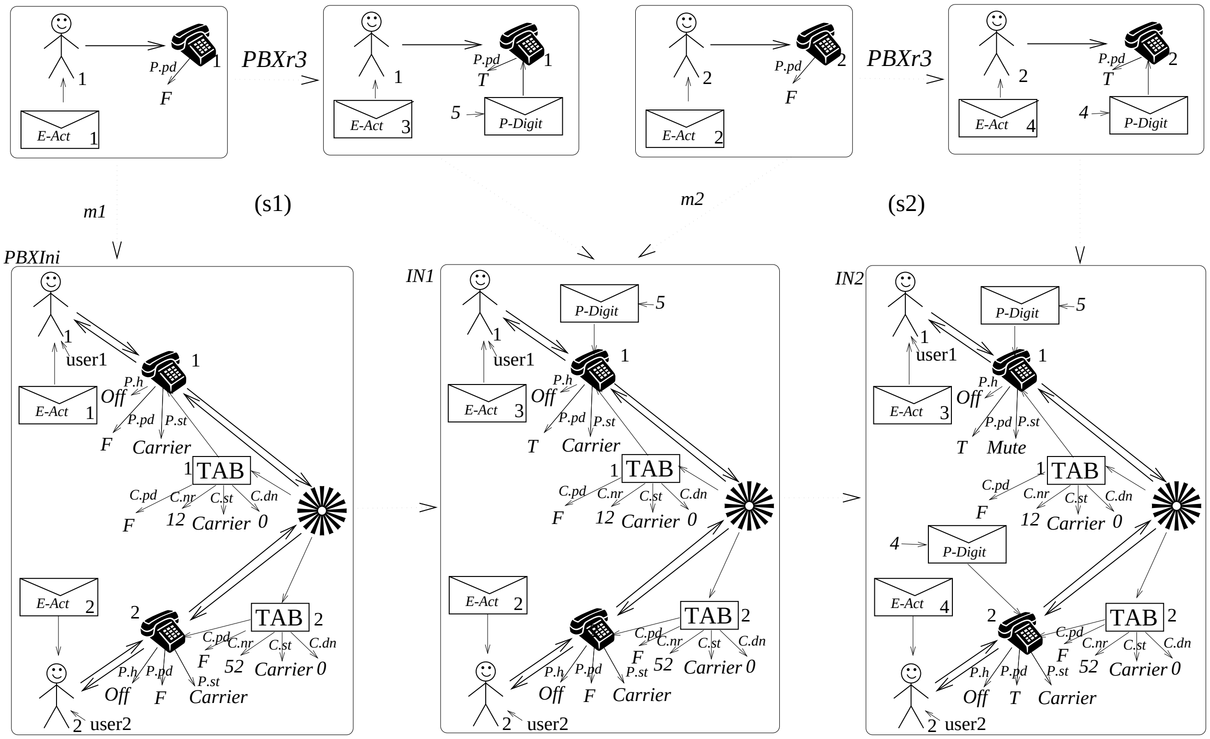


Figure 2.7: Derivation Sequence $\sigma 4$

In this derivation sequence the user of Phone 12 generates a `P-Digit(5)` message (step $s1$) and then the user of Phone 52 generates a `P-Digit(4)` message (step $s2$). Let derivations $\sigma 5$ and $\sigma 6$ be defined as follows: In derivation $\sigma 5$ these two messages are sent in the inverse order; and in derivation $\sigma 6$ the first step ($s5$) represents the generation of a `P-Digit(3)` message on Phone 12 and the second step ($s6$) represents the forwarding of this digit to the Central. We can observe that, among others, the following sequences belong to the sequential semantics $SeqSem(CGV)$ (";" denotes sequential composition):

$$\begin{aligned}
\sigma 1 &= (s1),\\
\sigma 2 &= (s3),\\
\sigma 3 &= (s5),\\
\sigma 4 &= (s1; s2),\\
\sigma 5 &= (s3; s4),\\
\sigma 6 &= (s5; s6)
\end{aligned}$$

As the telephone system is highly parallel, many derivation steps may occur concurrently. In a sequential semantics, concurrency is derived from arbitrary interleavings, i.e., if two actions a and b may occur in any order, then they may occur concurrently. This means that concurrency is the same as *sequential* independence. The notion of sequential independence of graph grammars can be used to find out which steps "mean" the same action. In the example, all steps $s1$ to $s6$ are different, but one can see that $s1$ and $s4$ (respectively $s2$ and $s3$) mean the same action: the sending of a `P-Digit(5)` (`P-Digit(4)`) message to Phone 12 (Phone 52). Formally, to find out whether two steps are sequential independent or not we have to try to prolongate the match of the second derivation until the graph where the first rule was applied. If we succeed, the steps are sequential independent and may be performed in the inverse order (and this means that they may also be performed in parallel). If not, the steps are sequentially dependent, and may only be performed in this order.

Concurrent Semantics

The main idea of the concurrent semantics of graph grammars is to use causal independence (instead of sequential independence) to describe concurrency. Roughly speaking, we say that one step $s2$ is causally dependent of a step $s1$, written $s1 \leq s2$, if $s1$ creates something that is needed for $s2$, i.e., , $s2$ can not occur if $s1$ had not occurred before. Thus, we are not anymore interested in sequences of steps, but in sets of actions related by a causal relationship. (We use the term 'step' only for a unit of a sequential derivation, and the term 'action' as a unit of a concurrent derivation.) A *concurrent derivation* is obtained from a sequential one by abstracting out from the intermediate graphs of the sequential derivation. The basic idea to construct a concurrent derivation is to glue all intermediate graphs into a graph C called *core graph* of a derivation (because it represents the kernel of this derivation). A concurrent derivation consists of an initial graph, a core graph and a set of rules, where the initial graph and each rule are connected (via morphisms) to the core graph. An *action* consists of a rule and the matches (morphisms) of the left- and right-hand sides of this rule into the core graph. These matches are called *pre-* and *post*-conditions of the action, respectively. For example, the sequential derivation $\sigma4$ gives raise to the concurrent derivation $\kappa4$, written $\sigma4 \leadsto \kappa4$, shown in Figure 2.8. Although in this example the concurrent and the sequential derivation are quite similar, this is not always true. No matter how long a sequential derivation is, its concurrent derivation always consists of one core graph, one initial graph (together with its morphism into the core graph) and a set of actions (rules together with morphisms into the core graph).

If we look at the concurrent derivation $\kappa4$, we can not say which of the actions $a1$ or $a2$ shall occur "first" (in a corresponding sequential derivation). This is because the pre- and post-conditions of these actions do not overlap in the core graph, i.e., the images of the pre- and post-conditions of these rules are disjoint. Moreover, $\kappa4$ is also the concurrent derivation of the sequential derivation $\sigma5$, i.e., $\sigma5 \leadsto \kappa4$. This stresses the fact that $\sigma4$ and $\sigma5$ represent in fact the same computation if we abstract from the sequential order. Let $\kappa6$ be the concurrent derivation generated from $\sigma6$ ($\sigma6 \leadsto \kappa6$). In the concurrent derivation $\kappa6$, the pre-condition of action $a6$ overlaps with the post-condition of action $a5$ on the item `C-Digit(3)` of the core graph, and this item was created by the action $a5$. This implies that action $a6$ is causally dependent of action $a5$, written $a5 \leq a6$, and thus there is only one possible sequential order in which these action can be observed: $a5; a6$. Thus, the following concurrent derivations are included in $ConcSem(CGV)$ ("," denotes that two actions are causally unrelated and $\leq$ denotes causal dependency):

$$\begin{aligned}\kappa1 &= (a1),\\ \kappa2 &= (a2),\\ \kappa3 &= (a5),\\ \kappa4 &= (a1, a2),\\ \kappa5 &= (a5 \leq a6)\end{aligned}$$

With respect to the sequential derivations described before, here there is one derivation less because $\sigma4$ and $\sigma5$ are represented by the same concurrent derivation ($\kappa4$).

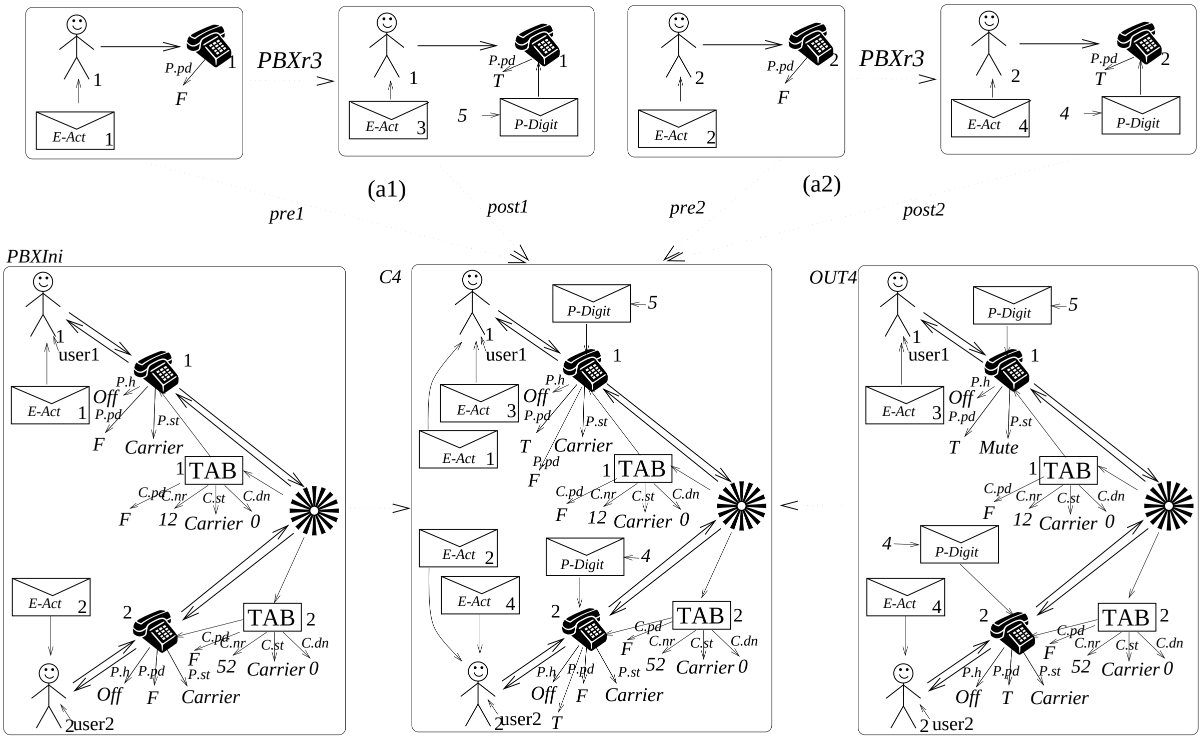


Figure 2.8: Concurrent Derivation $\kappa 4$

We observe that the core graph plays a role that is analogous to the role of the type graph of a grammar: it is a static structure in which all rules and the initial graph find compatible interpretations. But the core graph (as well as all rules and the initial graph of the grammar) are already typed (over the type graph $PBXType$). Therefore a concurrent derivation may be seen as a "doubly-typed" graph grammar (having the original type and the core graph as types and where the core graph itself is typed over $PBXType$). We will call this kind of grammars *doubly-typed graph grammars*. The fact that a concurrent derivation can be considered as a grammar gives an interesting view on the semantics of graph grammars: the behaviour of a graph grammar GG may be represented by a set of graph grammars (where each of these graph grammars represents one concurrent derivation of GG). As the application of the rules of one graph grammar may be non-deterministic, this raises the question whether it is possible to describe the behaviour of a graph grammar by only one graph grammar (intuitively as a suitable union of all graph grammars representing the concurrent derivations). Indeed, this is possible, and this kind of semantics will be called *unfolding semantics* here.

Unfoldings Semantics

As concurrent derivations are obtained by gluing the intermediate graphs from sequential derivations, they can not describe non-deterministic (conflict) situations explicitly, such situations are described by the non-existence of a derivation including the two "conflicting ones". In the example, $\kappa 1$ and $\kappa 2$ are not in conflict because there is a concurrent derivation, namely $\kappa 4$, that contains both. But $\kappa 1$ and $\kappa 5$ are in conflict because there can be no concurrent derivation containing both (due to the fact that both actions $a1$ and $a5$ delete the same item `E-Act` from the core graph, and an item can be deleted at most once in each derivation).

The interplay between non-determinism and concurrency gives a very rich description of the behaviour of a system. A well-accepted way to describe this interplay is by modeling a system using a causal and a conflict relationships (as it is done in event structures [Win87a]). The **unfolding semantics** of a system presented here is able to express these relationships in a natural way. Moreover, these relationships are defined not only between actions but also between items from the state graphs (this gives us a good basis for analysis of a grammar).

Graph grammars are usually cyclic, in the sense that the same rule may be applied many times (each time in a different context). The idea of the unfolding is to construct a grammar that represents all possible rule applications in all different contexts. The unfolding is constructed inductively starting with the initial graph of the grammar and in which in each step all possible applicable rules are applied to the results of the last step. As each item of the core graph can be created by at most rule, the unfolding is an acyclic grammar (each rule of the unfolding – that represents an application of a rule of the original grammar – can be applied at most once). We will show that the unfolding constructed inductively is actually the union of all concurrent derivations. The big advantage of the unfolding semantics with respect to the concurrent semantics is that conflicts are described explicitly.

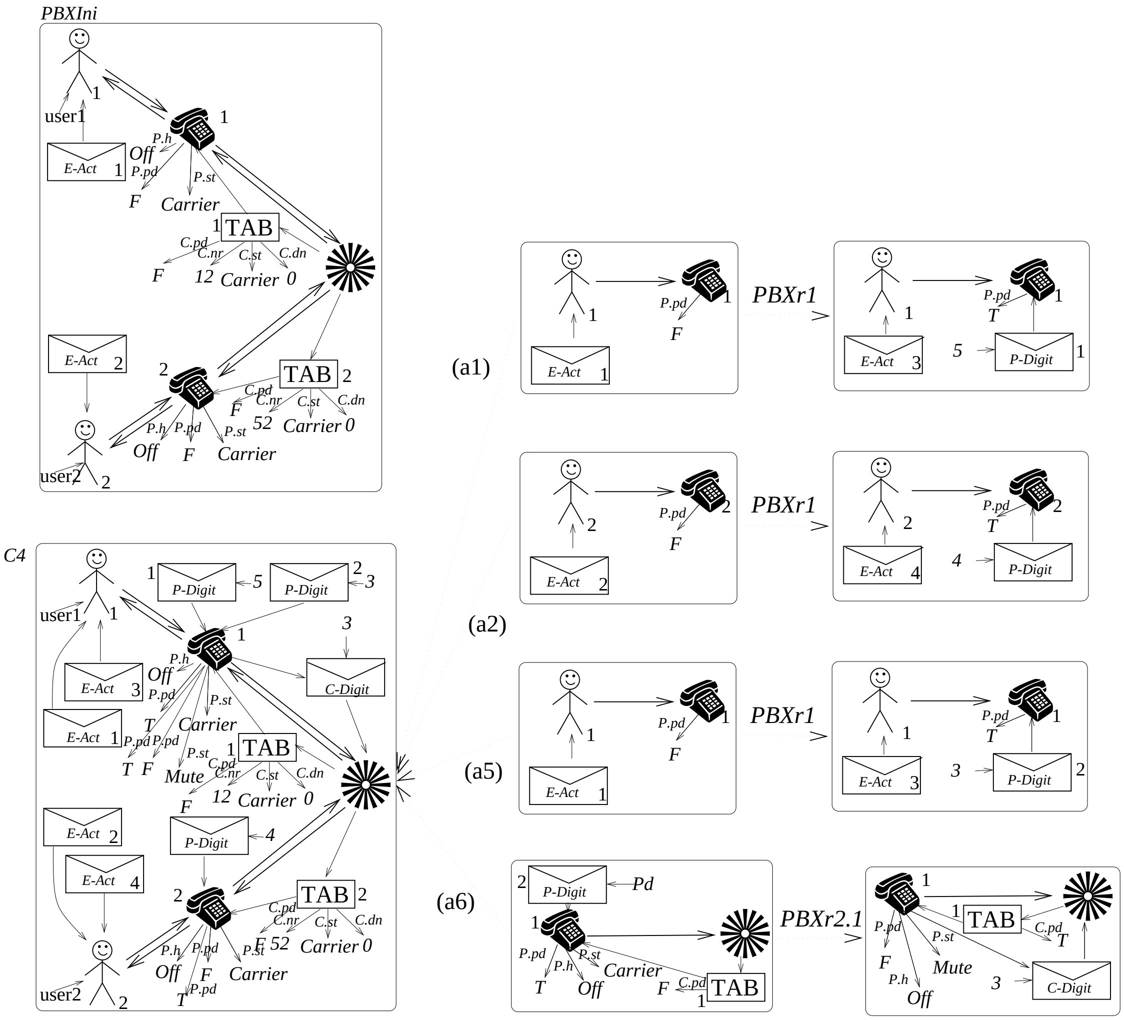


Figure 2.9: Part of the Unfolding of *CGV*

Figure 2.9 shows the (part of) the unfolding of the *CGV* grammar that includes the concurrent derivations $\kappa 1$ to $\kappa 5$. We can notice that there are only 4 actions (that are exactly the actions involved in the concurrent derivations). From the overlappings of these actions in the core graph, we can derive (among others) the following relationships between actions:

- **Causal Dependency**: based on overlappings of post- with pre-conditions of actions on created/deleted items. ($a5$ creates an item that is needed by $a6$.)

$$a5 \leq a6$$

- **Conflict:** based on overlappings of pre- with pre-conditions of actions on deleted items. ($a1$ and $a5$ delete the same item, as $a1$ is in conflict with $a5$ and $a6$ depends on $a5$, $a1$ in also in conflict with $a6$.)

$$a1 \overset{\#}{\Longleftrightarrow} a5, a1 \overset{\#}{\Longleftrightarrow} a6$$

3

Graph Grammars

In this chapter the basic definitions of *Single-Pushout graph grammars*[Löw90, Löw93], short SPO graph grammars, will be reviewed for the case of *typed graphs* [Kor93, CMR96a].

For practical applications, often graphs consisting only of vertices and edges lead to very low-level specifications of states (everything has to be coded into these two kinds of entities). Therefore many efforts had been made enriching the concept of a graph. The most basic enrichment is to add a label alphabet to a graph, over which vertices and edges may be labeled. A more sophisticated extension is to use variables and terms of some algebraic specification as attributes for vertices and edges. These kind of graphs are called attributed graphs [LKW93], and are of great practical relevance, not only because the readability of the graph grammar specification is increased but also because it reduces significantly the number of rules that are necessary to specify some problem. Other possible extension is to use different kinds of edges, like ,e.g., hyperedges [Hab92] or edges between edges [Löw90]. As we saw in the description of the PBX system (Chap. 2), graphs are very suitable to describe types and their relationships within a system. Formally, this can is captured by the notion of *typed graphs* [Kor93, CMR96a]. The basic idea of a typed graph is that, instead of a set of labels, a graph (called type graph) is used as a label for another graph, and the labeling mechanism, called typing here, must preserve the sources and targets of edges. Practically, this has the advantage that some inconsistent states of a system are ruled out by the typing (in)compatibility, i.e., the concept of typing offered by typed graphs is stronger than the the one offered by labeled graphs. For example, if there is no edge between two vertices in the type graph, there can not be a state of the corresponding grammar in which instances of these two vertices are connected by an edge. This kind of restriction can not be done directly by using labeled graphs (it must be required additionally). This kind of consistency check is used, for example, in the PROGRESS system [Sch91] (based on a programmed approach to graph grammars), where typed graphs have been successfully used for many applications. Theoretically, the use of type graphs brings the advantage that instances and types are represented in a uniform way. In this chapter we will show how this typing concept can be used as a basis for relationships between different graph grammars (these relationships, given by morphisms, will be defined in the next chapter). The theory we developed will be based on graph grammars in which states are described by typed graphs and rules by typed graph morphisms. Here we will use as type graph a simple graph (i.e., a graph consisting of a set of vertices, a set of edges and source and target functions), but we believe that the results can be generalized for other kinds of type graphs, for example, for attributed graphs or hypergraphs.

The main aims, definitions and results presented in this chapter are:

- Definition of basic notions of graph grammars: typed graphs (Sect. 3.1), rules (Sect. 3.2), graph grammars and their behaviour (Sect.3.3).
- Establishment of a relationship between type graph morphisms and correspondingly typed graphs: this is given by a *retyping construction* (Def. 3.8) that turns out to be a functor (Prop. 3.11). This is a very important result because it allows for a definition of relationships between graph grammars that are induced by relationships between the corresponding type graphs.
- Establishment of a relationship between type graph morphisms and sets of rules over these types: this is given by a functor transforming rules with respect to one type into rules with respect to the other type (Def. 3.17). Moreover, as rules will be translated up to isomorphism, this allows for a representation independent definition of relationships between graph grammars.
- Definition and discussion about the semantics of graph grammars (Sect. 3.3): here we will present the basic idea of changes of states specified by graph grammars, that are derivation steps. These can be combined sequentially, giving raise to a sequential semantics for graph grammars. Recently, another kind of semantics for SPO graph grammars was proposed in which the emphasis is put on describing the concurrency of the grammar. This semantics is based on concurrent, rather than sequential, derivations [Kor95, Kor96].

3.1 Typed Graphs

A graph G consists of a set V of vertices and a set E of edges connecting the vertices of V (this connections are expressed by total functions $source, target : E \to V$ assigning to each edge its source and target vertices respectively). A graph morphism expresses a structural compatibility between graphs. This structural compatibility may be total or partial, given by total and partial morphisms respectively. A graph morphism $f : G \to G'$ from a graph G into a graph G' consist of two components: a function f_V mapping vertices of G into vertices of G' and a function f_E mapping edges of G into edges of G'. These components must obey some compatibility restrictions: every edge that is mapped by f_E must be compatible with the mapping of its source and target vertices by f_V. Graphs can be seen as algebras with respect to a signature having two sorts (vertices and edges) and two operations (source and target functions from edges to vertices), and graph morphisms can be seen as (total or partial) algebra homomorphisms. This view on graphs as algebras gave raise to the *algebraic approach to graph grammars* [Ehr79].

We will use the Single-PushOut (SPO) approach to graph grammars, that is one of the algebraic approaches. The basic concept of a rule in the SPO approach is described by a partial graph morphism. The SPO approach was developed by Löwe in [Löw90, Löw93], and there it was described using a category of graphs seen as algebras with respect to (special) algebraic specifications and partial morphisms described by a total morphism from a subgraph of the source graph. To keep this work more comprehensible, we will avoid speaking about algebras, and will rather see graphs as tuples consisting of two sets (vertices and edges) and two total functions (source and target of edges). Relationships between graphs will be expressed by a pair of partial functions mapping vertices and edges that are weakly commuting with source and target functions (in [Kor96] it was shown that using a pair of weakly commuting functions yields the same category as the original definition of category of graphs and partial morphisms in [Löw90]).

Definition 3.1 (Weak Commutativity) *For a (partial) function $f : A \to B$ with domain $dom(f)$, let $f^{\blacktriangledown} : A \leftarrow dom(f)$ and $f! : dom(f) \to B$ denote the corresponding domain inclusion and the domain restriction. $(f^{\blacktriangledown}, f!)$ is called the* ***span*** *of f.*

$$\begin{tikzcd}
A \arrow[d, "a"'] & dom(f) \arrow[l, hook', "f^{\blacktriangledown}"'] \arrow[r, maps to, "f!"] \arrow[d, phantom, "="] & B \arrow[d, "b"] & & A \arrow[r, "f"] \arrow[d, "a"'] \arrow[dr, phantom, "\geq"] & B \arrow[d, "b"] \\
A' \arrow[rr, "f'"'] & & B' & \Longleftrightarrow & A' \arrow[r, "f'"'] & B'
\end{tikzcd}$$

Given functions as shown above, where a and b are total, we write $f' \circ a \geq b \circ f$ and say that the diagram ***commutes weakly*** *iff $f' \circ a \circ f^{\blacktriangledown} = b \circ f!$.* ☺

Remarks.

1. *If f is an injective function, its inverse is usually denoted by $(f)^{-1}$. The inverse of the domain restriction $f^{\blacktriangledown}$ is an injective and partial function. In particular $f = f! \circ (f^{\blacktriangledown})^{-1}$, and thus $((f^{\blacktriangledown})^{-1}, f!)$ is a* ***factorization*** *of f.*

2. *If f and f' are total, weak commutativity coincides with commutativity.*

☺

The compatibility condition defined above means that everything that is preserved (mapped) by the morphism must be compatible. The term "weak" is used because the compatibility is just required on preserved items, not on all items.

We assume that the reader is familiar with basic notions of category theory [AHS90, BW90]. Set-theoretical characterizations of all categorical constructions that we use can be found in Appendix B.

The mathematical notation can be found in Appendix A. By default, morphisms are *partial*, i.e., *morphism* denotes a partial morphism and *total morphism* denotes a total morphism.

With the notion of weak compatibility defined above, we can now define graphs and partial graph morphisms.

Definition 3.2 (Graph, Graph Morphism) *A* ***graph*** *$G = (V_G, E_G, source^G, target^G)$ consists of a set of vertices V_G, a set of edges E_G, and total functions $source^G$ and $target^G : E_G \to V_G$, called source and target operations respectively. $x \in G$ denotes an item $x \in V_G \cup E_G$. A graph is finite if V_G and E_G are finite.*

A ***(partial) graph morphism*** *$g : G \to H$ from a graph G to a graph H is a tuple $g = (g_V, g_E)$ consisting of two partial functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ which are weakly homomorphic, i.e., the diagrams below commute weakly.*

$$\begin{tikzcd} E_G \arrow[r, "g_E"] \arrow[d, "source^G"'] \arrow[dr, phantom, "\geq"] & E_H \arrow[d, "source^H"] \\ V_G \arrow[r, "g_V"'] & V_H \end{tikzcd} \qquad \begin{tikzcd} E_G \arrow[r, "g_E"] \arrow[d, "target^G"'] \arrow[dr, phantom, "\geq"] & E_H \arrow[d, "target^H"] \\ V_G \arrow[r, "g_V"'] & V_H \end{tikzcd}$$

A morphism g is called total, injective, surjective, inclusion or empty if both components are total, injective, surjective, inclusions or empty functions, respectively. A graph G is a subgraph of H, written $G \subseteq H$ if there is an inclusion $g : G \to H$.

The category of graphs and partial graph morphisms is denoted by **GraphP** *(identities and composition are defined componentwise). The subcategory of* **GraphP** *consisting of all graphs and total graph morphisms is denoted by* **Graph**. ☺

Remarks.

1. *Note that if commutativity instead of weak commutativity would be required, a morphism would not allow that an edge is not mapped if its source or target vertices are mapped.*

2. *If g is an inclusion it is denoted by $g^{\blacktriangledown}$. Like partial functions, each partial graph morphism f has a span representation $(f^{\blacktriangledown}, f!)$ where the components are componentwise domain inclusions and restrictions, and $((f^{\blacktriangledown})^{-1}, f!)$ is a factorization of f.*

☺

Example **3.3 (Graph,Graph Morphism)** The graph shown in Figure 3.1 is given by $G = (V, E, source, target)$, where $V = \{$☎, ✹, P-Digit, C-Digit, $3, 4\}$, $E = \{a, b, c, d, e, f, g\}$, $source = \{a \mapsto$ ☎, $b \mapsto$ ✹, $c \mapsto$ P-Digit, $d \mapsto 3, e \mapsto$ ☎, $f \mapsto$ C-Digit, $g \mapsto 4\}$, $target = \{a \mapsto$ ✹, $b \mapsto$ ☎, $c \mapsto$ ☎, $d \mapsto$ P-Digit, $e \mapsto$ C-Digit, $f \mapsto$ ✹, $g \mapsto$ C-Digit$\}$.

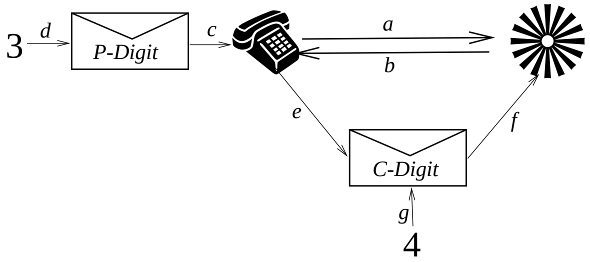


Figure 3.1: Graph

Graphical Notation: To simplify the graphical representation, we will usually only write the name of an edge in the picture if we need to differentiate it from another one (and the information about source and target is not enough) or if we want to make some comment on this edge. To differentiate vertices with the same graphical representation we will usually index them with numbers.

Figure 3.2 shows two mappings (f and g). The mapping f is a (partial) graph morphism. The mapping g is not a graph morphism because the source vertex of edge c is not mapped and source and/or target of the edges a and b are not mapped compatibly, i.e.,

$$\begin{aligned} f_V(source^G(c)) = undef \neq \text{✺} &= source^{G'}(g_E(c)) \\ f_V(target^G(a)) = \boxed{\text{P-Digit}}_1 \neq \boxed{\text{P-Digit}}_2 &= target^{G'}(f_E(a)) \\ f_V(source^G(b)) = \text{☎}_1 \neq \text{☎}_2 &= source^{G'}(f_E(b)) \\ f_V(target^G(b)) = undef \neq \text{✺} &= target^{G'}(f_E(b)) \end{aligned}$$

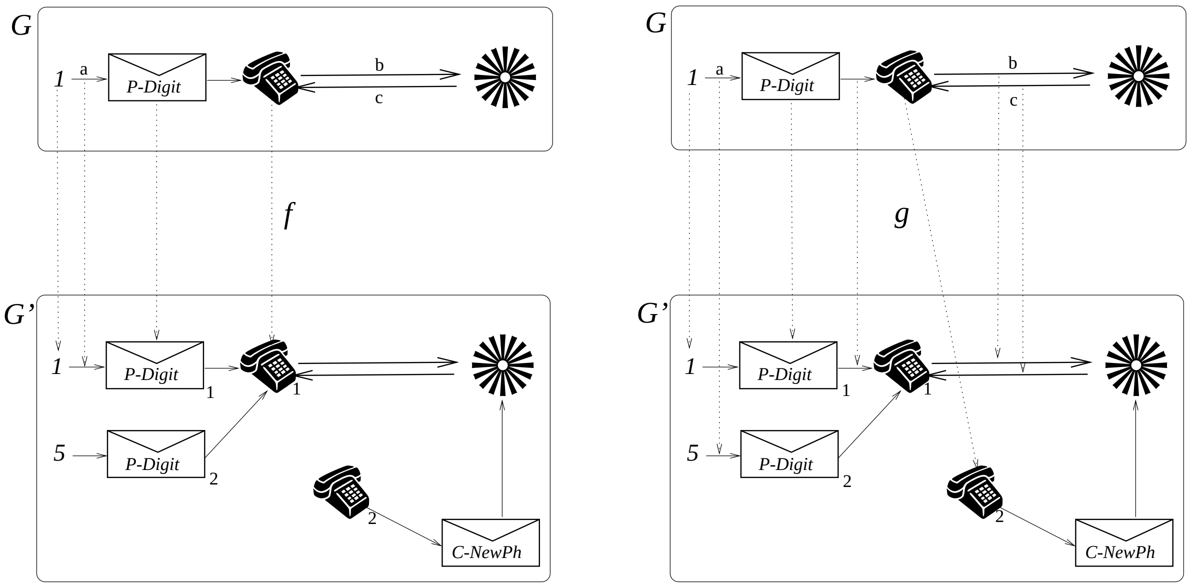


Figure 3.2: f: Graph Morphism g: Not a Graph Morphism

Graphical Notation: As the mapping of edges must be compatible with the mapping of vertices we will sometimes omit this mapping (when it is clear from the context). Items that are preserved via a morphism (are in the domain) will be indicated by corresponding indexes in source and target graphs (if there is no confusion, these indexes are omitted). ☺

Theorem 3.4 *The categories* **Graph** *and* **GraphP** *have limits and colimits and the inclusion functor $\mathcal{I}$:* **Graph** $\to$ **GraphP** *preserves colimits and pullbacks.* ☺

Proof. It is well known that **Graph** has limits and colimits. The proof that **GraphP** has limits and that $\mathcal{I}$ preserves pullbacks can be found in [Löw90]. The proof that **GraphP** has colimits can be found in [Kor96], where it is also shown that $\mathcal{I}$ preserves colimits. The proof that **GraphP** has limits and that $\mathcal{I}$ preserves pullbacks can be found in [Löw90]. $\surd$

Remark. *The inclusion functor $\mathcal{I}$ does not preserve initial objects, and therefore it does not preserve arbitrary limits (in particular, products are not preserved).* ☺

Definition 3.5 (Typed Graph) *A **typed graph** G^T is a tuple $G^T = (G, t^G, T)$ where G and T are graphs and $t^G : G \to T$ is a total graph morphism.*

Usually we will use $x \in G^T$ meaning that x is a vertex or an edge of G.

*A **typed graph morphism** $g^t : G^{T1} \to H^{T2}$ between typed graphs G^{T1} and H^{T2} is a pair of graph morphisms $g^t = (g, t)$ with $g : G \to H$ and $t : T1 \to T2$ such that the diagram below commutes weakly in* **GraphP**, *i.e., $t \circ t^G \circ g^{\blacktriangledown} = t^H \circ g!$.*

$$\begin{tikzcd}
G \arrow[r, "g"] \arrow[d, "t^G"'] \arrow[dr, phantom, "\geq"] & H \arrow[d, "t^H"] \\
T1 \arrow[r, "t"'] & T2
\end{tikzcd}$$

*A typed graph morphism is called an injective/surjective/total if both components are injective/surjective/total. A typed graph **automorphism** is an isomorphism $f^t : G^T \to G^T$ that is different from the identity on G^T.*

The category of typed graphs and typed graph morphisms is denoted by **TGraphP**. *If we fix one type graph T, a subcategory of* **TGraphP** *denoted by* **TGraphP(T)**, *is obtained having as objects all typed graphs over T and as morphisms all morphisms of* **TGraphP** *in which the type component is the identity. The restriction of the categories above to total morphisms are denoted by* **TGraph** *and* **TGraph(T)**, *respectively.* ☺

Notation: If it is clear from the context, we will denote a typed graph G^T by G, and a morphism $f^{id_T} = (f, id_T)$ in **TGraphP** by f^T.

Example **3.6 (Typed Graph, Morphism)** Let us consider the graph T shown in Figure 3.3 as the type graph of our system. Then an actual graph I is an instance of this type graph, i.e., a graph in which possibly many occurrences of each type occur. To describe the relationship between an instance I and the type graph T, we use a total graph morphism $t^I : I \to T$. This morphism describes the typing of all vertices and edges of the instance graph, and assures that the instance graph is consistent with respect to the type graph (via the compatibility requirements of morphisms). A typed graph will usually be denoted by I^T, where I is the instance graph having T as type.

Graphical Notation: Usually we will indicate the typing morphism by giving the same graphical symbols to vertices in the instances and in the type graph.

As graphs represent states, it is natural that state changes are represented by relationships between graphs. Again, we may use graph morphisms to express this relationships. In this case, we require that not only vertices and edges are mapped compatibly, but also that the typing information is preserved. This means, for example, that vertex of type ☎ can not be mapped via a morphism to a vertex of type ✳.

Figure 3.4 depicts a typed graph morphism f^T between the (typed-) graphs I^T and I'^T. This morphism deletes ☎$_2$, C-Digit and the corresponding edges (i.e., these items are not in the domain of f). Phone ☎$_1$, ✳ and P-Digit are preserved by the morphism (are in the domain of f).

Graphical Notation: As it is clear from the context, we will usually just indicate the existence of graph morphisms by an arrow between the corresponding graphs. When necessary, items in the

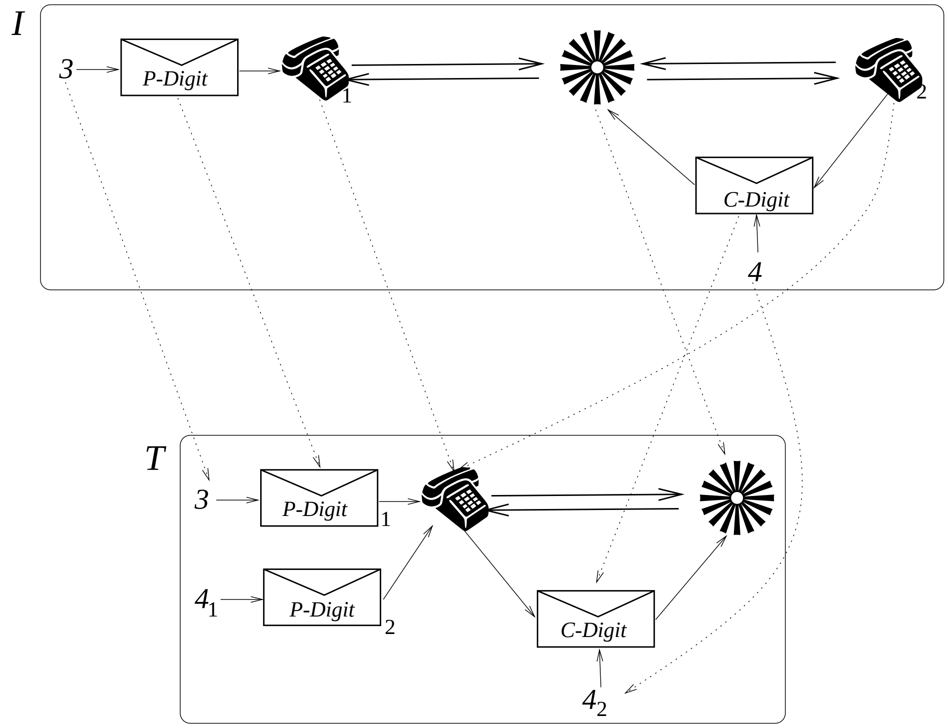


Figure 3.3: Typed Graph I^T

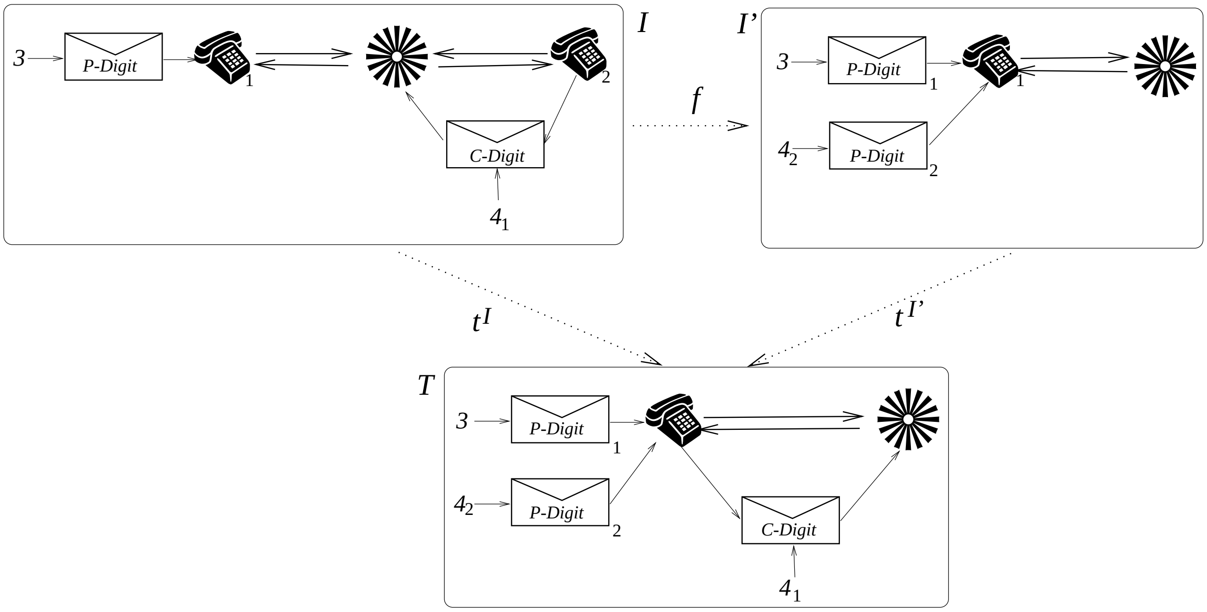


Figure 3.4: Typed Graph Morphism f^T

source and target of the morphisms will get indexes to avoid confusion. We will also often omit the type-graph if the mapping of instances into it is clear. ☺

The categorical constructions in the categories of typed graphs that are relevant for this work are given in Appendix. B.4.

In the next sections, we will be interested in relationships between graph grammars that may have different type graphs. Thus it is important to define how a graph (morphism) with respect to one type can be converted to a graph (morphism) with respect to another type based on a type morphism. Roughly speaking, the conversion of a typed graph $G1^{T1}$ into a typed graph $G2^{T2}$ based on a type graph morphism $f : T2 \rightarrow T1$ yields that $G2$ is the biggest subgraph of $G1$ whose elements are typed over elements of the type graph $T1$ that are in the range of f (in fact, if f is not injective, $G2$ may not be a subgraph of $G1$ because some elements of $G1$ may be splited in $G2$).

The *retyping* of graphs and morphisms must have some properties such that the preservation of derivation steps (and sequences) by graph grammar morphisms can be assured. The kind of retyping presented here is similar to the one presented in [CEL$^+$96a] for the DPO-approach. The basic differences are that the construction given below is a functor between categories of typed graphs and partial morphisms and that the morphism that induces this functor is a partial morphism, instead of a span of total morphisms as in [CEL$^+$96a]. In [CH95, HCEL96] a partial morphism was also used (in the DPO context) to induce the retyping of graphs, but there a partial morphism in the opposite direction as the one presented here was used. An important remark is that the retyping functor induced by a morphism is contravariant with respect to this morphism. This means that the retyping is done in fact in the opposite direction from the graph morphism that induced the retyping. The advantage of this kind of retyping is that it allows to split types (this splitting allows for the definition of suitable products in the category of graph grammars – this will be presented in Chap. 4).

The definition of the retyping construction will be based on pullbacks in the category **GraphP**. Pullbacks are only unique up to isomorphism and to define a deterministic construction we need exactly one result. Thus, we will use a "choice of pullbacks" [CEL$^+$96a], but without assuming associativity. A choice of pullbacks means that we take a concrete pullback for each pullback diagram. For the rest of the constructions, it is not relevant which is the concrete choice that was done to define the retyping, but that one was done. Therefore we will not take care of defining concretely such a choice of pullbacks.

Definition 3.7 (Choice of Pullbacks) *A* ***choice of pullbacks*** *in a category* **Cat** *that has pullbacks is a fixed pullback* $\overline{PB}(G3 \xrightarrow{g} G1 \xleftarrow{f} G2) = (G1 \xleftarrow{f^\bullet} G4 \xrightarrow{g^\bullet} G2)$ *for each diagram* $(G3 \xrightarrow{g} G1 \xleftarrow{f} G2)$ *in* **Cat**. ☺

General Assumption **3.1** *Let* $\overline{PB}$ *be a choice of pullbacks in the category* **GraphP**. ☺

Remark. *The pullbacks that we will construct in categories of graphs will be always from total graph morphisms. The construction of pullbacks in* **Graph** *is done componentwise (see Appendix B). In [Löw90] it was shown that the category* **GraphP** *has pullbacks and that the inclusion functor* $\mathcal{I}$: **Graph** $\rightarrow$ **GraphP** *preserves pullbacks. Although the pullbacks here* *will be always of total graph morphisms, we will use the category* **GraphP** *because in some situations it is necessary to find an universal morphism induced by the pullback in which one of the comparison morphisms is partial.* ☺

Definition 3.8 (Retyping Functor) *Let $f : T2 \to T1$ be a morphism in* **GraphP**. *Then there is a functor $\mathcal{T}_f$:* **TGraphP(T1)** $\to$ **TGraphP(T2)**, *called* ***retyping functor***, *induced by f.*

$$\begin{tikzcd}
T1 & \mathbf{TGraphP(T1)} \arrow[d, "\mathcal{T}_f"] \\
T2 \arrow[u, "f"] & \mathbf{TGraphP(T2)}
\end{tikzcd}$$

$\mathcal{T}_f$ is defined for each object $(G1, t^{G1}, T1)$ and morphism $g^{T1} : G1 \to H1$ in **TGraphP(T1)** *as*

- **Objects:** *$\mathcal{T}_f(G1, t^{G1}, T1) = (G2, f^\blacktriangledown \circ t^{G1\bullet}, T2)$, where (1) below is a choice of pullback in* **Graph**, *i.e., $(G1 \overset{f!^G}{\leftarrow} G2 \overset{t^{G1\bullet}}{\to} dom(f)) = \overline{PB}(G1 \overset{t^{G1}}{\to} T1 \overset{f!}{\leftarrow} dom(f))$.*

$$\begin{tikzcd}
G1 \arrow[r, "t^{G1}", |->] & T1 & \\
G2 \arrow[u, "f!^G", |->] \arrow[r, "t^{G1\bullet}"', |->] \arrow[ur, phantom, "(1)"] & dom(f) \arrow[u, "f!"', |->] \arrow[r, hook, "f^\blacktriangledown"'] & T2
\end{tikzcd}$$

- **Morphisms:** *$\mathcal{T}_f(g1^{T1}) = g2^{T2}$, where $g2 = g2! \circ (g2^\blacktriangledown)^{-1}$ is defined as follows: Let (3) be the pullback in* **GraphP** *of $g^\blacktriangledown : dom(g1) \to G1$ and $f!^G : G2 \to G1$ where the pullback morphism $g2^\blacktriangledown : dom(g2) \to G2$ is an inclusion. The morphism $g2! : dom(g2) \to H2$ is obtained as the universal morphism induced by pullback (2).*

$$\begin{tikzcd}
 & & dom(g1) \arrow[dll, hook, "g1^\blacktriangledown"'] \arrow[drrr, |->, "g1!"] & & & \\
G1 \arrow[rrrrr, |->, "g1"] \arrow[drrr, |->, "t^{G1}"] & & & & & H1 \arrow[dll, |->, "t^{H1}"'] \\
 & (3) & & T1 & & \\
 & & (1) & & (2) & \\
 & & dom(g2) \arrow[uuuu, |->, "f!^\bullet"] \arrow[dll, hook, "g2^\blacktriangledown"'] \arrow[drrr, |->, "g2!"] & & & \\
G2 \arrow[uuuu, |->, "f!^G"] \arrow[rrrrr, |->, "g2"] \arrow[drrr, |->, "t^{G1\bullet}"'] & & & & & H2 \arrow[uuuu, |->, "f!^H"'] \arrow[dll, |->, "t^{H1\bullet}"] \\
 & & & dom(f) \arrow[uuuu, |->, "f!"'] & &
\end{tikzcd}$$

☺

Remarks.

1. *The morphism $g2^{\blacktriangledown}$ may be chosen as an inclusion because by pullback properties it shall be injective and total, and thus, an inclusion up to isomorphism.*

2. *The morphism $g2!$ is defined in the following way: As (3) is a pullback we have $g1^{\blacktriangledown} \circ f!^{\bullet} = f!^{G} \circ g2^{\blacktriangledown}$. This implies that $t^{G1} \circ g1^{\blacktriangledown} \circ f!^{\bullet} = t^{G1} \circ f!^{G} \circ g2^{\blacktriangledown}$. As $g1^{T1}$ is a morphism and (1) commutes we obtain that $t^{H1} \circ g1! \circ f!^{\bullet} = f! \circ t^{G2} \circ g2^{\blacktriangledown}$. Therefore as (2) is a pullback there is a universal morphism $g2! : dom(g2) \to H2$ such that $f!^{H} \circ g2! = g1! \circ f!^{\bullet}$ and $t^{H2} \circ g2! = t^{G2} \circ g2^{\blacktriangledown}$. The latter assures that $g2^{T2}$ is a morphism in* **TGraphP(T2)** *(it assures the weak commutativity requirement).*

☺

This retyping will be illustrated in the following example.

Example **3.9 (Retyping Construction)** Figure 3.5 shows three examples of retyping. The basic idea is that types that are in $T2$ but not in $T1$ do not affect the translation (see example (1)), instances of types that are in $T1$ and not in the image of f are forgotten (see example (2)), and instances of types that are duplicated are also duplicated (see example (3)). ☺

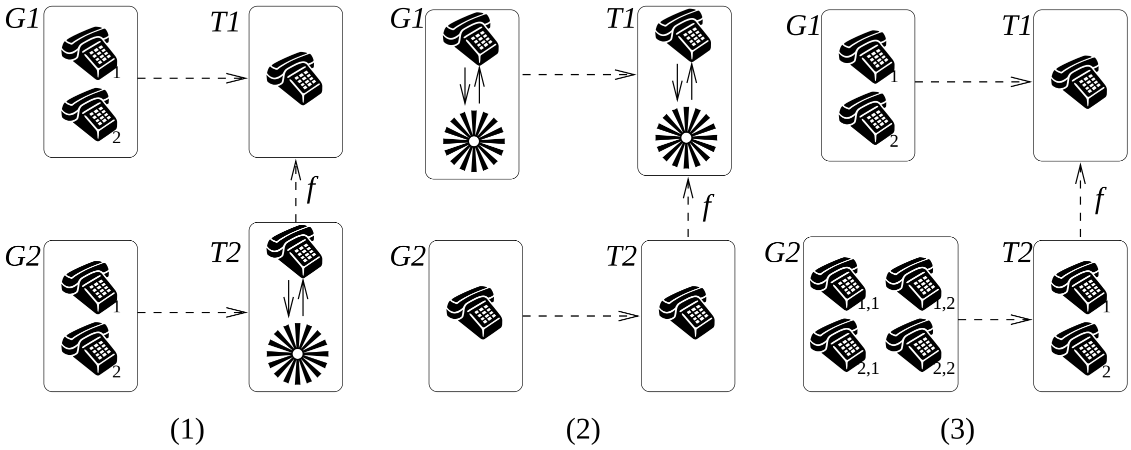


Figure 3.5: Retyping

Before we show that $\mathcal{T}_f$ is a functor, we will show that $g1 \circ f!^{G} = f!^{H} \circ g2$.

Proposition 3.10 *Let $g2^{T2} = \mathcal{T}_f(g1^{T1})$. Then the square (5) below commutes and square (4) is a pullback. If $g1$ is total then (5) is also a pullback.* ☺

$$\begin{tikzcd}
G1 \arrow[r, "g1"] & H1 \\
G2 \arrow[u, maps to, "f!^G"] \arrow[r, "g2"'] \arrow[ur, phantom, "(5)"] & H2 \arrow[u, maps to, "f!^H"']
\end{tikzcd}
\qquad
\begin{tikzcd}
 & dom(g1) \arrow[dl, hook, "g1^\blacktriangledown"'] \arrow[dr, maps to, "g1!"] & \\
G1 \arrow[rr, "g1"] \arrow[dr, phantom, "(3)"] & & H1 \\
 & dom(g2) \arrow[uu, maps to] \arrow[dl, hook, "g2^\blacktriangledown"'] \arrow[dr, maps to, "g2!"] \arrow[ur, phantom, "(4)"] & \\
G2 \arrow[uu, maps to, "f!^G"] \arrow[rr, "g2"'] & & H2 \arrow[uu, maps to, "f!^H"']
\end{tikzcd}
\qquad
\begin{tikzcd}
G1 \arrow[rr, "g1"] \arrow[dr, maps to, "t^{G1}"] & & H1 \arrow[dl, maps to, "t^{H1}"'] \\
 & T1 & \\
G2 \arrow[uu, maps to, "f!^G"] \arrow[rr, "g2"] \arrow[dr, maps to, "t^{G1\bullet}"'] \arrow[ur, phantom, "(1)"] & & H2 \arrow[uu, maps to, "f!^H"'] \arrow[dl, maps to, "t^{H1\bullet}"] \arrow[ul, phantom, "(2)"] \\
 & dom(f) \arrow[uu, maps to, "f!"'] &
\end{tikzcd}$$

Proof. First we will show that (5) commutes. Assume that $g1 \circ f!^G \neq f!^H \circ g2$. Then we have three cases:

1. $\exists e1 \in G2 : g1 \circ f!^G(e1) = \texttt{undef}$ and $f!^H \circ g2(e1) = e2$
   As $f!^G$ is total, there is $e3 = f!^G(e1) \in G1$. Then as $g1 \circ f!^G(e1) = \texttt{undef}$, we must have that $g1(e3) = \texttt{undef}$, i.e., $e3 \notin dom(g1)$. As $f!^H \circ g2(e1) = e2$, $e2 \in dom(g2)$. As (2) is a pullback and $g1^\blacktriangledown$ is an inclusion, there must be $e3 \in dom(g1)$. But this contradicts the fact that $g1(e3) = \texttt{undef}$.

2. $\exists e1 \in G2 : g1 \circ f!^G(e1) = e2$ and $f!^H \circ g2(e1) = \texttt{undef}$
   As $f!^H$ is total and $f!^H \circ g2(e1) = \texttt{undef}$, $e1 \notin dom(g2)$. As $g1 \circ f!^G(e1) = e2$, there is $e3 \in G1$such that $f!^G(e1) = e3$ and $g1(e3) = e2$. This means that $e3 \in dom(g1)$. As (2) is a pullback, there must be $e1 \in dom(g2)$, but this contradicts $e1 \notin dom(g2)$.

3. $\exists e1 \in G2 : g1 \circ f!^G(e1) = e2$, $f!^H \circ g2(e1) = e3$ and $e2 \neq e3$
   In this case $e1 \in dom(g2)$ and $f!^G(e1) = e4 \in dom(g2)$. As (2) is a pullback, $f!^{G\bullet}(e1) = e4$. As (3) must commute $f!^H \circ g2!(e1) = g1!(e4)$. By definition of $g1$ and $g2$ we have that $g2(e1) = g2!(e1)$ and $g1(e4)! = g1(e4)$ if $e1$ and $e4$ belong to the corresponding domains. Therefore $e2 = e3$.

As (3) and (1) are pullbacks, (3)+(1) is also a pullback. As $g1^{T1}$ is and $g2^{T2}$ are morphisms in $\mathbf{TGraphP(T1)}$ and $\mathbf{TGraphP(T2)}$, we have that $t^{G1} \circ g1^\blacktriangledown = t^{H1} \circ g1!$ and $t^{G1\bullet} \circ g2^\blacktriangledown = t^{H1\bullet} \circ g2!$. This implies that (3)+(1)=(4)+(2). Then as (4)+(2) and (2) are pullbacks and since (4) commutes, (4) is also a pullback.

If $g1$ is total then $g1! = g1$. This implies that $g2! = g2$ and as (4) is a pullback, (5) is also a pullback in this case. $\surd$

Proposition 3.11 *$\mathcal{T}_f$ is a well-defined functor.* ☺

Proof.

1. *$\mathcal{T}_f$ is well-defined*: Obviously, $\mathcal{T}_f(G1)$ is a typed graph in $\mathbf{TGraphP(T2)}$ ($G2$ is a graph and the typing morphism is total). As discussed in the remarks of Def. 3.8, $g2^{T2}$ is a morphism in $\mathbf{TGraphP(T2)}$.

2. $\mathcal{T}_t$ *preserves identities:* Let $g1^{T1} = id_{G1}^{id_{T1}}$. In this case, $G2 = H2$ and $f!^G = f!^H$ because they are pullback object and morphism of the same diagram. As the identity is total, $dom(id_{G1}) = G1$ and thus an isomorphism. Isomorphisms are inherited under pullbacks, what implies that $dom(g2) = G2$. Therefore the only possible morphism $g2!$ such that $t^{G2} \circ g2! = t^{G2} \circ id_{G2}$ and $f!^G \circ g2! = id_{G1} \circ f!^G$ is the identity of $G2$.

3. $\mathcal{T}_f$ *preserves composition:* Let $g1^{T1} : G1^{T1} \to H1^{T1}$ and $h1^{T1} : H1^{T1} \to I1^{T1}$ be morphisms in $\mathbf{TGraphP(T1)}$ and $(h1 \circ g1)^{T1}$ be their composition (by definition, obtained componentwise). Let $\mathcal{T}_f((h1 \circ g1)^{T1}) = k^{T2}$, $\mathcal{T}_f(h1^{T1}) = h2^{T2}$ and $\mathcal{T}_f(g1^{T1}) = g2^{T2}$. Then we have to show that $k^{T2} = h2^{T2} \circ g2^{T2}$. As the composition of morphisms is defined componentwise in $\mathbf{TGraphP(T2)}$, it suffices to show that $k = h2 \circ g2$.

$$\begin{tikzcd}
 & & dom(h1 \circ g1) \arrow[dll, hook, bend right] \arrow[dl, hook] \arrow[dr, mapsto] \arrow[drr, mapsto, bend left] \arrow[d, phantom, "(7)"] & & \\
G1 \arrow[dr, phantom, "(1)"] & dom(g1) \arrow[l, hook', "g1^\blacktriangledown"'] \arrow[r, mapsto, "g1!"] \arrow[dr, phantom, "(3)"] & H1 \arrow[dr, phantom, "(2)"] & dom(h1) \arrow[l, hook, "h1^\blacktriangledown"'] \arrow[r, mapsto, "h1!"] \arrow[dr, phantom, "(4)"] & I1 \\
G2 \arrow[u, mapsto, "f!^G"] & dom(g2) \arrow[u, mapsto] \arrow[l, hook, "g2^\blacktriangledown"] \arrow[r, mapsto, "g2!"'] & H2 \arrow[u, mapsto, "f!^H"'] & dom(h2) \arrow[u, mapsto] \arrow[l, hook', "h2^\blacktriangledown"] \arrow[r, mapsto, "h2!"'] & I2 \arrow[u, mapsto, "f!^I"'] \\
 & & dom(k) \arrow[ull, hook, bend left, "k^\blacktriangledown", "(8)"'] \arrow[ul, hook, "u1"'] \arrow[ur, mapsto, "u2"] \arrow[urr, mapsto, bend right, "k!"', "(9)"] \arrow[u, phantom, "(10)"] & &
\end{tikzcd}$$

By the definition of the retyping construction we have that (1), (2), (3) and (4) are pullbacks (see Prop. 3.10). As k is obtained by retyping $g1 \circ h1$, squares (5), with tips in $dom(g1)$, $dom(g2)$, $dom(h1 \circ g1)$ and $dom(k)$ (see below), and (6), with tips in $I1$, $I2$, $dom(h1 \circ g1)$ and $dom(k)$ are also pullbacks. Square (7) is a pullback because this is the characterization of the domain of a composed function. Based on pullbacks (1) and (4) we can find universal morphisms $u1 : dom(k) \to dom(g2)$ and $u2 : dom(k) \to dom(h2)$ such that (8) and (9) commute. As $g2^\blacktriangledown$ and $k^\blacktriangledown$ are inclusions and (8) commutes, $u1$ is also an inclusion. As $k!$ is total and (9) commutes, $u2$ is also total. As (3)–(7) commute (see diagrams above and below), (10) also commutes. Then as (5)+(7) and (2) are pullbacks and (10) commutes, (10) is also a pullback. The facts that (10) is a pullback and that $k^\blacktriangledown$ is an inclusion imply that $dom(k) = dom(h2 \circ g2)$. As (8), (9) and (10) commute we conclude that $k = h2 \circ g2$.

$$\begin{tikzcd}
 & & dom(h1 \circ g1) \arrow[dll, hook] \arrow[drr, "|"{marking}, ""] & & \\
dom(g1) \arrow[drrr, "|"{marking}] & & (7) & & dom(h1) \arrow[dl, hook] \\
 & (5) & & H1 & \\
 & & dom(k) \arrow[uuu] \arrow[dll, hook] \arrow[drr] & (6) & \\
dom(g2) \arrow[uuu] \arrow[drrr] & & & & dom(h2) \arrow[uuu] \arrow[dl, hook] \\
 & & & H2 \arrow[uuu] &
\end{tikzcd}$$

√

Proposition 3.12 states that the retyping functor preserves special pushouts (the ones that are considered as derivation steps). This fact will be used to prove that morphisms between graph grammars induce a translation of derivations (this will be shown in Sect. 3.3).

Proposition 3.12 *Let $r = (r1^{T1}) : L1 \to R1$ and $m = (m1^{T1}) : L1 \to G1$ be morphisms in* **TGraphP(TG1$^{\mathbf{T1}}$)** *where $r1$ is injective and $m1$ is total, and (1) below be a pushout in* **TGraphP(T1)** *of r and m. Let $f : T2 \to T1$ be a typed graph morphism. Then (2) is a pushout in* **TGraphP(T2)**.

$$\begin{tikzcd}
L1 \arrow[r, tail, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & R1 \arrow[d, "m^\bullet"] \\
G1 \arrow[r, tail, "r^\bullet"'] & H1
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{T}_f(L1) \arrow[r, tail, "\mathcal{T}_f(r)"] \arrow[d, "\mathcal{T}_f(m)"'] \arrow[dr, phantom, "(2)"] & \mathcal{T}_f(R1) \arrow[d, "\mathcal{T}_f(m^\bullet"] \\
\mathcal{T}_f(G1) \arrow[r, tail, "\mathcal{T}_f(r^\bullet)"'] & \mathcal{T}_f(H1)
\end{tikzcd}$$

☺

Proof. See Appendix C. √

Proposition 3.13 states that the retyping construction is (up to isomorphism) compatible with the composition of morphisms in the category **GraphP**. This is important for the well-definedness of the category of graph grammars (see Sect. 3.3).

Proposition 3.13 *Let $f : T1 \to T2$ and $g : T2 \to T3$ be morphisms in* **GraphP**. *Then*

$$\mathcal{T}_{g \circ f} \cong \mathcal{T}_g \circ \mathcal{T}_f$$

☺

Proof. See Appendix C. $\surd$

A stronger statement would be $\mathcal{T}_{g \circ f} = \mathcal{T}_g \circ \mathcal{T}_f$. This would require that the choice of pullbacks (Def. 3.7) on which the retyping functor is based is associative. As it is not clear whether there is a choice of pullbacks that is associative, we prefer to skip this requirement.

3.2 Rules

Graphs may be used to represent states of a system. In the graph grammar formalism, state changes are specified via *rules*. Rules describe a state change in the following way: the left-hand side describes what has to be present in a state such that the rule can be applied, and the right-hand side describes the changes that occur in the state via the application of the rule. The relationship between left- and right-hand sides expresses the basic operations involved in a change of state, namely *preservation*, *deletion* and *addition* of items by the rule. In the SPO approach to graph grammars, the relationship between left- and right-hand sides is described by a (partial) morphism.

The idea of using a partial morphism to describe a change of state is quite simple. Let $r : L \to R$ be a partial morphism. The basic operations involved in a change of state are described as follows:

Deletion : Everything that is in L and is not mapped to R via r is deleted.

Addition : Everything that is in R and is not in the range of r is created.

Preservation : Everything that is mapped via r is preserved by the rule.

We will put two restrictions on rules: the first restriction is that rules shall not identify items and the second is that rules shall delete something. Although these restrictions were motivated by theoretical reasons (some of the definitions/results would not be achieved without these restrictions), they are also reasonable from a practical point of view. The first restriction is formalized by requiring that rules must be injective. This allows rules to be "inverted", what is a necessary property for the construction of the concurrency semantics of grammars and for the construction and interpretation of the unfolding of grammars. The restriction to injective rules does not represent a strong limitation for many practical applications. The main purpose of the graph grammars that we have in mind in this thesis is the specification of concurrent and reactive systems, and not the generation of (graph) languages. Examples of reactive systems are object-oriented systems, actor systems and other systems in which message passing is the basis for calling and performing operations. Practically, a reaction to some message is often specified by deleting the message and doing the required actions. Non-deleting rules in this context would mean that (if the message requiring a reaction is not deleted) there may be infinitely many (isomorphic) reactions for the same action. If we then think that this system is concurrent, all these reaction are not in conflict with each other (because nothing is deleted) and may thus occur in parallel. Theoretically, if we are interested in true-concurrency semantics, non-deleting rules are problematic because they lead to interpretation problems with respect to the causal dependencies between actions of a grammar (also in other formalisms, e.g., Petri nets, the existence of transitions without pre-conditions is forbidden in order to obtain true-concurrency semantics models as unfoldings or event structures).

Definition 3.14 (Rule) *Let T be a graph. Then a* **rule** *with respect to T is a morphism $r^T : L^T \to R^T$ in* $\mathbf{TGraphP(T)}$ *iff*

1. *r^T is injective and*
2. *r^T is consuming (not total).*

The class of all rules with respect to one type graph is denoted by $Rules(T)$. If we replace the second requirement by

$2'$. r is not an isomorphism

we obtain a **general rule**. *The class of all general rules with respect to a type graph T is denoted by $GRules(T)$. Obviously, there is an inclusion $i : Rules(T) \to GRules(T)$. The sets of isomorphism classes of rules and of general rules with respect to a type graph T are denoted by $IRules(T)$ and $IGRules(T)$. The function c is defined for all $r \in Rules(T)$ as $c(r) = [r]$. The composition of $gc \circ i = gi \circ c$ is denoted by ir.*

$$\begin{tikzcd}
Rules(T) \arrow[r, hook, "i"] \arrow[d, "c"'] \arrow[dr, "ir"] & GRules(T) \arrow[d, "gc"] \\
IRules(T) \arrow[r, hook, "gi"'] & IGRulesx(T)
\end{tikzcd}$$

☺

Remark. *General rules were defined for technical reasons. For the definition of graph grammars, only rules are allowed, but the set $IGRules(T)$ of isomorphism classes of general rules is used to define the morphisms between graph grammars.* ☺

Example **3.15 (Rule)** The rule described in Figure 3.6 deletes the P-Digit and 4_1 vertices, preserves the PHONE and the CENTRAL and creates a C-Digit and a 4_2 vertices (obviously, the corresponding edges are also deleted/preserved/created). The type graph of this rule is the graph T of Figure 3.3. ☺

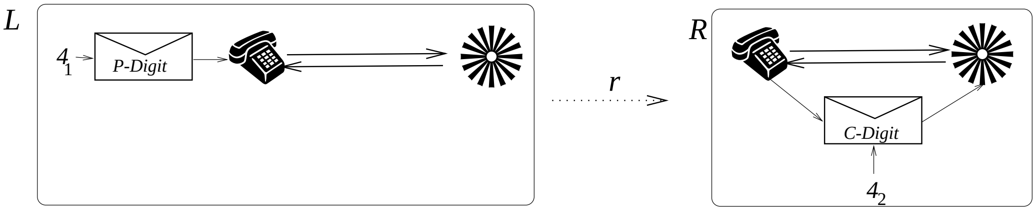


Figure 3.6: Rule

Definition 3.16 (Subrule) *Let $r1 : L1 \to R1$ and $r2 : L2 \to R2$ be rules over type T. Then $r1$ is a* **subrule** *of $r2$, denoted by $r1 \subseteq^r r2$, iff there are total and injective morphisms*

$i_L : L1 \to L2$ and $i_R : R1 \to R2$ such that the diagram below is a pushout in $\mathbf{TGraphP(T)}$

$$\begin{tikzcd}
L1 \arrow[r, "r1"] \arrow[d, hook, "i_L"'] \arrow[dr, phantom, "PO"] & R1 \arrow[d, hook, "i_R"] \\
L2 \arrow[r, "r2"'] & R2
\end{tikzcd}$$

and the safety condition below is satisfied by i_L and i_R.

*(Safety Condition) A total and injective morphism $i_L : L1 \to L2$ satisfies the **safety condition** iff for all element $e \in L2$: if $t^{L2}(e) \in rng(t^{L1})$ then there exists $p \in L1$ such that $i_L(p) = e$.*

Two rules $r1$ and $r2$ are **isomorphic**, *denoted by $r1 \cong r2$, if $r1 \subseteq^r r2$ and i_L and i_R are isomorphisms. The pair (id_L, id_R) is called the **rule-identity**. Let $r1 \subseteq^r r1$ using isomorphisms i_L and i_R, then the pair (i_L, i_R) is called **rule-automorphism** if (i_L, i_R) is not the rule-identity of $r1$. We denote by $[r1] \subseteq^R [r2]$ the extension of the subrule relationship to isomorphism classes of rules.* ☺

Remarks.

1. *The safety condition assures that $L2$ does not contain any element that has the same type as an element in the image of i_L and is not in the image of i_L.*

2. *The relation $[r1] \subseteq^R [r2]$ is well-defined because if $r1 \subseteq^r r2$ and $r2 \cong r2'$ then $r1 \subseteq^r r2'$ because total and injective morphisms compose, the composition of pushouts is again a pushout (if $r2 \cong r2'$ then the diagram above is trivially a pushout) and the condition on elements is transitive.*

3. *We will sometimes use the notion of a subrule when considering general rules or even isomorphisms.*

☺

Using the construction of sets of (isomorphism classes of) rules over some type, we can define a functor called *Rules functor*. The idea is to associate with each type graph its set of rules and with each type graph morphism the corresponding translation of rules (given by the retyping functor). As the retyping of graphs and morphisms is done in the opposite direction of the direction of the type graph morphism, the rules functor will be a contravariant functor. That is, it transforms objects and morphisms from the dual (or opposite) category of the category **GraphP**, called $\mathbf{GraphP^{OP}}$, into objects and morphisms of **SetP**. The category $\mathbf{GraphP^{OP}}$ has the same objects as **GraphP** and, except for the direction, the same morphisms of **GraphP** (see Appendix B).

Definition 3.17 (Rules Functor) *$IGRules$ extends to a functor $\mathcal{R} : \mathbf{GraphP^{OP}} \to \mathbf{SetP}$, defined for all objects $T1, T2$ and morphism $f^{OP} : T1 \to T2$ in $\mathbf{GraphP^{OP}}$ as follows*

- **Objects:** $\mathcal{R}(T1) = IGRules(T1)$

- **Morphisms:** $\mathcal{R}(f) = \mathcal{R}_f : IGRules(T1) \to IGRules(T2)$ *is defined for all* $[r] \in IGRules(T1)$ *as*
$$\mathcal{R}_f([r]) = \begin{cases} [\mathcal{T}_f(r)], \text{ if } \mathcal{T}_f(r) \text{ is not an isomorphism,} \\ \texttt{undef}, \text{ otherwise} \end{cases}$$

☺

Proposition 3.18 *$\mathcal{R}$ is well-defined.* ☺

Proof. By definition, $\mathcal{R}(T1)$ is a set and $\mathcal{R}(f)$ is a partial function. If f is the identity then, $\mathcal{R}(f)([r]) = [\mathcal{T}_f(r)] = [r]$ for all $[r] \in IRules^+(T1)$ by the definition of $\mathcal{T}_f$ and the fact that applying $\mathcal{T}_f$ to each general rule $r1^{T1}$ yields a general rule $r2^{T2}$ such that $r1 \cong r2$ (Lemma C.2). Let $f : T1 \to T2$ and $g : T2 \to T3$. By Prop. 3.13 we have that $\mathcal{T}_{g \circ f} \cong \mathcal{T}_g \circ \mathcal{T}_f$. Therefore $[\mathcal{T}_{g \circ f}(r)] = [\mathcal{T}_g \circ \mathcal{T}_f(r)]$. $\surd$

Proposition 3.19 *Let $f^{OP} : T1 \to T2$. Then $\mathcal{R}_f$ preserves subrules, that is if $[r1], [r1'] \in dom(\mathcal{R}_f)$ and $[r1] \subseteq^R [r1']$ then $\mathcal{R}_f([r1]) \subseteq^R \mathcal{R}_f([r1'])$.* ☺

Proof. By the definition of subrules (Def. 3.16) there are total and injective morphisms i_L and i_R connecting the left- and right-hand sides of $r1$ and $r1'$ such that the square obtained this way is a pushout and the safety condition is satisfied. Lemma C.1 assures that there are total and injective $\mathcal{T}_f(i_L)$ and $\mathcal{T}_f(i_R)$ between the corresponding retyped rules. Moreover, as $\mathcal{T}_f$ is a functor, the translated square commutes and Prop. 3.12 assures that it is also a pushout (because i_L is total and $r1$ is injective). As the safety condition is satisfied by iL and iR and the retyping yields exactly the translation of the source and target graphs of iL and iR (without any other elements), we conclude that $\mathcal{T}_f(i_L)$ and $\mathcal{T}_f(i_R)$ also satisfy the safety condition and thus $\mathcal{R}_f([r1]) \subseteq^R \mathcal{R}_f([r1'])$. $\surd$

Rules can be combined with each other, giving raise to more complex rules. There are many ways to combine rules, giving raise to parallel, amalgamated, synchronized, concurrent rules (see [EHK$^+$96, CMR$^+$96b] for an overview). Here we will present a slightly different approach to the construction of parallel and amalgamated rules. This difference arises from the fact that we here have a different motivation for these constructions, as will be explained below for the case of parallel rules.

The idea of the construction of a parallel rule from rules $r1$ and $r2$ is that the resulting rule $r1 + r2$ shall be able to simulate the effect of $r1$ and $r2$ acting in parallel. Standardly, a parallel rule is constructed based on rules of the same grammar (and thus having the same type), and the resulting rule has also the same type as the component rules. We are mostly interested in composing graph grammars, and this brings the necessity to compose rules that may belong to different grammars (and thus have possibly different type graphs). Like the standard construction of parallel rules, the construction presented here can also be seen as a disjoint union of rules and therefore we will call it also parallel rule. To make a distinction, we will use a different notation: we write $r1 || r2$ instead of the standard notation $r1 + r2$.

Definition 3.20 (Parallel Rule) *Let $r1^{T1} : L1^{T1} \to R1^{T1}$ and $r2^{T2} : L2^{T2} \to R2^{T2}$ be rules with respect to types $T1$ and $T2$ respectively, and T be the coproduct of $T1$ and*

$T2$ in **GraphP***(see Appendix B.3). Then the* **parallel rule** *of $r1^{T1}$ and $r2^{T2}$, denoted by $r1^{T1}||r2^{T2}$, is the morphism in* **TGraphP**

$$r1^{T1}||r2^{T2} : L^T \to R^T$$

where L^T is the coproduct of $L1^{T1}$ and $L2^{T2}$ in **TGraphP** *(see Appendix B.4), R^T is the coproduct of $R1^{T1}$ and $R2^{T2}$ in* **TGraphP***, and $r1^{T1}||r2^{T2}$ is the universal morphism induced by the coproduct of left-hand sides.*

$$\begin{tikzcd}
L1^{T1} \arrow[rrr, "r1^{T1}"] \arrow[dd, hook, "i1_L"'] & & & R1^{T1} \arrow[dd, hook, "i1_R"] & & \\
& & L2^{T2} \arrow[rrr, "r2^{T2}"] \arrow[dl, hook', "i2_L"] & & & R2^{T2} \arrow[dl, hook', "i2_R"] \\
& L^T \arrow[rrr, "r1^{T1}||r2^{T2}"'] & & & R^T &
\end{tikzcd}$$

If we fix a coproduct T of type graphs $T1$ and $T2$, we denote by $r1^{T1}||^T r2^{T2}$ the parallel rule using this fixed type graph. ☺

Example **3.21 (Parallel Rule)** Consider the rule $r2$ in Figure 2.2. The result of constructing the parallel rule $r2||r2$ is shown in Figure 3.7. The type graph of the parallel rule consists of two disjoint copies $T1$ and $T2$ of the type graph of Figure 2.1. The mapping into this type graph is indicated by corresponding indices on the elements of the left- and right-hand side of the parallel rule. This parallel rule describes a situation in which there are two independent PBX systems and that messages (in this case digits) from phones belonging to different centrals happen in parallel. Moreover, as the type of the parallel rule is the disjoint union of the types of the component rules ($T1$ and $T2$), there can be no connections between items from these two type graphs in the resulting type graph. In the example, this means that the two PBX systems that were put together are completely unconnected to each other (there can be no communication from phones belonging to different centrals). In Def. 3.23 another way of composing rules that allow such connections will be presented. ☺

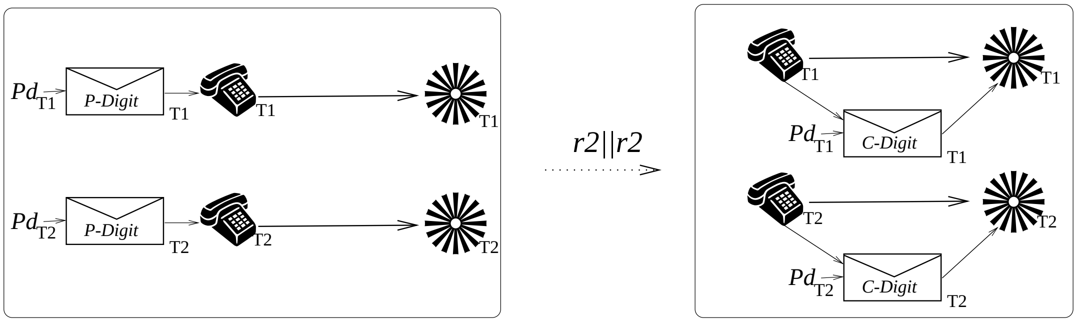


Figure 3.7: Parallel Rule

Proposition 3.22 *The parallel rule is well-defined.* ☺

Proof. Coproducts in **TGraphP** are constructed componentwise in **GraphP**. As the T was constructed as a coproduct, L^T and R^T are well-defined. Moreover, this implies that the type component of the parallel rule is id_T. Therefore $r1^{T1}||r2^{T2}$ is a morphism in **TGraphP(T)**. Now we have to show that $r1^{T1}||r2^{T2}$ is injective and consuming.

1. *Injectivity:* Follows from injectivity of $i1_R$, $i2_R$, $r1^{T1}$ and $r2^{T2}$.

2. *Consuming:* Follows from the consuming property of the component rules from the definition of the parallel rule as the corresponding universal morphism.

$\surd$

Now we will define the composition of two rules with respect to a third rule. This kind of composition is usually called *amalgamated rule* [BFH87]. Again, here we will consider amalgamation construction of rules of different grammars. The relationships between the interface rule and the rules to be composed will be described by total and injective (typed) graph morphisms. It would be possible to define more general amalgamated rules, but for the purposes of this paper, only this kind is necessary.

Definition 3.23 (Amalgamated Rule) *Let $r0^{T0} : L0^{T0} \to R0^{T0}$, $r1^{T1} : L1^{T1} \to R1^{T1}$ and $r2^{T2} : L2^{T2} \to R2^{T2}$ be rules with respect to types $T0$, $T1$ and $T2$ respectively. Let $f1^{t1}_L : L0^{T0} \to L1^{T1}$, $f1^{t1}_R : R0^{T0} \to R1^{T1}$, $f2^{t2}_L : L0^{T0} \to L2^{T2}$, $f2^{t2}_R : R0^{T0} \to R2^{T2}$ be total and injective typed graph morphisms such that the squares (1) and (2) in the diagram below commute. Let T be the pushout of $t1 : T0 \to T1$ and $t2 : T0 \to T2$ in* **GraphP**.

$$\begin{tikzcd}
L0^{T0} \arrow[rr, "r0^{T0}"] \arrow[dr, hook, "f2^{t2}_L"'] \arrow[dd, hook, "f1^{t1}_L"'] & & R0^{T0} \arrow[dr, hook, "f2_R"] \arrow[dd, hook, "f1^{t1}_R"' near end] \arrow[dl, phantom, "(2)"] & \\
& L2^{T2} \arrow[rr, "r2^{T2}"] \arrow[dd, hook] \arrow[dr, phantom, "(1)" near start] & & R2^{T2} \arrow[dd, hook] \\
L1^{T1} \arrow[rr, "r1^{T1}" near start] \arrow[dr, hook] & & R1^{T1} \arrow[dr, hook] & \\
& L^T \arrow[rr, "r1^{T1}||_{r0^{T0}}r2^{T2}"'] & & R^T
\end{tikzcd}$$

Then the ***amalgamated rule*** *of $r1$ and $r2$ with respect to $r0$, denoted by $r1^{T1}||_{r0^{T0}}r2^{T2}$, is defined as the morphism in* **TGraphP(T)**

$$r1^{T1}||_{r0^{T0}}r2^{T2} : L^T \to R^T$$

where L^T is the pushout of $f1^{t1}_L$ and $f2^{t1}_L$ in **TGraphP**, *R^T is the pushout of $f1^{t1}_R$ and $f2^{t1}_R$ in* **TGraphP**, *and $r1^{T1}||_{r0^{T0}}r2^{T2}$ is the universal morphism induced by the pushout of the left-hand sides of the rules.*

If we fix a pushout object T of $t1$ and $t2$, we denote by $r1^{T1}||^{T}_{r0^{T0}}r2^{T2}$ the amalgamated rule using this fixed type graph. ☺

Example **3.24 (Amalgamated Rule)** Consider the rules $r2$, $Pr2.1$ and $Cr2$ of Figures 2.2, 2.3 and 2.4. It is easy to see that the rule $r2$ is included in both $Pr2.1$ and $Cr2$,i.e., the type graph of $r2$ is included in the other two and the left- and right-hand sides of $r2$ are also included in the corresponding left- and right-hand sides of $Pr2.1$ and $Cr2$. To construct the amalgamated rule we can first construct the resulting type graph by gluing the type graphs of $Pr2.1$ and $Cr2$ along the type graph of $r2$. The resulting type graph can be seen in Figure 2.5. The we do the same procedure with the left- and right-hand sides of the rules, giving raise to the rule $PBXr2.1$ in Figure 2.6. This amalgamated rule, denoted by $Pr2.1||_{r2}Cr2$, describes a synchronization between the Phone and Central components: the Phone can only send a digit message to the Central if the Central is waiting for it. ☺

Proposition 3.25 *The amalgamated rule is well-defined.* ☺

Proof. Pushouts of total morphisms in **TGraphP** are constructed componentwise in **GraphP**(see Appendix B.4). As the T was constructed as a corresponding pushout, L^T and R^T are well-defined. Moreover, this implies that the type component of the amalgamated rule is id_T. Therefore $r1^{T1}||_{r0^{T0}}r2^{T2}$ is a morphism in **TGraphP(T)**. Now we have to show that $r1^{T1}||_{r0^{T0}}r2^{T2}$ is injective and consuming.

1. *Injectivity:* Follows from injectivity of $f1^{t1}_R$, $f2^{t2}_R$, $r0^{T0}$, $r1^{T1}$, $r2^{T2}$, from the definition of the amalgamated rule as a universal morphism, and from the fact that injectivity is inherited from pushout morphisms in **Graph**.

2. *Consuming:* Follows from the consuming property of the component rules and the componentwise definition of the parallel rule.

√

We just showed how to compose rules with and without an interface rule. Next we will define how to split (decompose) a rule according to a given splitting of its type. This will be done for splitting a type into two disjoint types, called *parallel decomposition*, and for splitting a type into two types and an interface, called amalgamated decomposition of rules. For example, if we decompose a parallel rule with respect to the original types, we get again the component rules (up to isomorphism). Depending on how the items of left- and right-hand sides of the rule are typed, it can be that we do not get two rules when we decompose a rule with respect to two types. But this decomposition of a rule always gives raise to at least one rule. This is very important for the proof that the category of (typed) graph grammars has products (Theo. 4.16), and that these products will be shown to correspond to a parallel composition of (typed) graph grammars (see Sect. 4).

Definition 3.26 (Parallel Decomposition of Rules) *Let* $T1 \xrightarrow{t1} T \xleftarrow{t2} T2$ *be a coproduct diagram in* **GraphP***and* $r \in GRules(T)$. *Then the* ***parallel decomposition*** *of* r *with respect to* $T1|T2$ *is defined as*

$$decomp_{T1|T2}(r) = (\mathcal{T}_{t1}(r), \mathcal{T}_{t2}(r))$$

$$\begin{tikzcd}
L1 \arrow[rr, hook, "r1"] \arrow[dr, hook, "t^{L1}"'] \arrow[drr, hook, "l1"] & & R1 \arrow[dl, "t^{R1}"] \arrow[drr, hook, "i1^{\bullet}"] & & L2 \arrow[rr, "r2"] \arrow[dll, hook', "l2"'] \arrow[dr, "t^{L2}"] & & R2 \arrow[dll, "i2^{\bullet}"'] \arrow[dl, "t^{R2}"] \\
 & T1 \arrow[drr, hook, "t1"'] & L \arrow[rr, "r"] \arrow[dr, "t^{L}"'] & & R \arrow[dl, "t^{R}"] & T2 \arrow[dll, hook', "t2"] & \\
 & & & T & & &
\end{tikzcd}$$

☺

Notation: When it is clear from the context the decomposition $decomp_{T1|T2}(r)$ will be denoted by $decomp(r)$.

Remarks.

1. *By the definition of the retyping (Def. 3.8), the four squares described by* (Li, Ti, L, T), (Ri, Ti, R, T), *for* $i = 1, 2$, *are pullbacks (because the coproduct inclusions are total).*
2. *The parallel decomposition is the inverse (up to isomorphism) of the parallel composition. This is assured by Lemma C.3 (see Appendix C).*

☺

Example **3.27 (Parallel Decomposition of Rules)** Consider the rule r^T in Figure 3.8. We want to decompose this rule with respect to to the decomposition of the type T into $T1$ and $T2$. To obtain the component with respect to $T1$ we just have to remove from r all items whose types are not in $T1$, in this case, we remove the CENTRAL from the left- and right-hand sides of r and obtain $r1$. $r2$ is obtained analogously. In this case, the result of the parallel decomposition is $decomp_{T1|T2}(r) = (r1, r2)$. Note that $r2$ is an isomorphism and thus not a rule. ☺

Proposition 3.28 *Let* $(r1, r2) \in \{decomp_{T1|T2}(r) | r \in GRules(T)\}$. *Then exactly one of the following cases is true:*

1. $r1 \in GRules(T1)$ *and* $r2$ *is an isomorphism;*
2. $r2 \in GRules(T2)$ *and* $r1$ *is an isomorphism;*
3. $r1 \in GRules(T1)$ *and* $r2 \in GRules(T2)$.

☺

Proof. By definition of $GRules$ (Def. 3.14), if $ri \in GRules(Ti)$, for $i = 1, 2$, then ri is not an isomorphism and vice versa. Therefore, the 3 cases are mutually exclusive. Thus it remains to show that there can't be the case in which $r1$ and $r2$ are both isomorphisms.

By Lemma C.3 we obtain that L is the coproduct of $L1$ and $L2$ and R is the coproduct of $R1$ and $R2$. Thus, r must be the parallel rule of $r1$ and $r2$ (because of uniqueness of universal morphisms), i.e., $r = r1||r2$. By hypothesis, $r \in GRules(T)$ and thus r is not an isomorphism and is injective. This means that there is at least one $x \in L$ such that $r(x)$ is undefined or $y \in R$ such that $y \notin rng(r)$. Assume we have the first case (x). As L is a

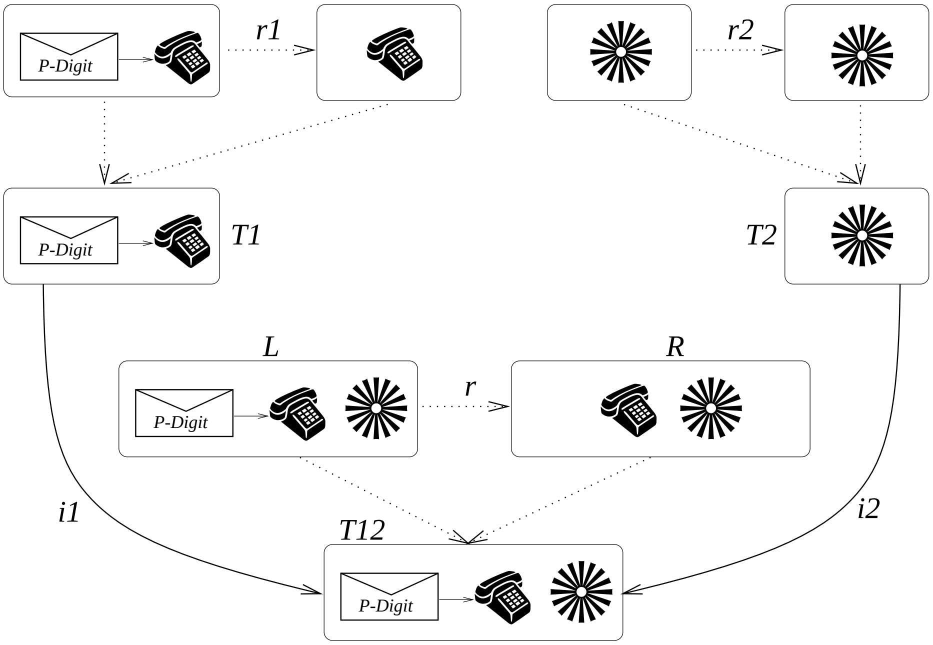


Figure 3.8: Parallel Decomposition of the Rule r

coproduct of $L1$ and $L2$, x must have a pre-image in one of these two graphs. Assume it has a pre-image in $L1$. Let $z \in L1$ such that $l1(z) = x$. As the coproduct morphisms $l1 : L1 \to L$ and $i1^{\bullet} : R1 \to R$ are total and $r \circ l1 = i1^{\bullet} \circ r1$ we conclude that $r1(z) = \texttt{undef}$. Thus $r1$ is not an isomorphism. If we assume that x has a pre-image in $L2$, we get that $r2$ can't be an isomorphism, and if we assume the second case ($y \in R$), we analogously obtain that either $r1$ or $r2$ can't be isomorphisms in this case. Thus, we conclude that if r is not an isomorphism, either $r1$ or $r2$ or both are not isomorphisms. $\surd$

The next proposition expresses the fact that the parallel rule construction is compatible with the subrule relation.

Proposition 3.29 *Let* $[r1], [r1'] \in \mathcal{R}(T1)$, $[r2], [r2'] \in \mathcal{R}(T2)$, $r1 \subseteq^r r1'$ *and* $r2 \subseteq^r r2'$. *Then* $[r1\|r2] \subseteq^R [r1'\|r2']$. ☺

Proof. The construction of the parallel rule is based on coproducts in **TGraphP**. The relations $r1 \subseteq^r r1'$ and $r2 \subseteq^r r2'$ mean that there are pushouts (1) and (2) in **TGraphP**(**T1**) and **TGraphP**(**T2**) using the rule morphisms and corresponding inclusions such that the safety condition is satisfied. Pushouts in the latter categories are also pushouts in **TGraphP**(see Def. B.10 and Def. B.14). Let (3) be the square consisting of the rules $r1\|r2$ and $r1'\|r2'$ and the total and injective morphisms $i_{L1} + i_{L2}$ and $i_{R1} + i_{R2}$, where the component morphisms are the corresponding inclusions of pushouts (1) and (2). Due to standard categorical results, if (1) and (2) are pushouts and (3) is obtained componentwise by coproducts, then (3) is also a pushout (see [BW90] for a proof). As the safety condition is satisfied by the components and the parallel rules are obtained as coproducts, we conclude that $[r1\|r2] \subseteq^R [r1'\|r2']$. $\surd$

Now, analogously to the parallel decomposition, we will define the amalgamated decomposition.

Definition 3.30 (Amalgamated Decomposition of Rules) *Let $i1 : T0 \to T1$ and $i2 : T0 \to T2$ be total and injective and (1) be a pushout. Let $r \in Rules^{+}(T)$. Then the* ***amalgamated decomposition*** *of r with respect to $T1|_{T0}T2$ is defined as*

$$decomp_{T1|_{T0}T2}(r) = (\mathcal{T}_{i2^\bullet}(r), \mathcal{T}_{i2^\bullet \circ i1}(r), \mathcal{T}_{i1^\bullet}(r))$$

$$\begin{tikzcd}
L0 \arrow[rrrr, "r0"] \arrow[dr, mapsto, "l2"] \arrow[dd, mapsto, "l1"'] \arrow[ddddrr, mapsto] & & & & R0 \arrow[dr, mapsto, "s2"] \arrow[dd, mapsto, "s1"] \arrow[ddddll, mapsto] & \\
& L2 \arrow[rrrr, "r2"] \arrow[dd, mapsto, "l1^\bullet"] \arrow[ddddrr, mapsto] & & & & R2 \arrow[dd, mapsto, "s1^\bullet"] \arrow[ddddll, mapsto] \\
L1 \arrow[rrrr, "r1"] \arrow[dr, mapsto, "l2^\bullet"] \arrow[ddddrr, mapsto] & & & & R1 \arrow[dr, mapsto, "s2^\bullet"] \arrow[ddddll, mapsto] & \\
& L \arrow[rrrr, "r"] \arrow[ddddrr, mapsto] & & & & R \arrow[ddddll, mapsto] \\
& & T0 \arrow[dr, mapsto, "i2"] \arrow[dd, mapsto, "i1"] & & & \\
& & (1) & T2 \arrow[dd, mapsto, "i1^\bullet"] & & \\
& & T1 \arrow[dr, mapsto, "i2^\bullet"] & & & \\
& & & T & &
\end{tikzcd}$$

☺

Remarks.

1. *All (four) squares connecting the square of left-hand sides to the square of type graphs, as well as the (four) squares connecting the square of right-hand sides to the square of type graphs, are pullbacks.*

2. *Lemma C.4 assures that the amalgamated decomposition is the inverse of the amalgamated composition of rules (up to isomorphism) if we use the pairs $(l1^{i1}, s1^{i1})$ and $(l2^{i2}, s2^{i2})$ as basis to construct the amalgamated rule, where the morphisms li and si, for $i = 1, 2$, are obtained by the retyping construction.*

☺

Example **3.31 (Amalgamated Decomposition)** Consider rule r in Figure 3.9. The type graph T is obtained as a gluing of $T1$ and $T2$ with respect to $T0$ (via the inclusions $i1$ and

$i2$). To decompose rule r with respect to this decomposition of the type we proceed analogously to the parallel decomposition. For example, $r1$ is obtained by deleting from r everything whose type is not present in $T1$. Thus the amalgamated decomposition of r with respect to$T0$,$T1$ and $T2$, is $decomp_{T1|_{T0}T2}(r) = (r1, r0, r2)$. ☺

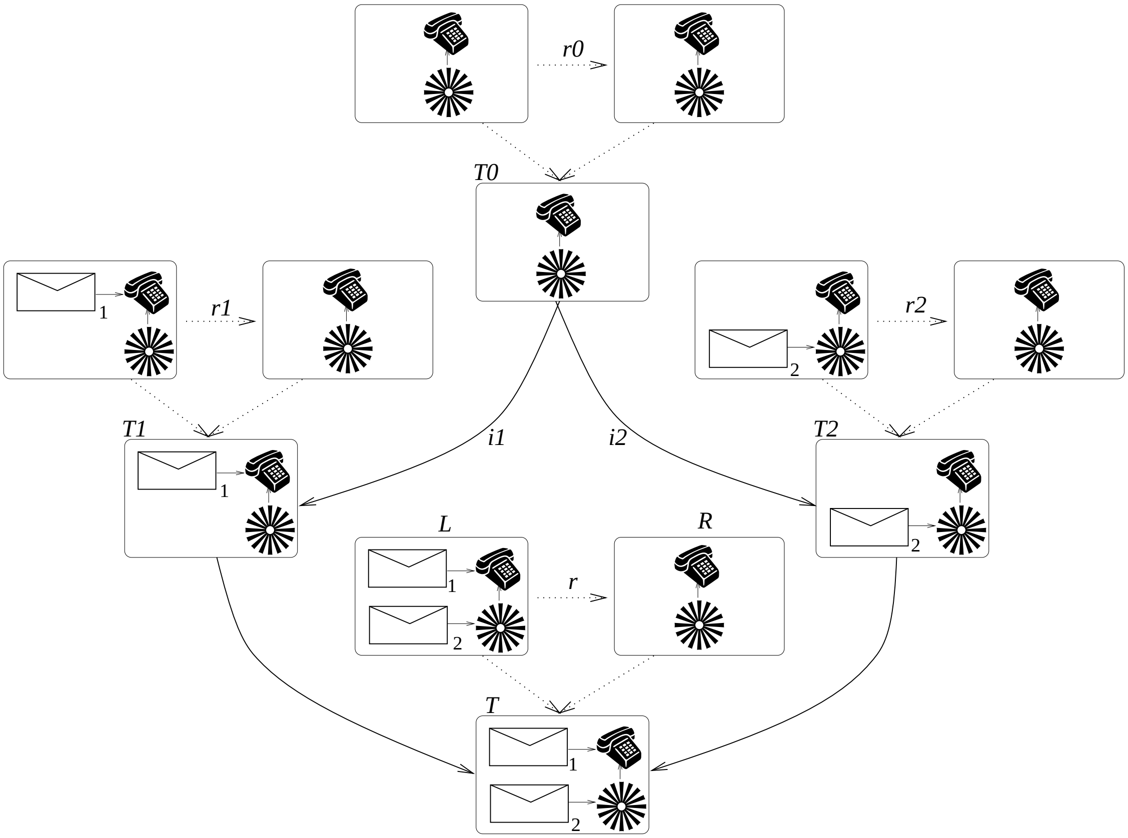


Figure 3.9: Amalgamated Decomposition of the Rule r

Proposition 3.32 *Let $(r1, r0, r2) \in \{decomp_{T1|_{T0}T2}(r) | r \in GRules(T)\}$. Then exactly one of the following cases is true:*

1. *$r1 \in GRules(T1)$ and $r0$ and $r2$ are isomorphisms;*
2. *$r2 \in GRules(T2)$ and $r0$ and $r1$ are isomorphisms;*
3. *$r1 \in GRules(T1)$, $r2 \in GRules(T2)$ and $r0$ is an isomorphism;*
4. *$r1 \in GRules(T1)$, $r2 \in GRules(T2)$ and $r0 \in GRules(T0)$*

☺

Proofidea. Analogous to proof of Prop. 3.28, using Lemma C.4. √

3.3 Graph Grammars and their Behaviour

The basic idea of graph grammars is to specify a system by specifying its initial state (by a graph) and the possible changes of states (by graph rules). The behaviour of a graph grammar is given by applying rules to graphs representing the states of the system, called state graphs in the following. The application of a rule to a state graph is possible if there is a *match* for this rule, i.e., there is a subgraph of the state that corresponds to the left-hand side of the rule (this correspondence do not have to be an isomorphism because different items from the left-hand side may be mapped to the same items in the state graph).

The operational semantics of a graph grammar is based on the applications of the rules to state graphs. In particular the algebraic approach to graph grammars [Ehr79, Löw93] stresses the fact that rules may be applied *in parallel* to a state graph. This may happen if the matches of the rules are not mutually exclusive in the sense that the rules do not delete the same items of the state graph. Moreover, graph grammars are inherently *non-deterministic*: if many rules are applicable to some state then the choice of which will be the next one to be applied is non-deterministic. Even the same rule may have different ways of being applied to the same graph (depending on the existing matches), and the choice of which match will be first applied is non-deterministic. If we have two rules $r1$ and $r2$ that can be applied to a graph G, there may be three different situations, that lead to three kinds of non-determinism:

Sequential Independent Situation: This is the case when, after applying one of the rules, we can still apply the other one and vice-versa. The kind of non-determinism described by this situation arises from the wish to apply these rules in some sequence, that is, the choice is made just in order to sequentialize the applications of these rules. Therefore, this kind of non-determinism will be called *interleaving non-determinism.*

Mutually Exclusive Situation: In this case, the application of one rule disables the application of the other and vice-versa. This happens if (at least) one item of the state graph is deleted by both rules.[1] This kind of non-determinism will be called *mutual exclusion non-determinism.*

Mixed Situation: The fact that graph grammars allow items to be preserved by the application of rules gives raise to the possibility of having situations in which, although two rules are not mutually exclusive, there is only one possible order in which we may apply these two rules. This happens if these two rules overlap on items of the state graph that are preserved by one rule and deleted by the other. On the one hand, these rules are not mutually exclusive because they do not delete the same items. But on the other hand, they can not be applied in any sequence because if the deleting rule is applied first, the other one can not be applied anymore. This situation was mostly investigated in the SPO approach, and is called a *weak parallel* situation [Kor95] (for the DPO approach see [CR96]). The kind of non-determinism described here will de called *asymmetric non-determinism.*

In the case of algebraic graph grammars, there is another kind of non-determinism that may occur: the application of a rule to a match may have infinitely many different results. All of these results are isomorphic, but nevertheless it makes a difference whether all of these

[1] In the DPO approach to graph grammars there may be other situations that lead to this kind of conflict. These are related to the gluing condition [Ehr79].

results are distinguished at the semantical level or not. If the semantics includes all these derivation steps, they shall be considered as different, what really means that the result of a derivation step is non-deterministic. If these derivation steps are identified at the semantical level, then they are just different syntactical representation for “the same” computation step. We will call this kind of non-determinism *representation non-determinism*.

A sequential view on a (specification of a) system is to see it as a description of the possible *sequences* of steps that may happen in this system. Looking at these sequences, we may notice that different forms of non-determinism may describe different phenomena. Usually, the interleaving non-determinism is the basis for describing the *concurrency* of a system (see [Mil89, MM90, MP92, CEL$^+$94a, WN94]) and the mutually exclusion non-determinism is used to describe the *conflicts* of a system. If we relax the requirement that all actions of a system must happen in some sequence, and instead say that actions that are *causally* dependent from others must happen in some order, but the other actions do not have to be represented in a sequence, we have a more concurrent view on the system. In such a view, interleaving non-determinism is not really a non-deterministic choice because there is only one possibility to describe this situation: as a concurrent application of the involved rules (note that ‘concurrent’ does not mean that they must happen together, but that they may happen together). In some approaches in which concurrency is not obtained from arbitrary interleavings, interleaving non-determinism is a consequence of concurrency (these approaches are mostly based on partial orders of actions [Pra86, BD87, Vog92, MMS94, Kor96]). Thus, forgetting for a moment about asymmetric non-determinism, in a concurrent semantics there is only one kind of non-determinism, and it describes the conflict situations of the system. Asymmetric non-deterministic situations can only occur if the formalism allows that items of a state are preserved from one state to the other, where ‘preservation’ of an item is different from ‘deletion and re-creation’ of some item. In formalisms like Petri nets [Pet80], rewriting logic [Mes92], various kinds of process calculus [Hoa85, Hen88, Mil89] there is usually no difference between preservation and deletion/re-creation. The great advantage of making this difference is that a bigger amount of parallelism is allowed in a system because various read-accesses of the same item may happen in parallel. In [MR95] an extension of Petri nets, called contextual nets, was presented adding this feature to Petri nets. In graph grammars, the ability to preserve items gave raise to a very rich theory of parallelism. However, the theory was mostly concerned about situations in which an item is read-accessed by many processes at the same time. The situation in which one item is write-accessed by one process and read-accessed by many others was mostly not investigated. Therefore there is no consense yet about which phenomena is described by asymmetric non-determinism. In [Kor96] it was used to describe concurrency, whereas in [CEL$^+$94a] it was used to describe conflicts.

The *sequential semantics* of graph grammars given in Def. 3.37 is based on sequences of applications of rules (sequential derivations). In such a semantics, non-determinism is described implicitly by different sequences having a common subsequence. Thus, concurrency and conflicts are described implicitly. Moreover, as this semantics is based on concrete graphs, there is representation non-determinism.

The *concurrent semantics* of graph grammars given in Def. 3.46 is based on concurrent applications of rules (concurrent derivations). In such a semantics, interleaving and asymmetric non-determinism are explicitly expressed by the absence of causal relationships between applications of rules. Thus concurrency is explicitly described. If two derivations have a common subderivation, a mutually exclusive situation may occur if there is no concurrent derivation containing these two. Thus, conflicts are described implicitly by the concurrent semantics. By the use of suitable equivalence classes of concurrent derivations, representation non-determinism is avoided.

In Chap. 6 we will present a semantics for graph grammars, namely the *unfolding semantics*, in which both concurrency and conflicts are described explicitly, and in which representation non-determinism is not present.

A *(typed) graph grammar* (see following definition) consists of

- a type graph, that specifies the type of all graphs involved in this grammar and is therefore called *type of the grammar*,
- an initial graph, that specifies the initial state of the system,
- a set of rule names, that shall be used to identify the rules of the grammar and
- a function associating with each rule name a corresponding rule.

  The initial graph and all rules must be typed according to the type of the grammar.

Definition 3.33 (Graph Grammar) *A* ***(typed) graph grammar*** *is a tuple* $GG = (T, I, N, n)$ *where*

- T *is a type graph (the* **type** *of the grammar),*
- I *is a typed graph in* $\mathbf{TGraphP(T)}$ *(the* ***initial graph*** *of the grammar),*
- N *is a set of* ***rule names****,*
- $n : N \to Rules(T)$ *is a total function (the* ***naming function****, assigning to each rule name a rule with respect to the type* T*).*

We denote by $\overline{n}$ *the extension of* n *to equivalence classes of rules, i.e.,* $\overline{n}(x) = [n(x)]$ *for each* $x \in N$*.*

A graph grammar GG *is* **safe** *if its initial graph* I *has no automorphisms and each of its rules* r *has no rule-automorphisms.* ☺

Remark. *The word 'rule' will be used in many contexts. It may sometimes mean an element of* N*, some times a morphism* $n(x)$*, and sometimes a pair* $(x, n(x))$*, for* $x \in N$*. A morphism* $n(x)$ *is usually be called* a rule pattern. ☺

Example **3.34 (Graph Grammar)** The grammar PBX described in Chap. 2 is an example of graph grammar having as components: i) type graph: $PBXType$ shown in Figure 2.5, ii) initial graph: $PBXIni$ shown in Figure 2.6, iii) rule names: $\{PBXr1, PBXr2.1, PBXr2.2, PBXr3\}$, iv) association of names to rules: given in Figure 2.6. ☺

3.3.1 Sequential Semantics

The operational behaviour of a graph grammar is defined in terms of *derivation steps*, that are applications of the rules of the grammar to some state graph. A rule r can be applied to a state IN if the pattern of the left-hand side L of the rule is found in IN. Formally, this is described by the existence of a total typed graph morphism m from L to T, called *match*. The application of this rule r to the match m consists, roughly speaking, of taking from IN everything that is deleted by the rule and adding everything that is created by the rule (see Sect. 3.2 for a description of how these operations are modeled by a rule). As there are some conflict situations that may occur (see [EHK+96]), we describe the process of application of a rule $r : L \to R$ to a graph IN via a match $m : L \to IN$ by first adding then deleting:

1. **Add** to IN all items that shall be added by the rule, that is, all elements e that are in the right-hand side R ($e \in R$) and that are not in the range of the rule morphism $e \notin (rng(r))$.

2. **Delete** from the result of the first step all elements that shall be deleted by the rule (elements $e \in L$ such that $e \notin dom(r)$), and the items that depend on deleted items (dangling edges).

Formally, the construction of the result of a rule application, called a *derivation step*, is given by a pushout of the rule r and the match m in the category of graphs typed over a fixed graph T.

A description of a *no-operation* step is done by the *empty steps*. There we use an empty-rule (that is an isomorphism) as a rule. The result is that nothing is deleted and nothing is created, i.e., we have the same output graph as the input. Empty steps will be useful for defining translations of derivation sequences from one grammar to another (based on graph grammar morphisms – see Sect. 4.1).

The following definition is the standard one for derivations in the single pushout approach, except for the fact that the derivation step is not only a pushout, but a pushout together with the name of the rule that was used (a corresponding definition for the DPO approach can be found in [CMR+96b]).

Definition 3.35 (Match, Derivation Step) *Given a rule $r : L \to R$ with respect to a type graph T, a* ***match*** *$m : L \to IN$ of r in a graph IN is a total morphism in* $\mathbf{TGraphP(T)}$. *A* ***derivation step*** *s of a graph IN_s with rule r_s with name nr_s at match m_s is a tuple $s = (nr_s, S)$, where S is a pushout diagram of m_s and r_s in* $\mathbf{TGraphP(T)}$ *(see Def. B.14 for the explicit construction).*

$$\begin{tikzcd}
L_s \arrow[r, "r_s"] \arrow[d, "m_s"'] \arrow[dr, phantom, "S"] & R_s \arrow[d, "m_s^\bullet"] \\
IN_s \arrow[r, "r_s^\bullet"'] & OUT_s
\end{tikzcd}$$

A derivation step is denoted by $IN_s \overset{nr_s:m_s}{\Longrightarrow} OUT_s$. IN_s, OUT_s, $r_s^\bullet$ and $m_s^\bullet$ are called ***input graph***, ***output graph***, ***co-rule*** *and* ***co-match*** *respectively.*

For a graph grammar GG, the class of all derivation steps using rules in GG is denoted by $Steps_{GG}$. Let $i : L \to R$ be an isomorphism and $m : L \to IN$ be a total morphism in

TGraphP(T). *The the pushout of i and m is called an **empty step** of GG and denoted by $IN \overset{i:m}{\Longleftrightarrow} OUT$. The class of all steps including the empty steps, denoted by $StepsI_{GG}$, is defined by $StepsI_{GG} = Steps_{GG} \cup \{s \mid s \text{ is an empty step}\}$.* ☺

Example **3.36 (Derivation Step)** Square $(s1)$ of Figure 2.7 described the application of the rule $PBXr1$ to the graph IN. The match is indicated by the indices, and describes that PHONE 12 receives a `P-Digit(5)` message. Following the steps to construct $IN1$, we first have to add to IN the messages `E-Act`$_3$ and `P-Digit(5)`, and the new vertex T together with a corresponding edge 5. Then, we delete the message `E-Act`$_1$ and the vertex F of PHONE$_1$ with the corresponding edge 5. ☺

A derivation step describes a unit of a computation using a graph grammar. Whole (sequential) computations may be described by sequences of derivation steps in which the output graph of one step is the input graph of the subsequent step. These sequences are called *derivation sequences*. The sequential semantics of a graph grammars is defined as the class of all derivation sequences of this grammar.

Notation: Let A be a set (or class). Then the set (or class) of all sequences over A is denoted by A^∞. The restriction of A^∞ to finite sequences is denoted by A^*. The empty sequence is denoted by λ. Let $\sigma \in A^\infty$. Then $|\sigma| \in \mathbb{N} \cup \{\omega\}$ denoted the length of σ. The i^{th} element of a sequence σ is denoted by σ_i. Concatenation of sequences $\sigma1$ and $\sigma2$ is denoted by $\sigma1 \bullet \sigma2$.

Definition 3.37 (Sequential Derivation, Sequential Semantics) *Given a graph grammar $GG = (T, I, N, n)$. The class of **sequential derivations** with respect to GG is defined by*

$$SDer_{GG} = \{\sigma \in Steps_{GG}^{\infty} \mid \sigma = \lambda \text{ or } IN_\sigma = IN_{\sigma_1} = I \text{ and } OUT_{\sigma_i} = IN_{\sigma_{i+1}} \text{ for all } 1 \leq i < |\sigma|\}$$

*If a sequential derivation $\sigma \in SDer_{GG}$ is finite, we define its **output graph** as $OUT_\sigma = OUT_{\sigma_{|\sigma|}}$. The class of **sequential derivations including empty steps** with respect to GG is defined by*

$$SDerI_{GG} = \{\sigma \in StepsI_{GG}^{\infty} \mid \sigma = \lambda \text{ or } IN_\sigma = IN_{\sigma_1} = I \text{ and } OUT_{\sigma_i} = IN_{\sigma_{i+1}} \text{ for all } 1 \leq i < |\sigma|\}$$

*The **sequential semantics** of GG is the class of sequential derivations of GG, i.e., $SDer_{GG}$.* ☺

Example **3.38** Figure 2.7 shows a sequential derivation of the grammar PBX consisting of two derivation steps $s1$ and $s2$. There we can see that the output graph of $s1$, namely $IN1$, is the same as the input graph of $s2$. ☺

For concurrent systems, the sequential semantics has at least three drawbacks:

1. As discussed in the introduction of this section, concurrency in a sequential semantics is usually expressed by the fact that the "same" derivation steps may be observed in

two different orders.[2] If we consider just the total order of derivation steps given by one sequential derivation, then it is impossible to say which steps can be observed in different orders. If we look 'inside' the sequential derivation, then we may find out which steps are independent from each other, but this task is not at all easy because the concepts of parallel and sequential independence of steps [EHK+96] are defined only for subsequent derivation steps and the generalization of these concepts for sequences is not straightforward. There are some approaches that go in the direction of defining classes of sequential derivations that are equivalent, like the canonical derivations [Kre77, LD94] and shift-equivalences [Sch94, CEL+94a], but neither of them gives the answer to the question whether two arbitrary steps of a sequential derivation are independent or not (the comparison is made on the level of sequences and not on the level of derivation steps). Therefore one can say that *concurrency is described implicitly* in the sequential semantics.

2. To find out whether two derivation steps are mutually exclusive, one has to search for a bigger sequential derivation containing these two steps. If such a derivation can be found, the steps are not mutually exclusive. Otherwise, they are mutually exclusive. Thus, *non-determinism is described implicitly.*

3. In the algebraic approach to graph grammars, the result of a derivation step is unique only up to isomorphism. This means that, if some rule creating a vertex is applicable to a graph IN, there will be infinitely many sequential derivations in the sequential semantics that represent this application, namely one for each possible (isomorphic) resulting graph. In the sequential semantics presented here, all these isomorphic results are present, that is *representation non-determinism occurs.*

Two of these drawbacks (1. and 3.) were solved by the introduction of the concurrent semantics of (SPO) graph grammars [Kor95, Kor96].

3.3.2 Concurrent Semantics

The main aim of developing a concurrent semantics for graph grammars was to represent concurrency explicitly. This is achieved by substituting sequential by concurrent derivations as the basis for the semantics. Moreover, this allows one to define morphisms between derivations leading to a category of concurrent derivations. This category has a special property that allows the definition of a quotient category based on isomorphism classes of concurrent derivations. This quotient category identifies computations that shall be "the same" (that are just different because different choices of results were done for each derivation step). Using this category as semantics, we avoid representation non-determinism.

A concurrent derivation defines a class of sequential derivations that are in some sense equivalent with respect to concurrency. The definitions given below are slight variations from the ones in [Kor96]: the difference is that here we include explicitly the component "names of actions" that was not present in the original version. This (syntactical) change does not affect the results obtained in [Kor96].

The construction of a concurrent derivation κ with respect to one sequential derivation σ will be done in three steps:

[2] To find out which derivation steps are the same, one may use concepts like parallel or sequential independence of derivations (see [EHK+96] for an overview).

1. For each derivation step s of σ, the input and output graphs are glued accordingly, giving raise to graphs IO_s and total morphisms embedding IN_s, OUT_s, L_s and R_s in IO_s.

2. Glue all IO graphs, giving raise to the *core graph* C and embeddings of all IN_s, OUT_s, L_s and R_s in C. The diagram consisting of the core graph and the corresponding embeddings is called *core structure* of σ. By construction, the core graph contains all occurrences of the left- and right-hand sides of used rules.

3. Build the concurrent derivation κ, consisting of the core graph, the embedding of the initial graph of σ in the core graph, a set of action names, and a function mapping action names to actions. The actions are obtained from the embeddings of the left- and right-hand sides of rules used in σ in the core graph. The action names consist of the name of the rule used in the action and the action itself. As the output graph can be derived from the other components, it will not be an explicit part of the concurrent derivation.

Remind that a derivation step of a grammar GG consists of a rule name and of a pushout in $\mathbf{TGraphP(T)}$, where T is the type of the grammar. Therefore the graphs contained in a concurrent derivation will also be typed over T.

Definition 3.39 (Core Structure) *Let GG be a graph grammar.*

1. *Given a step $s = (n, S) \in Steps_{GG}$, we define the* ***step-core*** *of s as $core(s) = (IO_s, in_s, out_s)$ constructed as follows (see diagram below):*

   a) *Construct pushout (1) of $L_s \overset{r_s^\blacktriangledown}{\hookleftarrow} dom(r_s) \overset{r_s!}{\rightarrow} R_s$ in $\mathbf{TGraphP(T)}$.*

   b) *This leads to a factorization of $r_s = (aR_s)^{-1} \circ aL_s$.*[3]

   c) *Thus the pushout S (of the derivation step) can be decomposed into pushouts (2) and (3).*

   d) *As $(aR_s)^{-1}$ is injective and surjective, the pushout morphism $(out_s)^{-1}$ is also injective and surjective and can be inverted, giving raise to the morphism out_s*

$$\begin{tikzcd}
 & & dom(r_s) \arrow[dll, hook', "r_s^\blacktriangledown"'] \arrow[drr, mapsto, "r_s!"] & & \\
L_s \arrow[rrrr, "r_s"] \arrow[rrrr, phantom, "(1)"'] \arrow[drr, mapsto, "aL_s"'] \arrow[dd, mapsto, "m_s"'] & & & & R_s \arrow[dll, hook', "aR_s"'] \arrow[dd, "m_s^\bullet"] \\
 & (2) & LR_s \arrow[urr, tail, "(aR_s)^{-1}"'] \arrow[dd, mapsto, "m_s^x"'] & (3) & \\
IN_s \arrow[rrrr, "r_s^\bullet"'] \arrow[drr, mapsto, "in_s"'] & & & & OUT_s \arrow[dll, hook', "out_s"] \\
 & & IO_s \arrow[urr, tail, "(out_s)^{-1}"] & &
\end{tikzcd}$$

[3] Due to the fact that inverting the morphisms $r_s^\blacktriangledown$ and aR_s yields also a pushout (see [Kor96]).

2. *Given a sequential derivation $\sigma \in SDer_{GG}$, let $inout(\sigma) = (core(\sigma_i))_{i \in \{1..|\sigma|\}}$ be the diagram containing all cores of steps of σ. Then the colimit $Core(\sigma) = (C, cin_i, c_i, cout_i)_{i \in \{1..|\sigma|\}}$ in* **TGraphP(T)** *is called the* ***core structure*** *of σ. Each colimit morphism c_i is called* ***core morphism*** *and the colimit graph C is called* ***core graph****.*

$$\begin{tikzcd}
L_j \arrow[rr, "r_j"] \arrow[dr, "aL_j"'] & & R_j \arrow[dl, "aR_j"] & & L_{j+1} \arrow[rr, "r_{j+1}"] \arrow[dr, "aL_{j+1}"'] & & R_{j+1} \arrow[dl, "aR_{j+1}"] \\
& LR_j \arrow[dd, "m_j^x"'] & & & & LR_{j+1} \arrow[dd, "m_{j+1}^x"'] & \\
IN_j \arrow[rrr, "r_j^\bullet"'] \arrow[dr, "in_j"] \arrow[ddrrr, bend right, "cin_j"'] & & & OUT_j = IN_{j+1} \arrow[dll, hook', "out_j"] \arrow[rrr, "r_{j+1}^\bullet"'] \arrow[drr, "in_{j+1}"'] & & & OUT_{j+1} \arrow[dl, hook', "out_{j+1}"] \\
& IO_j \arrow[drr, "c_j"] & & & & IO_{j+1} \arrow[dll, "c_{j+1}"'] & \\
& & & C & & &
\end{tikzcd}$$

☺

Remark. *The diagram $inout(\sigma)$ is a diagram containing only total morphisms. The inclusion functor $\mathcal{I}$:* **TGraph(T)** $\rightarrow$ **TGraphP(T)** *preserves colimits (see [Kor96]), and thus the colimit $Core(\sigma)$ is also the colimit of the same diagram in* **TGraph(T)***, i.e., all colimit morphisms are total. The fact that diagram $inout(\sigma)$ can be seen as a partial order (there are no cycles) and that all morphisms in this diagram are injective implies that all colimit morphisms are also injective (see [Kor96] for a proof).* ☺

A concurrent derivation describes a computation of a grammar in which some actions may occur in parallel. This computation is represented by the changes of states caused by the application of rules to an initial graph (initial state). States, as well as left- and right-hand sides of rules, are represented by typed graphs. The construction of the core structure provides a way to describe all state graphs in one (typed) graph, and the corresponding applications of rules as (total) morphisms having the core graph as target. Thus, we can see the core graph as a kind of type graph in which the initial graph and all rules of the considered derivation are interpreted. Therefore, the following definition will use the same notation as for typed graphs. In this context, a "typing morphism" is a (total) morphism that has the core graph as target. However, we shall remark here that the typing used in this definition is not the same as the one used on typed graphs because the core graph is already a typed graph (this kind of typing will be formally defined in Sect. 5.1 and called *double-typing*). Moreover, in Sect. 5.4 we will also show that a concurrent derivation can be seen as a graph grammar that has special properties (can be considered as a "deterministic" graph grammar).

Definition 3.40 (Concurrent Derivation) *Let $GG = (T, I, N, n)$ be a graph grammar, $\sigma \in SDer_{GG}$ and $Core(\sigma) = (C, cin_i, c_i, cout_i)_{i \in dom(\sigma)}$ be the* ***core structure*** *of σ. Then* *the* ***concurrent derivation*** $\kappa = (C^\kappa, I^\kappa, N^\kappa, n^\kappa)$ *corresponding to* σ*, written* $\kappa \rightsquigarrow \sigma$*, is defined as follows (see diagrams of Def. 3.39):*

- **Core graph:** $C^\kappa = C$.
- **Initial graph:** $I^\kappa = cin_1 : I \to C$.
- **Action names:** $N^\kappa = \{(nr_j, L_j^C \overset{r_j^C}{\to} R_j^C) \mid j \in \{1..|\sigma|\}$ *and* $n(nr_j) = (L_j^C \overset{r_j^C}{\to} R_j^C)\}$. *The typing morphisms are given by* $t^{L_j^C} = c_j \circ m_j^x \circ aL_j : L_j \to C$ *and* $t^{R_j^C} = c_j \circ m_j^x \circ aR_j : R_j \to C$.
- **Actions:** $n^\kappa : N^\kappa \to Rules(C)$ *is defined for all* $x = (nr, r) \in N^\kappa$ *as* $n^\kappa(x) = r$. *This function assigns for each action name a corresponding* ***action***.

For each $r^C : L^C \to R^C \in rng(n^\kappa)$ *we define* $pre^\kappa_{r^C} = t^{L^C}$, $r^\kappa_{r^C} = r^C$, $post^\kappa_{r^C} = t^{R^C}$*, called* ***pre-condition***, **rule** *and* ***post-condition*** *of* r^C *respectively.*

The ***length*** *of a concurrent derivation* κ *is defined by* $|\kappa| = card(N^\kappa)$. *If a concurrent derivation is finite (*N^κ *is finite), its* ***output graph*** OUT_κ *is defined as* $OUT_\kappa = out_{|\sigma|} : OUT_\sigma \to C$.

The class of all concurrent derivations obtainable from $SDer_{GG}$ *is denoted by* $CDer_{GG}$.

☺

Remarks.

1. *As* C *is a typed graph, the initial graph* I^κ *of* κ *is a typed graph that has a typed graph as type. We will call these kind of graphs* double-typed graphs, *and they will be defined in Sect. 5.* $Rules(C)$ *is the set of rules that have the typed graph* C *as type. The definition of* $Rules(C)$ *is thus analogous to Def. 3.14 of* $Rules(T)$ *in which* T *is a graph.*

2. *In [Kor95] it was shown that each concurrent derivation induces a partial order on its set of rules, that describes the causal relationships between them. This order is defined by the way in which the pre- and post- conditions of each rule overlap in the core graph. This dependency relation will be defined in Sect. 5.1.*

3. *In fact, usually there are infinitely many concurrent derivations with respect to the same sequential derivation because the construction of the core graph delivers a result that is unique up to isomorphism only. All these concurrent derivations are isomorphic, and this will be expressed via morphisms between concurrent derivations (Def. 3.42. For each sequential derivation, there exists exactly one abstract concurrent derivation (Def. 3.45) associated to it, where abstract concurrent derivations are defined as isomorphism classes of concurrent derivations.*

4. *An axiomatic characterization of concurrent derivations as special graph grammars will be given in Sect. 5.4.*

☺

Example **3.41 (Concurrent Derivation)** Concurrent derivations are constructed from sequential ones by identifying items in different intermediate graphs that shall be the same (are connected by morphisms). Figure 2.8 shows the concurrent derivation corresponding to the sequential derivation shown in Figure 2.7. We can see that the input (and output) graphs are the same, and the rules that were applied are also the same. The basic difference is that all matches of the rules have now the same graph as target, namely the *core graph* $C4$. Roughly speaking, this graph is obtained by gluing the graphs IN,$IN1$ and $IN2$ compatibly with the derivation morphisms. Note that no matter which length a sequential derivation has, the corresponding concurrent derivation will always consist of the initial graph of the derivation, the core graph and the set of rules that were applied (together with their matches into the core graph). The output graph will usually not be represented because it can be derived from the other components. From the overlappings of the left- and right-hand sides of rules in this graph, dependencies between rules can be derived. For example, as the matches $pre1/post1$ and $pre2/post2$ are completely disjoint, actions $a1$ and $a2$ are independent and may be executed concurrently. It is interesting to notice here that the total order of applications of rules described in the sequential derivation $\sigma4$ is not anymore present in $\kappa4$. It is replaced by a partial order that indicates the causal dependencies between rule applications. ☺

Relationships between concurrent derivations can be expressed by concurrent derivation morphisms. These relationships express a kind of *concurrent prefix* relationship between derivations, i.e., there may be a morphism f from a concurrent derivation $\kappa1$ to $\kappa2$ if these derivations have the same input graphs, the rules used in $\kappa1$ are included in the rules used in $\kappa2$ and the applications of the same rule in $\kappa1$ and $\kappa2$ are isomorphic (that means, they are the same action up to isomorphism). This "being the same action" can be expressed by commutativity of the corresponding images in the core graphs with the transformation of the core graph of $\kappa1$ into the core graph of $\kappa2$. Moreover, the embeddings of the input graph of $\kappa1$ and $\kappa2$ in the corresponding core graphs must also be compatible with the transformation of the core graph. The intuitive meaning of a concurrent derivation morphism $f : \kappa1 \rightarrow \kappa2$ is that $\kappa1$ is a *(concurrent) prefix* of $\kappa2$, that is, the computation $\kappa1$ may evolve (maybe in parallel with others) to the computation $\kappa2$.

Definition 3.42 (Concurrent Derivation Morphism) *Let GG be a graph grammar and $\kappa1 = (C1, I1^{C1}, N1, n1), \kappa2 = (C1, I2^{C2}, N2, n2) \in CDer_{GG}$. Then a* ***concurrent derivation morphism*** *$f : \kappa1 \rightarrow \kappa2$ is a pair $f = (f_C, f_N)$ where*

- *$f_C : C1 \rightarrow C2$ is a total and injective typed graph morphism in* $\mathbf{TGraphP(T)}$ *and*
- *$f_N : N1 \rightarrow N2$ is a total function*

such that

1. *$I1 = I2$ and (1) commutes*
2. *for all $x = (rn, L1^{C1} \overset{r1^{C1}}{\rightarrow} R1^{C1}) \in N1$, $n2 \circ f_N(x) = (rn, L2^{C2} \overset{r2^{C2}}{\rightarrow} R2^{C2})$, where $L2 = L1$, $R2 = R1$, $r2 = r1$ and (2) and (3) commute.*

$$\begin{tikzcd}
& I1 = I2 \arrow[dl, "in^{I1}"'] \arrow[dr, "in^{I2}"] \arrow[d, phantom, "(1)"] & \\
C1 \arrow[rr, hook, "f_C"'] & & C2
\end{tikzcd}
\quad
\begin{tikzcd}
& L1 = L2 \arrow[dl, "pre^{r1^{C1}}"'] \arrow[dr, "pre^{r2^{C2}}"] \arrow[d, phantom, "(2)"] & \\
C1 \arrow[rr, hook, "f_C"'] & & C2
\end{tikzcd}
\quad
\begin{tikzcd}
& R1 = R2 \arrow[dl, "post^{r1^{C1}}"'] \arrow[dr, "post^{r2^{C2}}"] \arrow[d, phantom, "(3)"] & \\
C1 \arrow[rr, hook, "f_C"'] & & C2
\end{tikzcd}$$

The category of all concurrent derivations with respect to GG and all concurrent derivation morphisms between them is denoted by $\mathbf{CDer_{GG}}$.

Two concurrent derivations are isomorphic when both components f_C *and* f_N *are isomorphisms.* ☺

Example **3.43 (Concurrent Derivation Morphisms)** Consider the concurrent derivations $\kappa 1$, $\kappa 2$ and $\kappa 4$ of Figure 3.10. These are the same derivations discussed in Sect. 2.2.5, we just omitted the context items from the graphical representation to ease the understanding. All these derivations have the same input graph IN, and use the same rule $PBXr1$ (in $\kappa 4$, this rule is used twice). Intuitively, we may expect that $\kappa 1$ and $\kappa 2$ may evolve to $\kappa 4$ because we may find in $\kappa 4$ actions that correspond to the ones performed in these two derivations. That is, $\kappa 1$ and $\kappa 2$ are (concurrent) prefixes of $\kappa 4$. But $\kappa 1$ may never evolve to $\kappa 2$ because they represent completely distinct actions. As morphisms shall represent this prefix relationship, we expect that there are morphisms $f : \kappa 1 \to \kappa 4$, $g : \kappa 2 \to \kappa 4$, but no morphism $h : \kappa 1 \to \kappa 2$.

To find out whether there is a concurrent derivation morphism between two concurrent derivations, we have to map all actions of one concurrent derivation into the other (f_N) and find a morphism f_C between the corresponding core graphs that commute with the pre and $post$ images of the rules in these core graphs. Consider f_C as being the inclusion of $C2$ into $C4$ and $f_N(a1) = a1'$. In this case, it is trivial to see that $f_C \circ pre1(x) = pre1'(x)$ for all items x of L1, and that the same holds for $post1$. Thus there is a morphism $f = (f_C, f_N) : \kappa 2 \to \kappa 4$. Analogously we may find a morphism $g : \kappa 1 \to \kappa 4$. However, there is no concurrent derivation morphism between $\kappa 1$ and $\kappa 2$ because the PHONE vertex of the rule $PBXr1$ is mapped by $pre1$ to PHONE_1 and by $pre2$ to PHONE_2. The only possibility to find a morphism $h : \kappa 1 \to \kappa 2$ would be to map PHONE_1 of $\kappa 1$ to PHONE_2 of $\kappa 2$ via h_C. But such a mapping wouldn't be compatible with the inclusion of the input graph IN into $C1$ and $C2$, and is thus forbidden. ☺

Proposition 3.44 *Let* $\kappa 1, \kappa 2 \in CDer_{GG}$. *Then there is at most one concurrent derivation morphism* $f : \kappa 1 \to \kappa 2$. ☺

Proofidea. For a formal proof of this Proposition, see [Kor96]. This proof is based on the fact that each application of a rule depends in a unique way on the initial graph and on the rules that created items that are necessary for this rule to be applied. The basic requirement to assure this uniqueness is that all rules must delete something. In a concurrent derivation, there can't be an item that is deleted by more than one rule. Assume that a rule r of $\kappa 1$ deletes an item x. As the core graph component of a morphism must be injective, there can be only one way (if there is one) to map r to a corresponding rule in another concurrent derivation. $\surd$

The fact that there is at most one concurrent derivation morphism between two concurrent derivations (in each direction) is quite useful as a suitable means to find out which computations are equivalent with respect to isomorphism non-determinism. Consider two derivations $\kappa 1$ and $\kappa 2$ and morphisms $f : \kappa 1 \to \kappa 2$ and $g : \kappa 2 \to \kappa 1$. Intuitively, this means that $\kappa 1$ is a (concurrent) prefix of $\kappa 2$ and vice versa, i.e., they represent the same computation. Formally, the existence of morphisms assures that both derivations have the same initial graphs and the same rules are used in the same way. Moreover, the core graphs are isomorphic (there are total and injective morphisms in both directions, and as the sets of used rules are the same, these morphisms must be surjective too). This means that $\kappa 1$ and $\kappa 2$ are the same derivation, except for the core graphs (that are isomorphic). But this implies that they are equivalent with respect to isomorphism non-determinism and shall be considered as the same computation. Therefore, the concurrent semantics of a graph grammar (Def. 3.46) is based on isomorphism classes of concurrent derivations (Def. 3.45).

Definition 3.45 (Abstract Concurrent Derivation) *Let $\kappa 1, \kappa 2 \in Cder_{GG}$ be concurrent derivations and $f : \kappa 1 \rightarrow \kappa 2$ be a concurrent derivation morphism. Then an **abstract concurrent derivation** $[\kappa 1]$ is the class of concurrent derivations isomorphic to $\kappa 1$. An **abstract concurrent derivation morphism** is the equivalence class $[f]$ of morphisms between elements of $[\kappa 1]$ and $[\kappa 2]$.*

Abstract concurrent derivations and morphisms form a category denoted by $\mathbf{ADer_{GG}}$. ☺

Remark. *Prop. 3.44 assures that $\mathbf{ADer_{GG}}$ is well-defined.* ☺

Definition 3.46 (Concurrent Semantics) *The **concurrent semantics** of a graph grammar GG is given by the category $\mathbf{ADer_{GG}}$.* ☺

As discussed at the end of last section, the concurrency semantics does not have two of the drawbacks of the sequential one:

1. *Concurrency is represented explicitly.* If two actions may happen concurrently, there is only one concurrent derivation that represent this computation. Moreover, due to the partial order on actions induced by a concurrent derivation, one can see directly whether each two actions belonging to a concurrent derivation are or not independent from each other. (This partial order will be defined in Chap. 5 for a more general class of concurrent derivations, called occurrence graph grammars.)

3. *There is no representation non-determinism* because isomorphic derivations are considered to be the same computation.

However, there is still one point, namely point 2., that is not yet suitably described by the concurrent semantics. This item is concerned about mutually exclusive situations. To find out whether two concurrent derivations $\kappa 1$ and $\kappa 2$ are mutually exclusive or not one has to search for a bigger concurrent derivation that is an evolution of both of them. If such a derivation can be found, $\kappa 1$ and $\kappa 2$ are not mutually exclusive. Otherwise, they are mutually exclusive. Thus, *non-determinism is described implicitly* by the non-existence of some upper bounds. The unfolding semantics presented in Chap. 6 will also solve this drawback.

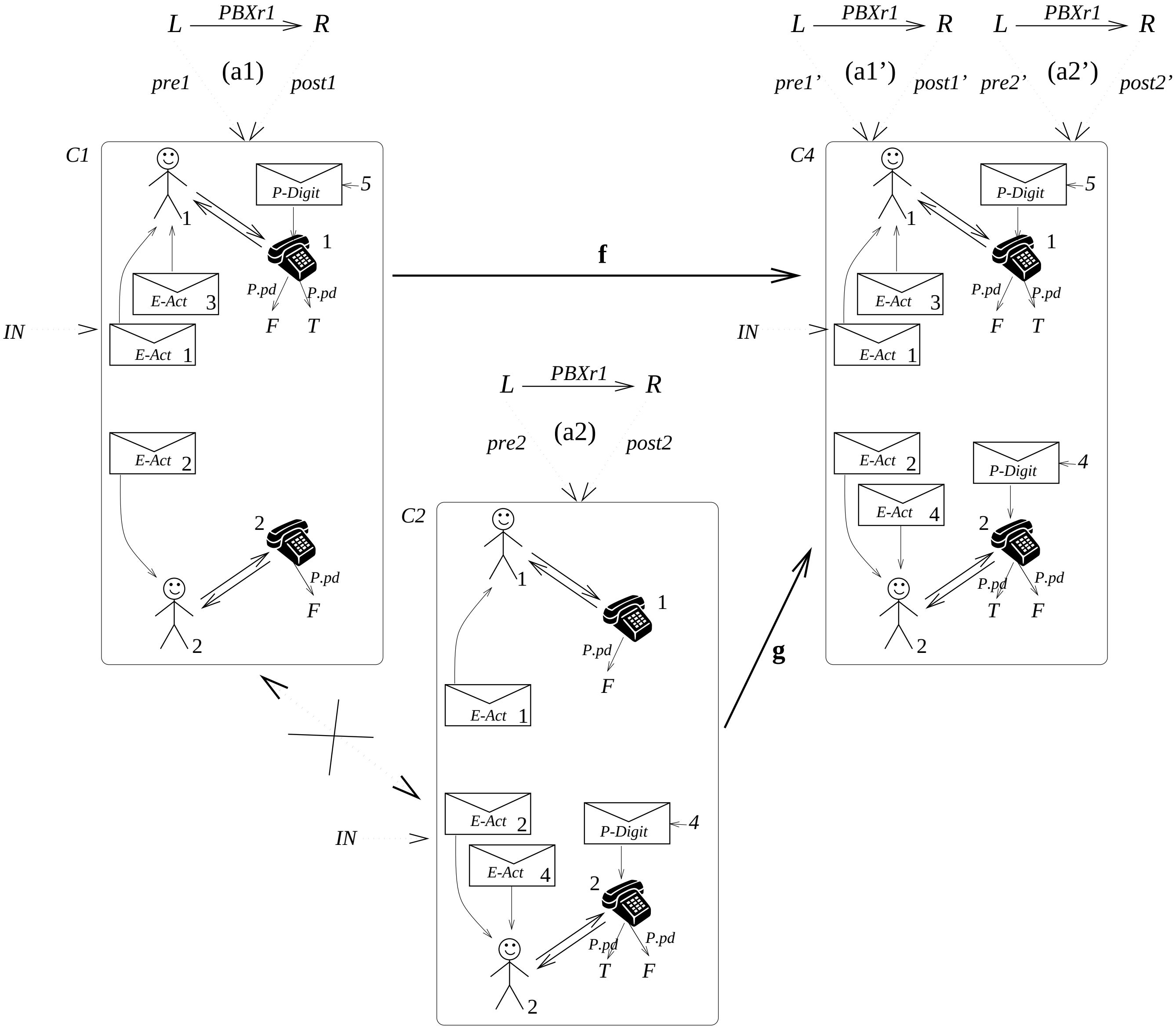


Figure 3.10: Concurrent Derivation Morphisms

4

Parallel Composition of Graph Grammars

The specification of complex systems is usually done by the "divide and conquer" idea: the system is divided into smaller, less complex components that are developed separately and then merged in some way to form the specification of the whole system. A suitable formalism that supports such a development shall assure that the composition operators used to merge the component specifications are compatible with the semantics of the system. Compatibility here means that the behaviour of the whole system can be derived from the behaviours of its components, i.e., the composed system does not shows a behaviour that is not specified in any of its components. This property is very desirable for a specification formalism, but it is not easy to achieve in formalisms in which initial states are considered, such as graph grammars, Petri nets with initial markings and transition systems. The reasons for this will be discussed in Sect. 4.2, where parallel composition operators for graph grammars will be presented. Specially the *cooperative parallel composition* (Sect. 4.2.2) seems to be a very promising kind of composition of graph grammars. This kind of composition formalizes the intuitive idea of divide and conquer described above: an abstract description of a system is divided into components that are further specialized and then merged together to form the specification of the whole system. The important requirement is that each component is a kind of conservative extension of the abstract description of the system, in the sense that the specialization of the abstract view defined in the component does not imply that the behaviour of the abstract level would change. Such specializations are formalized by *graph grammar morphisms* (Sect. 4.1). These morphisms are not only interesting to describe specializations (refinements) of grammars, but also to express structural and behavioural compatibilities between graph grammars.

The main aims, definitions and results of this chapter are:

- Definition of syntactical relationships (morphisms) between graph grammars (Def. 4.1) that are compatible with their operational behaviour (Theo. 4.11). This is presented in Sect. 4.1.

- Definition of two parallel composition operators for graph grammars: pure parallel composition (Def. 4.12), that is a composition without any interface, and cooperative parallel composition (Def. 4.20). Moreover, it is shown that these composition operators correspond to the product (Theo. 4.16) and the pullback (Theo. 4.24) in the category of graph grammars, respectively. From the fact that these operators correspond to these categorical constructions it follows that these operators are associative and compatible with each other. Moreover, this fact will be used to show that the unfolding semantics is compatible with the parallel composition.

As discussed above, suitable composition operators should be compatible with semantics. In Chap. 6 (Theo. 6.18) it will be shown that the parallel composition operators defined here are compatible with the unfolding semantics of graph grammars, that is a semantics specially suitable for concurrent systems.

4.1 Graph Grammar Morphisms

Different definitions of morphisms may describe different kinds of relationships between objects. The 'right' definition of a morphism is therefore dependent on which kinds of relationships one expect to describe. The basic application of morphisms we have in mind is to use them to compose graph grammars. Therefore, we are interested in relationships that express a structural (syntactical) compatibility between grammars. Moreover, as we expect that composition is compatible with semantics, morphisms should imply a corresponding semantical compatibilities. Many kinds of morphisms between graph grammars (and graph transformation systems) have already been defined [CH95, CEL$^+$96a, HCEL96, Kor96, PP96]. Most of them were defined for the DPO approach to graph grammars, but the aims were different than the aim of the morphisms that will be defined here. In [PP96] the aim is to transform graph grammars in such a way that the graph language is preserved, but not necessarily the sequential derivations. For reactive and concurrent systems often the way in which computations evolve is more important than the states that are reached [MP92]. Therefore, as our aim is to model this kind of systems, we will rather concentrate on morphisms that preserve derivations. In [CEL$^+$96a] a powerful notion of morphism based on spans of graph morphisms was defined and was shown to preserve sequential derivations. But as the category of graph-spans is known not to have all pushouts [Wag93, Mue95], interesting constructions like the cooperative parallel composition (see Sect. 4.2.2) would probably not be possible in this more general framework. The kind of morphism presented here is a specialization of this kind of morphism by substituting the span by a partial graph morphism (contrastingly to [CEL$^+$96a], here we allow rules to be mapped to arbitrary ones that are isomorphic to the corresponding retyping). In [CH95, HCEL96] a different specialization of the morphisms of [CEL$^+$96a] was done, namely to substitute the span by a partial morphism in the other direction as we do (that is, the partial morphism here goes in the direction of the graph transformation system morphism). This kind of morphism was defined for graph transformation systems (graph grammars without initial graph) and were also shown to preserve sequential derivations. Moreover, disjoint union and union with interface of graph transformation systems were presented and were shown to be compatible with the sequential derivations (in fact, with the sequential derivations starting from any graph). As the initial graph is an important component of the specification of a system (see discussion in Sect. 4.2.2), it is worthwhile to investigate such composition operators for this case (that is not a trivial extension of the case without initial graph). In [Kor96] morphisms between concurrent derivations, that are special graph grammars, were defined. The purpose of these morphisms is to express relationships between concurrent computations of one grammar (see Sect. 3.3.2). These morphisms not only preserve sequential derivations, but also the dependencies between the steps (or actions). However, these morphisms are very restricted to be used as a general kind of relationship between graph grammars.

The following are the main requirements that should be satisfied by the definition of a suitable morphism of graph grammars in view of composition operators:

1. The morphisms shall describe reasonable syntactical relationships between the rules and initial graphs of grammars.

2. Morphisms shall be compatible with derivation sequences, i.e., if two grammars are connected by a morphism, there should be a reasonable way to translate derivations of the first grammar into derivations of the second one.

3. Morphism shall be composable such that they are able to express transitive relationships between grammars (allowing the definition of a category).

4. Morphisms shall allow for the definition of a category in which some syntactical constructions exist (e.g. products and pullbacks) and these constructions shall have a reasonable interpretation in terms of operations on graph grammars (e.g. parallel composition).

The definition of graph grammar morphism presented here was inspired from the definition of morphisms between transition systems in [WN94] and between Petri nets in [MMS96]. A graph grammar morphism $f : GG1 \to GG2$ consists of two components: a (partial) graph morphism $f_T : T2 \to T1$ of types and a (partial) function $f_N : N1 \to N2$ of rule names. The mapping of types allows to split types, whereas the mapping of rule names allows to identify rules. The fact that types and rule names are mapped in opposite directions expresses a kind of duality between types, that represent *static* aspects of a system, and rules, that represent *dynamic* aspects of a system. Moreover, f_T and f_N must be compatible with each other, and this compatibility assures a structural relationship between these two grammars. Compatibility here means that the rule associated to a translated rule name must be a subrule of the translation of the corresponding rule induced by the type translation, and that the initial graph of $GG2$ is isomorphic to the translation of the initial graph of $GG1$ induced by f_T. Both f_N and f_T may be partial, what implies that $GG1$ may have more rules and types than $GG2$, and a bigger initial graph. The compatibility condition of the morphism assures additionally that the rules of $GG1$ that are not mapped do not affect the behaviour of the target grammar (i.e., it is possible to translate derivations of $GG1$ into derivations of $GG2$). $GG2$ may also have more rules and types that are not in the image of f, but the retyping of its initial graph must be (up to isomorphism) a subgraph of the initial graph of $GG2$.

Formally, we will define a graph grammar morphism as being a pair of morphisms that go in the same direction, but the mapping of types will be a morphism in the dual category of **GraphP**, and therefore usually denoted by f_T^{OP}. We will denote the corresponding morphism in **GraphP** by f_T.

Definition 4.1 (Graph Grammar Morphism) *Let $GG1 = (T1, I1^{T1}, N1, n1)$ and $GG2 = (T2, I2^{T2}, N2, n2)$ be two graph grammars. Then a* ***graph grammar morphism*** *$f : GG1 \to GG2$ is a pair $f = (f_T^{OP}, f_N)$ where $f_T^{OP} : T1 \to T2$ is morphism in* **GraphP$^{\mathbf{OP}}$** *and $f_N : N1 \to N2$ is a partial function such that the following conditions are satisfied:*

1. Sub-commutativity: $\overline{n2} \circ f_N \subseteq^R \mathcal{R}_{f_T} \circ \overline{n1}$, *where* $\overline{n1}$ *and* $\overline{n2}$ *are the extensions of* $n1$ *and* $n2$ *(see Def. 3.33) and* $\subseteq^R$ *is the subrule relation (Def. 3.16).*

$$\begin{tikzcd} N1 \arrow[r, mapsto, "\overline{n1}"] \arrow[d, "f_N"'] \arrow[dr, phantom, "\subseteq^R"] & \mathcal{R}(T1) \arrow[d, "\mathcal{R}_{f_T}"] \\ N2 \arrow[r, mapsto, "\overline{n2}"'] & \mathcal{R}(T2) \end{tikzcd}$$

2. $I2^{T2} \cong \mathcal{T}_{f_T}(I1^{T1})$

A morphism in which f_N *and* $(f_T)^{-1}$ *are inclusions and is called* ***inclusion***.[1] *We say that* $GG1 \subseteq GG2$ *is there is an inclusion* $i : GG1 \to GG2$. ☺

Remark. *The informal meaning of the conditions of the morphism are:*

1. *Whenever a rule name* nr *is mapped via* f_N *then the rule* $r2$ *associated to* $f_N(nr)$ *(i.e.,* $n2 \circ f_N(nr) = r2$*) must be a subrule of the translation of the rule associated to* nr *(*$f_N(nr)$*). This means that graph grammar morphisms allow to map a rule to a subrule (up to translation and isomorphism). The sub-commutativity condition also assures that if a rule name is not mapped then the corresponding rule pattern is also not mapped. The use of* $IGRules$ *instead of* $IRules$ *(that is considering also general rules) assures that is a rule name is not mapped, the corresponding "rule" in the second grammar would be an isomorphism (that is not a rule). Intuitively this means that rules that are not mapped necessarily lead to a no-op (no-operation) in the target grammar. This requirement (using general rules) is necessary for the preservation of derivations via morphisms (this will be discussed in Example 4.4).*

2. *The initial graph of* $GG2$ *is isomorphic to the translation of the initial graph of* $GG1$ *to* $GG2$*. This means that the initial graph of* $GG2$ *must be in some sense included in the one of* $GG1$*.*

☺

Example **4.2 (Graph Grammar Morphism)** Consider the graph grammars AGV and PLV depicted in Figures 2.2 and 2.3 resp. A graph grammar morphism $f : PLV \to AGV$ can be defined by the following components: f_T is the inclusion of $Type$ (Figure 2.1) into $PType$ and $f_N = \{Pr1 \mapsto r1, Pr2.1 \mapsto r2, Pr2.2 \mapsto r2, r3 \mapsto r3\}$. The compatibility conditions of a morphism require that:

1. $r1$ is a subrule of the rule obtained from $Pr1$ by forgetting all items that are typed over items $PType$ that are not in $Type$ (in this example, $r1$ is isomorphic to the retyping of $Pr1$). This "forgetting" is done by the retyping construction induced by f_T. Analogously, this requirement must also be satisfied for the other rules that are mapped via f_N.

2. Ini must be isomorphic to the graph obtained from $PIni$ by forgetting all items that are typed over items $PType$ that are not in $Type$.

[1] Note that an inclusion morphism is only possible if f_T is injective and surjective.

In the same way, we may find graph grammar morphisms $g : CLV \rightarrow AGV$, $f^\bullet : CGV \rightarrow CLV$ and $g^\bullet : CGV \rightarrow PLV$. ☺

Proposition 4.3 *Graph grammars and graph grammar morphisms form a category, denoted by* **GG**, *in which identities and composition are defined componentwise.* ☺

Proof.

1. *Identities are well-defined morphisms:* Let $GG = (T, I^T, N, n)$ be a graph grammar. Then we have to show that the pair $id = (id_T^{OP}, id_N)$ is a graph grammar morphism, i.e., it must satisfies conditions i) and ii) of the Def. 4.1

   (a) Diagram (1) commutes because $\mathcal{R}$ is a functor, and thus transforms identities into identities. Therefore, we trivially obtain that $\overline{n} \circ id_N \subseteq^R \mathcal{R}_{id_T} \circ \overline{n}$.

$$\begin{tikzcd} N \arrow[r, maps to, "\overline{n}"] \arrow[d, "id_N"'] \arrow[dr, phantom, "(1)"] & \mathcal{R}(T) \arrow[d, "\mathcal{R}_{id_T^{OP}}"] \\ N \arrow[r, maps to, "\overline{n}"'] & \mathcal{R}(T) \end{tikzcd}$$

   (b) Let $G^T = \mathcal{T}_{id_T}(I^T)$. As the diagram below is a pullback, i is an isomorphism and thus $G^T \cong I^T$.

$$\begin{tikzcd} I \arrow[r, maps to, "t^I"] \arrow[dr, phantom, "(PB)"] & T \\ G \arrow[u, "i"] \arrow[r, maps to, "t^G"'] & T \arrow[u, "id_T"'] \end{tikzcd}$$

2. *Composition is well-defined:* Let $GGi = (Ti, Ii^{Ti}, Ni, ni)$ be graph grammars, for $i = 1..3$, and $f = (f_T^{OP}, f_N) : GG1 \rightarrow GG2$ and $g = (g_T^{OP}, g_N) : GG2 \rightarrow GG3$ be graph grammar morphisms. Then we have to show that $g \circ f = (g_T^{OP} \circ f_T^{OP}, g_N \circ f_N)$ is a well-defined morphism:

   (a) Diagrams (2) and (3) sub-commute because f and g are morphisms. As (3) sub-commutes we have that $\overline{n3} \circ g_N \circ f_N \subseteq^R \mathcal{R}_{g_T} \circ \overline{n2} \circ f_N$. Prop. 3.19 assures that $\mathcal{R}_{g_T}$ preserves the subrule relation, and thus as (2) sub-commutes we get that $\overline{n3} \circ g_N \circ f_N \subseteq^R \mathcal{R}_{g_T} \circ \mathcal{R}_{f_T} \circ \overline{n1}$, i.e., (2)+(3) sub-commute.

$$\begin{tikzcd} N1 \arrow[r, maps to, "\overline{n1}"] \arrow[d, "f_N"'] \arrow[dr, phantom, "(2)"] & \mathcal{R}(T1) \arrow[d, "\mathcal{R}_{f_T}"] \\ N2 \arrow[r, maps to, "\overline{n2}"] \arrow[d, "g_N"'] \arrow[dr, phantom, "(3)"] & \mathcal{R}(T2) \arrow[d, "\mathcal{R}_{g_T}"] \\ N3 \arrow[r, maps to, "\overline{n3}"'] & \mathcal{R}(T3) \end{tikzcd}$$

   (b) $I3^{T3} = \mathcal{T}_g \circ \mathcal{T}_f(I1^{T1}$. Let $I3'^{T3} = \mathcal{T}_{g_T \circ f_T}(I1^{T1})$. By Prop. 3.13 we conclude that $I3^{T3} \cong I3'^{T3}$.

3. Neutrality of identity and associativity of composition follow from these properties in $\mathbf{GraphP^{OP}}$ and **SetP** and from the componentwise construction of morphisms.

$\surd$

Now, until the end of this section, we will prove that graph grammar morphisms are compatible with the sequential and concurrent semantics of graph grammars, i.e., if there is a morphism $f : GG1 \to GG2$, we can translate the (concurrent and sequential) derivations of $GG1$ into corresponding ones of $GG2$. This is stated in Theo. 4.11. The basic idea of this translation will be illustrated in the following example.

Example **4.4 (Translation of Derivation Sequences)** Consider the graph grammar $GG1 = (T1, I1, \{r1, r2, r3\}, n1)$ and $GG2 = (T2, I2, \{r1', r3'\}, n2)$ where $T1 = (\{\bullet, \bigstar, \blacksquare\}, \emptyset, \emptyset, \emptyset)$, $T2 = (\{\circ, \square\}, \emptyset, \emptyset, \emptyset)$ and the initial graphs and rules are depicted in Figure 4.1. Let $f : GG1 \to GG2$ be a graph grammar morphism having the following components $f_T = (\{\circ \mapsto \bullet, \square \mapsto \blacksquare\}, \emptyset)$ and $f_N = \{r1 \mapsto r1', r2 \mapsto \mathtt{undef}, r3 \mapsto r3'\}$ (i.e., rule $r2$ is not mapped by the morphism). The sequential derivation $\sigma1$ of $GG1$ depicted in Figure 4.1 can be translated to the sequential derivation $\sigma2$ of $GG2$. This is done in 2 steps:

1. *Translate* each derivation step of $\sigma1$ according to the morphism f. This step gives raise to a sequence l that is not a sequential derivation of $GG2$ because the empty rule $r2'$ is not a rule of $GG2$.

2. Delete the empty steps from l. This step is called *normalization* (in the sense that a sequential derivation is a normal form of a sequence including empty steps) and gives raise to the sequential derivation $\sigma2$. It may also be necessary to substitute the initial graph of l by the initial graph of $GG2$ (they must be isomorphic due to condition 2. of graph grammar morphisms).

Note that, if the rule $r2$ would create something, say a $\blacksquare$ such that the rule $r3$ would become dependent on this rule (by changing the current match to this new $\blacksquare$), then the retyping $\mathcal{T}_{f_T}(r2)$ of $r2$ would yield a rule $L \to R$ where L is the empty graph and R has a $\blacksquare$ vertex. In this case, $\mathcal{T}_{f_T}(r2)$ would not be an isomorphism and its corresponding isomorphism class would belong to $\mathcal{R}_{f_T}(T2)$. Therefore f would not be a graph grammar morphism (condition 1. is violated: $\overline{n2} \circ f_N(r2) = \mathtt{undef} \neq [\mathcal{T}_{f_T}(r2)]$). An analogous situation would occur if $r2$ would delete some item created by the rule $r1$. ☺

The next lemma will be used to prove that graph grammar morphisms preserve sequential semantics. The basic idea is that rules that are not mapped by the morphism must lead to a no-operation in the second grammar. Isomorphisms denote no-operations because the application of a "rule" that is an isomorphism would leave the input graph unchanged.

Lemma 4.5 *Let $f : GG1 \to GG2$ be a graph grammar morphism and $x \in N1$ be a rule name in $GG1$ such that $f_N(x)$ is undefined (i.e., $x \notin dom(f_N)$). Then $\mathcal{T}_{f_T}(n1(x))$ is an isomorphism.* ☺

Proof. Let $n1(x) = r1$ and $[r1] = \overline{n1}(x)$ (this is defined because $n1$ and $\overline{n1}$ are total). By definition of a graph grammar morphism (Def 4.1) we have that $\overline{n2} \circ f_N \subseteq^R \mathcal{R}_{f_T} \circ \overline{n1}$. As $\overline{n1}$

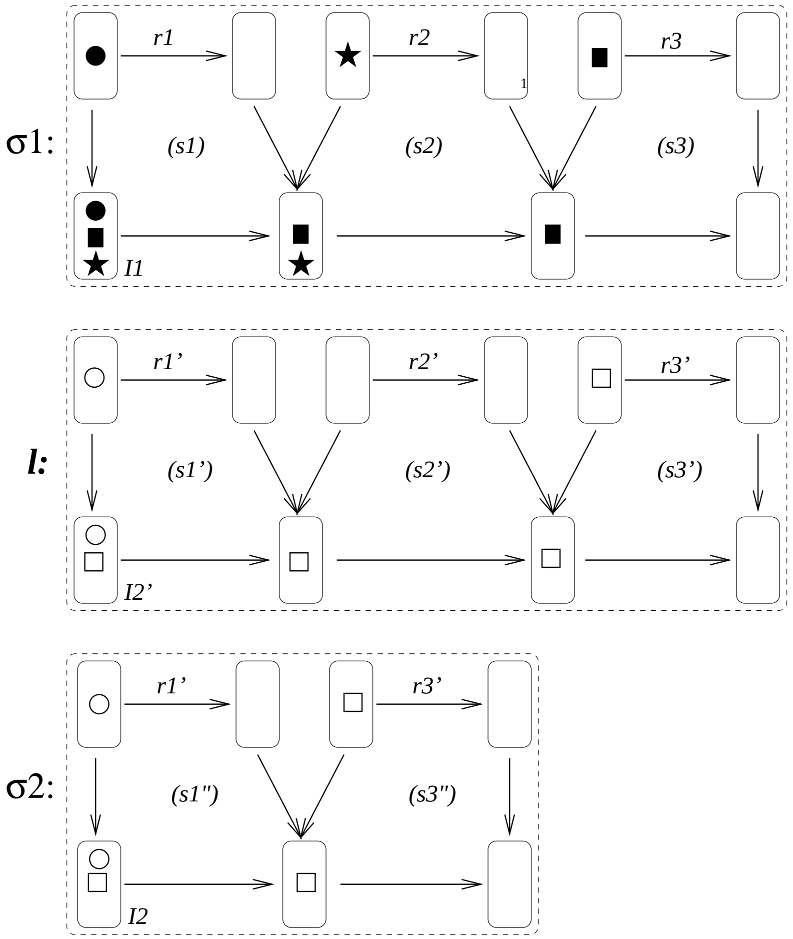


Figure 4.1: Translation of Derivation Sequences

is total and $f_N(x)$ is undefined, $\mathcal{R}_{f_T}([r1])$ must be undefined too. By definition of the functor $\mathcal{R}$ (Def. 3.14), this can only be the case if $\mathcal{T}_{f_T}(r1)$ is an isomorphism. $\surd$

Before we can prove Theo. 4.11, we have to define how sequential derivations of one grammar can be translated into sequential derivation of a second grammar using a graph grammar morphism. This translation will be based on the translation of derivation steps.

Proposition 4.6 *Let $f = (f_{TOP}, f_N) : GG1 \to GG2$ be a graph grammar morphism. Then a total function $f^s : Steps_{GG1} \to StepsI_{GG2}$ is called* ***translation of derivation steps*** *iff for all $s1 = (nr_{s1}, S1) \in Steps_{GG1}$, where $S1 = (IN1 \overset{nr_{s1}:m_{s1}}{\Longrightarrow} OUT1))$, we have a subrule* *relation* $(i_L, i_R) : n1(nr_{s1}) \to n2 \circ f_N(nr_{s1})$ *and*

$$f^s(s1) = \begin{cases} (f_N(x_{s1}), S2), \text{ if } x1 \in dom(f_N), \text{ where} \\ \qquad S2 = (\mathcal{T}_{f_T}(IN1) \overset{f_N(x_{s1}):m_{s2}}{\Longrightarrow} \mathcal{T}_{f_T}(OUT1)) \text{ with} \\ \qquad m_{s2} : L_{s2} \to \mathcal{T}_{f_T}(IN1) = i_L \circ \mathcal{T}_{f_T}(m_{s1}), \\ S2, \text{ if } x1 \notin dom(f_N), \text{ where} \\ \qquad S2 = (\mathcal{T}_{f_T}(IN1) \overset{r_{s2}:m_{s2}}{\Longleftrightarrow} \mathcal{T}_{f_T}(OUT1)) \text{ with} \\ \qquad r_{s2} = \mathcal{T}_{f_T}(r_{s1}), m_{s2} = \mathcal{T}_{f_T}(m_{s1}) \end{cases}$$

☺

Proof. We have to show that $f^s(s1) \in StepsI_{GG2}$.

1. $x_{s1} \in dom(f_N)$: In this case, there is $x_{s2} = f_N(x_{s1})$. Let $n1(x_{s1}) = r_{s1}$. By definition of graph grammar morphisms (Def. 4.1) we have that $n2(x_{s2}) = r_{s2} \subseteq^r \mathcal{T}_{f_T}(r_{s1})$. Thus there are total and injective morphisms i_L and i_R such that (1) below is a pushout. As m_{s1} is a match, it is total and thus by Lemma C.1 $\mathcal{T}_{f_T}(m_{s1})$ is also total. Thus, $m_{s2} = i_L \circ \mathcal{T}_{f_T}(m_{s1})$ is a match for r_{s2}. Diagram (3) ($\mathcal{T}_{f_T}(S1)$) is a pushout because $S1$ is a pushout of an injective and a total morphisms and the functor $\mathcal{T}_{f_T}$ preserves these pushouts (Prop. 3.12). Thus, $S2 = (1) + (3) = (\mathcal{T}_{f_T}(IN1) \overset{x_{s2}:m_{s2}}{\Longrightarrow} \mathcal{T}_{f_T}(OUT1))$ is also a pushout. As r_{s2} is a rule of $GG2$ and m_{s2} is a match for r_{s2}, the derivation step $(x_{s2}, S2)$ is in $Steps_{GG2} \subseteq StepsI_{GG2}$.

$$\begin{tikzcd}
\mathcal{T}_{f_T}(L_{s1}) \arrow[rr, "\mathcal{T}_{f_T}(r_{s1})"] \arrow[dd, "\mathcal{T}_{f_T}(m_{s1})"'] \arrow[d, phantom, "(2)"] \arrow[rrd, phantom, "(1)"] & & \mathcal{T}_{f_T}(R_{s1}) \arrow[dd] & \\
& L_{s2} \arrow[ul, hook, "i_L"'] \arrow[rr, "r_{s2}"'] \arrow[dl, "m_{s2}"] \arrow[dr, phantom, "(3)"] & & R_{s2} \arrow[ul, hook, "i_R"'] \arrow[dl, "m^\bullet_{s2}"] \\
\mathcal{T}_{f_T}(IN1) \arrow[rr] & & \mathcal{T}_{f_T}(OUT1) &
\end{tikzcd}$$

2. $x_{s1} \notin dom(f_N)$: By Lemma 4.5, if $x_{s1} \notin dom(f_N)$ then $\mathcal{T}_{f_T}(n1(x_{s1})) = \mathcal{T}_{f_T}(r_{s1})$ is an isomorphism. Like for the first case, we have that $\mathcal{T}_{f_T}(m_{s1})$ is total because m_{s1} is total. As pushout of an injective and a total morphism are preserved by $\mathcal{T}_{f_T}$, we conclude that $f^s(s1) = S2$ is a pushout in **TGraphP(T2)**, and as pushouts of an isomorphism and a total match are empty steps (see Def. 3.35), $f^s(s1) \in StepsI_{GG2}$.

$$\begin{tikzcd}
\mathcal{T}_{f_T}(L_{s1}) \arrow[rr, "\mathcal{T}_{f_T}(r_{s1})"] \arrow[dd, "\mathcal{T}_{f_T}(m_{s1})"'] \arrow[d, phantom, "(2)"] \arrow[rrd, phantom, "(1)"] & & \mathcal{T}_{f_T}(R_{s1}) \arrow[dd] & \\
& L_{s2} \arrow[ul, equal, "i_L=id"'] \arrow[rr, "r_{s2}"'] \arrow[dl, "m_{s2}"] & & R_{s2} \arrow[ul, equal, "i_R=id"'] \arrow[dl, "m^\bullet_{s2}"] \\
\mathcal{T}_{f_T}(IN1) \arrow[rr] & & \mathcal{T}_{f_T}(OUT1) &
\end{tikzcd}$$

√

The next construction will be used to transform derivation sequences from one grammar into derivation sequences of another grammar. This transformation will be done in two steps:

1. *Translation:* Translate the derivation sequences of one grammar into sequences of steps of the second grammar according to a given graph grammar morphism. As morphisms are allowed to be partial, this translation may lead to sequences including empty steps.

2. *Normalization:* Remove the empty steps from the derivation obtained in the first step. This step will give raise to a sequential derivation of the second grammar.

Definition 4.7 (Normalization) *Let GG be a graph grammar and $l \in StepsI^{\infty}_{GG}$ such that $l = \lambda$ or $OUT_{i-1} = IN_i$, for all $i = 2..|l|$. Then the* **normalization** *of l, denoted by $norm(l)$, is defined as follows*

$$norm(l) = \begin{cases} \lambda, & \text{if } (l = \lambda) \text{ or } (l = s1 \text{ and } r_{s1} \text{ is an isomorphism}) \\ s1 \bullet norm(l1), & \text{if } l = s1 \bullet l1 \text{ and } r_{s1} \text{ is not an isomorphism} \\ norm(l1'), & \text{if } l = s1 \bullet l1 \text{ and } r_{s1} \text{ is an isomorphism, where} \\ & l1' = \begin{cases} \lambda, \text{ if } l1 = \lambda \\ s2' \bullet l2, \text{ if } l1 = s2 \bullet l2 \end{cases} \end{cases}$$

$$\begin{tikzcd}
L_{s1} \arrow[r, "r_{s1}"] \arrow[d, "m_{s1}"'] & R_{s1} \arrow[dr] & & L_{s2} \arrow[r, "r_{s2'}=r_{s2}"] \arrow[dl, "m_{s2}"'] \arrow[dlll, bend right, "m_{s2'}=(r^{\bullet}_{s1})^{-1}\circ m_{s2}"'] & R_{s2} \arrow[d, "m^{\bullet}_{s2}=m^{\bullet}_{s2'}"] \\
IN_{s1} \arrow[rr, shift right, "r^{\bullet}_{s1}"'] \arrow[rrrr, bend right, "r^{\bullet}_{s2'}=r^{\bullet}_{s2}\circ r^{\bullet}_{s1}"'] & & OUT_{s1} = IN_{s2} \arrow[ll, shift right, "(r^{\bullet}_{s1})^{-1}"'] \arrow[rr, "r^{\bullet}_{s2}"'] & & OUT_{s2}
\end{tikzcd}$$

☺

Remark. *The 3 cases of the definition of the normalization can be explained as follows:*

$norm(l) = \lambda$: *This is the case if l is the empty list or a list with only one empty step.*

$norm(l) = s1 \bullet norm(l1)$: *In this case the first step of the list l is not an empty step. Then the normalization of this list leaves this step as it is and concatenate it with the normalization of the rest of the list.*

$norm(l) = norm(l1')$: *Here the head of the list ($s1$) was an empty step. Therefore this step shall not be part of the result. But as we want to get as a result a list in which the output graph of one step coincides with the input graph of the following step, we have to make the input graph of the next step of the list become the input graph of the empty step that will be deleted (this step is possibly the output of some other step). As $s1$ is an empty step, its rule r_{s1} is an isomorphism and therefore $r^{\bullet}_{s1}$ is also an isomorphism and can be inverted. Thus the match of the new step $s2'$ is defined as $m_{s2'} = (r^{\bullet}_{s1})^{-1} \circ m_{s2}$, and the co-rule is defined as $r^{\bullet}_{s2'} = r^{\bullet}_{s2} \circ r^{\bullet}_{s1}$. As $(r^{\bullet}_{s1})^{-1}$ is an isomorphism, the resulting diagram $S2'$ is also a pushout.*

☺

The next proposition shows that the normalization of l is a sequence of derivation steps of the grammar GG that doesn't include empty steps. Moreover, the output of each step in this sequence is equal to the input of the subsequent step.

Proposition 4.8 *Let GG be a graph grammar and $l \in StepsI^{\infty}_{GG}$ such that $l = \lambda$ or $OUT_{i-1} = IN_i$, for all $i = 2..|l|$. Then $norm(l) \in Steps^{\infty}_{GG}$ and $norm(l) = \lambda$ or $OUT_{i-1} = IN_i$, for all $i = 2..|norm(l)|$. Moreover, if $l \in SDerI_{GG}$ then $norm(l) \in SDer_{GG}$.* ☺

Proof. See Appendix C. √

Definition 4.9 (Normalized-Translation of Derivation Sequences) *Let $f : GG1 \rightarrow GG2$ be a graph grammar morphism, f^s be a corresponding translation of derivation steps and $\sigma \in SDer_{GG1}$. Then the* ***normalized translation*** *of σ induced by f^s, denoted by $ntran_{f^s}(\sigma)$ is defined as follows:*

1. Translation:
$$tran_{f^s}(\sigma) = \begin{cases} \lambda, & \textit{if } \sigma = \lambda \textit{ or} \\ f^s(s1), & \textit{if } \sigma = s1 \\ f^s(s1) \bullet tran_{f^s}(\sigma 1), & \textit{if } \sigma = s1 \bullet \sigma 1 \end{cases}$$

2. Normalization: $ntran_{f^s}(\sigma) = norm(tran_{f^s}(\sigma))$

☺

The next proposition describes the fact that if there is a morphism $f : GG1 \rightarrow GG2$ then there is a corresponding translation of derivation sequences $f^D : SDer(GG1) \rightarrow SDer(GG2)$.

Proposition 4.10 *Let $f : GG1 \rightarrow GG2$ be a graph grammar morphism and f^s be a translation of derivation steps. Then there is a total function $f^S : SDer(GG1) \rightarrow SDer(GG2)$, called a* ***translation of derivation sequences****, such that for all $\sigma 1 \in SDer_{GG1}$ as $f^S(\sigma 1) \cong ntran_{f^s}(\sigma 1)$.* ☺

Proof. The translation of derivation steps f^s exists due to Prop. 4.6. Let $ntran_{f^s}(\sigma 1) = \sigma 1'$. The initial graph of $GG2$ must be isomorphic to the initial graph of the derivation $\sigma 1'$ because f is a graph grammar morphism (and thus there is an iso i_f – see diagram below) and the translation of derivation steps also guarantees this (there is an iso i_I). Thus, we can change the initial graph of σ' and maintain the rest as in σ', obtaining a derivation sequence $\sigma 2$ in $GG2$. The initial graph of $\sigma 2$ is the initial graph of $GG2$ by construction and all rules used in $\sigma 2$ are rules of $GG2$ (assured by the normalization construction). For all subsequent steps i and $i+1$ of $\sigma 2$, we have that $OUT_i = IN_{i+1}$ because $\sigma 1$ is a sequential derivation, and thus fulfills this requirement, the translation preserves this property because $\mathcal{T}_{f_T}$ is a functor and the normalization construction preserve this property due to Prop. 4.8.

$$\begin{tikzcd}
\mathcal{T}_{f_T}(IN_{GG1}) & L_{s1} \arrow[r, "r_{s1}"] \arrow[d, tail, "m_{s1}"] & R_{s1} \arrow[d] \\
IN_{GG2} \arrow[u, tail, "i_f"] \arrow[r, tail, "(i_I)^{-1} \circ i_f"'] & IN_{\sigma 1'} \arrow[r] \arrow[ul, tail, "i_I"'] & OUT_{s1}
\end{tikzcd}$$

√

Theorem 4.11 *Let $f : GG1 \to GG2$ be a graph grammar morphism. Then f is compatible with the sequential semantics of graph grammars: All sequential derivations of $GG1$ can be translated according to a translation of derivation sequences f^S to sequential derivations of $GG2$.* ☺

Proof. According to Prop. 4.10, a morphism between graph grammars induces a corresponding translation of sequential derivations. $\surd$

In fact, graph grammar morphisms are also compatible with the concurrent semantics of graph grammars. Concurrent derivations are obtained from sequential derivations by gluing items (the intermediate graphs). Therefore the translation of sequential derivations could be used to define a corresponding translation of concurrent derivations. We will establish this relationship by using the unfolding of a graph grammar. In Chap. 6, it will be shown that the unfolding of a graph grammar includes all its concurrent derivations and that the unfolding construction induces a functor from the category of graph grammars into the category of occurrence graph grammars (in which all concurrent derivations are included). The application of this functor to a graph grammar morphism gives automatically a translation of concurrent derivations.

4.2 Parallel Composition of Graph Grammars

In this section we will present two kinds of parallel composition of graph grammars: *pure parallel composition* and *cooperative parallel composition.*

The *pure parallel composition* describes a composition of grammars without any interface. The resulting grammar is the disjoint union of the component grammars plus the parallel rules that may be built using one rule of each component grammar. These parallel rules describe explicitly the possibility of rules of both grammars to be applied in parallel. Thus the rules of the composed grammar may express *synchronous* operations between the component grammars (by means of parallel rules) as well as *asynchronous* operations (via the rules that belong to one of the components).

The *cooperative parallel composition* can be used to find a common extension to two different extensions of a grammar. The basic idea is that we have a grammar GG that represents a description of a whole system, called *abstract view*, and this grammar is *specialized* (or refined) in two different ways, giving raise to grammars $GG1$ and $GG2$. A specialization of GG may add new types, rules and have a bigger initial graph than GG. These specialization relationships are described by (special) graph grammar morphisms $s1 : GG1 \to GG$ and $s2 : GG2 \to GG$. Specialization morphisms assure that $GG1$ and $GG2$ are in a sense conservative extensions of GG, i.e., the added parts of $GG1$ and $GG2$ with respect to GG do not change the behaviour of the abstract view. The cooperative parallel composition of $GG1$ and $GG2$ with respect to GG is constructed as a union of these three grammars: the type and initial graph of the composition are union of the corresponding type and initial graphs, and the rules are the rules obtained as the union of corresponding rules in GG, $GG1$ and $GG2$ (amalgamated rules), the rules that are in $GG1$ and $GG2$ and not in GG and the parallel rules obtained from the latter ones. The amalgamated rules put together the different specializations made in $GG1$ and $GG2$ of the same rule of GG. Therefore we say that $GG1$ and $GG2$ *cooperate* to build the description of the whole system (described by the resulting composed graph grammar). The parallel and other rules represent, as in the pure parallel composition, the synchronous and asynchronous compositions respectively. Obviously, the pure parallel composition is a special case of the cooperative parallel composition, namely when the abstract view is empty.

4.2.1 Pure Parallel Composition

In this section the pure parallel composition of graph grammars will be defined. As discussed above, the type and initial graphs of the composition is the disjoint union of the types and initial graphs of the components, and the rules of the composition gives us more than just the disjoint union of the rules of the component, namely all possible synchronized behaviours of rules belonging to the component grammars. These synchronized behaviours are expressed by parallel rules. The basic idea is that the parallel composition of grammars $GG1$ and $GG2$ yields a grammar $GG12$ containing all rules of each component grammar and all parallel rules that can be build using one rule of each component grammar.

For graph grammars such a parallel composition has not been defined yet. Most of the binary composition operators presented until now are based on disjoint union constructions (e.g., [CH95, TS95, Jan96, KK96, PP96]), what implies that only the asynchronous parallelism between the components is expressed by the composed system. In some other formalisms, similar kinds of parallel composition as defined here for graph grammars have already been defined. For transition systems such a parallel composition was defined in [WN94] and for place/transition nets in [MMS96].

Definition 4.12 (Pure Parallel Composition) *The* ***pure parallel composition*** $GG1||GG2$ *of two grammars* $GGi = (Ti, Ii^{Ti}, Ni, ni)$, *for* $i = 1, 2$, *is constructed as follows* $GG1||GG2 = (T, I^T, N, n)$ *where:*

- T *is the coproduct of* $T1$ *and* $T2$ *in* **GraphP**, *i.e., the disjoint union of type graphs (see Def. B.8)*
- I^T *is the coproduct of* $I1^{T1}$ *and* $I2^{T2}$ *in* **TGraphP**, *i.e., the disjoint union of initial graphs (see Def. B.10*
- $N = N1 \uplus N2 \uplus (N1 \times N2)$,*i.e., the product of* $N1$ *and* $N2$ *in* **SetP***(see Def. B.2)*
- $n : N \to Rules(T)$ *is defined for all* $nr \in N$ *as follows*

$$n(nr) = \begin{cases} \mathcal{T}_{(p1_T)^{-1}}(n1(nr)), & \text{if } nr \in N1 \\ \mathcal{T}_{(p2_T)^{-1}}(n2(nr)), & \text{if } nr \in N2 \\ n1(a)||^T n2(b), & \text{if } nr = (a, b) \in N1 \times N2 \end{cases}$$

where $p1_T : T1 \to T12$ *and* $p2_T : T2 \to T12$ *are inclusions induced by the coproduct* T.

☺

Example **4.13 (Pure Parallel Composition – See Def. 4.12)** The parallel composition of the grammars PLV and CLV shown in Figures 2.3 and 2.4 respectively is the grammar $PLV||CLV = (T, I^T, N, n)$ where T is the disjoint union of $PType$ and $CType$, I is the disjoint union of $PIni$ and $CIni$ and the rules all the ones of PLV and CLV plus the rules $Pr1||Cr1$, $Pr1||Cr2$, $Pr1||Cr3$, $Pr2.1||Cr1$, $Pr2.1||Cr2$, $Pr2.1||Cr3$, $Pr2.2||Cr1$, $Pr2.2||Cr2$, $Pr2.2||Cr3$, $Pr3||Cr1$, $Pr3||Cr2$, $Pr3||Cr3$. three of these rules can be seen in Figure 4.2, where the indices C and P indicate to which type graph (Central or Phone) the items belong (the items representing data types shall be mapped correspondingly). Note that in the composed grammar $PLV||CLV$, whenever a parallel rule, say $Pr1||Cr2$, can be applied, so do the component rules $Pr1$ and $Cr2$ because the matches must be completely disjoint (because the component rules are typed over disjoint parts of the composed type graph). Moreover, the results of applying the parallel rule and applying the components in any order will be the same. Obviously, as rules in this case may be applied concurrently, $Pr1$ and $Cr2$ may be applied concurrently (asynchronously). The parallel rule application forces a synchronization between $Pr1$ and $Cr2$, i.e., these two rules must start begin applied and end at the same time. The fact that these rules may be applied asynchronously means that there can be a situation in which we observe that these two rules are being applied, but they don't need necessarily to start and end together. ☺

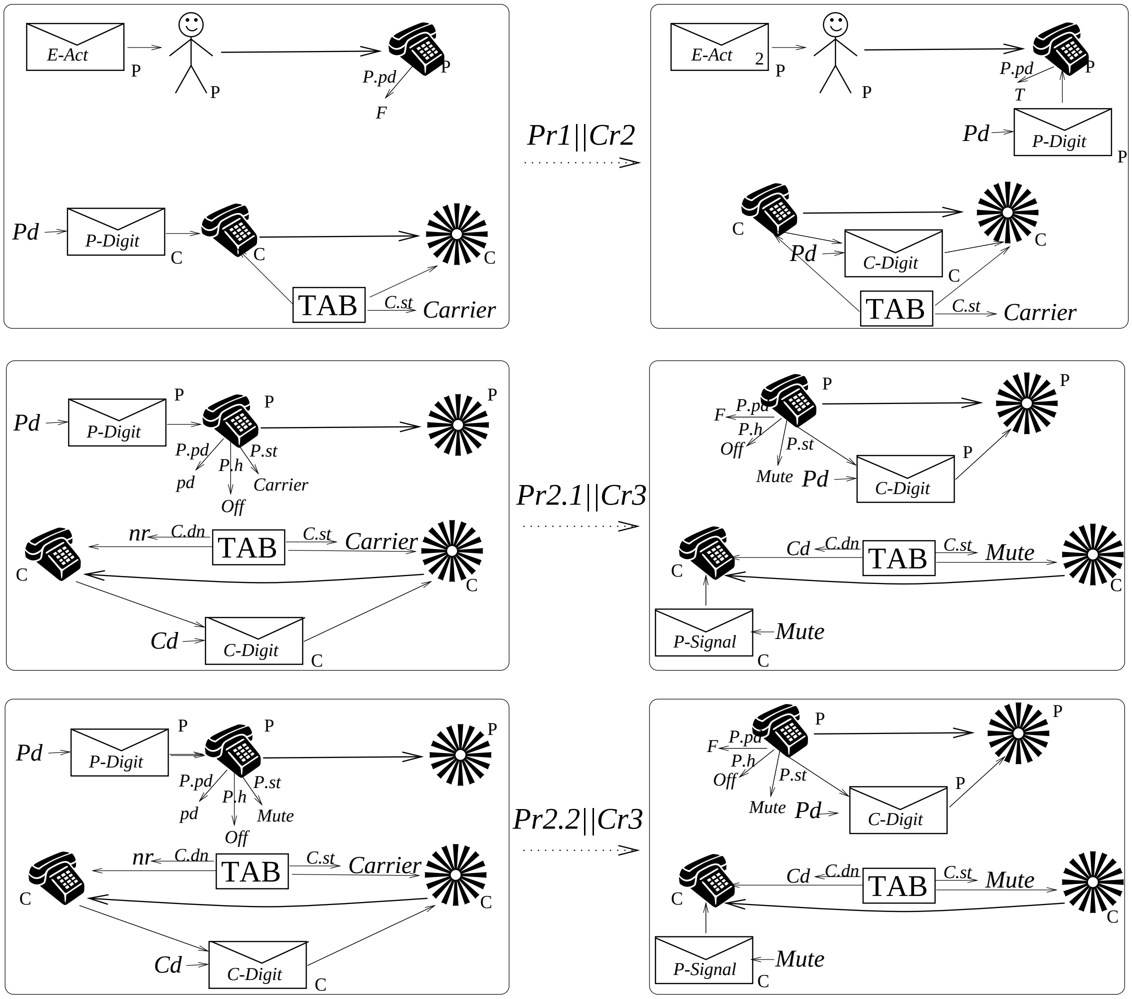


Figure 4.2: Rules belonging to $PVL||CLV$

Proposition 4.14 *The parallel composition of graph grammars is well-defined.* ☺

Proof. We have to show that $GG1||GG2$ is a graph grammar. The type graph of this grammar is T. As coproducts in **TGraphP** are constructed componentwise, the graph I is typed over T. As N is a set by construction, we just have to show that the images of n are rules with type graph T. As $p1_T$ is total and injective it can be inverted, giving raise to $(p1_T)^{-1} : T \to T1$. Moreover, $(p1_T)^{-1}$ is injective and surjective, what implies (by Lemma C.2) that the retype construction $\mathcal{T}_{(p1_T)^{-1}}$: **TGraphP(T1)** $\to$ **TGraphP(T12)** transform each rules r^{T1} into a morphism r'^{T12} where $r \cong r'$, what implies that r'^{T} is a rule and the type graph is T. Therefore, for all $x \in N1$, $n(x) = \mathcal{T}_{(p1_T)^{-1}}(n1(x))$ is a rule with type graph T. Analogously, all images of $n(x)$ for $x \in N2$ are also rules with respect to T. Let $(a, b) \in (N1 \times N2)$. By Prop. 3.22, the parallel rule is a rule having as type the coproduct to the component types. Thus, $n1(a)||^T n2(b) = n12(x)$ is a rule with type T. $\surd$

The following proposition assures that the morphisms of the coproduct of type graphs and of the product of sets of rule names give raise to graph grammar morphisms $p1 : GG \to GG1$ and $p2 : GG \to GG2$. In Theo. 4.16 it will be shown GG is the categorical product of $GG1$ and $GG2$ and that these morphisms are the corresponding projection morphisms.

Proposition 4.15 *Let $GG1$ and $GG2$ be graph grammars, and $GG = GG1||GG2$ be their pure parallel composition. Let $p1_T$ and $p2_T$ be the morphisms of the coproduct of type graphs and $p1_N$ and $p2_N$ the projection morphisms of the product of sets of rule names. Then $p1 = (p1_T^{OP}, p1_N) : GG \to GG1$ and $p2 = (p2_T^{OP}, p2_N) : GG \to GG2$ are graph grammar morphisms.* ☺

Proof. We have to show that the pairs $p1$ and $p2$ satisfy the conditions of graph grammar morphisms (see Def. 4.1). In fact, we will show that in this case, the first condition of subcommutativity can be substituted by commutativity (that is a stronger requirement). We will show this for $p1$, the proof for $p2$ can be done analogously.

1. *Commutativity*:

$$\begin{tikzcd}
N1 \arrow[d, "\overline{n1}"'] \arrow[dr, phantom, "(1)"] & N \arrow[l, two heads, "p1_N"'] \arrow[d, "\overline{n}"] \\
\mathcal{R}(T1) & \mathcal{R}(T) \arrow[l, two heads, "\mathcal{R}_{p1_T}"]
\end{tikzcd}$$

Let $x \in N$ and $n(x) = r^T$. Then we have 3 cases:

(a) $x \in N1$:

$$\begin{aligned}
\mathcal{R}_{p1_T} \circ \overline{n}(x) &= \mathcal{R}_{p1_T}([r^T]) && \text{Def. of } \overline{n1} \text{ and } n(x) = r^T \\
&= [\mathcal{T}_{p1_T}(r^T)] && \text{Def. 3.17} \\
&= [r^{T1}] && \text{Lemma C.2} \\
&= \overline{n1}(x) && \text{Def. of } \overline{n1} \\
&= \overline{n1} \circ p1_N(x) && p1_N(x) = x \text{ if } x \in N1
\end{aligned}$$

(b) $x \in N2$:

$$\begin{aligned}
\mathcal{R}_{p1_T} \circ \overline{n}(x) &= \mathcal{R}_{p1_T}([\mathcal{T}_{p2_T}(n2(x))]) && \text{Def. 4.12} \\
&= \texttt{undef} && \text{Def. 3.17 and } rng(p1_T) \cap rng(t^L) \cap rng(t^R) = \emptyset
\end{aligned}$$

(c) $x = (a, b) \in (N1 \times N2)$:

$$\begin{aligned}
\mathcal{R}_{p1_T} \circ \overline{n}(x) &= \mathcal{R}_{p1_T}([n1(a) \|^T n2(b)]) && \text{Def. 4.12}\\
&= [\mathcal{T}_{p1_T}([n1(a) \|^T n2(b)])] && \text{Def. 3.17}\\
&= [n1(a)] && \text{Prop. 3.28 and Def. 3.20}\\
&= \overline{n1}(a) && \text{Def. of } \overline{n1}\\
&= \overline{n1} \circ p1_N(x) && p1_N(x) = a \text{ if } x = (a, b) \in N1 \times N2
\end{aligned}$$

Therefore (1) commutes.

2. *Isomorphism of initial graphs:* By Lemma C.3 (2) is a pullback. As $p1_T$ is total and (2) is a pullback, by Def. 3.8, $\mathcal{T}_{p1_T}(I^T) \cong I1^{T1}$.

$$\begin{tikzcd}
I1 \arrow[r, hook, "p1_T^{\bullet}"] \arrow[d, "t^{I1}"'] \arrow[dr, phantom, "(2)"] & I \arrow[d, "t^I"] \\
T1 \arrow[r, hook, "p1_T"'] & T
\end{tikzcd}$$

√

The next theorem shows that the construction of the parallel composition $GG1 \| GG2$ given above is the categorical product in the category of graph grammars **GG**. This fact implies that the pure parallel composition describes the maximal degree of parallelism that the two component grammars may have when composed. This is assured by the universal property of products, that says that whenever there is another grammar that includes $GG1$ and $GG2$, this grammar is also included in the composition $GG1 \| GG2$ (this assertion holds not only for inclusions). Moreover, the fact that the pure parallel composition is a product will be very useful to prove that the unfolding semantics of a graph grammar that will be defined in Chap. 6 is compatible with this kind of composition.

Theorem 4.16 *The parallel composition of graph grammars is the categorical product (object) in the category of graph grammars.* ☺

Proof. Let the product morphisms $p1 = (p1_T^{OP}, p1_N) : GG1 \| GG2 \to GG1$, $p2 = (p2_T^{OP}, p2_N) GG1 \| GG2 \to GG2$ be defined componentwise in the categories $\mathbf{GraphP^{OP}}$ and **SetP**, where the components are the product morphisms induced by the products of type graphs and sets of rule names. Let X be a graph grammar and $x1 = (x1_T^{OP}, x1_N) : X \to GG1$ and $x2 = (x2_T^{OP}, x2_N) : X \to GG2$ be graph grammar morphisms. Then $u = (u_T^{OP}, u_N) : X \to GG1 \| GG2$ is defined componentwise where the components are the corresponding universal morphisms induced by the products of types and rule names. Due to Prop. 4.15, $p1$ and $p2$ are graph grammar morphism. Thus it remains to show that u is a graph grammar morphism and that the universal property is satisfied.

1. *u is a graph grammar morphism:*

(a) Commutativity: By Prop. 4.15 we have that squares (1) and (2) below commute. As $x1$ and $x2$ are graph grammar morphisms, squares (3) and (4) below subcommute. As u_N is the universal morphism induced by the coproduct of sets of rule names, we have

(5) $p1_N \circ u_N = x1_N$
(6) $p2_N \circ u_N = x2_N$
As u_T is the universal morphism induced by the coproduct of type graphs and $\mathcal{R}$ is a functor we have
(7) $\mathcal{R}_{p1_t} \circ \mathcal{R}_{u_T} = \mathcal{R}_{x1_T}$
(8) $\mathcal{R}_{p2_t} \circ \mathcal{R}_{u_T} = \mathcal{R}_{x2_T}$
We have to show that (9) sub-commutes, i.e., $\overline{n} \circ u_N \subseteq^R \mathcal{R}_{u_T} \circ \overline{nx}$.

$$\begin{tikzcd}
N1 \arrow[r, mapsto, "\overline{n1}"] \arrow[drr, phantom, "(3)"] & \mathcal{R}(T1) \arrow[dr, phantom, "(1)"] & & & N2 \arrow[r, mapsto, "\overline{n2}"] \arrow[dl, phantom, "(2)"] & \mathcal{R}(T2) \arrow[dll, phantom, "(4)"] \\
& & N \arrow[r, mapsto, "\overline{n}"] \arrow[ull, "p1_N"'] \arrow[urr, "p2_N"] \arrow[dr, phantom, "(9)"] & \mathcal{R}(T) \arrow[ull, "\mathcal{R}_{p1_T}"'] \arrow[urr, "\mathcal{R}_{p2_T}"'] & & \\
& & NX \arrow[r, mapsto, "\overline{nx}"'] \arrow[u, "u_N"] \arrow[uull, bend left, "x1_N"] \arrow[uurr, "x2_N"'] & \mathcal{R}(TX) \arrow[u, "\mathcal{R}_{u_T}"'] \arrow[uull, "\mathcal{R}_{x1_T}"] \arrow[uurr, bend right, "\mathcal{R}_{x2_T}"'] & &
\end{tikzcd}$$

Assume that this does not hold. Let $ex \in NX$. As $\overline{n} \circ u_N \not\subseteq^R \mathcal{R}_{u_T} \circ \overline{nx}$ by assumption, one of the following cases must be true:

i. $\overline{n} \circ u_N(ex) = \texttt{undef}$ and $\mathcal{R}_{u_T} \circ \overline{nx}(ex) = e$:
As $\overline{n}$ is total we have that $u_N(ex) = \texttt{undef}$. Then (5) implies that $x1_N(ex) = \texttt{undef}$, and thus $\mathcal{R}_{x1_T} \circ \overline{nx}(ex) = \texttt{undef}$ because of (3). Analogously we conclude that $\mathcal{R}_{x2_T} \circ \overline{nx}(ex) = \texttt{undef}$. By Prop. 3.28 and Def. 3.17 $e \in dom(\mathcal{R}_{p1_T})$ or $e \in dom(\mathcal{R}_{p2_T})$ or both. Assume $e \in dom(\mathcal{R}_{p1_T})$. Then by (7) we conclude that $\mathcal{R}_{x1_T} \circ \overline{nx}(ex)$ must be defined, and this contradicts $\mathcal{R}_{x1_T} \circ \overline{nx}(ex) = \texttt{undef}$. If we assume that $e \in dom(\mathcal{R}_{p2_T})$ we get analogously a contradiction. Therefore this case (i.) can never happen.

ii. $\overline{n} \circ u_N(ex) = e'$ and $\mathcal{R}_{u_T} \circ \overline{nx}(ex) = \texttt{undef}$:
Analogous to the first case.

iii. $\overline{n} \circ u_N(ex) = e'$, $\mathcal{R}_{u_T} \circ \overline{nx}(ex) = e$ and $e' \not\subseteq^R e$:
By Prop. 3.28 the rules of $\mathcal{R}(T)$ can be decomposed, yielding a rule in $\mathcal{R}(T1)$ or in $\mathcal{R}(T2)$ or both. Let $e = [r]$, $e' = [r']$, $decomp(r) = (r1, r2)$ and $decomp(r') = (r1', r2')$. Then we have the following cases:

- $r1$, $r2$, $r1'$ and $r2'$ are not isomorphisms: By Def. 3.17 and 3.26 we have that $\mathcal{R}_{p1_T}(e) = [r1^{T1}]$ and $\mathcal{R}_{p1_T}(e') = [r1'^{T1}]$. As (7) commutes we have that $\mathcal{R}_{x1_T} \circ \overline{nx}(ex) = \mathcal{R}_{p1_T} \circ \mathcal{R}_{u_T} \circ \overline{nx}(ex) = \mathcal{R}_{p1_T}(e) = [r1^{T1}]$. Then (3) yields that $\overline{n1} \circ x1_N(ex) \subseteq^R [r1^{T1}]$. By definition of e' we have that $\mathcal{R}_{p1_T}(e') = \mathcal{R}_{p1_T} \circ \overline{n} \circ u_N(ex)$. Then (1) yields that $\mathcal{R}_{p1_T}(e') = \overline{n1} \circ p1_N \circ u_N(ex)$. This yields that $\mathcal{R}_{p1_T}(e') = \overline{n1} \circ x1_N(ex)$ by (5). Then by (3) we conclude that $\mathcal{R}_{p1_T}(e') = [r1'^{T1}] \subseteq^R [r1^{T1}] = \mathcal{R}_{p1_T}(e)$. Analogously, we obtain that $\mathcal{R}_{p2_T}(e') = [r2'^{T2}] \subseteq^R [r2^{T2}] = \mathcal{R}_{p2_T}(e)$. Thus by Prop. 3.29 we conclude that $e' \subseteq e$, what contradicts the hypothesis that $e \not\subseteq^R e'$.
- $r1, r1'$ are isomorphisms, $r2, r2'$ are not isomorphisms: By Def. 3.17 and 3.26 we have that $\mathcal{R}_{p1_T}(e) = \texttt{undef}$, $\mathcal{R}_{p1_T}(e') = \texttt{undef}$, $\mathcal{R}_{p2_T}(e) = [r2^{T2}]$ and $\mathcal{R}_{p2_T}(e') = [r2'^{T2}]$. Analogously to the previous case we obtain that $\mathcal{R}_{p2_T}(e') = [r2'^{T2}] \subseteq^R [r2^{T2}] = \mathcal{R}_{p2_T}(e)$. As $e' \in rng(\overline{n})$ and

$e' \notin dom(\mathcal{R}_{p1_T})$ we have that $r1'^{T1} = \emptyset^{T1}$ (by definition of pure parallel composition) and this is trivially a subrule of $r1^{T1}$. Therefore, by Prop. 3.29 we conclude that $e' \subseteq^R e$, what contradicts the hypothesis.

- $r2, r2'$ are isomorphisms, $r1, r1'$ are not isomorphisms: Analogous to the previous case.
- Other cases: Let $r1^{T1}$ be an isomorphism. Then $\mathcal{R}_{p1_T}(e) = \texttt{undef}$. As (7) commutes we have that $\mathcal{R}_{x1_T} \circ \overline{nx}(ex) = \texttt{undef}$, and this implies (by (3)) that $\overline{n1} \circ x1_N(ex) = \texttt{undef}$. As (5) commutes, $\overline{n1} \circ p1_N \circ u_N(ex) = \texttt{undef}$. Thus (1) yields that $\mathcal{R}_{p1_T} \circ \overline{n} \circ u_N(ex) = \texttt{undef}$, i.e., $\mathcal{R}_{p1_T}(e') = \texttt{undef}$. By Def. 3.17 this implies that $r1'^{T1}$ is an isomorphism. Thus we have again case ii. If we start with $r1'^{T1}$ we obtain that $r1^{T1}$ is also an isomorphism. If $r2^{T2}$ is an isomorphism we obtain analogously that $r2'^{T2}$ is also an isomorphism and vice versa. By Prop. 3.28, there can not be the case in which $r1$ and $r2$ are isomorphisms.

(b) Isomorphism of initial graphs: Here we have to prove that (1) is a pullback and (2) commutes.

$$\begin{tikzcd}
 & I \arrow[r, mapsto, "u_I"] \arrow[dl, "t^I"'] \arrow[d, "a"] \arrow[dl, phantom, "(2)" near end] \arrow[dr, phantom, "(1)"] & IX \arrow[d, "t^{IX}"] \\
T & dom(u_T) \arrow[l, hook', "u_T^\blacktriangledown"] \arrow[r, mapsto, "u_T!"'] & TX
\end{tikzcd}$$

As $x1$ is a morphism, there is a total morphism $a1 : I1 \to dom(x1_T)$ such that $t^{I1} = a1 \circ x1_T^\blacktriangledown$ (the pullback morphism from the retyping construction). The same holds for $x2$, yielding a morphism $a2 : I2 \to dom(x2_T)$. $u_T : T \to TX$ is the universal morphism induced by the coproduct T using comparison morphisms $x1_T$ and $x2_T$. Therefore $dom(u_T)$ consists of all elements of T that are mapped via one of these comparison morphisms, and thus $dom(u_T)$ is the disjoint union (coproduct) of $dom(x1_T)$ and $dom(x2_T)$. As I is a coproduct, there is a (total) morphism $a : I \to dom(u)$ such that (5) and (6) commute. Thus, by Lemma C.3, (5) and (6) are pullbacks. Squares (7) and (8) are pullbacks because $x1$ and $x2$ are graph grammar morphisms. Now we can apply Lemma C.5 having as hypothesis the facts that I and $dom(u_T)$ are coproducts and squares (5)–(8) are pullbacks in **Graph**, obtaining that (1) in also a pullback in **Graph**. As T is a coproduct, $u^\blacktriangledown$ as well as t^I are coproduct morphism, and by uniqueness of coproduct morphisms, we conclude that $t^I = u_T^\blacktriangledown \circ a$, i.e., diagram (2) commutes.

$$\begin{tikzcd}
I1 \arrow[drr, hook] \arrow[d, mapsto, "a1"] \arrow[dd, mapsto, bend right=60, "t^{I1}"'] & & & & I2 \arrow[dll, hook'] \arrow[d, mapsto, "a2"'] \arrow[dd, mapsto, bend left=60, "t^{I2}"] \\
dom(x1_T) \arrow[d, hook, "x1_T^\blacktriangledown"] \arrow[drr, hook] & (5) & I \arrow[d, mapsto, "a"] & (6) & dom(x2_T) \arrow[d, hook, "x2_T^\blacktriangledown"'] \arrow[dll, hook'] \\
T1 \arrow[drr, hook] & = & dom(u_T) \arrow[d, hook, "u_T^\blacktriangledown"] & = & T2 \arrow[dll, hook'] \\
 & & T12 & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & TX & & \\
dom(x1_T) \arrow[urr, mapsto, "x1_T!"] & (7) & IX \arrow[u, mapsto, "t^{IX}"] & (8) & dom(x2_T) \arrow[ull, mapsto, "x2_T!"'] \\
I1 \arrow[u, mapsto, "a1"] \arrow[urr, mapsto] & & & & I2 \arrow[u, mapsto, "a2"'] \arrow[ull, mapsto]
\end{tikzcd}$$

2. *Universal property:* Follows from the componentwise construction of u and from the corresponding properties of the components.

$\surd$

We will now introduce a new way of composing grammars using an interface grammar. As already discussed in the example (Chap. 2), it is very useful for practical applications to allow the splitting of a specification into smaller parts, that can than be worked out separately. Obviously, one wants to have the property that, when the pieces are put together, the behaviour of the whole can be totally inferred from the behaviours of the parts. One of the biggest problems in concurrent and distributed system is to assure this property, because in many cases we observe that putting components together in a naive way leads to behaviours that can not be observed in any of the components. Thus, the question that arises is: *Which kinds of restrictions have to be made to define a composition in which this property can be assured?* In the next section, we give one possible answer for this question.

4.2.2 Cooperative Parallel Composition

Very often in the literature, composition of parts of a system with respect to some interface are defined via colimit constructions (gluing all components together on the common interface). This is the case e.g., in algebraic specification [EM90], Petri nets [Jen92, PER95], and graph transformation systems [Jan93, CH95, TS95, PP96]. For the case of graph grammars, i.e., transformation systems with initial graph, although this kind of gluing seems to be a natural way of putting parts together, it is well known that it is not compatible with the most basic semantics of grammars, namely sequential derivations. This is also the case if we think of Petri nets with initial markings or even transition systems with initial state. The reason for this is that the semantics of a grammar usually only consider applications of rules over graphs that can be generated by the grammar itself. To illustrate this fact, we will give an example.

Example **4.17** Consider the graph grammars shown in Figure 4.3, where the type graphs of all of them is the graph T. Grammar $GG3$ is the union (gluing)[2] of $GG1$ and $GG2$ over $GG0$. As grammar $GG0$ has no rules, its sequential semantics $SDer_{GG0}$ is the set consisting only of the empty derivation sequence. The rule $r1$ of grammar $GG1$ can not be applied to graph $IN1$, and therefore $SDer_{GG1} = \{\lambda\}$. Analogously, $SDer_{GG2} = \{\lambda\}$. In grammar $GG3$, both rules can be applied, and thus its sequential semantics $SDer_{GG3}$ consists of the empty derivation sequence, and derivation sequences corresponding to all applications of rules $r1$ and $r2$ (due to the fact that isomorphic copies are represented, these are infinitely many). Thus we can notice that, although $GG3$ can be obtained by gluing $GG1$ and $GG2$ along $GG0$, its semantics can not be obtained as a suitable composition of the semantics of the components. ☺

In [CH95] a composition operation similar to the one described in the example above was presented, just that the grammars do not have initial graphs, and as thus called *graph transformation systems*. In this case, the sequential semantics consists of all possible sequences of applications of rules starting at all possible graphs [CEL$^+$96a]. If we take our example

[2] The kind of union of grammars used in this example is analogous to the ones defined for graph transformation systems, for example in [CH95], but gluing also the initial graphs. This example is used just to illustrate the effect of such a composition on graph grammars with initial graphs.

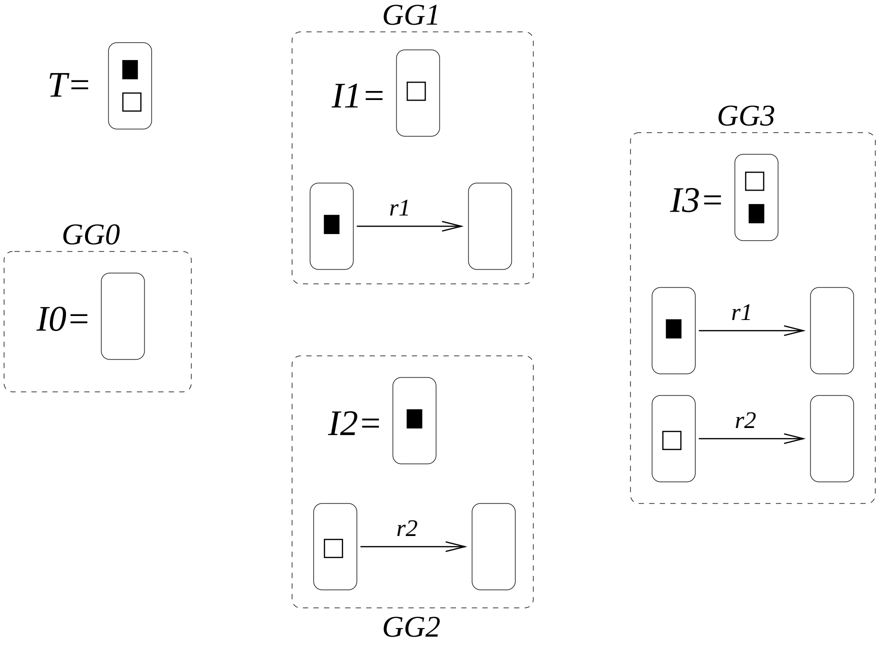


Figure 4.3: Rules belonging to $PVL||CLV$

forgetting the initial graphs, we will observe that the semantics of the transformation system described by the rules of $GG3$ can be obtained as a suitable composition of the semantics of the corresponding components.

One of the reasons why *initial graphs* are used for specification is to restrict the derivations to the "wanted" ones, in other words, to *specify which derivations* describe the desired behaviour of a system. As graph grammars specify a system in terms of changes of state, from a specification point of view, the description of the initial state of the system is an interesting feature. Moreover, different initial states usually lead to different behaviours of systems, what implies that if we are interested in some kinds of correctness properties of systems, like, e.g., that some error states are never reached, the specification of the initial state in mandatory.

As the initial states complement a set of rules for the specification of a system and specification of a whole system based on components is a desired property of a specification formalism, suitable means of composing grammars (including initial state) with respect to an interface shall be developed. The basic idea for the composition developed here is that the components shall not add rules that would modify the behaviour of the interface, because this could lead to problems in other components. The interface, called *abstract view*, shall be seen as a global, abstract specification of the whole system. Each component is a specialization of the interface. This specialization may be done in two ways: i) by adding new items to rules/types/initial graph to the ones in the abstract view, ii) by adding new rules. The condition for the first way of specialization is that the new added items belong only to types that are not in the abstract view. Using this kind of specialization, the same rule may be specialized in many different ways. The condition for the second kind of specialization is that the items that are created/deleted by the newly added rules are local, in the sense that they only use local types.This idea of is captured by the notion of specialization morphisms.

The next definition (*specialization* morphisms) is (with the appropriate changes of approach) a specialization of the definition of morphisms called *refinements* of DPO-graph transformation systems presented in [HCEL96]. With respect to these refinement morphisms, the main restrictions made here are that specialization morphisms are required to be injective on types and that the translation of rules induced by type morphism must be isomorphic to the corresponding rule of the target grammar.

Definition 4.18 (Specialization Morphism) *Let $f = (f_T^{OP}, f_N) : GG1 \to GG2$ be graph grammar morphism. Then f is called a* ***specialization morphism****, and $GG1$ and $GG2$ are called* ***specialized grammar*** *and* ***abstract grammar****, respectively, iff*

1. *f_T is total and injective,*
2. *f_N is surjective,*
3. *$\forall nr \in dom(f_N) : n2 \circ f_N(nr) \cong \mathcal{T}_{f_T} \circ n1(nr)$, and*
4. *$\forall nr \notin dom(f_N) : \mathcal{T}_{f_T} \circ n1(nr)$ is the empty typed graph morphism, i.e., $\emptyset^{id_{T2}} : \emptyset^{T2} \to \emptyset^{T2}$.*

☺

Remarks. *The intuitive meaning of the conditions is:*

1. *The type graph of the specialized grammar includes the type of the abstract grammar. This is reasonable because this means that there are no elements of the abstract grammar that are "forgotten" in the specialization.*
2. *All rules of the abstract grammar are in the specialized grammar. Again, this means that the specialized grammar does not forget rules. But note that the same rule of the abstract grammar may specialized by many rules of the specialized grammar. This allows us to model situations in which actions that seems to be the same at a more abstract level are distinguishable if we look at a more concrete level.*
3. *This requirement assures that the rules of the components only add elements belonging to local types to the rules of the abstract grammar.*
4. *This requirement assures that rules that are local to a component do not use elements of types belonging to the abstract grammar, that is, they may only use local types.*

☺

Example **4.19 (Specialization Morphisms)** The grammars PLV and CLV (Figures 2.3 and 2.4) are two different specializations of the grammar AGV (Figure 2.2). In grammar PLV, no new rule is added with respect to rules that were in the interface, but the same rule of the interface $r2$ in specialized in two different ways ($Pr2.1$ and $Pr2.2$). One can check whether this is a specialization or not by forgetting from the rules (type, initial graph) of PLV items that have types that are not in the type of AGV (Figure 2.1). In this example, we have that this forgetting from PLV yields the grammar AGV (notice that $r2$ is yielded by forgetting corresponding items from $Pr2.1$ as well as from $Pr2.2$). ☺

As discussed in Chap. 2, the idea of composition introduced here is based on a top-down development of the system: first an abstract description of the components and their interconnections is fixed, then each component is specialized separately and at the end they are put together. This composition is called *cooperative parallel composition* because the the components cooperate to do the specification of the whole system. This cooperation can be reflected in the fact that both specializations done in the components do not change the behaviour of the abstract view, and that in the resulting system, specializations made by different components for the same rule of the abstract view are glued together (via an amalgamated rule). One could ask whether this kind of concept for specification of systems is adequate for practical applications. Many discussions with the company Nutec have revealed that such a concept can be quite useful for the development of concurrent/reactive systems (that are the ones they use to develop). The main aspect is that such a global, abstract specification of the system as we used here as interface is anyway necessary for the development of a product. It serves as a basis for the communication between members of different teams. Moreover, it makes explicit which changes may affect other components. For example, if a developer wants to specialize the interface in a way in which the interface would get a new rule, the formalism makes explicit that this is a "critical change" because the behaviour of other parts may change. "Safe" changes or specializations are given by graph grammar specialization morphisms. This means that whenever there is a specialization morphism between a component and the abstract view, this component is a safe extension of the abstract view, what implies that the cooperative parallel composition of this component with other ones with respect to this abstract view will not show any unexpected (unspecified) behaviour.

$$\begin{tikzcd}
& \text{Abstract View} = GG0 & \\
\text{Component } 1 = GG1 \arrow[ur, "\text{Specialization } s1"] & (PB) & \text{Component } 2 = GG2 \arrow[ul, "\text{Specialization } s2"'] \\
& System = GG3 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Formally, this idea of composition gave raise to a definition based on pullbacks: if we have an interface $GG0$ and two specialization morphisms $s1 : GG1 \rightarrow GG0$ and $s2 : GG2 \rightarrow GG0$, we can build the cooperative parallel composition of $GG1$ and $GG2$ with respect to $GG0$ as a pullback in the category of graph grammars. This composition consists of the gluings of the types and initial graphs of $GG1$ and $GG2$ with respect to $GG0$ and the following set of rules: rules that are in the components and not in the abstract view and rules that are in the abstract view specialized according to the specializations done in the components. For each of the rules r that is in the abstract view, there is a rule r' (or more, depending on the morphisms $s1$ and $s2$) in the composition that corresponds to it. This rule r' contains the specializations done in $GG1$ and $GG2$ to the rule r, that is, r' is constructed as the amalgamated rule of the rules in $GG1$ and $GG2$ that correspond to r with respect to r.

Definition 4.20 (Cooperative Parallel Composition) *Let $GGi = (Ti, Ii^{Ti}, Ni, ni)$, for $i = 0, 1, 2$ be graph grammars, $GG0$ be safe, and $s1 = (s1_T^{OP}, s1_N) : GG1 \rightarrow GG0$, $s2 = (s2_T^{OP}, s2_N) : GG2 \rightarrow GG0$ be specialization morphisms. Then the* ***cooperative parallel composition*** *$GG1||_{GG0}GG2$ of $GG1$ and $GG2$ with respect to $GG0$ using $s1$ and $s2$ is constructed as follows $GG1||_{GG0}GG2 = (T3, I3^{T3}, N3, n3)$ where:*

- *$T3$ is the pushout (object) of $s1_T$ and $s2_T$ in* **GraphP**, *i.e., the union with respect to $T0$ of the types $T1$ and $T2$ (see Def. B.8)*

- *$I3^{T3}$ is the pushout of $i1^{s1_T}$ and $i2^{s2_T}$ in* **TGraphP**, *i.e., is the union with respect to $I0^{T0}$ of $I1^{T1}$ and $I2^{t2}$ (see Def. B.10), where $i1 : I0 \to I1$ and $i2 : I0 \to I2$ are the pullback morphisms of pullbacks (1) and (2) below.*

$$\begin{tikzcd}
 & T0 \arrow[dl, hook', "s1_T"'] \arrow[dr, hook, "s2_T"] & \\
T1 \arrow[dr, hook', "s2^\bullet_T"'] & (PO) & T2 \arrow[dl, hook, "s1^\bullet_T"] \\
 & T3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & I0 \arrow[dl, hook', "i1"'] \arrow[dr, hook, "i2"] \arrow[ddd, bend left, "t^{I0}"] & \\
I1 \arrow[dr, hook', "i2^\bullet"] \arrow[dddd, "t^{I1}"'] & & I2 \arrow[dl, hook, "i1^\bullet"'] \arrow[dddd, "t^{I2}"] \\
 & I3 \arrow[ddd, bend right, "t^{I3}"'] \arrow[l, phantom, "(1)"] \arrow[r, phantom, "(2)"] & \\
 & T0 \arrow[dl, hook', "s1_T"'] \arrow[dr, hook, "s2_T"] \arrow[l, phantom, "(3)"] \arrow[r, phantom, "(4)"] & \\
T1 \arrow[dr, hook', "s2^\bullet_T"'] & & T2 \arrow[dl, hook, "s1^\bullet_T"] \\
 & T3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & I0 \arrow[dl, hook', "i1"'] \arrow[dr, hook, "i2"] & \\
I1 \arrow[dr, hook', "i2^\bullet"'] & (PO) & I2 \arrow[dl, hook, "i1^\bullet"] \\
 & I3 &
\end{tikzcd}$$

- *$N3 = PB^{\mathbf{GraphP}}(N1 \overset{s1_N}{\to} N0 \overset{s2_N}{\leftarrow} N2)$ is the pullback of $N1$ and $N2$ with respect to $N0$ in* **SetP***(see Def. B.2)*

$$\begin{tikzcd}
 & N0 & \\
N1 \arrow[ur, "s1_N"] & (PB) & N2 \arrow[ul, "s2_N"'] \\
 & N3 \arrow[ul, "s2^\bullet_N"] \arrow[ur, "s1^\bullet_N"'] &
\end{tikzcd}$$

- *$n3 : N3 \to Rules(T3)$ is defined for all $nr \in N3$ as follows*

$$n3(nr) = \begin{cases} \mathcal{T}_{(s2^\bullet_T)^{-1}}(n1(nr1)), & \text{if } s2^\bullet_N(nr) = nr1 \text{ and } s1^\bullet_N(nr) = \texttt{undef} \\ \mathcal{T}_{(s1^\bullet_T)^{-1}}(n2(nr2)), & \text{if } s1^\bullet_N(nr) = nr2 \text{ and } s2^\bullet_N(x) = \texttt{undef} \\ n1(nr1)||^{T3}_{e^{TO}} n2(nr2), & \text{if } s2^\bullet_N(nr) = nr1, s1^\bullet_N(nr) = nr2, \\ & nr1 \notin dom(s1_N) \text{ and } nr2 \notin dom(s2_N) \\ n1(s2^\bullet_N(nr))||^{T3}_{n0(nr0)} n2(s1^\bullet_N(nr)), & \text{if } s1_N \circ s2^\bullet_N(nr) = s2_N \circ s1^\bullet_N(nr) = nr0 \end{cases}$$

where $(s1^\bullet_T)^{-1}$ and $(s2^\bullet_T)^{-1}$ are the inverses of the pushout morphisms $s1^\bullet_T$ and $s2^\bullet_T$, $n1(nr1)||^{T3}_{e^{TO}} n2(nr2)$ is the amalgamated rule using morphisms $(\emptyset^{si_T}, \emptyset^{si_T}) : e^{T0} \to ni(nri)$, for $i = 1, 2$, with e being the empty graph morphism, and $n1(s2^\bullet_N(x))||_{n0(x0)} n2(s1^\bullet_N(x))$ is the amalgamated rule using morphisms $(ki_L^{si_T}, ki_R^{si_T}) : r0^{T0} \to ri^{Ti}$, for $i = 1, 2$, with $(ki_L^{si_T}, ki_R^{si_T})$ being the composition of the retyping morphisms of ri with the unique subrule relation from $r0$ to $\mathcal{T}_{(si_T)^{-1}}(ri)$.

$$\begin{tikzcd}
T0 \arrow[d, hook, "si_T"'] \arrow[rd, phantom, "(5)"] & L0 \arrow[l, mapsto, "t^{L0}"'] \arrow[r, "r0"] \arrow[d, hook, "ki_L"] & R0 \arrow[r, mapsto, "t^{R0}"] \arrow[d, hook, "ki_R"'] \arrow[rd, phantom, "(6)"] & T0 \arrow[d, mapsto, "si_T"] \\
Ti & Li \arrow[l, hook, "t^{Li}"] \arrow[r, "r1"'] & Ri \arrow[r, mapsto, "t^{ri}"'] & Ti
\end{tikzcd}$$

☺

Remarks.

1. *Diagrams (1) and (2) are pullbacks because $s1$ and $s2$ are graph grammar morphisms in which the type components $s1_T$ and $s2_T$ are total and injective. This implies that morphisms $i1$ and $i2$ are also total and injective.*

2. *Analogously, diagrams (5) and (6) are also pullbacks.*

3. *Due to the facts that $GG0$ is safe and to condition 3. of specialization morphisms, there is a unique subrule relation from $r0$ to $\mathcal{T}_{(si_T)^{-1}}(ri)$.*

☺

Example **4.21 (Cooperative Parallel Composition)** The grammar CGV shown in Figure 2.6 is thecooperative parallel composition of grammars PLV and CLV with respect to AGV using the corresponding specialization morphisms (as described in the previous example). ☺

Proposition 4.22 *The cooperative parallel composition is well defined.* ☺

Proof. Here we have to show that $GG3$ is a well-defined graph grammar (Def. 3.33). This can be done completely analogously to the proof of Prop. 4.14, where we proved that the pure parallel composition of grammars is well-defined. Here the proof is based on the facts that pushouts in **TGraph** can be constructed componentwise (see Def. B.4), retyping preserves rules (Lemma C.1) and amalgamated rules are rules having as type the pushout of the types of the components (Prop. 3.25). The uniqueness (up to isomorphism) of this result is due to the safeness of $GG0$ and condition 3. of specialization morphisms. √

Proposition 4.23 *Let $GG0$, $GG1$ and $GG2$ be graph grammars, $s1 : GG1 \to GG0$ and $s2 : GG2 \to GG0$ be specialization morphisms and $GG1||_{GG0}GG2$ be their cooperative parallel composition. Let $s1^\bullet_T$ and $s2^\bullet_T$ be the pullback morphisms type graphs and $s1^\bullet_N$ and $s2^\bullet_N$ the pullback morphisms of sets of rule names. Then $s1^\bullet = (s1^{\bullet OP}_T, s1^\bullet_N) : GG \to GG1$ and $s2^\bullet = (s2^{\bullet OP}_T, s2^\bullet_N) : GG \to GG2$ are graph grammar morphisms.* ☺

Proof. We have to show that the pairs $s1^\bullet$ and $s2^\bullet$ satisfy the conditions of graph grammar morphisms (see Def. 4.1). We will show this for $s1^\bullet$, the proof for $s2^\bullet$ can be done analogously.

1. *Sub-commutativity*:

$$\begin{tikzcd}
N1 \arrow[d, "\overline{n1}"'] \arrow[dr, phantom, "(1)"] & N3 \arrow[l, two heads, "s2^{\bullet}_{N}"'] \arrow[r, two heads, "s1^{\bullet}_{N}"] \arrow[d, "\overline{n3}"] \arrow[dr, phantom, "(2)"] & N2 \arrow[d, "\overline{n2}"] \\
\mathcal{R}(T1) \arrow[dr, "\mathcal{R}(s1_T)"'] & \mathcal{R}(T3) \arrow[l, two heads] \arrow[r, two heads] & \mathcal{R}(T2) \arrow[dl, "\mathcal{R}(s2_T)"] \\
& \mathcal{R}(T0) &
\end{tikzcd}$$

Let $x \in N3$ and $\overline{n3}(x) = [r]$. As $s1_N$ and $s2_N$ are surjective, we have 4 cases (see Def. B.2 for pullback construction in **SetP**) – in the cases below $a, b, c \neq \perp$:

(a) $x = (a, \perp, \perp)$: Analogous to item 1. of proof of Lemma 4.15.

(b) $x = (\perp, \perp, c)$: Analogous to item 2. of proof of Lemma 4.15.

(c) $x = (a, \perp, c)$: Analogous to item 3. of proof of Lemma 4.15.

(d) $x = (a, b, c)$:

$$\begin{aligned}
\mathcal{R}_{s2^{\bullet}_{T}} \circ \overline{n3}(x) &= \mathcal{R}_{s2^{\bullet}_{T}}([n1(a)||_{n0(b)}n2(c)]) && \text{Def. 4.20} \\
&= [\mathcal{T}_{s2^{\bullet}_{T}}([n1(a)||_{n0(b)}n2(c)])] && \text{Def. 3.17} \\
&= [n1(a)] && \text{Prop. 3.32} \\
&= \overline{n1}(a) && \text{Def. of } \overline{n1} \\
&= \overline{n1} \circ s2^{\bullet}_{N}(x) && s2^{\bullet}_{N}(x) = a \ (N3 \text{ is a pullback})
\end{aligned}$$

2. *Isomorphism of initial graphs*: Analogous to the proof of Prop. 4.15, using Lemma C.4 and the facts that $s1$ and $s2$ are specialization morphisms.

√

Theorem 4.24 *Let $GGi = (Ti, Ii, Ni, ni)$, for $i = 0, 1, 2$ be graph grammars, and $s1 = (s1^{OP}_{T}, s1_N) : GG1 \to GG0$, $s2 = (s2^{OP}_{T}, s2_N) : GG2 \to GG0$ be specialization morphisms. Then the cooperative parallel composition $GG3 = GG1||_{GG0}GG2$ using $s1$ and $s2$ is the pullback (object) of $(GG1 \xrightarrow{i1} GG0 \xleftarrow{i2} GG2)$ in the category of graph grammars* **GG**.

$$\begin{tikzcd}
& GG0 & \\
GG1 \arrow[ur, "s1"] & (PB) & GG2 \arrow[ul, "s2"'] \\
& GG3 \arrow[ul, "s3"] \arrow[ur, "s4"'] &
\end{tikzcd}$$

☺

Proof. Let the pullback morphisms $p1 = (p1^{OP}_{T}, p1_N) : GG3 \to GG1$, $p2 = (p2^{OP}_{T}, p2_N) : GG3 \to GG2$ be defined componentwise in the categories $\mathbf{GraphP^{OP}}$ and **SetP**, where the components are the pullback morphisms induced by the pullbacks of types and rule names. Let X be a graph grammar and $x1 = (x1^{OP}_{T}, x1_N) : X \to GG1$ and $x2 = (x2^{OP}_{T}, x2_N) : X \to GG2$ be graph grammar morphisms. Then $u = (u^{OP}_{T}, u_N) : X \to GG3$ is defined componentwise where the components are the corresponding universal morphisms induced by the pullbacks of type graphs and sets of rule names. By Prop. 4.23, $p1$ and $p2$ are graph grammar morphisms. We have to show that u is a graph grammar morphism and that the universal property is satisfied. The universal property follows from the componentwise construction of u and the corresponding properties of the components. The proof that u is a graph grammar morphism can be found in Appendix C. $\surd$

5

Occurrence Graph Grammars

The concepts introduced in this and in the next chapters are closely related to corresponding notions for the area of Petri nets. In fact, they were inspired by these concepts for nets. In the following we will briefly discuss this relationship to motivate the definitions in this chapter. A more detailed discussion will be done in Sect. 7.3.

Petri nets [Pet62] is a well-established formalism for specifying concurrent systems. The basic idea is that the structure (control flow) of a system is represented by a bipartite graph, where the vertices are called places and transitions (or conditions and events, depending on the kind of net we are looking at). States are represented by sets (or multisets) of places and the current state is described by tokens lying on places. A change of state is described by the switching of a transition of the net. This switching removes some tokens of some places (the pre-conditions of a transition) and creates some new tokens in other places (the post-conditions of a transition). Like for graph grammars, there are many different kinds of semantics for Petri nets. Usually, the sequential semantics of nets is based on switching sequences. For the concurrent semantics, there are many different approaches, for example [Pet77, NPW81, GR83, BD87, MM90, Vog92]. In particular, the notion of a net process (originally introduced in [GR83]) gives a semantical description of a net in which the occurrences of events (switchings of transitions) in computations are ruled by *causal dependency relationships* between them. The interesting feature of the process semantics of nets is that the behaviour of a net is explained by a set of "special nets". These special nets are very simple nets that enjoy a lot of properties (deterministic, acyclic, safe). Such semantical nets were introduced in [NPW81] with the name of *(deterministic) occurrence nets.* They are called occurrence nets because each place of such a net represents the occurrence of some token in some place of the (syntactical) net, and each transition of the occurrence net represents the occurrence of a switching of a transition of the (syntactical) net. In an occurrence net one can identify directly the relationships between the switchings of transitions of a net. These relationships are typically the *causal dependency relation*, the *conflict relation* and the *concurrency relation* (where the third relation is derived from the first two). These relations give a very good basis for the formal analysis of a net [GW91, PL91, McM92, McM95].

Graph grammars can be considered as a generalization of Petri nets in the sense that graphs (instead of sets) represent the states and the pre-conditions of some transition are allowed to be read-only accessed, i.e., preserved. The latter feature can be also found in an extension of Petri nets called contextual nets [MR95]. Process semantics of graph grammars have already been defined in [Kre83, KW86, Kor95, KR95, CMR96a], but only the process semantics defined in [CMR96a] (for the DPO approach to graph grammars) introduces a concept that can be seen as the counterpart of deterministic occurrence nets for the case of (DPO) graph grammars. There a causal dependency relation between the rules (and type graph) of a graph grammar was defined in such a way that it corresponds exactly to the definition in the Petri nets case. But in this way, one of the most important features of graph grammars, namely the ability to preserve items, was not very much explored, as it will be explained as follows. Consider two rules $r1$ and $r2$ and matches into a graph G that overlap only in one item x that is preserved by $r1$ and deletes by $r2$. There is a relationship between these rules because $r1$ can not occur after $r2$ has occurred. In such a situation, it seems that $r2$ causally depends on $r1$. But, on the other hand, $r2$ may happen independently of $r1$ because the items x is already present in G. Therefore, the relation between these two rules is not causality but an occurrence order ($r1$ may only occur before $r2$). Obviously, the causal dependency between rules implies a particular occurrence order. In [CMR96a] this occurrence order was confused with causal order. Here we will distinguish between occurrence and causal orders. As we will see in Sect. 5 this is necessary to define a suitable notion of (non-deterministic) occurrence graph grammars. Moreover, the interplay between these relations (causal dependency, occurrence order and conflict relation) within a graph grammar yields a valuable means of reasoning about the semantics of this grammar.

We are aiming at a semantics of a graph grammar GG that describes all possible states and changes of states of GG and is a graph grammar itself. So the first problem is to find a way to represent all possible states of GG in a suitable way. One could take a set of graphs to represent the states, and then describe changes of states as relationships on this set. But then the specific way in which a state is related to another is lost. To preserve this information, we may relate the states through morphisms (the derivation morphisms). Thus we get a graph in which the states are (typed) graphs and the arrows are (typed) graph morphisms. This kind of semantics for graph grammars is called transition system semantics and was presented in [CEL$^+$96a]. But using this semantics it is very difficult to check which actions are dependent, independent or in conflict because the same item may be represented many times: each time an item is preserved by a derivation step, this item is represented in the input and in the output graphs. This discussion is completely analogous to the one that led from sequential to concurrent derivations. So, what we really would like to have as a representation of the states is one graph, analogous to the core graph. One action then would be a rule whose left- and right-hand sides are interpreted in terms of this core graph, as in the case of concurrent derivations. But, contrastingly to concurrent derivations, we would like to be able to represent non-deterministic actions in one object, what allows the description of the whole semantics of a graph grammar in one object (instead of using a category as in the case of the concurrent semantics). In this way, not only concurrent but also non-determinism is represented explicitly.

If we see the core graph as a type, and the set of actions as rules, we may notice that the definition of a concurrent derivation is very similar to the definition of a graph grammar. The only difference is that the core graph is a typed graph, whereas in a graph grammar the type is a simple graph (an object in **GraphP**). The reason for the fact that the core graph is typed is that the core graph represents all states involved in a sequential derivation, and these states are typed graphs. Thus, as the semantical entities represented by concurrent derivations shall give raise to grammars having core graphs as types, it seems reasonable to define this kind of grammars, and this will be done in Sect. 5.1. If we see the concept of a graph more abstractly, we can define a notion of 'graph' grammar in which the type graph is already a typed graph. One state of such a grammar will be then a typed graph that is typed into the type typed graph. We will call these kind of grammars *doubly-typed grammars*, and these will serve as a basis for defining a class of (doubly-typed) grammars that can be seen as a semantical framework for (typed) graph grammars, namely the *occurrence graph grammars* (Sect. 5). Although this concept was inspired by the corresponding one for Petri nets, our definitions and results do not match directly the corresponding ones for Petri nets. The choice of following this different way was done to assure the compatibility of the unfolding semantics with the parallel composition operators defined in Chap. 4 (a more detailed discussion on this topic will be done in Sects. 7.1 and 7.3).

The main aims, definitions and results of this chapter are:

- Definition of various relationships between the rules of a (doubly-typed) graph grammar (Defs. 5.20, 5.23, 5.25 and 5.27). These relations will be used to identify a class of (doubly-typed) graph grammars that can be seen as (possibly non-deterministic) computations of a graph grammar. These relationships are presented in Sect. 5.2 (the basic definition of doubly-typed graph grammars are introduced in Sect. 5.1).

- Definition of *occurrence graph grammars* and occurrence graph grammar morphisms (Sect. 5.3). It is shown in Prop. 5.37 that occurrence graph grammar morphisms preserve some (independency) relations. Moreover, a class of occurrence graph grammar morphism, namely *prefix morphisms* is defined. Prefix morphisms enjoy a lot of properties. Particularly important are the preservation of many relations by prefix morphisms (Prop. 5.41), and the fact that there can be at most one prefix morphism between two occurrence graph grammars (Prop. 5.43).

- In Sect. 5.4, concurrent derivations and concurrent derivation morphisms are characterized as special kinds of occurrence graph grammars and graph grammar morphisms (Theo. 5.45 and Theo. 5.46). This will be used to establish a relationship between the concurrent and the unfolding semantics of graph grammars that will be introduced in Chap. 6.

- Definition of a folding construction for occurrence grammars (Def. 5.47). This folding establishes a relationship between occurrence graph grammars and typed graph grammars. In particular, it is shown that an occurrence graph grammar represents the derivations of its folded grammar (Prop. 5.50 and 5.51). This is presented in Sect. 5.5.

5.1 Doubly-Typed Graph Grammars

The definitions and results from this section are analogous to the ones in Sect. 3.1 and 3.3, where typed graphs and (typed) graph grammars were defined. To get the category of doubly-typed graphs, one has just to take the corresponding definition of typed graphs (Def. 3.5) and substitute the category **GraphP** by **TGraphP**(**T**). Again, colimits are constructed componentwise in the basis category, i.e., **TGraphP**(**T**) in this case, followed by a totalization construction (that makes the typing morphism total). The definitions concerning graph grammars are then obtained by substituting **TGraphP** by **DTGraphP**(**T**) (category of doubly-typed graphs) in the corresponding definitions from Sect. 3.3. Although we believe that it would be possible to carry over all results to this case, we will only define explicitly what we will need in the rest of this thesis.

Definition 5.1 (Doubly-Typed Graph) *Let T be a graph. A* ***doubly-typed graph*** *$G^{TG \nearrow T}$ over T is a tuple $G^{TG \nearrow T} = (G^T, t^{G^T}, TG^T)$ where G^T and TG^T are typed-graphs and $t^{G^T} : G^T \to TG^T$ is a total typed-graph morphism in* $\mathbf{TGraphP(T)}$. *The typed graph TG^T is called* ***double-type graph****.*

We denote by $x \in G^{TG \nearrow T}$ an element $x \in V_G \cup E_G$. ☺

Example **5.2 (Doubly-Typed Graph)** The graph depicted in Figure 5.1 is a doubly-typed graph with double-type $C1^{T1}$. Usually we will draw only the typing morphisms from $G1$ to $C1$ and from $C1$ to $T1$ (because the typing morphism from $G1$ to $T1$ is the composition of the other two). ☺

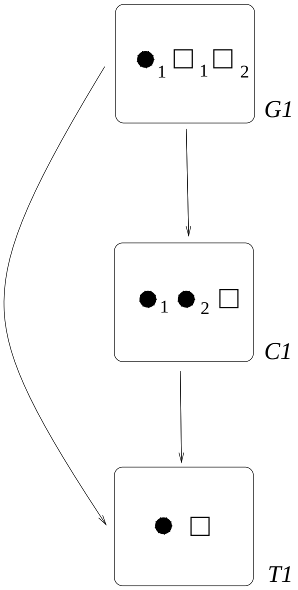


Figure 5.1: Doubly-Typed Graph $G1^{C1 \nearrow T1}$

We will define three kinds of morphisms between double-typed graphs: the first one between (doubly-typed) graphs over different type graphs; the second one between (doubly-typed) graphs over the same type graph; and the third between double-typed graphs over the same double-type graph. The second kind of morphism is a special case of the first kind, and the third a special case of the second. These three kinds will be used to define morphisms between doubly-typed graph grammars, morphisms between concurrent derivations (and unfoldings) of the same grammar and rules/matches of one grammar, respectively.

Definition 5.3 (Doubly-Typed Graph Morphisms) *Let $G^{TG \nearrow T}$ and $H^{TH \nearrow T'}$ be doubly-typed graphs and $g^{t'} : G^T \to H^{T'}$ and $t^{t'} : TG^T \to TH^{T'}$ be typed graph morphisms (morphisms in* $\mathbf{TGraphP}$*).*

$$\begin{tikzcd}
G^T \arrow[r, dashed, "g^{t'}"] \arrow[d, "t^{G^T}"'] \arrow[dr, phantom, "(1)"] & H^{T'} \arrow[d, "t^{H^{T'}}"] \\
TG^T \arrow[r, dashed, "t^{t'}"'] & TH^{T'}
\end{tikzcd}
\qquad
\begin{tikzcd}
G^T \arrow[r, dashed, "g^T"] \arrow[d, "t^{G^T}"'] \arrow[dr, phantom, "(2)"] & H^T \arrow[d, "t^{H^T}"] \\
TG^T \arrow[r, dashed, "t^T"'] & TH^T
\end{tikzcd}
\qquad
\begin{tikzcd}
G^T \arrow[r, dashed, "g^T"] \arrow[d, "t^{G^T}"'] \arrow[dr, phantom, "(3)"] & H^T \arrow[d, "t^{H^T}"] \\
TG^T \arrow[r, dashed, "id^T_{TG}"'] & TG^T
\end{tikzcd}$$

1. *The pair $(g^{t'}, t^{t'})$ is a* ***doubly-typed graph morphism****, denoted by $g^{t\nearrow t'} : G^{TG\nearrow T} \to H^{TH\nearrow T'}$, iff diagram (1) commutes weakly.*

2. *If $g^{t\nearrow t'}$ is a doubly-typed graph morphism and $t' = id_T$ we say that $g^{t\nearrow t'}$ is a* ***T-doubly-typed graph morphism****, denoted by $g^{t\nearrow T}$. In this case, the weak commutativity requirement reduces to the one shown in diagram (2).*

3. *If $g^{t\nearrow T}$ is a T-doubly-typed graph morphism and $t = id_{TG}$ we say that $g^{t\nearrow T}$ is a* ***TG^T-doubly-typed graph morphism****, denoted by $g^{TG\nearrow T}$. In this case, the weak commutativity requirement reduces to the one shown in diagram (3).*

The categories of doubly-typed graphs and doubly-typed graph morphisms, T-doubly-typed graph morphisms and TG^T-doubly-typed graph morphisms are denoted by **DTGraphP**, **DTGraphP(T)** *and* **DTGraphP(TG$^\mathbf{T}$)***, respectively.* ☺

Remark. *The weakly commuting diagrams shown in the previous definition are diagrams in* **TGraphP(T)**. *The corresponding diagrams in* **GraphP***(obtained by making all typing morphisms explicit) are shown below. As $t^{G^T} = (t^{G\nearrow}, id_T)$ and $t^{H^{T'}} = (t^{H\nearrow}, id_T)$ are total typed graph morphisms, diagrams marked with "=" commute. As $g^{t'}$ and $t^{t'}$ are typed graph morphisms, diagrams marked with "$\leq$" weakly commute. Thus, weak commutativity of the morphism $g^{t\nearrow t'}$ means concretely that the outer squares commute weakly, i.e., $t^{H\nearrow} \circ g \leq t \circ t^{G\nearrow}$ (in the right-most diagram we have $t = id_{TG}$, and thus the weak commuting square reduces to a weak commuting triangle).*

$$\begin{tikzcd}
G \arrow[rr, dashed, "g"] \arrow[dd, "t^{G\nearrow}"'] \arrow[dr] & & H \arrow[dd] \arrow[dr] & \\
{=} & T \arrow[rr, dashed, "t'"', "="] \arrow[ur, phantom, "\geq"] \arrow[dr, phantom, "\geq"] & {t^{H\nearrow}} & T' \\
TG \arrow[rr, dashed, "t"'] \arrow[ur] & & TH \arrow[ur] &
\end{tikzcd}
\quad
\begin{tikzcd}
G \arrow[rr, dashed, "g"] \arrow[dd, "t^{G\nearrow}"'] \arrow[dr] & \geq & H \arrow[dd, "t^{H\nearrow}"] \arrow[dl] \\
{=} & T & {=} \\
TG \arrow[rr, dashed, "t"'] \arrow[ur] & \geq & TH \arrow[ul]
\end{tikzcd}
\quad
\begin{tikzcd}
G \arrow[rr, dashed, "g"] \arrow[dr] \arrow[ddr, "t^{G\nearrow}"'] & \geq & H \arrow[dl] \arrow[ddl, "t^{H\nearrow}"] \\
{=} & T & {=} \\
& TG \arrow[u] &
\end{tikzcd}$$ ☺

Example **5.4 (Doubly-Typed Graph Morphisms)** Figure 5.2 shows two doubly-typed graph morphisms $f^{\nearrow} = (f^{t1}, t2^{t1})$ and $g^{\nearrow} = (g^{id_{T2}}, id_{C2}^{id_{T2}})$. The latter is a morphism in **DTGraphP(C2$^\mathbf{T2}$)**. ☺

Pushouts in **DTGraphP(TG$^\mathbf{T}$)** can be constructed componentwise in **TGraphP(T)**, where the second component is the identity of TG^T. For the explicit construction see Construction B.16.

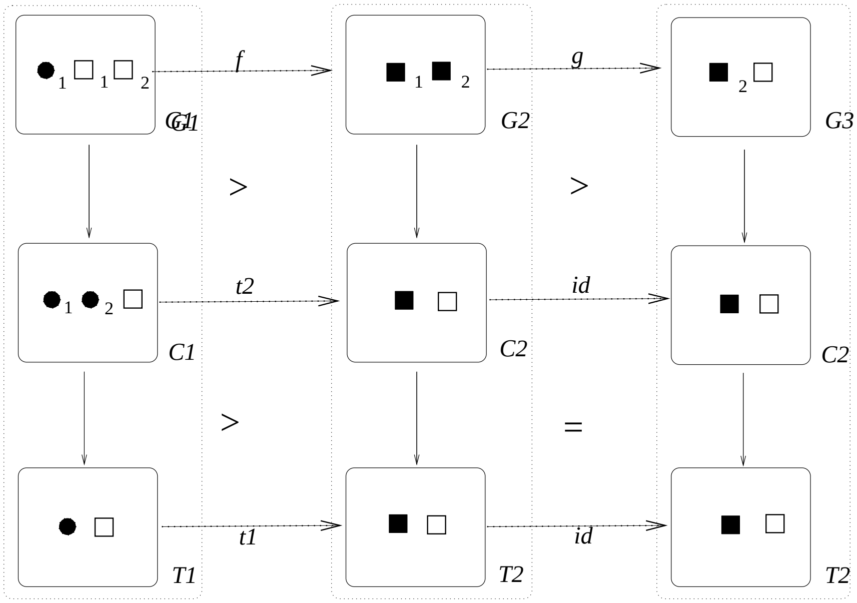


Figure 5.2: Doubly-Typed Graph Morphisms

Definition 5.5 (ReDoubletyping Functor) *Let $f^t : TG2^{T2} \to TG1^{T1}$ be a morphism in* **TGraphP**. *Then there is a functor $\mathcal{DT}_{f^t}$:* $\mathbf{DTGraphP(TG1^{T1})} \to \mathbf{DTGraphP(TG2^{T2})}$ *induced by f^t.*

$$\begin{tikzcd}
TG1^{T1} & \mathbf{DTGraphP(TG1^{T1})} \arrow[d, "\mathcal{DT}_{f^t}"] \\
TG2^{T2} \arrow[u, "f^t"] & \mathbf{DTGraphP(TG2^{T2})}
\end{tikzcd}$$

$\mathcal{DT}_{f^t}$ is defined for all object $G1^{TG1\nearrow T1} = (G1^{T1}, t^{G1^{T1}}, TG1^{T1})$ with typing morphism $t^{G1^{T1}} = (t^{G1\nearrow}, id_{T1})$ and morphism $g^{TG1\nearrow T1} = (g^{TG1}, id_{TG1^{T1}}) : G1^{TG1\nearrow T1} \to H1^{TG1\nearrow T1}$ in $\mathbf{DTGraphP(TG1^{T1})}$ *as follows:*

- **Objects:** *$\mathcal{DT}_{f^t}(G1^{TG1\nearrow T1}) = (G2^{T2}, t^{G2^{T2}}, TG2^{T2})$, where $G2^{TG2} = \mathcal{T}_f(G1^{TG1})$ with typing morphism $t^{G2\nearrow} : G2 \to TG2$, $t^{G2} = t^{TG2} \circ t^{G2} : G2 \to T2$ and $t^{G2^{T2}} = (t^{G2\nearrow}, id_{T2})$.*

$$\begin{tikzcd}
G1 \arrow[rr, "t^{G1\nearrow}"] \arrow[rrr, bend left, "t^{G1}"] \arrow[rrr, phantom, bend left=15, "="] & \overline{PB} & TG1 \arrow[r, "t^{TG1}"] & T1 \\
& dom(f) \arrow[ur, "f!"] \arrow[dr, "f^{\blacktriangledown}"'] \arrow[r, phantom, "="] & \arrow[r, phantom, "\leq"] & \\
G2 \arrow[uu, "f!^G"] \arrow[ur, "(t^{G1\nearrow})^{\bullet}"] \arrow[rr, "t^{G2\nearrow}"'] \arrow[rrr, bend right, "t^{G2}"'] \arrow[rrr, phantom, bend right=15, "="] & = & TG2 \arrow[uu, "f"] \arrow[r, "t^{TG2}"'] & T2 \arrow[uu, "t"']
\end{tikzcd}$$

- **Morphisms:** $\mathcal{DT}_{f^t}(g^{TG1\nearrow T1}) = g'^{TG2\nearrow T2} = (g'^{TG2}, id_{TG2^{T2}})$, *where* $g'^{TG2} = \mathcal{T}_f(g^{TG1})$.

$$\begin{tikzcd}
G1 \arrow[rr, "g"] \arrow[dr, |->, "t^{G1\nearrow}"] & & H1 \arrow[dl, |->, "t^{H1\nearrow}"'] \\
 & TG1 & \\
G2 \arrow[uu, |->, "f!^G"] \arrow[rr, "g'"] \arrow[dr, |->, "(t^{G1\nearrow})^\bullet"'] \arrow[ur, phantom, "\overline{PB}"] & & H2 \arrow[uu, |->, "f!^H"'] \arrow[dl, |->, "(t^{H1\nearrow})^\bullet"] \arrow[ul, phantom, "\overline{PB}"'] \\
 & dom(f) \arrow[uu, |->, "f!"'] \arrow[d, hook, "f^{\blacktriangledown}"] & \\
 & TG2 &
\end{tikzcd}$$

☺

Proposition 5.6 $\mathcal{DT}_{f^t}$ *is well-defined.* ☺

Proof. By construction, $G2$ is a graph and t^{G2} is a total graph morphism. Therefore, $G2^{T2}$ is a typed graph over $T2$. It remains to show that $t^{G2^{T2}} = (t^{G2}, id_{T2})$ is a total typed graph morphism, what is true by definition of t^{G2} ($t^{G2} = t^{TG2} \circ t^{g2\nearrow}$). For the well-definedness of morphisms we have to show that $t^{H2\nearrow} \circ g' \leq t^{G2\nearrow}$. This is assured by the fact that $g'^{TG2} = \mathcal{T}_f(g^{TG1})$ is a morphism in **TGraphP(T2)** ($\mathcal{T}_f$ is well-defined – Prop. 3.11). $\surd$

Example **5.7 (Double-Retyping Construction)** Figure 5.3 shows the retyping of the double-typed graph $G1^{C1\nearrow T1}$ using the typed graph morphism c^t. In this retyping, each □-vertex is duplicated because there are two elements in $dom(c)$ that are mapped to the same element of $C1$ (remind that the morphisms are indicated by using same symbols and/or indices). The •-vertex is forgotten because this type is not in $C2$. Note that through a double-retyping, usually graphs can only become "smaller", in the sense that all elements of the retyped graph are in the original graph (maybe some are duplicated). Now look at Figure 5.4. There the retyping of the morphism $(g1^{T1}, C1^{T1})$ with respect to the same typed graph morphism c^t is shown. The construction is done in the following way: first the source and target graphs of the morphism are retyped (pullbacks (1) and (2)), then the domain of the retyped morphism is constructed (pullback (3)) and is mapped to the target graph in a compatible way. ☺

Proposition 5.8 *Let* $r = (r1^{TG1\nearrow T1}) : L1 \to R1$ *and* $m = (m1^{TG1\nearrow T1}) : L1 \to G1$ *be morphisms in* **DTGraphP(TG1^T1)** *where* $r1$ *is injective and* $m1$ *is total, and (1) below be a pushout in* **DTGraphP(TG1^T1)** *of* r *and* m. *Let* $f : TG2^{T2} \to TG1^{T1}$ *be a typed graph morphism. Then (2) is a pushout in* **DTGraphP(TG2^T2)**.

$$\begin{tikzcd}
L1 \arrow[r, tail, "r"] \arrow[d, |->, "m"'] \arrow[dr, phantom, "(1)"] & R1 \arrow[d, "m^\bullet"] \\
G1 \arrow[r, tail, "r^\bullet"'] & H1
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{DT}_f(L1) \arrow[r, tail, "\mathcal{DT}_{f(r)}"] \arrow[d, |->, "\mathcal{DT}_{f(m)}"'] \arrow[dr, phantom, "(2)"] & \mathcal{DT}_f(R1) \arrow[d, "\mathcal{DT}_{f(m^\bullet}"] \\
\mathcal{DT}_f(G1) \arrow[r, tail, "\mathcal{DT}_{f(r^\bullet)}"'] & \mathcal{T}_f(H1)
\end{tikzcd}$$

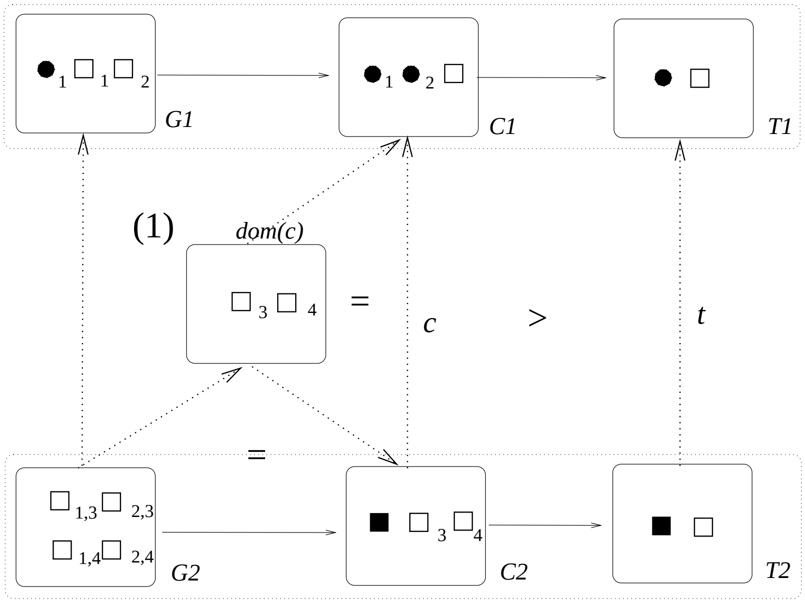


Figure 5.3: Double-Retype of a Double-Typed Graph

☺

Proof. As pushouts in $\mathbf{DTGraphP(TG^T)}$ are constructed componentwise (see Construction B.16) and $\mathcal{T}_f$ preserves pushouts of one injective and one total morphisms (Prop. 3.12), we conclude that $\mathcal{DT}_{f^t}$ also preserves pushouts. √

The next definition establishes a relationships between doubly-typed and (single) typed graphs and morphisms via a functor, called *type-forgetful functor*. As the name says, this functor forgets one of the types of a double-type graph. Let $G^{TG \nearrow T}$ be a doubly-typed graph. By definition, this graph consists of two typed graphs, G^T and TG^T and a typed graph morphism $t^T : G^T \to TG^T$ connecting them. The type-forgetful functor forgets the component TG^T and the typing morphism, yielding the typed graph G^T. This functor will be used in Sect. 5.5 to define a functor between the categories of occurrence graph grammars (that are grammars having a typed-graph as type) and of typed graph grammars.

Definition 5.9 (Type-forgetful Functor) *Let* **DTGraphP** *and* **TGraphP** *be categories of doubly-typed and typed graphs, respectively. Then the* ***type-forgetful functor*** $\mathcal{V}_T$: **DTGraphP** $\to$ **TGraphP** *is defined as:*

- **Objects:** $\mathcal{V}_T(G^{TG \nearrow T'}) = G^{T'}$, *for any* $G^{TG \nearrow T'} = (G^{T'}, t^{G^{T'}}, TG^{T'}) \in$ **DTGraphP**.
- **Morphisms:** $\mathcal{V}_T(f^{t \nearrow t'}) = f^{t'}$, *for any morphism* $f^{t \nearrow t'} = (f^{t'}, t^{t'}) \in$ **DTGraphP**.

☺

Proposition 5.10 *The type-forgetful functor* $\mathcal{V}_T$ *is well-defined.* ☺

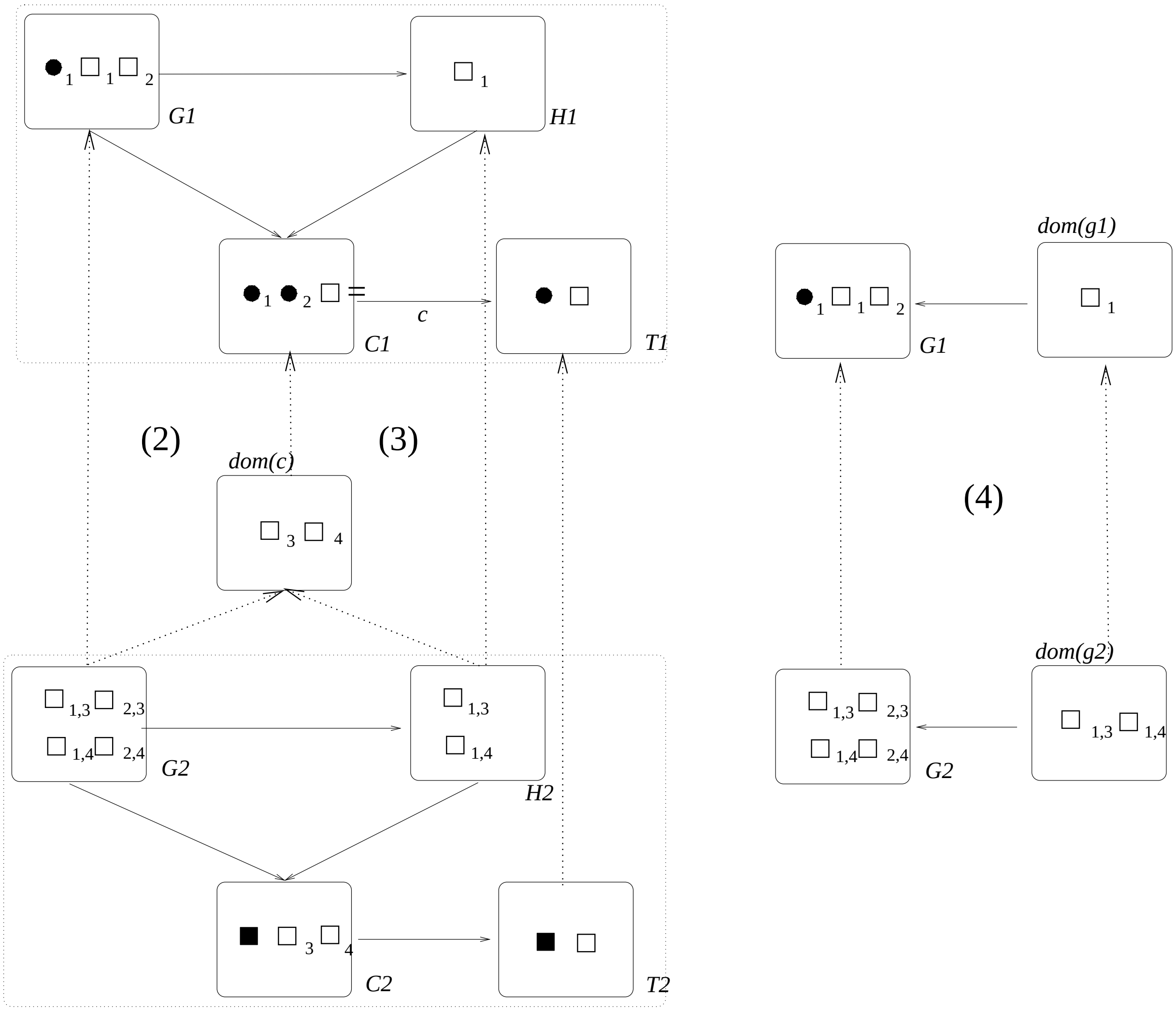


Figure 5.4: Double-Retype of a Double-Typed Graph Morphism

Proof. Obviously, $\mathcal{V}_T$ yields objects/morphisms in **TGraphP**. As identities, composition in **DTGraphP** are constructed componentwise in **TGraphP** and $\mathcal{V}_T$ is a projection, $\mathcal{V}_T$ preserves identities and composition. $\surd$

Example **5.11 (Type-Forgetful Functor)** Figure 5.3 shows the result of applying the type-forgetful functor to the doubly-typed graph morphisms of Figure 5.2. ☺

Definition 5.12 ((Doubly-typed) Rule) *Given a doubly-type graph TG^T. Then a* **rule** *with respect to TG^T is a morphism $a = r^{TG\nearrow T} = (r^T, id_{TG^T}) : L^{TG\nearrow T} \to R^{TG\nearrow T}$ in* **DTGraphP($\mathbf{TG^T}$)** *iff r^T is a rule in* **TGraphP** *(see Def. 3.14).*

We denote by $Rules(TG^T)$ the class of rules over the double-type graph TG^T and $DRules(TG^T)$ the corresponding extension to isomorphism classes of general rules.

Let the typing morphisms from $L^{TG\nearrow T}$ and $R^{TG\nearrow T}$ be t^{L^T} and t^{R^T}, respectively. Then we define

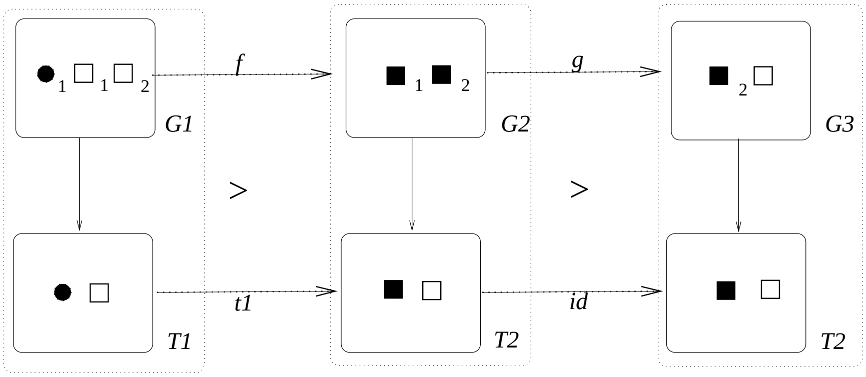


Figure 5.5: Result of Applying the Type Forgetful Functor

- $L_a = L^T$ *and* $R_a = R^T$, *the left- and right-hand sides of the rule* $a = r^{TG\nearrow T}$
- $pre_a = t^{L^T} : L^T \to TG^T$, *the* **pre-condition** *of the rule* $a = r^{TG\nearrow T}$
- $post_a = t^{R^T} : R^T \to TG^T$, *the* **post-condition** *of the rule* $a = r^{TG\nearrow T}$
- $r_a = r^T$, *the* **rule pattern** *of the rule* $a = r^{TG\nearrow T}$

A doubly-typed rule $r1^{TG\nearrow T}$ *is a* **subrule** *of a doubly-typed rule* $r2^{TG\nearrow T}$, *denoted by* $r1^{TG\nearrow T} \subseteq^r r2^{TG\nearrow T}$, *iff* $r1^{TG}$ *is a subrule of* $r2^{TG}$ *(see Def. 3.16).* ☺

Definition 5.13 (D-Rules Functor) *DRules extends to a functor* $\mathcal{DR}$: $\mathbf{DTGraphP^{OP}} \to \mathbf{SetP}$, *defined for all object* TG^T *and morphism* $f^t : TG2^{T2} \to TG1^{T1}$ *of* **DTGraphP** *as follows:*

- **Objects:** $\mathcal{DR}(TG^T) = IRules^+(TG^T)$
- **Morphisms:** $\mathcal{DR}(f^t) = \mathcal{DR}_{f^t} : IRules^+(TG1^{T1}) \to IRules^+(TG2^{T2})$ *is defined for all* $[r] \in IRules^+(TG1^{T1})$ as

$$\mathcal{R}_{f^t}([r]) = \begin{cases} [\mathcal{DT}_{f^t}(r)], \text{ if } \mathcal{DT}_{f^t}(r) \text{ is not an isomorphism}, \\ \texttt{undef}, \text{ otherwise} \end{cases}$$

☺

Proposition 5.14 $\mathcal{DR}$ *is well-defined.* ☺

Proof. Analogous to the proof of the well-definedness of $\mathcal{R}$ (Prop. 3.18), based on Lemma C.2, on Prop. 3.13 and the componentwise construction of identities and composition in **DTGraphP**. $\surd$

The next definition introduces doubly-typed graph grammars. These are graph grammars in which the type graph is a typed graph (compare Def. 3.33).

Definition 5.15 ((Doubly-Typed) Graph Grammar, Morphism) *A* ***doubly-typed graph grammar*** *is a tuple* $GG = (TG^T, I^{TG\nearrow T}, N, n)$ *where*

- *TG^T is a double-type graph (the **type** of the grammar),*
- *$I^{TG\nearrow T}$ is a doubly-typed graph in $\mathbf{DTGraphP(TG^T)}$ (the **initial graph** of the grammar),*
- *N is a set of **rule names**,*
- *$n : N \to Rules(TG^T)$ is a total function (the **naming function**, assigning to each rule name a rule with respect to the type TG^T).*

We denote by in_{GG} the typing morphism of the initial graph, i.e., $in_{GG} = t^{I^T} : I^T \to TG^T$.

*Let $GG1 = (TG1^{T1}, I1^{TG1\nearrow T1}, N1, n1)$ and $GG2 = (TG2^{T2}, I2^{TG2\nearrow T2}, N2, n2)$ be two graph grammars. Then a **doubly-typed graph grammar morphism** is a pair $f = (f_T^{OP}, f_N)$ where f_T^{OP} is morphism in $\mathbf{DTGraphP^{OP}}$ and f_N is a function such that the following conditions are satisfied*

1. *The following diagram sub-commutes, where $\overline{n1}$ and $\overline{n2}$ are the extensions of $n1$ and $n2$ (see Def. 3.33), i.e., for all $x \in domf_N$ or $x \in dom(\mathcal{DR}_{f_T} \circ \overline{n1})$, $\overline{n2} \circ f_N(x) \subseteq^R \mathcal{DR}_{f_T} \circ \overline{n1}(x)$.*

$$\begin{tikzcd}
N1 \arrow[r, mapsto, "\overline{n1}"] \arrow[d, "f_N"'] \arrow[dr, phantom, "\subseteq^R"] & \mathcal{DR}(TG1^{T1}) \arrow[d, "\mathcal{DR}_{f_T}"] \\
N2 \arrow[r, mapsto, "\overline{n2}"'] & \mathcal{DR}(TG2^{T2})
\end{tikzcd}$$

2. *$I2^{TG2\nearrow T2} \cong \mathcal{DT}_{f_T}(I1^{TG1\nearrow T1})$.*

The category of doubly-typed graph grammars is denoted by **DTGG**. ☺

Example **5.16 (Doubly-Typed Graph Grammar/Morphism)** In Figure 5.6 we can see two doubly-typed graph grammars ($GG1$ and $GG2$) and a doubly-typed graph grammar morphism $f = (f_T^{OP}, f_N) : GG1 \to GG2$, where $f_T = c^t$ and f_N maps the only rule in $GG1$ to the only rule in $GG2$. We will call the typing morphisms from the left- and right-hand sides of rules by *pre* and *post* morphisms. The initial graph of $GG2$ is exactly the result obtained by (double-) retyping the initial graph of $GG1$. The rule $r2$ is not exactly the retyping of the rule $r1$, but a subrule of it (the retyping of $r1$ is $\mathcal{T}(r1)$ in the figure). ☺

Definition 5.17 (Match,Derivation Step) *Given a rule $r : L \to R$ with respect to a double-type TG^T, a **match** $m : L \to IN$ of r in a doubly-typed graph IN is a total morphism in $\mathbf{DTGraphP(TG^T)}$. A **derivation step** of a graph IN_s with rule r_s with name nr_s at match m_s, denoted by $IN_s \overset{nr_s:m_s}{\Longrightarrow} OUT_s$, is a tuple $s = (n_s, S)$, where S is a pushout $IN_s \overset{r_s^\bullet}{\to} OUT_s \overset{m_s^\bullet}{\leftarrow} R_s$ of m_s and r_s in $\mathbf{DTGraphP(TG^T)}$ (see Construction B.16). The components IN_s, OUT_s, $r_s^\bullet$ and $m_s^\bullet$ are called **input graph**, **output graph**, **co-rule** and **co-match**, respectively.*

$$\begin{tikzcd}
L_s \arrow[r, "r_s"] \arrow[d, "m_s"'] \arrow[dr, phantom, "PO"] & R_s \arrow[d, "m_s^\bullet"] \\
IN_s \arrow[r, "r_s^\bullet"'] & OUT_s
\end{tikzcd}$$

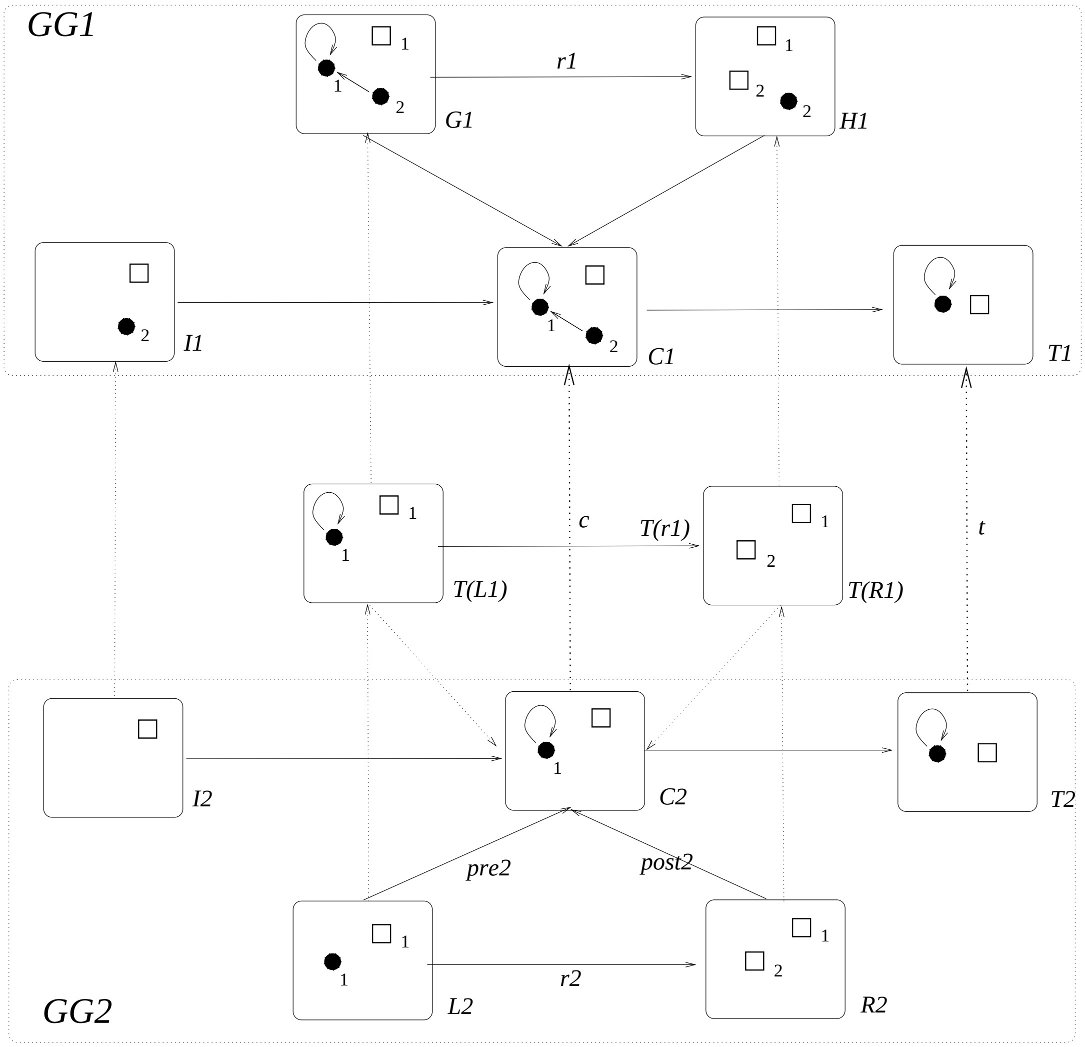


Figure 5.6: Doubly-Typed Graph Grammar Morphism

☺

5.2 Relations within a (Doubly-Typed) Graph Grammar

In this section we will investigate important relationships between rules and items of the (double-type) of a grammar. These relationships will be used in Sect. 5 to define a suitable subclass of doubly-typed graph grammars that can be considered as "behaviours" of (single) typed graph grammars, namely *occurrence graph grammars*. The basic idea is to interpret the graph TG of a double-type graph TG^T as a graph describing *occurrences* of items that are typed over T. These items represent the vertices and edges that may occur in derived graphs of a typed graph grammar. This idea corresponds to the notion of a *core graph* of a concurrent derivation (see Def. 3.39). Rather than constructing a core graph, as it was the case for concurrent derivations, we will define it axiomatically. In Sect. 5.4 we will show that concurrent derivations are special kinds of occurrence graph grammars, and that the core graph of a concurrent derivation is a core graph in the sense of the next definition (and therefore we used the same name).

As a core graph shall describe the occurrences of items (vertices/edges), each element of a core graph must have a origin: either it has been present in the initial graph of the grammar or it was created by one rule of the grammar. Moreover, the origin of each item in the core graph must be unique. This captures the idea that the same vertex can not be created twice in some derivation of a grammar.

Definition 5.18 (Core Graph) *Let $GG = (C^T, I^{C\nearrow T}, N, n)$ be a doubly-typed graph grammar. Then C^T is called a* ***core graph*** *iff it satisfies the following condition:*

$\forall x \in C^T$: $\exists ! y \in (I^T \uplus (\biguplus_{ai \in rng(n)} R_{ai}))$*such that*

$$x = \begin{cases} in_{GG}(y), \text{ if } y \in I^T, \\ post_{ai}(y), \text{ if } y \in R_{ai} \text{ and } y \notin rng(r_{ai}) \end{cases}$$

If C^T is a core graph, then each element in $rng(n)$ is called an ***action*** *of GG. An action a* ***creates*** *an element $e \in (V_C \cup E_C)$ iff $e \in rng(post_a)$ and $e \notin rng(post_a \circ r_a)$. Action a* ***deletes*** *e iff $e \in rng(pre_a)$ and there is $el \in L_a$ such that $pre_a(el) = e$ and $el \notin dom(r_a)$.* ☺

Remark. *As the term "rule", the term "action" is also overloaded: sometimes we will speak about an action meaning the element associates to a name via the naming function of a grammar, sometimes meaning the name itself of the action and sometimes meaning the pair (name,action).* ☺

Notation: Let GG be a doubly typed graph grammar whose double-type graph is a core graph. Then the names of its components will be by default C^T for the core graph, $I^{C\nearrow T}$ for the initial graph, N for the set of rule names and n for the naming function. If the name of the grammar is indexed, for example $GG1$, the components will be indexed accordingly.

Although it would be possible to define the relationships between rules and/or items of the (double-)type graph of arbitrary doubly-typed graph grammar, we will here stick to the grammars whose double-type is a core graph. This has the advantage that the interpretation of the relations in this case is directly the one that will be used in the next sections. For example, the causal dependency relation of a grammar that has a core graph as type (Def. 5.20) relates actions that necessarily must occur before others within some derivation of the corresponding (single-typed) grammar. If the double-type is not a core graph, this relation expresses a "potential" causal relationship: there may be a derivation in which the corresponding actions are causally related. Moreover, these relationships could have been defined for typed graph grammars and we believe that these more general definitions may be useful for proving properties of systems specified with typed (or double-typed) graph grammars. But as proving properties of grammars is out of the scope of this thesis, we preferred to study in more detail the (very relevant) special case of grammars with core graphs as types.

Some relations between rules and/or elements of the type graph of a grammar have already been defined in other works. In [Kor96] the causal dependency and the weak conflict relations as defined here were defined between actions of a concurrent derivation (that is a special type of doubly-typed graph grammar – see Sect. 5.4). In [CMR96a] a causal relationship was defined for (strongly safe DPO) graph grammars. But causality there has a different meaning than here. There a rule that preserves some elements that are needed by another rule are considered as cause of this second rule. Here only rules that create some item that is needed by another one can be considered as causes of this second rule. This stresses the idea that a cause provides the necessary conditions for some action. The causal relation in [CMR96a] corresponds here to the so-called occurrence relation, that describes possible orders in which the actions may occur.

The first relationship of a graph grammar that will be investigated here is the *causal dependency relation*. The intuitive idea of this relation is that

> An action a is a (direct) cause of an action b if a creates some item that is needed (preserved/deleted) by b. This implies that b can only happen after a has happened.

The causal dependency relation is also defined between types: we say that an item x of the core graph is a cause of an item y if the "deletion" of x causes the "creation" (i.e., there is some action that deletes x and creates y).

Notation:

1. The set of ***pre-conditions*** of a element $a \in A$ with respect to a relation $R \subseteq A \times A$, denoted by $Pre^R(a)$, is defined by
$$Pre^R(a) = \{a' | a'Ra\}$$
2. The set of ***minimal elements*** with respect to a relation R, denoted by Min^R, is defined by
$$Min^R = \{x| \nexists y : yRx\}$$
3. The restriction of a relation R to a set of elements $S \subseteq A$, denoted by $R|_S \subseteq S \times S$, is defined by
$$R|_S = \{aRb | a, b \in S\}$$

Example **5.19 ((Causal) Dependency Relation)** Consider the double-typed graph grammar $GG1$ depicted in Figure 5.7. The typed graph $C1^{T1}$ is a core graph: the black and white circles were present in the initial graph, the white square was created by rule $r1$, the edge was created by rule $r2$ and the black square was created by rule $r3$. To understand the causal dependencies between the actions of $GG1$ better we draw the typing morphisms explicitly from the rules to the core graph. The overlappings of these morphisms in the core graph can be used to find out which actions causally depend on other ones. For example, we can notice that the actions $a1$ and $a2$ overlap in the item □. This white square is created by action $a1$ (is in the right-hand side, but not in the left-hand side) and is deleted by action $a2$ (is in the left-hand side, but not in the right-hand side). Therefore we can say that $a2$ is causally dependent of $a1$, denoted by $a1 \leq a2$. The same relationship can be observed between actions $a2$ and $a3$: the edge is created by $a2$ and deleted by $a3$, therefore $a2 \leq a3$. Although actions $a2$, $a3$ and $a4$ overlap in the white circle, actions $a2$ and $a4$ (and $a3$ and $a4$) are not causally dependent because action $a2$ does not create anything that is needed by $a4$. In fact, action $a4$ can occur without any other action occurring first. Therefore in this example we have the following dependencies between actions: $a1 \leq a2$, $a2 \leq a3$, $a1 \leq a3$ (obviously, is $a3$ depends on $a2$ and $a2$ depends on $a1$ we must have that $a3$ depends on $a1$).

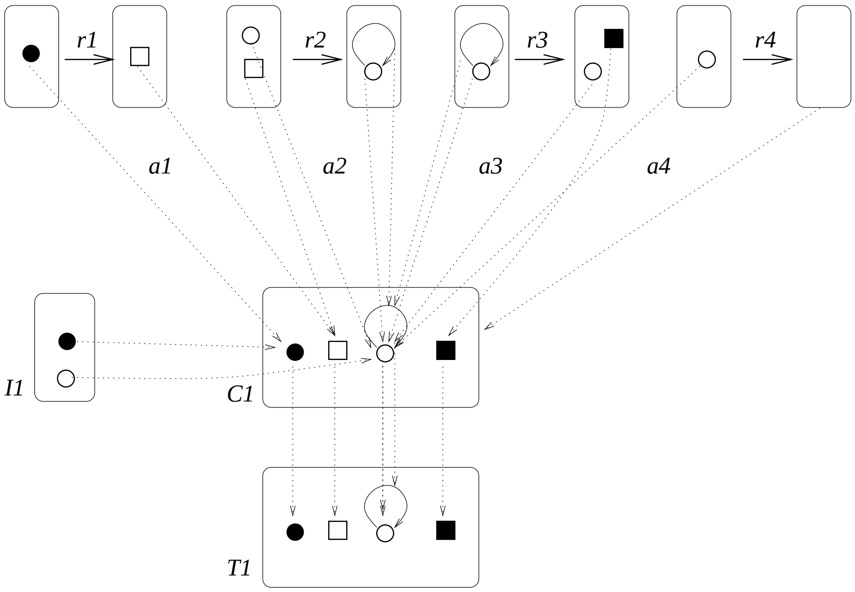


Figure 5.7: Doubly-Tuped Grammar $GG1$

The dependencies between the elements of the core graph is derived from the deletion/creation relation imposed by the rules. For example, □ depends on • because □ can only be created if a • is deleted (described by action $a1$). In the same way we get the following dependencies between types: $\bullet \leq \square$, $\square \leq\rightarrow$, $\bullet \leq\rightarrow$, $\rightarrow\leq \blacksquare$, $\square \leq \blacksquare$ and $\bullet \leq \blacksquare$. ☺

The formalization of this intuitive idea of causal dependency is based on the application of an "inverse" rule: if we want to find out whether action $a2$ is dependent on action $a1$ we remove everything that was added by $a1$ from the core graph, then if all elements needed by $a2$ are in the resulting core graph $a2$ is not dependent on $a1$, otherwise $a2$ depends on $a1$. This removal of items of the core graph can be done by applying the inverse of the rule of $a1$. The definition given below corresponds to the notion of weak parallel dependency, that is a relation defined between derivation steps of a graph grammar.

Definition 5.20 ((Causal) Dependency Relation) *Let $GG = (C^T, I, N, n)$ be a doubly-typed graph grammar and C^T be a core graph. Let $n1, n2 \in N$ and $n(n1) = a1 = r1^{TG\nearrow T}$, $n(n2) = a2 = r2^{TG\nearrow T}$, $a1 \neq a2$. Let $e1, e2 \in C$, $e1 \neq e2$*[1].

1. *The action $n2$ is* **directly (causally) dependent** *of $n1$, written $n1 \trianglelefteq^N n2$, iff $(r_{a1})^{-1\bullet} \circ pre_{a2}$ is not total, where (1) is a pushout in* **TGraphP(T)**. *Otherwise, $n2$ is* **not directly dependent** *of $n1$, denoted by $n1 \ntrianglelefteq^N n2$. If $n1 \ntrianglelefteq^N n2$ and $n2 \ntrianglelefteq^N n1$, we say that $n1$ and $n2$ are* **directly independent**.

$$\begin{tikzcd}
L_{a1} \arrow[d] & R_{a1} \arrow[l, "(r_{a1})^{-1}"'] \arrow[dr, "post_{a1}"] \arrow[dl, phantom, "(1)"] & & L_{a2} \arrow[r, "r^T_{a2}"] \arrow[dl, "pre_{a2}"] & R_{a2} \\
X1 & & C^T \arrow[ll, "(r_{a1})^{-1\bullet}"] & &
\end{tikzcd}$$

[1] Remind that $e1, e2 \in C$ means $e1, e2 \in V_C \cup E_C$.

2. *The (causal)* ***dependency relation between actions*** *of a doubly-typed graph grammar $\leq^N \subseteq (N \times N)$ is the reflexive and transitive closure of $\trianglelefteq^N$.*

3. *The element $e2$ is* **directly (causally) dependent** *of $e1$ written $e1 \trianglelefteq^T e2$, iff there is an action $n1 \in N$ such that $n1$ deletes $e1$ and creates $e2$.*

4. *The (causal)* ***dependency relation between types*** *of a doubly-typed graph grammar $\leq^T \subseteq (C \times C)$ is the reflexive and transitive closure of $\trianglelefteq^T$.*

5. *The (causal)* ***dependency relation*** *of a doubly-typed graph grammar $\leq \subseteq (N \times N) \cup (C \times C)$ is defined by $(\leq^N \cup \leq^T)$.*

☺

Remarks.

1. *Note that, in the application of the inverse rule there can be no conflicts between deletion and preservation because the rule is injective and the elements added by a rule are not identified in the core graph (otherwise the double-type would not be a core graph). There may be edges from the core graph that are deleted by the application of the inverse rule, although they are not specified in the right-hand side of the original rule (dangling edges). But this is not a problem because actions that need these edges will also need the action that created the corresponding vertex.*

2. *If it is clear from the context, we will usually omit the superscripts N and T from $\leq$.*

3. *We could have defined the causal dependency relation relating also rules and elements of the core graph (as in [CMR96a]), but, as this will not be needed later on, we stick to this definition (that makes is some cases the reasoning about the relationships easier).*

☺

Example **5.21** In Figure 5.8 we can see that action $a3$ is causally dependent on action $a2$ (see grammar $GG1$ in Figure 5.7). This is due to the fact that we can not prolongate the pre-condition of $a3$ to graph $H1$ (the edge can not be mapped). ☺

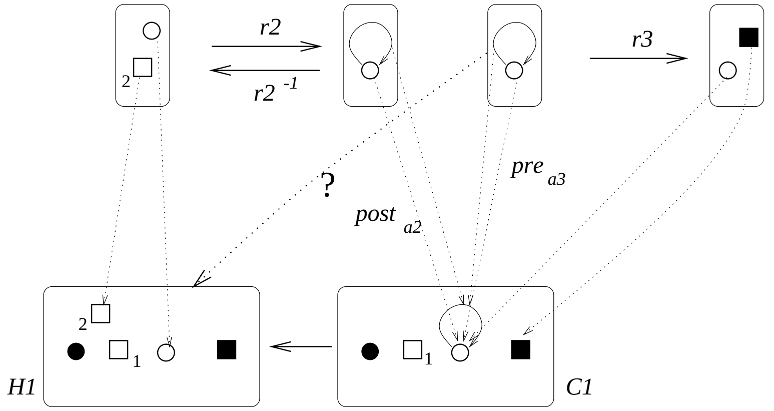


Figure 5.8: Causal Dependent Situation

The second relationship of a graph grammar that will be investigated here is the *weak conflict relation*. The intuitive idea of this relation is that

> An action a is in weak conflict with an action b if a deletes some item that is needed (preserved/deleted) by b. This implies that b can not happen if a has happened (a excludes the occurrence of b).

This relation is called a weak conflict because it only states that the occurrence of one of the actions exclude the occurrence of the other (and not vice versa, as the usual definition of a conflict). In the case that a deletes something that is also deleted by b, we will obtain that b is also in weak conflict with a, and they are thus in *(classical) conflict*: they are mutually exclusive. Asymmetric situations of weak conflict arise from the fact that items may be preserved (read-only accessed) by rules. The ability to model the preservation of items is a very important feature of graph grammars. It allows for the modeling of highly parallel systems and in particular in which not only completely disjoint actions may occur in parallel [Tae96]. Items that are preserved by some action can be considered as "read-accessed" and therefore many other actions that also read these items may occur in parallel with the first one. In the SPO approach, it is even allowed that one write-access (deletion) may occur in parallel with many read-accesses (preservation) of some item, and this allows even more parallelism within a system specified using SPO graph grammars. Although the preservation of items is a very important feature for the specification of parallel systems, it raises some situations between actions that can not be identified as conflicts in the classical sense (because they do not mutually exclude each other), are not causally dependent, but can not be observed in any order.

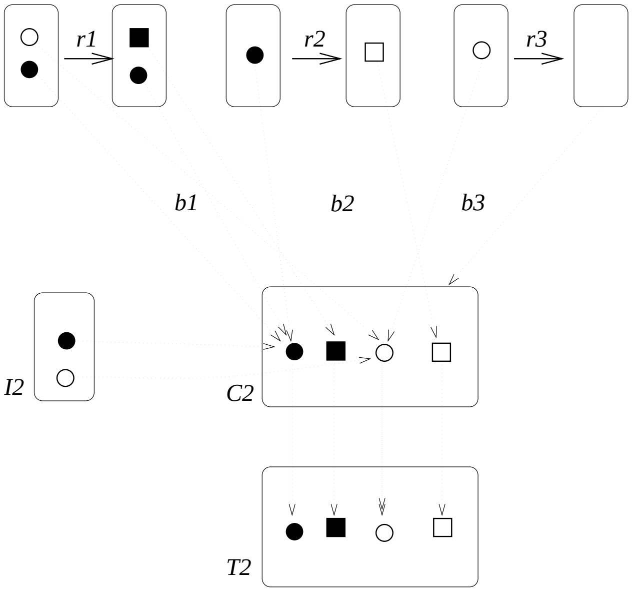


Figure 5.9: Grammar $GG2$

Example **5.22 (Weak Conflict Relation)** Consider the doubly-typed graph grammar depicted in Figure 5.9. The action $b1$ preserves the item •, that is deleted by action $b2$. Moreover, action

$b1$ does not create anything that is needed by $b2$ and vice versa,i.e., these two actions are causally independent. As they do not delete the same items, one would expect that they may occur in any order is we consider some derivation of this grammar. But this is not true because if $b2$ occurs, the • vertex is deleted, and then $b1$ can not occur anymore. On the contrary, if $b1$ occurs, $b2$ can still occur because the • vertex will still be there after the occurrence of $b1$. In such a situation we say that $b2$ in in weak conflict with $b1$, and denote this by $b1 \xrightarrow{\#} b2$ (the direction of the arrow indicates which is the possible occurrence order for these actions). Now consider actions $b1$ and $b3$. We can observe that they exclude each other because both delete the vertex ∘. Thus, the weak conflict relation between actions of this grammar contains the following elements $b1 \xrightarrow{\#} b2$, $b1 \xrightarrow{\#} b3$ and $b3 \xrightarrow{\#} b1$. ☺

The weak conflict relation is also defined between types, and the intuitive idea is that an item is in weak conflict with another one if the creation of the second item excludes the creation of the first one. In the example above, we have that $\blacksquare \leq^{\#} \square$.

The weak conflict relation between actions will be defined analogously to the causal dependency relation: based on the overlappings of actions in the core graph. But here we look for overlappings between the pre-conditions of actions.

Definition 5.23 (Weak Conflict Relation) *Let $GG = (C^T, I, N, n)$ be a doubly-typed graph grammar and C^T be a core graph. Let $n1, n2 \in N$ and $n(n1) = a1 = r1^{TG\nearrow T}$, $n(n2) = a2 = r2^{TG\nearrow T}$, $a1 \neq a2$. Let $e1, e2 \in C$, $e1 \neq e2$.*

1. *The action $n2$ is in* **weak conflict** *with $n1$, written $n1 \xrightarrow{\#}^{N} n2$ iff $r^{\bullet}_{a2} \circ pre_{a1}$ is not total, where (1) is a pushout in* **TGraphP(T)**. *Otherwise, $n2$ is* **not in direct weak conflict** *with $n1$, denoted by $n1 \not\xrightarrow{\#}{}^{N} n2$.*

$$\begin{tikzcd}
R_{a1} & L_{a1} \arrow[l, "r_{a1}"'] \arrow[dr, "pre_{a1}"'] & & L_{a2} \arrow[dl, "pre_{a2}"'] \arrow[r, "r_{a2}"] \arrow[dr, phantom, "(1)"] & R_{a2} \arrow[d, "pre^{\bullet}_{a2}"] \\
& & C^T \arrow[rr, "r^{\bullet}_{a2}"'] & & H2
\end{tikzcd}$$

2. *The element $e2$ is in* **weak conflict** *with $e1$, written $e1 \xrightarrow{\#}^{T} e2$ iff there are actions $n1$ and $n2$ that create $e1$ and $e2$, respectively, and $n1 \xrightarrow{\#}^{N} n2$.*

3. *The* **weak conflict relation** *of a doubly-typed graph grammar $\xrightarrow{\#} \subseteq (N \times N) \cup (C \times C)$ is defined by $(\xrightarrow{\#}^{N} \cup \xrightarrow{\#}^{T})$.*

☺

Example **5.24** In Figure 5.10 we can see that action $b2$ is in weak conflict with action $b1$ (see grammar $GG2$ in Figure 5.9). This is due to the fact that we can not prolongate the pre-condition of $b1$ to graph $H2$ (the •-vertex can not be mapped). ☺

As discussed before, weak conflict situations give raise to *conflict* situations when they are symmetric. Therefore we will define a conflict relation of a graph grammar based on its

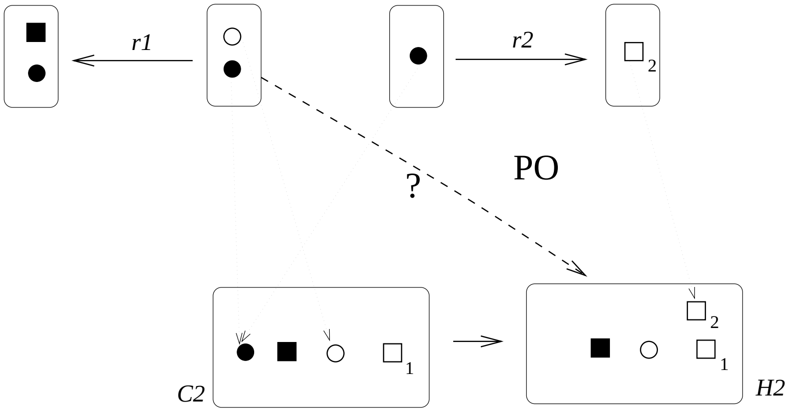


Figure 5.10: Weak Conflict Situation

weak conflict relation. Conflicts are inherited under causal dependencies: if an action a is in conflict with an action b then all actions that depend on a will also be in conflict with b because if b occurs, all of them can not occur anymore (symmetrically, all actions that depend on b will also be in conflict with a).

Definition 5.25 (Conflict Relation) *Let $GG = (C^T, I, N, n)$ be a doubly-typed graph grammar and C^T be a core graph. Let $x1, x2, x3 \in (N \times N) \cup (C \times C)$.*

1. *The **inherited weak conflict relation** of GG, denoted by $\overset{\#}{\Longrightarrow}$, is defined by $\overset{\#}{\Longrightarrow} \subseteq (N \times N) \cup (C \times C)$ such that*

$$x1 \overset{\#}{\Longrightarrow} x2 \text{ iff } \exists x3 : x1 \overset{\#}{\longrightarrow} x3 \text{ and } x3 \leq x2$$

2. *The **conflict relation** of GG, denoted by $\overset{\#}{\Longleftrightarrow}$, is defined by $\overset{\#}{\Longleftrightarrow} \subseteq (N \times N) \cup (C \times C)$ such that*

$$x1 \overset{\#}{\Longleftrightarrow} x2 \text{ iff } x1 \overset{\#}{\Longrightarrow} x2 \text{ and } x2 \overset{\#}{\Longrightarrow} x1$$

☺

Example **5.26 (Conflict Relation)** Consider the grammar $GG3$ depicted in Figure 5.11. Although it may not look like at first glimpse, actions $c1$ and $c3$ are in conflict. They do not exclude each other directly (the overlappings of their pre-conditions in the core graph is empty), but there can be no sequential derivation of this grammar in which these two actions occur. This is because is action $c1$ occurs, action $c4$ must have occurred first (it creates the ○-vertex that is needed by $c1$). But on the other hand, if $c4$ occurs, $c3$ can not occur anymore (the □-vertex needed by $c3$ is deleted by $c4$). Moreover, we observe that $c4$ is not in conflict with $c3$ because there may be a sequential derivation including both $(c2, c3, c4)$. The possible sequential derivations of this grammar will be discussed in Example 5.28. ☺

In many formalisms, for example Petri nets, transition systems and event structures, the possible orders in which a conflict-free set of actions (transitions,events) may occur in some sequential computation depend only on the causal dependency relation: any total order

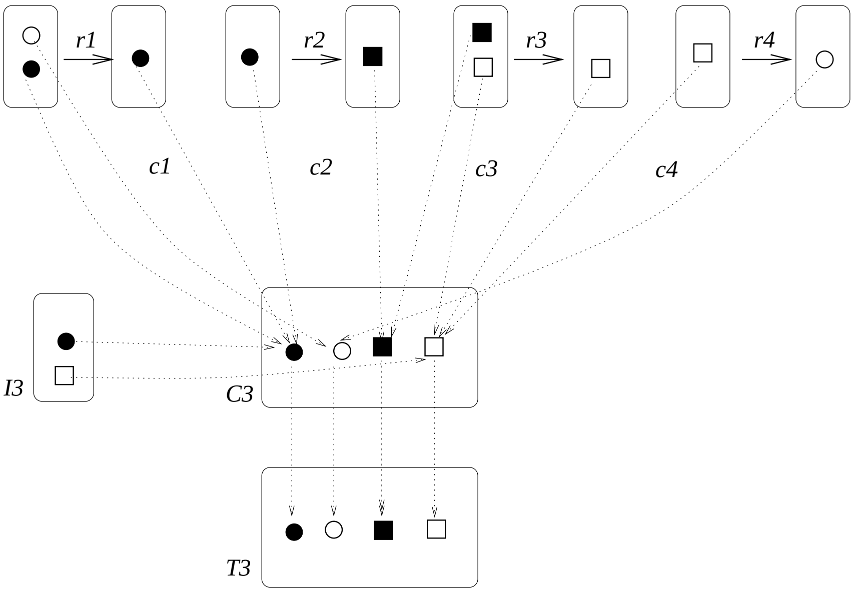


Figure 5.11: Grammar $GG3$

that is compatible with the dependency relation yields a possible sequentialization of these actions. In the case of nets with inhibitor arcs, it was noticed in [JK91] that these relations were not enough to describe suitably the computations of such a net. In graph grammars these relations are not enough due to the weak conflicts. We can observe that both relations, $\leq$ and $\xrightarrow{\#}$, impose some restrictions on the order in which actions may occur in derivation sequences. If $a \leq b$ then in any sequential derivation we must have that a occurs before b because b needs something that is created by a. If $a \xrightarrow{\#} b$ then in any sequential derivation that includes both we must have that a occurs before b because b excludes the occurrence of a. That is, in the case of graph grammars we have to take not only causal but also sequential dependencies (given by weak conflict) into account in searching for an occurrence order of actions of a grammar. Therefore the combination of these two relations gives us the possible occurrence orders of actions. This relation will be called *occurrence relation*, and the basic idea is that

> If the pair of actions (a, b) is in the occurrence relation then in any sequential derivation including both action, b occurs after a.

This relation is very important because it will be the basis for recognizing possible derivation sequences within a set of actions.

Definition 5.27 (Occurrence Relation) *Let $GG = (C^T, I, N, n)$ be a doubly-typed graph grammar, C^T be a core graph and $\leq$ and $\xrightarrow{\#}$ be the dependency and weak conflict relations of this grammar, respectively. Then the* **occurrence relation** *of GG, denoted by $\leq^{\#}$, is defined by $\leq^{\#} \subseteq (N \times N) \cup (C \times C)$ such that $\leq^{\#}$ is the reflexive and transitive closure of $(\leq \cup \xrightarrow{\#})$.*

Let $a \in N$. Then the **local occurrence relation** *with respect to the action a, denoted by $\leq^{\#}_a$, is the reflexive and transitive closure of $(\leq \cup \xrightarrow{\#})|_{Pre^{\leq}(a)}$. Analogously, for a* *set of actions* $A \subseteq N$, $\leq^{\#}_{A}$ *is the reflexive and transitive closure of* $(\leq \cup \xrightarrow{\#})|_{P}$, *where* $P = \{a' | a \in A \text{ and } a' \leq a\}$. ☺

Remark.

1. *Although weak conflict relationships are not transitive, if we have that* $a \xrightarrow{\#} b$ *and* $b \xrightarrow{\#} c$ *then* a *must occur before* c *in any sequential derivation that includes these 3 actions. Therefore, for the occurrence relations (where we are interested in possible occurrence orders of actions) it is reasonable to consider a transitive closure of the weak conflict relation.*

2. *The local occurrence relation expresses the occurrence relation between the causes of an action* a. *Note that to define this relation only the weak conflict and causal dependencies among the causes of* a *were used (this relation is not the restriction of the occurrence relation to the elements of* $Pre^{\leq}(a)$*).*

☺

Example **5.28 (Occurrence Relation)** Consider the grammar $GG1$ of Figure 5.7. The occurrence relation between actions of this grammar is (disconsidering the pair due to reflexivity): $\bullet \leq^{\#} \square$, $\square \leq^{\#} \rightarrow$, $\bullet \leq^{\#} \rightarrow$, $\rightarrow \leq^{\#} \blacksquare$, $\square \leq^{\#} \blacksquare$, $\bullet \leq^{\#} \blacksquare$, $a1 \leq^{\#} a2$, $a1 \leq^{\#} a3$, $a1 \leq^{\#} a4$, $a2 \leq^{\#} a3$, $a2 \leq^{\#} a4$ and $a3 \leq^{\#} a4$. In this case, we can find a total order that is compatible with the occurrence relation, namely $a1 < a2 < a3 < a4$. Therefore this is a possible sequential derivation of this grammar.

Now consider the grammar $GG2$ of Figure 5.9. The occurrence relation between actions of this grammar is (again without the pairs due to reflexivity): $b1 \leq^{\#} b2$, $b1 \leq^{\#} b3$ and $b3 \leq^{\#} b1$. This means that there can be no sequential derivation that contains $b1$ and $b3$ because there is no possible total order including these two actions that is compatible with $\leq^{\#}$.

Look now at grammar $GG3$ of Figure 5.11. As we have that $c4 \leq c1$, $c1 \xrightarrow{\#} c2$, $c2 \leq c3$ and $c3 \xrightarrow{\#} c4$, each action is related to the other three via the occurrence relation. This implies that there can be no total order that is compatible with the occurrence relation that includes these 4 actions (because there is a cycle). In particular, the sequences including $c1$ and $c3$ can not become derivations of the grammar because if these actions are present, the actions $c4$ and $c2$ must also belong to this derivation ($c1$ and $c3$ causally depend on these actions), and this is not possible because, as discussed above, there can be no total order of this actions that is compatible with their (causal and sequential) dependencies. The next example is to illustrate the difference between the occurrence relation and the local occurrence relation. In the grammar $GG4$ (Figure 5.12) we can observe the following (direct) relations between actions: $d1 \leq d2$, $d2 \leq d3$ and $d3 \xrightarrow{\#} d1$. If we then build the occurrence order of $GG4$ we will obtain, within others, the relationships $d1 \leq^{\#} d2$ and $d2 \leq^{\#} d1$. This hints on the fact that a sequential derivation using all actions of this grammar is not possible because there are cycles in the occurrence relation. In the local occurrence relation with respect to the action $d2$ we would obtain the $d1 \leq^{\#}_{d2} d2$, but $d2 \not\leq^{\#}_{d2} d1$. This means that there is a sequential derivation using the pre-conditions of $d2$ (and $d2$ itself). The local occurrence relation is used to find out if for each action there is at least one possible sequential derivation that including it. ☺

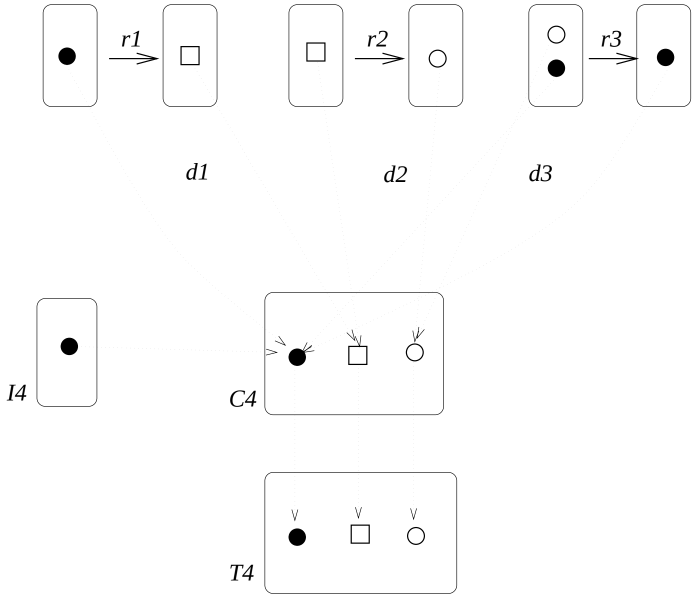


Figure 5.12: Grammar $GG4$

5.3 Occurrence Graph Grammars

The main motivation for the definition of occurrence graph grammars is to identify a class of doubly-typed graph in which each object represents a (concurrent, non-deterministic) computation of a typed graph grammar. Such a class can then be used as a semantical domain where all behaviours of typed graph grammars are represented. The advantages of such a semantical domain are twofold. On the one hand, it eases the understanding of the semantics: if a developer understands the basic operational behaviour of a graph grammar, the semantics will be also clear to him/her because it is also a graph grammar. On the other hand, theoretical investigations can be done within a uniform framework. In fact this kind of tight relationship between syntax and semantics can be also found in other formalisms. For example, in algebraic specifications [EM85] the quotient term algebra can be considered as the semantics of an algebraic specification. This algebra is a quite syntactical one obtained by generating all terms that are possible with respect to the corresponding signature and then grouping them into equivalence classes using the equations of the specification. In Petri nets [Pet62, Rei85], the semantics of a net can also be represented by a net itself (or by a set of nets), the unfolding net [WN94, MMS96]. Each place of the unfolding net correspond to a token of a reachable marking of the original net, and each transition to a possible switching of a transition of the original net.

For graph grammars, the idea we follow here is: the semantics shall describe all possible derivable graphs and all possible derivations. The semantics of a graph grammar will be a special kind of doubly-typed graph grammar, called *occurrence graph grammar*. In an occurrence graph grammar, the derivable graphs of the original grammar are represented in the double-type graph and derivations of the original grammar are described by actions (the rules of the occurrence graph grammar). As discussed in the introduction of this chapter, doubly-typed graph grammars are a good choice to obtain such semantical grammars because they are able to represent not only the occurrences of some type but also the type itself, such that the relationship to the original grammar is maintained. The advantage is that we can directly consider an occurrence graph grammar as a process of its underlying graph grammar (that can be obtained by folding the occurrence graph grammar). This approach is in accordance with the definition of graph grammar processes done in [Kre83, KW86], where a graph grammar process is a partial order of direct derivations (these are here represented by the actions of an occurrence graph grammar). Moreover, using this approach it will be possible to show that the unfolding semantics (that is an occurrence graph grammar) is compatible with parallel composition of graph grammars.

In fact, we will see that concurrent derivations, that can be used to describe the concurrent semantics of a graph grammar, are (deterministic) occurrence graph grammars. An occurrence graph grammar representing a computation of a typed graph grammar GG has two types: the type T of GG and a second type C representing the instances or *occurrences* of the items of T that appeared in the corresponding derivation(s) of GG. But this means that the second type (C) is not any type, but must consist only of items that are in the initial graph of GG or that were created in some derivation of GG. This means, C must be a *core graph* (Def. 5.18). Moreover, each action (rule) of an occurrence graph grammar shall represent a derivation step of some derivation sequence of GG. This means that each action must be applicable (and can not occur twice) in some derivation of the occurrence graph grammar. This can be assured by requiring that the causal dependency relation of an occurrence graph grammar is a partial order and that all causes of each action are not in conflict. We will formalize this idea of an occurrence graph grammar in the next definition.

Definition 5.29 (Occurrence Graph Grammar) *Let $GG = (C^T, I^{C\nearrow T}, N, n)$ be a doubly-typed graph grammar. Then GG is an* **occurrence graph grammar** *iff it satisfies the following conditions:*

1. *Acyclic local occurrence relations: $\forall a \in N : \leq^{\#}{}_a$ is antisymmetric.*
2. *No self-conflicts: $\overset{\#}{\Longleftrightarrow}$ is irreflexive.*
3. *Finite (action) causes: $\forall a \in N$: $Pre^{\leq^N}(a)$ is finite.*
4. *Core graph: C^T is a core graph (see Def. 5.18).*
5. *Deterministic action output: $\forall (nr, a), (nr, b) \in N$: $pre_a = pre_b$ and $r_a = r_b \Rightarrow post_a = post_b$.*
6. *The elements of N are of the form (x, a), where a is a doubly-typed graph morphism, $n(x, a) = a$ and for all $(x, a1), (x, a2) \in N$: $\mathcal{V}_T(a1) = \mathcal{V}_T(a2)$, where $\mathcal{V}_T$ is the type-forgetful functor (see Def. 5.9).*

☺

Remarks. *The informal interpretation of the axioms of an occurrence graph grammar are:*

1. *Acyclic local occurrence relations: This axiom assures that the local occurrence relation with respect to each action of GG is a partial order (by definition, this relation is reflexive and transitive). This means that there is a total oder that is compatible with this occurrence relation and thus this action occurs in some sequential derivation of*

this grammar. This axiom implies that the dependency relation $\leq$ is a partial order, that is, there are no dependency cycles of actions or of types (due to axiom 3., $\leq$ is well-founded).

2. *No self-conflicts: There is no action that is in conflict with itself. As the conflict relationship is inherited with respect to the dependency relationship, this also means that whenever two actions are causes of a third one, these two can not be in conflict.*
3. *Finite causes: Each action a has a finite number of causes (actions that must occur first such that action a may occur).*
4. *Core graph: Each item of C^T has exactly one origin: either it has a pre-image in the initial graph or in exactly one right-hand side of a rule of GG. This implies that if some item that was created by some action a appears in the right-hand side of another action b, it must be preserved by b. Analogously for the items that were in the initial graph. Moreover, this means that all actions must use items that either belong to the initial graph or were created by some other action(s) only using (directly or indirectly) items of the initial graph. As all rules are consuming, this implies that all the minimal elements with respect to $\leq$ belong to the initial graph.*
5. *Deterministic action output: The same rule is applied at most once at the same match. This makes sure that rule applications are deterministic with respect to representation non-determinism (see discussion in Sect. 3.3),i.e., the application of a rule to a match yields one result (and not infinitely many isomorphic results).*
6. *This item justifies the name "occurrence" graph grammar: each item (x, a) of N is an occurrence (a) of the rule x. The names of the occurrences are pairs containing the name of the corresponding rule that was used and the corresponding occurrence itself (used as an index to differentiate different occurrences of the same rule). Moreover, all occurrences having the same name of rule must use the same rule (this is modeled by the fact that, forgetting the concrete occurrence, the rule must be the same). This item is a technical convenience that will be used later on in some constructions using occurrence grammars.*

☺

Example **5.30 (Occurrence Graph Grammars)** If we consider the set of action names $N1 = \{(r1, a1), (r2, a2), (r3, a3), (r4, a4)\}$, the grammar $GG1$ of Figure 5.7 is an occurrence graph grammar. This also holds for grammars $GG2$ and $GG3$ (Figures 5.9 and 5.11). The grammar $GG4$ (Figure 5.12) is not an occurrence graph grammar because the first and second axioms are violated by action $d3$: its local occurrence relation is not antisymmetric (we have, for example, $d1 \leq^{\#}_{d3} d3$ and $d3 \leq^{\#}_{d3} d1$) and the conflict relation is no irreflexive ($d3 \overset{\#}{\Longleftrightarrow} d3$ because $d3 \xrightarrow{\#} d1$ and $d1 \leq d3$). ☺

Relationships between occurrence graph grammars will be expressed by *occurrence graph grammar morphisms.* These morphisms are doubly-typed graph grammar morphisms that preserve the special structure of the occurrence graph grammar. As each element of the core graph C^T represents an occurrence of some element of T, the morphisms between occurrence graph grammars shall map these occurrences compatibly, that is, using the retyping induced by the mapping of T. Moreover, the occurrence graph grammar morphisms shall respect the names of the actions: if some action name (n_a, a) is mapped to an action name (n_b, b), all other actions that use the rule name n_a must also be mapped to actions that use the rule name n_b. This guarantees the different occurrence of the same rule in the source occurrence grammar will be mapped to occurrences of the same rule in the target occurrence graph grammar.

Definition 5.31 (Occurrence Graph Grammar Morphism) *Let $Occ1$ and $Occ2$ be occurrence graph grammars and $f = (f_T^{OP}, f_N) : Occ1 \to Occ2$ be a doubly-typed graph grammar morphism, where $f_T = c^t : C1^{T1} \to C2^{T2}$. Then f is an **occurrence graph grammar morphism** iff the following conditions are satisfied:*

1. *$\forall (rn, a1), (rn, b1) \in dom(f_N) : f_N(rn, a1) = (rn1, a2), f_N(rn, b1) = (rn2, b2) \Rightarrow rn1 = rn2$.*
2. *Diagram (1) below is a pullback in **GraphP**(see Def. B.6 in the appendix), where $t^{dom(c)} = (t^\blacktriangledown)^{-1} \circ t^{C2} \circ c^\blacktriangledown$ (this morphism must be total because c^t weakly commutes).*

$$\begin{tikzcd}
C1 \arrow[r, mapsto, "t^{C1}"] & T1 \\
dom(c) \arrow[u, mapsto, "c!"] \arrow[r, mapsto, "t^{dom(c)}"'] \arrow[d, hook, "c^\blacktriangledown"'] \arrow[ur, phantom, "(1)"] \arrow[dr, phantom, "="] & dom(t) \arrow[u, mapsto, "t!"'] \arrow[d, hook, "t^\blacktriangledown"] \\
C2 \arrow[r, mapsto, "t^{C2}"'] & T2
\end{tikzcd}$$

The category of occurrence graph grammars and occurrence graph grammar morphisms is denoted by **OccGG**. ☺

Proposition 5.32 OccGG *is well-defined.* ☺

Proof. **OccGG** is a subcategory of **DTGG**, that is well-defined. Therefore, we just have to show that the identities of **DTGG** are occurrence graph grammar morphisms and the composition of occurrence graph grammar morphisms yields again an occurrence graph morphisms.

1. Let $id = (id_T^{OP}, id_N)$ be the identity of the occurrence grammar Occ. The first condition of occurrence graph grammar morphism is trivially satisfied by id_N. The second condition requires that diagram (1) below is a pullback, what is obviously true.

$$\begin{tikzcd}
C1 \arrow[r, mapsto, "t^{C1}"] & T1 \\
dom(id_{C1}) = C1 \arrow[u, mapsto, "id_{C1}"] \arrow[r, mapsto, "t^{C1}"'] \arrow[ur, phantom, "(1)"] & dom(id_{T1}) = T1 \arrow[u, mapsto, "id_{T1}"']
\end{tikzcd}$$

2. Let $f = (f_T^{OP}, f_N) : Occ1 \to Occ2$ and $g = (g_T^{OP}, g_N) : Occ2 \to Occ3$ be occurrence graph grammar morphisms, where $f_T = c1^{t1}$ and $g_T = c2^{t2}$. Then we have to show that

$g \circ f = ((g_T \circ f_T)^{OP}, g_N \circ f_N)$. Let $(rn, a1), (rn, a2) \in dom(g_N \circ f_N)$. Let $f_N(rn, a1) = (rn1, a2), f_N(rn, b1) = (rn2, b2)$ and $g_N(rn1, a2) = (rn3, a3), g_N(rn2, b2) = (rn4, b3)$. As f is an occurrence graph grammar morphism we conclude that $rn1 = rn2$. As g is also an occurrence graph grammar morphism we conclude that $rn3 = rn4$. Therefore, the first condition of occurrence graph grammar morphisms is satisfied. For the second condition, we have to show that diagram (2)=(3)+(7) below is a pullback, where

(7) is the square with tips in $dom(c2 \circ c1)$, $dom(c1)$, $dom(t1)$ and $dom(t2 \circ t1)$.

As f and g fulfill this condition, squares (3) and (4) are pullbacks. Moreover, squares (5) and (6) are also pullbacks (the domain of a composed function is standardly constructed as a pullback – see [Ken91]). Pullbacks (5) and (4) can be composed yielding a pullback (5)+(4). Square (7) commutes because $g_T \circ f_T$ is a morphism in **TGraphP**(this follows from the weak commutativity requirement of morphisms in this category and from the definition of $t^{dom(g_T \circ f_T)}$ – see Def. 5.31). As (5)+(4) and (6) are pullbacks and (7) commutes, decomposition of pullbacks yields that (7) is also a pullback. Therefore, (2) is obtained as a composition of pullbacks (3) and (7) and is thus also a pullback.

$$\begin{tikzcd}
 & C1 \arrow[rr, mapsto, "t^{C1}"] & & T1 & \\
 & & (3) & & \\
 & dom(c1) \arrow[uu, mapsto, "c1!"] \arrow[rr, mapsto, "t^{dom(c1)}"'] \arrow[dd, hook, "c1^{\blacktriangledown}"'] & & dom(t1) \arrow[uu, mapsto, "t1!"'] \arrow[dd, hook', "t1^{\blacktriangledown}"] & \\
 & & = & & \\
 & C2 \arrow[rr, mapsto, "t^{C2}"'] & & T2 & \\
 & & (4) & & \\
(5) & dom(c2) \arrow[uu, mapsto, "c2!"] \arrow[rr, mapsto, "t^{dom(c2)}"'] & & dom(t2) \arrow[uu, mapsto, "t2!"'] & (6) \\
dom(c2 \circ c1) \arrow[uuuuur, mapsto] \arrow[ur, hook] \arrow[rrrr, mapsto, "t^{dom(c2 \circ c1)}"'] & & & & dom(t2 \circ t1) \arrow[uuuuul, mapsto] \arrow[ul, hook']
\end{tikzcd}$$

$\surd$

The elements of the core graph of an occurrence graph grammar represent instances of the elements of the type graph that have been created by some action or have been present in the initial graph. An occurrence graph grammar shall represent a computation and the core graph shall represent all states that were reached by this computation. But as the core graph is only one graph where all these intermediate states are glued, the question arises which subgraphs of the core graph can be really found in some state. For example, if an element x was created because an element y was deleted, these two elements can never occur in the same reachable graph. The next definition gives the conditions for the subgraphs G of a core graph C such that G may occur as a subgraph of some computation. These graphs will be called *concurrent graphs* (as they are the analogous to the concurrent places of Petri nets [MMS96]). The fact that the concurrent graphs of an occurrence grammar can be reached by some derivation will be shown in Prop. 5.50.

The intuitive idea of the definition of a concurrent graph is that all its elements shall not depend on each other nor be in conflict with each other.

Definition 5.33 (Concurrent Graph) *Let $Occ = (C, I, N, n)$ be an occurrence graph grammar and G be a subgraph of C. Let $A = \{a \in N | a \in Pre^{\leq}(x) \text{ and } x \in G\}$. Then G is a* ***concurrent graph*** *iff the following conditions are satisfied for all $x, y \in G$:*

1. $x \not\leq y$ and $y \not\leq x$.
2. $x \overset{\#}{\not\Longleftrightarrow} y$.
3. $\leq^{\#}{}_A$ *is antisymmetric.*
4. $Pre^{\leq^{\#}{}_A}(x)$ *is finite.*

☺

Example **5.34 (Concurrent Graph)** Consider the occurrence grammar depicted in Figure 5.13. In this grammar, the causal dependency relation between actions $\leq^N$ is empty because all actions depend only on items that were present in the initial graph (note that the two $\circ$-vertices on the left-hand side of $r1$ are matched to the same vertex of C). The dependency relation between elements of the type graph is: $\circ_{1,2} \leq$ ■,$\circ_{1,2} \leq \rightarrow$, $\bullet_1 \leq circ_3$ and $\Box \leq \bullet_2$. The weak dependency relation is: $a1 \xrightarrow{\#} a2$, $a2 \xrightarrow{\#} a3$, $a3 \xrightarrow{\#} a1$, $\rightarrow \xrightarrow{\#} \circ_2$, ■ $\xrightarrow{\#} \circ_2$ and $\circ_2 \xrightarrow{\#} \bullet_2$. By the definition of the conflict relation based on these two relations, we obtain that there are no conflicts in this grammar, i.e., $\overset{\#}{\Longleftrightarrow} =$. Now consider the subgraph $G1$ of C. The components of this graph are not related via the causal dependency relation, and there are obviously no conflicts between them (because the grammar is conflict-free). Moreover, there is a possible sequence in which actions $a1$ and $a2$ (that are necessary to obtain $G1$) can be applied. As this set of actions $\{a1, a2\}$ is finite, we conclude that $G1$ is a concurrent graph of GG. The same holds for $G2$. Now consider the graph $G3$. As we have that $\Box \leq \bullet_2$, the first condition for concurrent graphs is violated, and thus $G3$ is not a concurrent graph. Intuitively this means that this graph can never occur in some derivation because the the creation of $\bullet_2$ implies that $\Box$ has been deleted. Although the the vertices in graph $G4$ are independent from each other with respect to $\leq$, $G4$ is also not a concurrent graph because the existence of this graph implies that the actions $a1$, $a2$ and $a3$ must have occurred, and the occurrence relation considering these three action is not antisymmetric (there is a cycle of weak conflicts). Graph $G5$ is not a concurrent graph for the same reason of $G3$: the edge $\rightarrow$ is dependent on its (source and target) vertex $\circ_{1,2}$. This means that there can be no derivation in which this edge is created. The reason why this edge is in the core graph is that the post-conditions of a rule must be total, otherwise there may be elements that have no type. Such a situation happens whenever there are conflicts between preservation and deletion (what may lead to the fact that some items that should be created by the rule are not created). The presence of this edge in the core graph is a technical convenience to make a derivation of some grammar to be again a graph grammar (where the typing is still given by total morphisms). ☺

The next proposition shows that the pre-conditions (as well as the post-conditions) of each action in an occurrence grammar are concurrent graphs. This will be used later on to show that each action in an occurrence graph grammar can really occur in some derivation (Prop. 5.51).

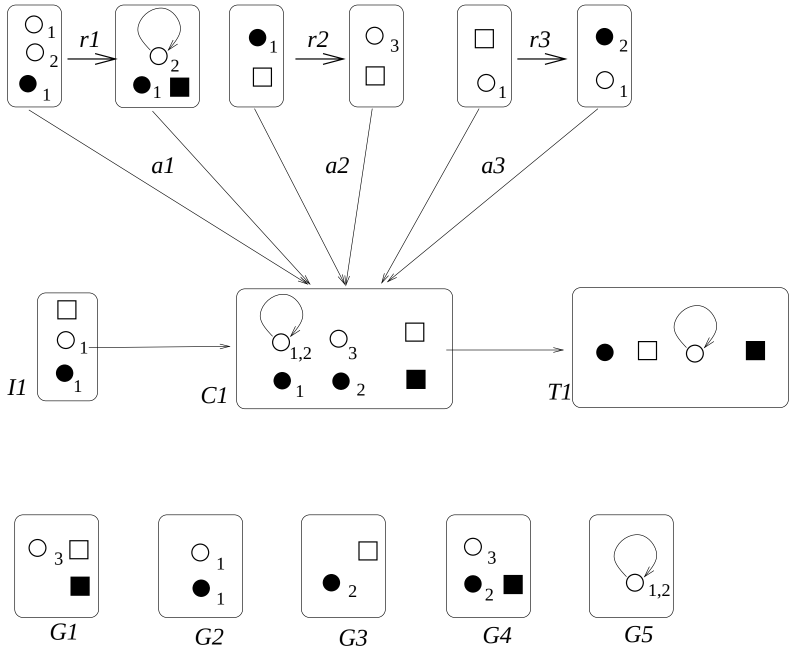


Figure 5.13: Concurrent ($G1, G2$) and Non-Concurrent Graphs ($G3, G4, G5$)

Proposition 5.35 *Let $Occ = (C, I, N, n)$ be an occurrence graph grammar and $(na, a) \in N$. Then $pre_a(L_a)$ and $post_a(R_a)$ are concurrent graphs.* ☺

Proof. We have to show that $pre_a(L_a)$ and $post_a(R_a)$ fulfill the four requirements of concurrent graphs (Def. 5.33). Assume that there are $x, y \in pre_a(L_a)$ and $x \leq y$. By definition of causal dependency (Def. 5.20), this means the creation of y depends on the deletion of x. Therefore there must be actions ax and ay that deletes x and creates y, respectively, and $ax \leq ay$. As $y \in Pre_a(L_a)$, $ay \leq a$. Thus by transitivity of $\leq$ we obtain that $ax \leq a$. But as x is needed by a and ax deletes x we must have that $a \xrightarrow{\#} ax$, and this implies that $a \overset{\#}{\Longleftrightarrow} a$ (by of the conflict relation – Def. 5.25). As Occ is an occurrence grammar, axiom 2. assures that there are no self-conflicts. Therefore, we must have $x \not\leq y$. The other three requirements can be shown analogously, using additionally axioms 1. and 3. of occurrence grammars.

√

As the relations of an occurrence graph grammar (dependency, weak conflict, conflict, occurrence) are special ones, the question arises whether they are preserved (or reflected) by occurrence graph grammar morphisms. In fact, independencies are preserved, but dependencies are not preserved. This will be illustrated by the next example.

Example **5.36 (Dependency Relations and Occurrence Graph Grammar Morphisms)** Figure 5.14 shows two actions $a1$ and $a1'$ of an occurrence graph grammar having as core graph $C1^{T1}$. Action $a1'$ is causally dependent of action $a1$ because it needs the vertex • that is created by $a1$. On the bottom of this figure we find two actions of an occurrence graph grammar $Occ2$. These actions are independent because their overlapping in the core graph $C2$ is empty. The pair

(c, t), where $c : C2 \to C1$ and $t : T2 \to T1$, is a possible type component of an occurrence graph grammar morphism: it is a morphism in **TGraphP** and domain restrictions yield a pullback (as required by axiom 2. of occurrence graph grammar morphisms). Moreover, the action component may map $a1$ to $a2$ and $a1'$ to $a2'$ because the rules of $a2$ and $a2'$ are exactly the translation of the rules of $a1$ and $a2$ with respect to the mapping of the type graphs (in fact, they could be even subrules of the corresponding translated rules). As $a1 \leq a1'$ and $a2 \not\leq a2'$, we conclude that occurrence graph grammar morphisms do not preserve the dependency relation. This occurs because the rules in the target occurrence grammar are allowed to be subrules of the original rules, and thus if the dependency between two actions was based on some items that was "forgotten" then the corresponding actions in the target occurrence grammar may be independent. Now let us consider the opposite situation: two actions $a2$ and $a2'$ are independent and are mapped via an occurrence graph grammar morphism to actions $a3$ and $a3'$ respectively. This is shown in Figure 5.15. The morphisms c and t in this figure identify both ■-vertices and both ○-vertices, and map □ to □. Although the rules in the target grammar have more elements than the original rules (because some elements of the type graph were splitted), the actions in the target grammar are also independent. The only way to make them dependent would be to force some overlapping of $post3$ and $pre3'$ in the core graph $C3$, but this would make either diagram (1) or diagram (2) non-commuting, what implies that the corresponding mapping is not a doubly-typed graph grammar morphism (because the translation of rules must be according to the core graphs – see Def. 5.5). This means that, if two actions are independent in one occurrence grammar, the corresponding actions in the image of an occurrence grammar morphism will still be independent.

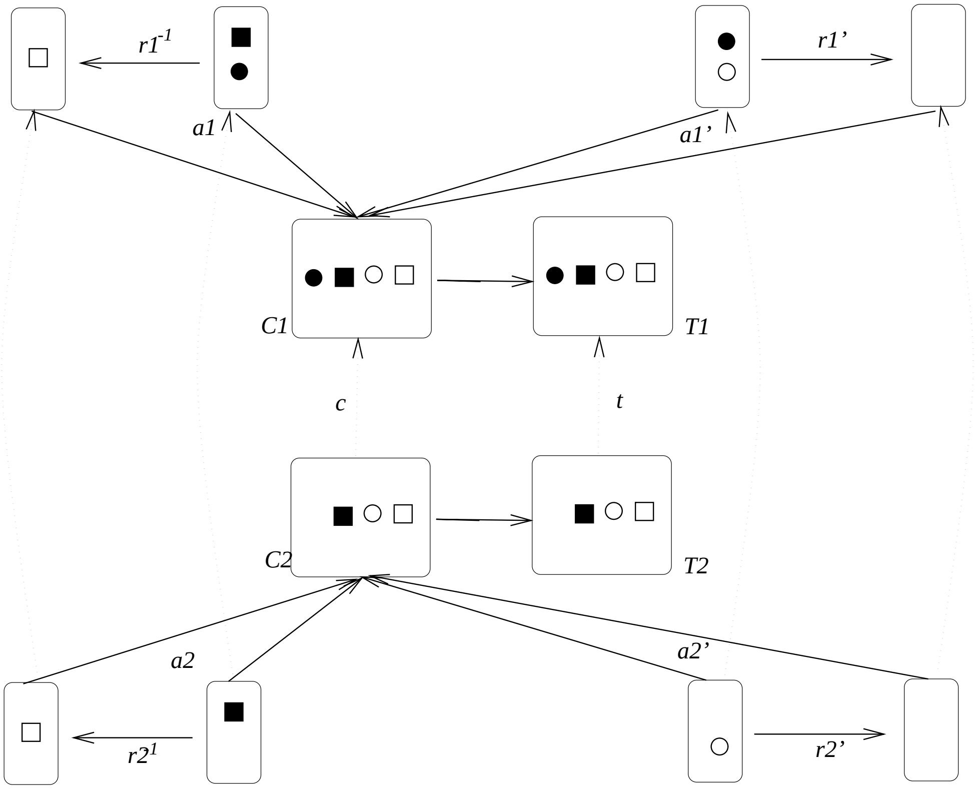


Figure 5.14: Occurrence Grammar Morphisms and Dependence

☺

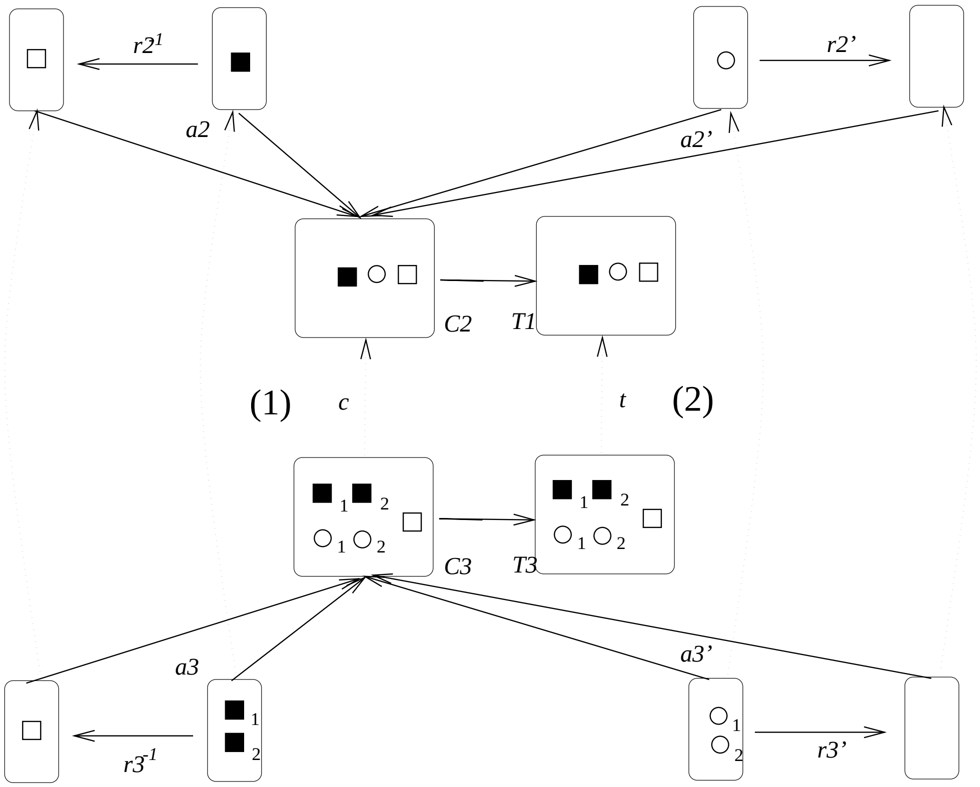


Figure 5.15: Occurrence Grammar Morphisms and Independence

Proposition 5.37 *Occurrence graph grammar morphisms preserve $\ntrianglelefteq$, $\ntrianglelefteq^{\#}$ and $\overset{\#}{\not\Leftrightarrow}$.* ☺

Proof. Let $f = (f_{TOP}, f_N) : Occ1 \to Occ2$ be an occurrence graph grammar morphism, $a1, b1 \in dom(f_N)$, $f_N(a1) = a2$ and $f_N(b1) = b2$. Let $f_T = c^t$.

Causal independence : Let $a1 \ntrianglelefteq b1$. Assume that $a2 \trianglelefteq b2$. This means that there must be $e \in C2$ such that e is created by $a2$ and needed by $b2$. Diagrams (1)–(4) commute because f is a double-typed graph grammar morphism. As (1) commutes, $e \in dom(c)$. As (2) commutes, $c(e)$ is created by $a1$. Analogously, we conclude that $c(e)$ is needed by $b1$. But this implies that $a1 \trianglelefteq b1$, what contradicts the hypothesis. Thus we conclude that $a2 \ntrianglelefteq b2$, i.e., f preserves $\ntrianglelefteq$.

$$\begin{tikzcd}
L_{a1} & R_{a1} \arrow[l, "(r_{a1})^{-1}"'] \arrow[dr, tail, "post_{a1}"] & & L_{b1} \arrow[r, "r_{b1}"] \arrow[dl, tail, "pre_{b1}"'] & R_{b1} \\
& & C1^{T1} & & \\
(2) & (1) & dom(c) \arrow[u, mapsto, "c!"] \arrow[d, hook, "c^{\blacktriangledown}"'] & (3) & (4) \\
L_{a2} \arrow[uuu, mapsto] & R_{a2} \arrow[l, "(r_{a2})^{-1}"] \arrow[uuu, mapsto] \arrow[r, tail, "post_{a2}"'] \arrow[ur, tail, "="] & C2^{T2} & L_{b2} \arrow[l, tail, "pre_{b2}"] \arrow[r, "r_{b2}"'] \arrow[uuu, mapsto] \arrow[ul, tail, "="'] & R_{b2} \arrow[uuu, mapsto]
\end{tikzcd}$$

Now let $a1 \not\leq b1$. This means that for all $c1 \in Pre^{\leq}(b1)$, $a1 \ntrianglelefteq c1$. As f preserves $\ntrianglelefteq$, $a2 \ntrianglelefteq f_N(c1)$. That is, for all pre-conditions of $b2$ that are in the image of f_N, these actions are independent from $a2$. If there is no action in $Pre^{\leq}(b1)$ that is not in the image of f_N, then we are ready. Assume that there is $c2 \in Pre^{\leq}(b1)$ and $c2 \notin rng(f_N)$. This means that $c2$ creates some item e that is (directly or indirectly) needed by $b2$. As $c2 \notin rng(f_N)$ we must have that $e \notin dom(c)$ (because f is a doubly-typed grammar morphism). But this implies that all actions that directly depend on $c2$ can not be in the image of f_N, and thus all actions that depend on these, like $b2$ can not also be in the image of f_N, what contradicts the hypothesis. Therefore we conclude that $a2 \not\leq b2$.

Occurrence Relation : Analogously to the first point, using the overlappings of the pre-conditions of actions, we obtain that $a2 \overset{\#}{\not\rightarrow} b2$ if $a1 \overset{\#}{\not\rightarrow} b1$. Now let $d1 \in dom(f_N)$ and $a1 \not\leq^{\#}_{d1} b1$. By the definition of $\leq^{\#}_{d1}$ (Def. 5.27), we have that there can be no chain of actions starting in $a1$ and ending in $b1$ in which the actions are related by causal dependency or weak conflict relationships. As $\not\leq$ and $\overset{\#}{\not\rightarrow}$ are preserved by f and f is a double-typed graph grammar morphism, we conclude that $a2 \not\leq^{\#}_{d2} b2$, where $d2 = f_{N(d1)}$.

Conflict Relation : Let $a1 \overset{\#}{\not\Leftrightarrow} b1$. This means by definition of conflict (Def 5.25) that $a1 \overset{\#}{\not\Rightarrow} b1$ or $b1 \overset{\#}{\not\Rightarrow} a1$. Assume the first case. By the definition of inherited weak conflict we have that for all $c1 \in N1$, $a1 \overset{\#}{\not\rightarrow} c1$ or $c1 \not\leq b1$. As $\overset{\#}{\not\rightarrow}$ and $\not\leq$ are preserved by f and f is a double-typed grammar morphism (what implies that all causes of an action that is in the image of a morphism are also in the image) we conclude that$a2 \overset{\#}{\not\Leftrightarrow} b2$.

$\surd$

The notion of a concurrent graph is mostly based on independence of items/actions. As morphisms preserve independence, they also preserve concurrent graphs, as the next proposition will show.

Proposition 5.38 *Occurrence graph grammar morphisms preserve concurrent graphs.* ☺

Proof. Directly follows from the definition of concurrent graphs (Def. 5.33), Prop. 5.37 and the fact that if some element is in the image of an occurrence grammar morphism, all its pre-conditions with respect to the dependency relation must also be in the image of this morphism (see Prop. 5.37). $\surd$

Although dependencies and weak conflicts are not preserved by arbitrary occurrence graph grammar morphisms, there are some that do preserve these relations. The next definition identifies a special class of occurrence graph grammar morphism, called *prefix morphisms*. If we consider occurrence graph grammars as (possibly non-deterministic) computations, prefix morphisms can be seen as prefix relations,i.e., if a morphism $p : Occ1 \to Occ2$ is a prefix morphism then the computation described by the occurrence grammar $Occ1$ is a “beginning” of the computation described by the occurrence grammar $Occ2$. Prefix morphisms enjoy a lot of properties (see Props. 5.40, 5.41, 5.42 and 5.43), and will be mainly used to relate different computations of the same typed graph grammar. Therefore, it is reasonable to require that the type graph, the initial graph and the rules that are used are the same (because they are the rules of the typed grammar that originated these computations).

Definition 5.39 (Prefix Morphisms) *Let $Occ1 = (C1^{T1}, I1^{C1 \nearrow T1}, N1, n1)$ and $Occ2 = (C2^{T2}, I2^{C2 \nearrow T2}, N2, n2)$ be two occurrence graph grammars and $p = (p_T^{OP}, p_N) : Occ1 \to Occ2$ be a morphism in* **OccGG**, *where $p_T = c^t : C2^{T2} \to C1^{T1}$. Then p is a prefix morphism iff the following conditions are satisfied*

1. $I2^{T2} = I1^{T1}$.
2. $t = id_{T1}$.
3. $\forall (rn, a1) \in N1$: $p_N(rn, a1) = (rn', a2) \Rightarrow rn = rn'$ *and* $r_{a1} = r_{a2}$.

☺

Proposition 5.40 *Let $p = (p_{T^{OP}}, p_N) : Occ1 \to Occ2$ be a prefix morphism and $p_T = c^t$. Then c is injective and surjective and f_N is total and injective.* ☺

Proof. As p is an occurrence grammar morphism, (1) is a pullback. By definition of a prefix morphism, $t = id_T$. As id_T is an isomorphism and (1) is a pullback, $c!$ is also an isomorphism. Therefore $c = c! \circ (c^{\blacktriangledown})^{-1}$ is injective and surjective. As $c!$ is surjective, the first requirement of doubly-typed graph grammar morphisms (Def. 5.15) implies that p_N must be total. Now assume that there are two actions $a1, b1 \in N1$ such that $p_N(a1) = p_N(b1) = a2$. By the third condition of prefix morphisms, the rules used in these three actions must be the same, i.e., $r_{a1} = r_{b1} = r_{a2}$. As $a1$ and $b1$ are mapped to $a2$, diagram (2) must commute with pre_{a1} and with pre_{b2}, what implies that $pre_{a1} = pre_{b1}$. Thus, using axiom 5. of occurrence graph grammars (deterministic action output), we conclude that $a1 = b1$.

$$\begin{tikzcd}
C1 \arrow[r, mapsto, "t^{C1}"] & T \\
dom(c) \arrow[u, "c!"] \arrow[r, mapsto, "t^{dom(c)}"'] \arrow[d, hook, "c^{\blacktriangledown}"'] \arrow[ur, phantom, "(1)"] & dom(t) = T \arrow[u, "id_T"'] \arrow[d, hook, "id_T"] \\
C2 \arrow[r, mapsto, "t^{C2}"'] \arrow[ur, phantom, "="] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L_{a1} = L_{b1} = L_{a2} \arrow[dl, "pre^{a2}"'] \arrow[dr, shift left, "pre^{a1}"] \arrow[dr, shift right, "pre^{b1}"'] \arrow[d, phantom, "(3)"] & \\
dom(c) \arrow[rr, hook, "c!"'] & & C2^{T2}
\end{tikzcd}$$

√

In Example 5.36 it was shown that occurrence grammar morphisms in general do not preserve the causal dependency relation. The reason was mainly that the rules in the target occurrence grammar were allowed to be subrules of the (translation of) original rules. The next proposition shows that prefix morphisms preserve not only the dependency relation, but also the local occurrence and conflict relations. Thus, the existence of a prefix morphism between two occurrence grammars expresses the existence of a very tight relationship between these two grammars.

Proposition 5.41 *Prefix morphisms preserve $\leq$, $\leq_a^{\#}$ and $\overset{\#}{\Longleftrightarrow}$.* ☺

Proof. Let $p = (p_T^{OP}, p_N) : Occ1 \to Occ2$ be a prefix morphism with $p_T = c^t : C2^{T2} \to C1^{T1}$. Then by Prop. 5.40, p^T is injective and surjective (because both components are injective and surjective) and p_N is a total and injective.

Let $a1, b1 \in N1$ and $a1 \neq b1$. As p_N is total and injective there are $a2 = p_N(a1)$ and $b2 = p_N(b1)$ and $a2 \neq b2$.

$$\begin{tikzcd}
L_{a1} \arrow[d, "y1"] \arrow[ddd, bend right=60, "id_{L1}"'] & R_{a1} \arrow[l, "(r_{a1})^{-1}"'] \arrow[dr, "post_{a1}"] \arrow[ddd, "id_{R1}"] \arrow[dl, phantom, "(1)"] & & L_{b1} \arrow[dl, "pre_{b1}"'] \arrow[ddd, bend left, "id_{L2}"] \\
H1 \arrow[d, "u"] & & C1^T \arrow[ll, "x1"'] \arrow[d, hook, "p_T^{\blacktriangledown}"'] & \\
H2 & (8)\ (7)\ (4) & C2^T \arrow[ll, "x2"] & (5) \\
L_{a2} \arrow[u, "y2"] & R_{a2} \arrow[l, "(r_{a2})^{-1}"] \arrow[ur, "post_{a2}"'] \arrow[ul, phantom, "(2)"] & & L_{b2} \arrow[ul, "pre_{b2}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
L_{a1} \arrow[d, "id_{L1}"'] & R_{a1} \arrow[l, "(r_{a1})^{-1}"'] \arrow[d, "id_{R1}"] \arrow[dl, phantom, "(3)"] \\
L_{a2} & R_{a2} \arrow[l, "(r_{a2})^{-1}"]
\end{tikzcd}$$

Dependency relation: Let $a1 \leq b1$. Assume that $b1$ is directly dependent of $a1$, i.e., $a1 \trianglelefteq^N b1$. By definition of direct dependency (Def. 5.20) this means that the morphism $x1 \circ pre_{b1}$ in the diagram above is not total, where (1) is a pushout in **TGraphP(T)**. Let (2) be the pushout of $(L_{a2} \overset{(r_{a2})^{-1}}{\leftarrow} R_{a2} \overset{post_{a2}}{\rightarrow} C2^{T2})$ in **TGraphP(T)**. We have to show that $x2 \circ pre_{b2}$ is not total. As p is a prefix morphism, there are identities id_{L1}, id_{L2} and id_{R1}. Moreover diagram (3) commutes because p $(r_{a2} = r_{a1})$ is thus trivially a pushout. As p is a doubly-typed graph grammar morphism, the translated rules are compatible with the mapping of the doubly-type, i.e., $post_{a1} \circ (id_{R2})^{-1} = post_{a1} = p_T \circ post_{a2}$ and $pre_{b1} \circ (id_{L2})^{-1} == pre_{b1} = p_T \circ pre_{b2}$. Let $p_T^{\blacktriangledown}$ be the inverse of p_T ($p^{\blacktriangledown}$ is total and injective because p_t is injective and surjective). As $p_T^{\blacktriangledown} \circ p_T = id_{C1^T}$ and p_T is surjective we obtain commutativity of (4). Analogously we obtain commutativity of (5). We can compose pushouts and (2) and (3) yielding pushout (6)=(2)+(3). As (6) and (4) commute and (1) is a pushout we obtain an universal morphism $u : H1 \to H2$ that makes (7) commute. As $x1 \circ pre_{b1}$ is not total $u \circ x1 \circ pre_{b1}$ is also not total. Therefore we obtain that $u \circ x1 \circ pre_{b1} = x2 \circ f_T^{\blacktriangledown} \circ pre_{b1} = x2 \circ pre_{b2} \circ id_{L2}$ is also not total, and as id_{L2} is an isomorphism and thus total, $x2 \circ pre_{b2}$ must be not total. This means that $a2 \trianglelefteq^N b2$.

If $b1$ is not directly dependent of $a1$ there must be a sequence $a1 \trianglelefteq^N c1 \trianglelefteq^N \ldots \trianglelefteq^N cn \trianglelefteq^N b1$, where all $ci \in N1$, for $i = 1..n$. As p_N is total, all these actions must be mapped to $Occ22$ and the direct dependencies must be preserved. Therefore by transitivity of $\leq^N$ we obtain that $a2 \leq^N b2$.

The dependency relation between types is derived from the fact that rules create/delete something. As the rules are mapped to identical ones and p_T is injective and surjective, $\leq^T$ is also preserved. Thus, we conclude that $\leq$ is preserved by p.

Weak conflict relation: Analogous to the first item, considering the morphisms pre_{a1} and pre_{b1}.

Conflict and Occurrence Relations: As these relations are obtained from the dependency and the weak conflict relations (see Defs. 5.25 and 5.27) and the latter are preserved by p, the conflict and occurrence

$\surd$

Proposition 5.42 *Prefix morphisms preserve and reflect concurrent graphs.* ☺

Proof. Preservation is due to Prop. 5.38. The fact that prefix morphisms reflect concurrent graphs follows from the fact that prefix morphisms preserve $\leq$, $\overset{\#}{\Longleftrightarrow}$ and $\leq^{\#}{}_{a}$. $\surd$

A very important property of prefix morphisms will be proven in the next proposition, namely that there can be at most one prefix morphism between two occurrence grammars. The reason for this is that each element of the core graph is uniquely determined by its history, that is, by the elements that were deleted and created by rules until this element was created. Note that, if non-consuming rules would be allowed or axiom 5. of occurrence grammars would be dropped, this proposition would not hold because there may be elements of the core graph that do not depend on the existence of any other item, and thus the histories of two of these elements would be the same, i.e., they are indistinguishable. The fact that there can be only one prefix morphism between two occurrence grammars will be used in Chap. 6 to define suitable equivalence classes of occurrence grammars, giving raise to a well-defined category of abstract occurrence grammars.

Proposition 5.43 *There is at most one prefix morphism between two occurrence grammars.* ☺

Proof. Let $Occ1$ and $Occ2$ be graph grammars as in Def. 5.39, and $f = (f_T^{OP}, f_N)$ and $g = (g_T^{OP}, g_N)$ be prefix morphisms $f, g : Occ1 \to Occ2$. We have to show that $f = g$, that is, that both components are equal:

Type component: Let $x2 \in C2^{T2}$. By axiom 4. (core graph) of occurrence graph grammars (Def 5.29) we can have two cases:

1. $x2 = in_{Occ2}(y), y \in I2^{T2}$: As f is a prefix morphism, $I1^{T1} = I2^{T2}$. Diagram (1) must commute using f_T and g_T because both are doubly-typed graph grammar morphisms (in fact, (1) must be a pullback). Let $f_T(x2) = x1$. As in_{Occ1}, in_{Occ2} and g_T are injective and (1) commutes using g_T, we conclude that $g_T(x2) = x1 = f_T(x2)$.

$$\begin{tikzcd} & I1^{T1} = I2^{T2} \arrow[dl, "in_{Occ1}"'] \arrow[dr, "in_{Occ2}"] \arrow[d, phantom, "(1)"] & \\ C1^{T1} & & C2^{T2} \arrow[ll, shift right, "f_T"'] \arrow[ll, shift left, "g_T"] \end{tikzcd} \qquad \begin{tikzcd} & R_{c1_0} = R_{c2_0} = R_{c2'_0} \arrow[dl, "post_{c1_0}"'] \arrow[dr, "post_{c2_0} = post_{c2'_0}"] \arrow[d, phantom, "(2)"] & \\ C1^{T1} & & C2^{T2} \arrow[ll, shift right, "f_T"'] \arrow[ll, shift left, "g_T"] \end{tikzcd}$$

2. $x2 = post_{a2}(y), y \in R_{a2}, y \notin rng(r_{a2})$: Here we have two cases:

   (a) $x2 \in dom(f_T)$: Let $x1 = f_T(x2)$. By axiom 4. of occurrence graph grammars, there is a unique action $a1 \in N1$ that creates $x1$. As f is a prefix morphism, requirement 1. of Def 5.39 assures that $r_{a2} = r_{a1}$. As g is also a prefix morphism it is total (Prop. 5.40). Let $(n_{a2'}, a2') = g_N(n_{[a1]}, a1)$. Again, requirement 1. of prefix morphisms assures that $r_{a2'} = r_{a1} = r_{a2}$. By axiom 1. of occurrence grammars, the local occurrence relation $\leq^{\#}_{a1}$ is a partial order. Let $\lll$ be a total oder that is compatible with $\leq^{\#}_{a1}$. This total oder is well-founded and finite due to axiom 3. of occurrence graph grammars. We will prove that $f_T(x2) = g_T(x2)$ by induction on $\lll$. The idea is that each element of a core graph is uniquely determined by its history (the initial graph and all rules that were applied to obtain this element). Therefore, if we use the same initial graph and the same rules, there can be only one way to relate the corresponding core graphs that is compatible with all actions and initial graph mappings. Remind that, if an action is mapped via an occurrence graph grammar morphism, all its pre-conditions must also be mapped (see Prop. 5.37).

   Ind. Basis : $c1_0$ is the minimal element of $\lll$:
   Let $c2_0 = f_N(c1_0)$ and $c2'_0 = g_N(c1_0)$. As $c1_0$ is the minimal element of $\lll$, it does not depend on any other actions. This means that all pre-conditions of $c1_0$ are in the initial graph. As for all elements $e \in I1^{T1} = I2^{T2}$ we have that $f_N(e) = g_N(e)$, we conclude that $pre_{c2_0} = pre_{c2'_0}$ (the commutativity requirement for morphisms requires that $f_T \circ pre_{c2_0} = pre_{c1_0} = g_T \circ pre_{c2'_0}$). Therefore axiom 5. of occurrence grammars (deterministic action output) yields that $post_{c2_0} = post_{c2'_0}$. Diagram (2) commutes with f_T and g_T because f and g are doubly-typed graph grammar morphisms. Let $x2'$ be an item created by $c2_0$ and $f_T(x2') = x1'$. As $f_T \circ post_{c2_0} = post_{a1} = g_T \circ post_{c2_0}$, we conclude that $g_T(x2') = x1'$.

   Ind. Hyp. : $c1_i$ is the i^{th} element of $\lll$ and for all e created by $R_{c2_i} = R_{c2'_i}$ we have that $f_T(e) = g_T(e)$.

   Ind. Step : Analogous to the induction basis, we obtain that $pre_{c2_{i+1}} = pre_{c2'_{i+1}}$ because all elements needed in $pre_{c1_{i+1}}$ must have been created in previous actions, and for the elements created by these actions it holds that $f_T = g_T$. Then we can use axiom 4. of occurrence grammars and obtain that for all e created by $c2_{i+1} = c2'_{i+1}$ we have that $f_N(e) = g_N(e)$.

(b) $x2 \notin dom(f_T)$: As f_T is surjective, there is no element in $C1^{T1}$ that have no pre-image in $C2^{T2}$ under f_T. Assume $x2 \in dom(g_T)$. Then there must be an action $a1 \in Occ1$ that creates $g_T(x2)$ ($g_T(x2)$ can be an element of the initial graph of $Occ1$ because this would immediately imply that $x2 \in dom(f_T)$). Analogously to the item before, we conclude that $a1$ must have an image in $Occ2$, what implies that diagram (2) (using $a1$ instead of $c1_0$) must commute with f_T and therefore $x2 \in dom(f_T)$, what is a contradiction.

Therefore, we conclude that $f_T = g_T$.

Action component: Let $(n_{a1}, a1) \in N1$. As f_N and g_N are total, there are $f_N(n_{a1}, a1) = (n_{a2}, a2)$ and $f_N(n_{a1}, a1) = (n_{a2'}, a2')$. As f and g are prefix morphisms we have that $r_{a2} = r_{a1} = r_{a2'}$ and $n_{a2} = n_{a1} = n_{a2'}$. Diagram (3) commutes with the pre-conditions of $a2$ and $a2'$, respectively, because f and g are graph grammar morphisms ($f_T = g_T$ by item 1. of this proof). As f_T is injective and surjective and $f_T \circ pre_{a2} = f_T \circ pre_{a2'} = pre_{a1}$, we conclude that $pre_{a2} = pre_{a2'}$. Then, using axiom 5. of occurrence graph grammars we obtain that $post_{a2} = post_{a2'}$, i.e., $f_N(n_{a1}, a1) = (n_{a2}, a2) = (n_{a2'}, a2') = g_N(n_{a1}, a1)$. That is, $f_N = g_N$.

$$\begin{tikzcd}
 & L_{a1} = L_{a2} = L_{a2'} \arrow[dl, |->, "pre^{a1}"'] \arrow[dr, |->, shift left, "pre^{a2}"] \arrow[dr, |->, shift right, "pre^{a2'}"'] \arrow[d, phantom, "(3)"] & \\
C1^{T1} & \, & C2^{T2} \arrow[ll, tail, two heads, "f_T = g_T"]
\end{tikzcd}$$

$\surd$

5.4 Concurrent Derivations and Occurrence Graph Grammars

In this section we will show that concurrent derivations are occurrence graph grammars. Thus, the concurrent semantics of a graph grammar can be described by its concurrent derivations, and, if the latter are doubly-typed graph grammars, they should belong to the class of "semantical grammars", that is, occurrence grammars. As one concurrent derivations is obtained from a sequential one, this concurrent derivation does not contain any conflict (because otherwise there would be two steps of the corresponding sequential derivations that are in conflict with each other, and this is not possible). But weak conflicts may occur. Therefore, concurrent derivations are special occurrence grammars. Vice versa, we may restrict to a class of occurrence graph grammars that can be considered as concurrent derivations. The main condition for these grammars is that the conflict relation is empty. Moreover, the occurrence relation of the grammar shall be a partial order. This implies that there exist a total order of actions that may be considered as a sequential derivation. The third condition is that the pre-conditions of each action with respect to the occurrence relation shall be finite. This assures that there is a sequential derivation with a countable number of steps that is represented by this occurrence grammar. Because there are no conflicts, this special class of occurrence grammars will be called *deterministic occurrence grammars*. The term deterministic shall be understood in the sense that if all actions of a deterministic occurrence grammar appear in some sequential derivation, then the result is (up to isomorphism) the same. This kind of occurrence grammars have been also defined for the DPO-approach in [CMR96a] (where only finite occurrence grammars were considered).

Definition 5.44 (Deterministic Occurrence Graph Grammars) *Let $Occ = (C^T, I^{C \nearrow T}, N, n)$ be an occurrence graph grammar. Then Occ is* ***deterministic*** *iff it satisfies the following conditions:*

1. *$\leq^{\#}$ is antisymmetric.*
2. *$\overset{\#}{\Longleftrightarrow} = \emptyset$.*
3. *$\forall a \in N$: $Pre^{\leq^{\#}}(a)$ is finite.*

☺

Remarks. *Note that these conditions imply the corresponding conditions (given by same numbers) of the definition of occurrence graph grammars (Def 5.29). If $\leq^{\#}$ is antisymmetric then all restrictions of it must be also antisymmetric, and in particular $\leq^{\#}{}_a$ must be antisymmetric, for all $a \in N$. If $\overset{\#}{\longleftrightarrow} = \emptyset$ then $\overset{\#}{\Longleftrightarrow}$ must also be the empty relation that is trivially irreflexive. If for all actions we have that $Pre^{\leq^{\#}}(a)$ is finite, we also have that $Pre^{\leq}(a)$ is finite because $Pre^{\leq}(a) \subseteq Pre^{\leq^{\#}}(a)$. These 3 conditions assure that there is a possible total order using all actions of Occ that is compatible with $\leq^{\#}$.* ☺

The next two theorems show that concurrent derivations are deterministic occurrence grammars and that concurrent derivation morphisms are prefix morphisms. These results will be used to relate the concurrency and the unfolding semantics of typed graph grammar (Theo. 6.9).

Theorem 5.45 *Concurrent derivations are deterministic occurrence grammars and vice versa.* ☺

Proof. **⇒:** This direction follows basically from the construction of a concurrent derivation (Def. 3.40). By construction, a concurrent derivation is a doubly-typed graph grammar with type C^T (the core graph is the type). Thus it remains to show that the conditions for deterministic occurrence grammars (axioms 1–3 of Def. 5.44 and axioms 4–6 of Def. 5.29) are satisfied. In [Kor96] it was shown that for each concurrent derivation there is at least one sequential derivation where all actions of the concurrent derivation are represented by derivation steps. Therefore, as it is possible to find a total order including all actions of the concurrent derivation, the weak dependency relation must be antisymmetric (axiom 2. is satisfied). Obviously, as one sequential derivation can not have conflicting derivation steps (i.e., there can not exist two steps that can not be sequentialized in any order because a sequential derivation is a total order of steps), a concurrent derivation also do not have conflicting actions. Thus, axiom 3. is satisfied. A concurrent derivation is constructed from a sequential one, where each action necessarily occurs after a finite number of actions (thus, axiom 1. is satisfied).

Axiom 4. follows from the construction of the core graph as a colimit. Axiom 5. is satisfied because all rules are consuming, and therefore by the construction of a concurrent derivation based on a colimit and on the fact that there are no conflicts, the same item of the core graph can be deleted by at most one rule. Item 6. follows from the construction of a concurrent derivation (Def. 3.40).

$\Leftarrow$: Here we start with an occurrence graph grammar $Occ = (C^T, I^{C\nearrow T}, N, n)$ and have to show that there is a sequential derivation whose concurrent derivation is Occ. To construct this sequential derivation, we take some total order $\lll$ of the actions in Occ that is compatible with $\leq^{\#}$, and start with the initial graph applying the corresponding rules in this total order (this total order exists because $\leq^{\#}$ is a partial order). Moreover, this total order is well-founded because of axiom 1. of Def. 5.44. If Occ has no actions, then the corresponding sequential derivation is the empty one. If there are actions, the corresponding sequential derivation σ will be constructed inductively as follows:

Ind. Basis: Let (n_a, a) be the minimal element of $\lll$.
Then $\sigma_1 = s1$ and $u_1 = in_{Occ}$,where $s1 = (na, S1)$, $S1$ the pushout depicted in diagram (1), $I^T = \mathcal{V}_T(I^{C\nearrow T})$ and $m_a = (in_{Occ})^{-1} \circ pre_a$. Axiom 4. of Def. 5.29 guarantees that m_a is total (action (na, a) is minimal with respect to $\leq \subseteq \lll$, and therefore can not use elements created by other actions).

$$\begin{tikzcd}
L_a \arrow[r, "r_a"] \arrow[d, "m_a"', |->] \arrow[dr, phantom, "(1)"] & R_a \arrow[d, "m_a^\bullet", |->] \\
I^T \arrow[r, "r_a^\bullet"'] & O_1^T
\end{tikzcd}$$

Ind. Hypothesis: Let i and (n_i, a_i) be the i^{th} element according to the total order $\lll$. Then there is a sequential derivation σ_i in which all actions $(n_a, a) \in N$ such that $(n_a, a) \lll (n_{ai}, ai)$ and an injective and total morphism $ui : O_i^T \rightarrow C^T$

Ind. Step: Let $j = i + 1$ and (n_{aj}, aj) be the j^{th} element according to the total order $\lll$. Then $sj = (n_{aj}, Sj)$, where Sj is the pushout depicted in diagram (2) and $m_{aj} = (u_i)^{-1} \circ pre_{aj}$. As the total order $\lll$ is compatible with $\leq^{\#}$, all actions on which action (n_{aj}, aj) depends must have occurred before. Therefore the only possibility for m_{aj} not to be total would be that some action, say ak, deleted an element that is needed by aj. This means that $aj \xrightarrow{\#} ak$, and thus $aj \leq^{\#} ak$ by the definition of $\leq^{\#}$ (Def 5.27). But this is not possible because the order $\lll$ is compatible with $\leq^{\#}$. Thus we conclude that m_{aj} is a match.

$$\begin{tikzcd}
L_{ai} \arrow[r, tail, "r_{ai}"] \arrow[d, "m_{ai}"', |->] & R_{ai} \arrow[dr] & & L_{aj} \arrow[r, tail, "r_{aj}"] \arrow[dl, "m_{aj}"', |->] \arrow[ddl, bend right, "pre_{aj}", |->] \arrow[dr, phantom, "(2)"] & R_{aj} \arrow[d, |->] \\
I_{si} \arrow[rr, tail, "r_{ai}^\bullet"'] & & O_{si} = I_{sj} \arrow[rr, tail, "r_{aj}^\bullet"'] \arrow[d, hook, "ui"'] & & O_{sj} \arrow[dll, bend left, "post_{aj}"] \\
& & C^T & &
\end{tikzcd}$$

Now we have to find an injective and total morphism $uj : O_{sj}^T \rightarrow C^T$. Consider the step-core of sj given by $core(s) = (IO_{sj}, in_{sj}, out_{sj})$ (see Def. 3.39 and diagram below). By construction, (3) and (4) are pushouts. As $r_{aj}^{C \nearrow T}$ is a morphism in $\mathbf{DTGraphP}(\mathbf{C^T})$, it must hold that $pre_{aj} \circ r_{aj}^{\blacktriangledown} = post_{aj} \circ r_{aj}!$ (weak commutativity requirement – see Def. 5.3). Therefore there must be an universal morphism $u^{LR} : LR_{sj} \to C^T$ induced by the pushout (3) such that $ou^{LR} \circ aL_{sj} = pre_{aj}$ and $u^{LR} \circ aR_{sj} = post_{aj}$. By definition of m_{aj} we have that $ui \circ m_{aj} = pre_{aj}$. Therefore there must be an universal morphism $u^{IO} : IO_{sj} \to C^T$ induced by pushout (4) such that $u^{IO} \circ m^x_{sj} = u^{LR}$ and $u^{IO} \circ in_{sj} = ui$.

$$\begin{tikzcd}
 & & dom(r_{aj}) \arrow[dl, hook, dashed, "r_{aj}^{\blacktriangledown}"'] \arrow[dr, dashed, "r_{aj}!"] & & \\
 & L_{aj} \arrow[rr, "r_{aj}"] \arrow[rr, phantom, "(3)"', shift right=2] \arrow[dr, dashed, "aL_{sj}"'] \arrow[dl, "m_{aj}"'] \arrow[ddddl, bend left=10, "pre_{aj}"] & & R_{aj} \arrow[dl, hook', dashed, "aR_{sj}"'] \arrow[dd] \arrow[ddddlll, thick, bend left=10, "post_{aj}"] & \\
 & & LR_{sj} \arrow[dd, dashed, "m^x_{sj}"] \arrow[ddll, dotted, "u^{LR}"'] \arrow[dl, phantom, "(4)"] & & \\
I_{sj} \arrow[rrr, "r^{\bullet}_{aj}"'] \arrow[drr, dashed, "in_{sj}"] \arrow[dd, hook, "ui"'] & & & O_{sj} \arrow[dl, hook', dashed, "out_{sj}"] \arrow[ddlll, hook', bend left=30, "uj"] & \\
 & & IO_{sj} \arrow[dll, dotted, "u^{IO}"'] & & \\
C^T & & & &
\end{tikzcd}$$

The morphism uj is than defined by $uj = u^{IO} \circ out_{sj}$. If both components are total and injective, uj is also total and injective. The component out_{sj} is total and injective by construction (see Def. 3.39). The morphism u^{IO} is total because all pushout morphisms and comparison morphisms used to get u^{IO} were total. To show that u^{IO} is injective, we will use the facts that in_{sj} is injective and that all elements that are created by R_{aj} are injectively included in IO_{sj}. Let $x, y \in IO_{sj}$, $x \neq y$. Assume $x, y \in rng(in_{sj})$. Then as ui and in_{sj} are injective and $in_{sj} \circ u^{IO} = ui$, we conclude that $u^{IO}(x) \neq u^{IO}(y)$. Now assume that $x, y \in rng(m^X_{sj}) - rng(in_{sj})$. In this case there are no items in L_{aj} that are pre-image for x and y. As LR_{sj} is a pushout, this means that x and y must have pre-images zx and zy in R_{aj}. Moreover, this means that rule r_{aj} creates zx and zy. Axiom 4. of occurrence graph grammars guarantees then that $post_{aj}(zx) \neq post_{aj}(zy)$, what implies that $u^{IO}(x) \neq u^{IO}(y)$. Now consider that $x \in rng(in_{sj})$ and $y \in rng(m^x_{sj}) - rng(in_{sj})$. Analogously to the previous case, we have here an element $z \in R_{aj}$ that is created by r_{aj}. Therefore, axiom 4. assures that there is no other action that creates $post_{aj}(z)$ and $post_{aj}(z)$ is not an item of the initial graph. If there is an element $k \in I_j$ such that $ui(k) = post_{aj}(z)$, then k must have been created by some action that occurred before aj, what contradicts axiom 4. Therefore we conclude that u^{IO} is injective.

The morphism uj is than defined by $uj = u^{IO} \circ out_{sj}$. As both components are total and injective, uj is also total and injective.

Now it remains to show that C^T is the colimit of the corresponding core structure diagram of σ. By construction, the morphisms ui obtained above make all diagrams of the core structure of σ commute. All these morphisms are injective and total, as the core morphisms must be. By axiom 4., there can not be any element in C^T that was not in the initial graph and is not the image of a right-hand side of a rule. Therefore, we all ui must be together surjective on C^T. Thus, C^T is the colimit of the constructed diagram.

$\surd$

To relate concurrent derivation morphisms with prefix morphisms, we have to substitute a total and injective morphism $f_C : C1^{T1} \to C2^{T2}$ from the definition of a concurrent derivation by its inverse, that is an injective and surjective one in the other direction.

Theorem 5.46 *Let* $\kappa 1 = (C1^T, I^{C1 \nearrow T}, N1, n1)$ *and* $\kappa 2 = (C2^T, I^{C2 \nearrow T}, N2, n2)$ *be concurrent derivations with respect to a grammar GG,* $f_C : C1^{T1} \to C2^{T2}$ *be a morphism in* $\mathbf{TGraphP}(\mathbf{T})$ *and* $f_N : N1 \to N2$ *be a morphism in* $\mathbf{SetP}$*. Then*

$f = (f_C, f_N)$ *is a concurrent derivation morphism* $\iff$ $p = ((f_C)^{-1}, f_N)$ *is a prefix morphism.*

☺

Proof. $\Rightarrow$: Let $f = (f_C, f_N) : \kappa 1 \to \kappa 2$ be a concurrent derivation morphism. By definition (see Def. 3.42) $f_C = c^t$ is total and injective and f_N is total. In [Kor96] it was shown that f_N is also always injective. Therefore we may define a pair $p = (p_T^{OP}, p_N)$ where $p_T = (f_C)^{-1}$ and $p_N = f_N$. As rule names are mapped via f_N to identical rule names and the corresponding rule morphism must be identical, the commutativity requirement of doubly-typed graph grammar morphisms is trivially satisfied (see Def. 4.1). As concurrent derivations have the same initial graph of their originating grammar, the second requirement for doubly-typed graph grammar morphisms is also satisfied. Thus, p is a doubly-typed graph grammar morphism. The first requirement for occurrence grammar morphisms is satisfied because of condition 2. of concurrent derivation morphisms. The second requirement is satisfied because f_T is total and injective (then $(f_T)^{-1}!$ is an isomorphism). Moreover, as f_C is a morphism in $\mathbf{TGraphP}(\mathbf{T})$, we have that $t = id_T$. As $p_T = (f_C)^{-1}$, it is injective and surjective, and as $p_N = f_N$, it is total and injective. The requirement 2. of prefix morphisms that rules to can only be mapped to rules having the same name and same rule morphism is satisfied by the requirement 2. of the definition of concurrent derivation morphisms (Def. 3.42).Thus, p is a prefix morphism.

$\Leftarrow$: Let $p = ((f_C)^{-1}, f_N) : \kappa 1 \to \kappa 2$ be a prefix morphism. We have to show that the pair $f = (f_C, f_N)$ is a concurrent derivation morphism. By Prop. 5.40, f_N is total and $(f_C)^{-1}$ is injective and surjective. Therefore, f_C is total and injective. The commutativity requirements from the definition of concurrent derivation morphisms can be derived from the commutativity requirement of the (double-)retyping construction and the requirements that the initial graphs of $\kappa 1$ and $\kappa 2$ are the same (item 1. of Def. 5.39) and that rule names and rule morphisms must be mapped to identical ones (item 3. of Def. 5.39). Therefore we conclude that f is a concurrent derivation morphism.

$\surd$

5.5 Folding of Occurrence Graph Grammars

Occurrence graph grammars shall serve as semantical models for graph grammars. Therefore we will define a relationship between occurrence grammars and typed graph grammars. The idea is that is an occurrence graph grammar Occ is related to a grammar GG, the Occ describes a (possibly non-deterministic) computation of GG. This relation will be expressed by a functor from the category of occurrence grammars **OccGG** into the category of (typed) graph grammars **GG**. This functor will be called *folding functor* because the different occurrences of elements of the same element of the type graph as well as different occurrences of the same rule will be folded together to obtain the grammar GG.

Definition 5.47 (Folding Functor) *Let* **OccGG** *and* **GG** *be the categories of occurrence and typed graph grammars respectively. Let $Occ = (C^T, I, N, n)$ be an object in* **OccGG** *and $f = (f_T^{OP}, f_N) : Occ1 \to Occ2$ be an occurrence graph grammar morphism, with $f_T = c^t$. Then the* ***folding functor*** *$\mathcal{F}$:* **OccGG** *$\to$* **GG** *is defined as:*

- **Objects:** *$\mathcal{F}(Occ) = (T, \mathcal{V}_T(I), N', n')$, where $N' = \{x | (x, a) \in N\}$, $n'(x) = \mathcal{V}_T(n(x, a))$ and $\mathcal{V}_T$:* **DTGraphP** *$\to$* **TGraphP** *is the type-forgetful functor (see Def. 5.9).*
- **Morphisms:** *$\mathcal{F}(f) = (t^{OP}, f'_N)$, where $f'_N(x) = x'$ if $f_N(x, a) = (x', a')$.*

☺

Proposition 5.48 *The folding functor is well-defined.* ☺

Proof. $\mathcal{F}(Occ)$ is obviously a graph grammar. Axiom 6. of occurrence graph grammars assure that f'_N is a well-defined function. The second requirement of occurrence graph grammar morphisms assures that the double-retyping (see Def. 5.5) of some doubly-typed graph $G1^{C1\nearrow T1}$ is compatible with the type-forgetful functor $\mathcal{V}_T$ (because the composition of pullbacks is again a pullback). That is,

$$\mathcal{DT}_{f^t}(G1^{C1\nearrow T1}) = G2^{C2\nearrow T2} \Rightarrow \mathcal{T}_f(\mathcal{V}_T(G1^{C1\nearrow T1})) \cong \mathcal{V}_T(G2^{C2\nearrow T2})$$

$$\begin{tikzcd}
G1 \arrow[r, mapsto, "t^{G1^{C1}}"] \arrow[dr, phantom, "PB"] & C1 \arrow[r, mapsto, "t^{C1}"] \arrow[dr, phantom, "PB"] & T1 \\
G2 \arrow[u, "c!^G"] \arrow[r, mapsto] & dom(c) \arrow[u, mapsto, "c!"] \arrow[r, mapsto] & dom(t) \arrow[u, mapsto, "t!"']
\end{tikzcd}$$

As both conditions of doubly-typed graph grammar morphisms are fulfilled by Occ and these conditions are based on double-retyping constructions, the corresponding conditions will be also satisfied by $\mathcal{F}(Occ)$. √

Example **5.49 (Folding Functor)** Figure 5.16 shows an occurrence graph grammar Occ and its folding $\mathcal{F}(Occ)$. Note that the two different actions $a1$ and $a2$ were folded to the same rule of $\mathcal{F}(Occ)$ because, if we forget the different matches in the core graph, the resulting rules can not be distinguished anymore, and as a grammars consists of a set of rules, they must be the same rule. The requirement 6. of occurrence graph grammars assures that actions that have the same rule name use also the same rule, and therefore there can not be the case that two rule names should be mapped to two different rules in the folded grammar. ☺

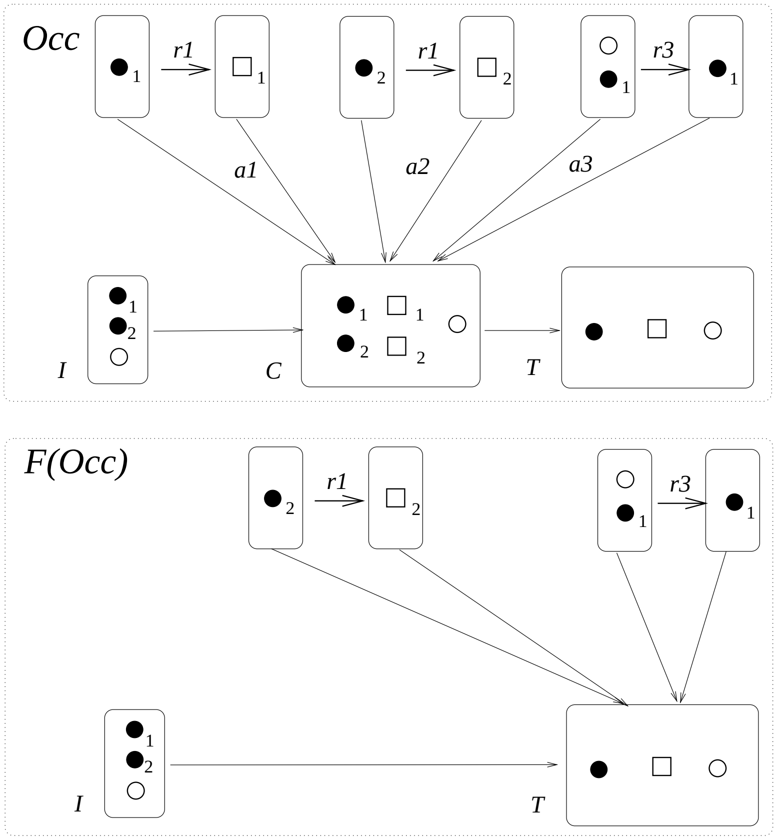


Figure 5.16: Folding of Occurrence Graph Grammars

The next two propositions confirm our choice of axioms for an occurrence graph grammar, in the sense that they show that an occurrence graph grammar represents derivations of a grammar. The first proposition is concerned about the elements of the core graph and their relationship with the derivable graphs of a grammar and the second with the actions and their relationship to the derivation steps of a grammar. Prop. 5.50 shows that for each concurrent graph of an occurrence graph grammar, there is a concurrent derivation that has this graph as an output graph with respect to the folding of this occurrence grammar. This means that each concurrent graph is (a subgraph of) some derivable graph. Prop. 5.51 shows that each action of an occurrence graph grammar can be used in at least one derivation its folded grammar.

Proposition 5.50 *Let $Occ = (C^T, I^{C\nearrow T}, N, n)$ be an occurrence graph grammar, $G \subseteq C$, and G be a concurrent graph. Then there is a concurrent derivation κ of $\mathcal{F}(Occ)$ such that $G \subseteq OUT_K$.* ☺

Proof. Let $A = \{(n_a, a) \in N \mid \exists x \in G : a'$ creates x and $(n_a, a) \leq (n_{a'}, a')\}'$. Then we define $\kappa = (C_K^T, I^{C_K \nearrow T}, N_K, n^K)$, where

- c_K^T is the smallest subgraph of C^T such that $rng(in_{Occ}) \cup rng(pre_a) \cup rng(post_a) \in C_K^T$, for all $(n_a, a) \in A$. Let $i : C_K^T \to C^T$ be the corresponding inclusion morphism.

- $I^{C_\kappa \nearrow T}$ is a doubly typed graph having as type morphism $t_\kappa^{I^T} = t^{id_T}$, where $t = (i)^{-1} \circ t_{Occ}^{I^T}$.
- $N_\kappa = \{(n_a, a') \mid (n_a, a) \in A, r_a = r_{a'}, pre_{a'} = (i)^{-1} \circ pre_a, post_{a'} = (i)^{-1} \circ post_a\}$

$$\begin{tikzcd}
L_a = L_{a'} \arrow[rr, mapsto, "r_a = r_{a'}"] \arrow[dr, mapsto, "pre_a"] \arrow[ddr, mapsto, "pre_{a'}"'] & & R_a = R_{a'} \arrow[dl, mapsto, "post_a"'] \arrow[ddl, mapsto, "post_{a'}"] \\
& C^T & \\
& C_\kappa^T \arrow[u, hook, "i"'] &
\end{tikzcd}$$

- $n^\kappa(n_a, a') = a'$, for all $(n_a, a') \in N_\kappa$

Now, if κ is a deterministic occurrence graph grammar then it is a concurrent derivation of the grammar $\mathcal{F}(\kappa) \subseteq \mathcal{F}(Occ)$.

The construction of κ yields that the initial graph and all rules are typed over C_κ^T. The typing morphisms are total due to the construction of C_κ^T. The rules are well-defined because the rules of Occ are well-defined. Thus, κ is a doubly-typed graph grammar. Conditions 4–6 of occurrence graph grammars (Def. 5.29) follow from the construction and from the fact that Occ satisfies these properties (for condition 4. we additionally need that A is closed under causal dependencies). Conditions 1–3 of deterministic occurrence grammars (Def. 5.44), that subsume the corresponding conditions of occurrence grammars, and the fact that $G \subseteq OUT_\kappa$ follow from the properties of concurrent graphs (Def. 5.33). $\surd$

Proposition 5.51 *Let $Occ = (C^T, I, N, n)$ be an occurrence graph grammar. Then for all $a \in N$ there exists at least one concurrent derivation κ with respect to $\mathcal{F}(Occ)$ such that there is a prefix morphism $p : \kappa \to Occ$ and $a \in rng(p_N)$.* ☺

Proof. By Prop.5.35 we have that $pre_a(L_a)$ is a concurrent graph. This implies that $\leq^{\#}{}_{Pre^{\leq}(a)}$ is antisymmetric, $Pre^{\leq^{\#}{}_{Pre^{\leq}(a)}}$ is finite and there are no conflicts between the elements of $Pre^{\leq}(a)$. Therefore we can build a deterministic occurrence graph grammar (concurrent derivation) κ by restricting the actions of Occ to the actions of $Pre^{\leq}(a)$ (analogous to the construction given in the proof of Prop. 5.50) and the corresponding inclusion in Occ will be then a prefix morphism. $\surd$

The following definitions and propositions of this chapter will be used as auxiliary definitions and proofs for the proofs of theorems 6.9 and 6.15 of Chap. 6. Definitions 5.52 and 5.53 and Prop. 5.54 are concerned with identifying subgrammars of an occurrence grammar. In Def. 5.55 the *maximal prefix* derivation with respect to a (special) diagram of concurrent derivations is defined. This maximal prefix is the least-upper bound of this diagram (in [Kor96] it was shown that least-upper bounds of diagrams of concurrent derivations exist).

Definition 5.52 (Depth of an Occurrence Graph Grammar) *Let $Occ = (C, I, N, n)$ be an occurrence graph grammar. Then the* **depth** *of an element $a \in N$ is given by*

$$depth(a) = \begin{cases} 0, & \text{if } a \in Min^{\leq} \\ max\{depth(b) \mid b \leq a\} + 1, & \text{otherwise} \end{cases}$$

The **depth of an occurrence graph grammar**, *denoted by $Depth(Occ)$ is defined as $Depth(Occ) = max\{depth(a) \mid a \in N\}$, if max is defined, otherwise $Depth(Occ) = \omega$.* ☺

Remarks.

1. *max delivers the maximum value of a set of values. If there is no maximum value, max is undefined.*

2. *The well-definedness of the depth of an action is due to axiom 3. of occurrence graph grammars (finite action causes).*

☺

Definition 5.53 (Subgrammar of Depth d) *Let $Occ = (C, I^C, N, n)$ be an occurrence graph grammar and $N' = \{x \in N \mid depth(x) \leq d\}$. Then its* **subgrammar of depth** *d, denoted by $Occ^{(d)}$, is defined as follows: $Occ^{(d)} = (Cd, Id^{Cd}, Nd, nd)$ where*

- *Cd is the smallest subgraph of C such that for all $(na, a) \in N'$: $(i_T)^{-1} \circ pre_a$, $(i_T)^{-1} \circ post_a$ and $(i_T)^{-1} \circ in_{Occ}$ are total, where $i_T : Cd \to C$ is the obvious inclusion.*

- *$Id^{Cd} = I^{Cd}$ with typing morphism $t^{Id} = (i_T)^{-1} \circ in_{Occ}$.*

- *$Nd = \{(na, a') \mid (na, a) \in N', r_{a'} = r_a, pre_{a'} = (i_T)^{-1} \circ pre_a, post_{a'} = (i_T)^{-1} \circ post_a\}$.*

- *$nd(na, a') = a'$, for all $(na, a') \in Nd$.*

The induced **subgrammar inclusion** *$i : Occ^{(d)} \to Occ$ is defined by $i = (i_T^{OP}, i_N)$ where i_T is the inclusion defined above and $i_N(na, a') = (na, a)$ where $r_a = r_{a'}, pre_a = i_T \circ pre_{a'}, post_a = i_T \circ post_{a'}$* ☺

Remarks.

1. *$Occ^{(d)}$ is obviously a graph grammar (all rules and the initial graph are typed over Cd). The requirements for an occurrence graph grammar are also satisfied because Occ is an occurrence graph grammar and by the construction of $Occ^{(d)}$ (see the proof of Prop. 5.50).*

2. *The inclusion i is trivially a prefix morphism.*

☺

Proposition 5.54 *Let Occ be a deterministic occurrence graph grammar. Then a subgrammar $Occ^{(d)}$ for each d is a deterministic occurrence graph grammar.* ☺

Proof. As $Occ^{(d)}$ is a subgrammar of Occ, there is a subgrammar inclusion $i : Occ^{(d)} \to Occ$, that is a prefix morphism. By Prop. 5.41, these morphisms preserve $\leq$, $\leq_a^{\#}$ and $\overset{\#}{\Longleftrightarrow}$. Therefore, $Occ^{(d)}$ must be a deterministic occurrence grammar, too. √

Definition 5.55 (Maximal Prefix) *Let GG be a graph grammar and κ be a concurrent derivation of depth $n+1$ of GG. Let D be a diagram having as objects all concurrent derivations κi of GG such that there is a prefix morphism $pi : \kappa i \to \kappa$, and as morphisms all prefix morphisms $px : \kappa i \to \kappa j$, for all $\kappa i, \kappa j \in D$. Let $\kappa^P \in D$ and $p : \kappa^P \to \kappa$ be its corresponding prefix morphism into κ. Then κ^P is called* ***maximal prefix derivation*** *of κ iff for all $\kappa i \in D$ there are prefix morphisms $pi^P : \kappa i \to \kappa^P$ such that diagram (1) below commute. The morphism p is then called* ***maximal prefix morphism****.*

$$\begin{tikzcd}
\kappa i \arrow[rr, "pi^P"] \arrow[dr, "pi"'] & {} \arrow[d, phantom, "(1)"] & \kappa^P \arrow[dl, "p"] \\
& \kappa &
\end{tikzcd}$$

☺

Proposition 5.56 *Let GG be a graph grammar and κ be a concurrent derivation of depth $n+1$ of GG. Let D be a diagram having as objects all concurrent derivations κi of GG such that there is a prefix morphism $pi : \kappa i \to \kappa$, and as morphisms all prefix morphisms $px : \kappa i \to \kappa j$, for all $\kappa i, \kappa j \in D$. Then there is a maximal prefix derivation κ^P for D and $depth(\kappa^P) = n$.* ☺

Proof. The proof for the existence of such a construction can be found in [Kor96]. The concurrent derivation κ^P is constructed as a colimit of the diagram D in the category of concurrent derivations with respect to GG. Now assume that κ^P is not of depth n. As there are prefix morphisms $pi^P : \kappa i \to \kappa^P$ and $p : \kappa^P \to \kappa$, the only possibility is that the depth of κ^P is $n+1$. This means that there is at least one action a in κ^P that depends on an action a' of depth n. As κ^P is constructed as a colimit the action a must belong to some concurrent derivation $\kappa i \in D$. But such a concurrent derivation can not be in D because this would mean that $depth(\kappa i) = n+1$. Therefore we conclude that $depth(\kappa^P) = n$. √

Proposition 5.57 *Let $p = (p_T^{OP}, p_N) : Occ1 \to Occ2$ be a maximal prefix morphism. Then $p_T \circ pre_a$ is total, for all $a \in N2$.* ☺

Proof. Let $a \in rng(p_N)$. Then the (sub-)commutativity requirement of graph grammar morphisms morphisms assures that $pre_a \in dom(p_T)$. Let $a \notin rng(p_N)$. Assume that $pre_a \notin dom(p_T)$. This means that some item needed by a is created by some action in $N2$, say a', that is not in $rng(p_N)$. As $Occ1$ is the maximal prefix derivation of depth n, we must have that $depth(a') = n+1$. But this leads to a contradiction because then we must have that $depth(a) = depth(a') + 1 = n+2$. Therefore we conclude that $pre_a \in dom(p_T)$. √

6

Unfolding Semantics of Graph Grammars

One of the main features of graph grammars is that each change a of state can be described in a very detailed way: we may have items that are deleted, items that are added and items that are preserved. In many of the other formalisms for concurrent systems, e.g., Petri nets, CCS [Mil89], transition categories [Gro96], conditional rewriting systems [Mes92], there is no difference between deletion followed by re-creation and preservation of items. Therefore, it seems to be adequate for graph grammars, even more that for other formalisms, to have these changes of state explicitely at the semantical level. This means that we do not want to "forget" all states and define a semantics for graph grammars based only on actions, like, e.g., an event structures semantics [Win89]. Obviously there may be applications in which this level of abstraction would be more adequate, and in fact it is possible to define suitable event structure semantics for graph grammars [CEL$^+$94a, Sch94, Kor95, CEL$^+$96b]. This kind of semantics gives a very nice way to reason about actions (or events) and relationships between them (dependencies and conflicts)[1]. But here we are mostly interested in defining a semantics in which it is possible, in addition to actions, to analyze the states and possible relationships between components of states. This semantics will be thus quite rich, and from it it is still possible to get the corresponding event structure of a grammar (by abstracting from the states) and also the language semantics of a grammar (by abstracting from the actions).

In the last chapter we introduced the concept of an occurrence grammar and showed that such a grammar is able to describe suitably the derivations of a graph grammar, including rules and derived graphs. In this chapter we will, starting from a graph grammar GG, construct an occurrence graph grammar that represents all its computations. Such an occurrence grammar is called *unfolding* of a grammar. In the area of Petri nets, unfolding of nets have been presented in [NPW81, MMS94, Sas94], and they are particularly suited for the investigation of reachability and deadlock properties of nets [McM92]. Moreover, another significant advantage is that the unfolding avoids (to a certain extent) the state explosion

[1] However, one should remark that the usual configurations analysis used for event structures does not reflect faithfully the kind of parallelism that may be possible in graph grammar because the ability to preserve items may give raise to asymmetric conflicts, that in turn lead to "non interleavable" parallelism (i.e., two actions that may occur in parallel can be only sequentialized in one order).

problem in constructing the semantics of a net (it was shown in [McM92] that the unfolding size grows linearly, whereas the number of states needed in the state space grows exponentially). Therefore, an unfolding semantics for graph grammars seems to be promising as a basis for analysis.

The main definition, aims and results of this chapter are:

- Definition of the unfolding of a graph grammar (Def. 6.7). This unfolding is constructed in two steps: first, the finite unfoldings are constructed inductively by applying at each step all rules of the grammar to the result of the previous step; and second, the unfolding is constructed as a colimit of the finite unfoldings in the category of graph grammars. Prop. 6.6 and Prop. 6.8 show that the unfolding of a graph grammar is an occurrence graph grammar. The unfolding construction is presented in Sect. 6.1.

- Establish a relationship between the unfolding and the concurrent semantics of a grammar (Sect 6.2). This is done by showing that the unfolding is the colimit of all concurrent derivations of a grammar (Theo. 6.9), what implies that the unfolding describes all deterministic computations of a graph grammar.

- Establishment of a tight relationship between the categories of graph grammars and of (abstract) occurrence graph grammars (Sect 6.3). This relationship is given by an adjunction where the functor involved are the folding and the unfolding functors (Theo. 6.15). The fact that the unfolding functor is a right-adjoint implies that all computations (deterministic and non-deterministic) of a grammar are represented in the unfolding of this grammar (Theo. 6.16).

- Investigation of the compatibility of the unfolding semantics with parallel composition operators on grammars defined in Chap. 4 (Sect. 6.4). It is shown in Theo. 6.18 that the unfolding semantics is compatible with parallel composition. This means that the parallel composition operators and unfolding semantics seem to be good candidates to be used as a basis for a suitable module concept for graph grammars.

6.1 Construction of the Unfolding of a Graph Grammar

The finite unfoldings of a graph grammar will be obtained by an inductive construction. Let GG be a graph grammar. To construct the unfolding of GG, we start with the empty unfolding (the unfolding of depth 0). In the next step, we have to check which rules of GG are applicable at the initial graph of GG. This search for the set of applicable rules at some unfolding step is defined in Def. 6.1. Then we have to apply this rules to the initial graph. We will apply all these rules at once, by applying a corresponding parallel rule (Def. 6.3). But, as we want to have all derived graphs represented in the unfolding, no item shall be deleted from the initial graph. This will be achieved by applying the parallel rule only from its domain (thus, considering only the items that are preserved and that are added). The next step is to “make” this result become an occurrence grammar. This means practically that we have to make sure that all rules and initial graph are typed over the same core graph. This can be done by just composing some morphisms. In the second step, the search for the applicable rules is more difficult because we have to identify subgraphs of the core graph of the first step that are concurrent, i.e., that may occur in some derivation step. It is enough to identify the subgraphs of some derivable graph because if there is a match to a subgraph of a derivable graph, it is also a match in the derivable graph. Therefore, we can use the definition of concurrent graph (Def. 5.33) to find the applicable rules. Then we apply the parallel applicable rule as in the first step, and again have to make the result an occurrence graph grammar. But at this step we have to take into account also the rules that were applied in the step before: these rules must also be mapped to the new core graph. This construction can be summarized by the following steps: The unfolding of depth d of a graph grammar GG is obtained by induction on the depth d:

Ind. Basis (Depth 0) The unfolding of depth 0 is the empty unfolding. It consists of the initial graph I^T of GG as a core graph, I^T typed with the identity morphism as initial graph, the empty set of rule names and the empty naming function.

Ind. Step (Depth $i+1$) : The unfolding of depth $i+1$ is constructed in four steps:

1. Construct the set $ApplRules^{i+1}$ of rules that are applicable to some output graph of the unfolding step i. This set consists of triples of a rule name, a rule and a corresponding match (see Def. 6.1).
2. Apply all the rules in $ApplRules^{i+1}$ to the core graph of step i in such a way that nothing is deleted, only the items created by these rules are added. This means to apply a parallel rule (see Def. 6.3 containing all the rules in $ApplRules^{i+1}$ to the corresponding parallel match, but starting not from the left-hand side but from the domain of this parallel rule (this way nothing will de deleted).
3. Retype the left-hand sides from rules in $ApplRules^{i+1}$ to the new core graph (the right-hand sides are already correctly typed by the construction of the new type graph).
4. Retype all actions that were present in unfolding i to the new core graph.

Definition 6.1 (Set of Applicable Rules) *Let Occ be an occurrence graph grammar with core graph C^T and $GG = (T, I, N, n)$ be a typed graph grammar. Then the* ***set of applicable rules*** *of GG into C^T is defined as*

$$\begin{aligned} ApplRules(Occ, GG) = \{ \quad & (nr, r^T, m^T) | nr \in N, n(nr) = r^T : L^T \to R^T, m^T = m^{OUT} : L^T \to C^T, \\ & m^{OUT} \textit{ is a match and } rng(m^{OUT}) \textit{ is a concurrent graph}\} \end{aligned}$$

☺

Definition 6.2 (Set of Applied Rules) *Let $Occ = (C^T, I, N, n)$ be an occurrence graph grammar. Then the set of applied rules of Occ is defined by*

$$Applied(Occ) = \{(n_a, r_a, pre_a) | (n_a, a) \in N\}$$

☺

Definition 6.3 (Parallel Applicable Rule) *Let $A = ApplRule(Occ, GG)$ be the set of applicable rules of GG into the core graph C^T of Occ. Let L^+ be the coproduct in* **TGraphP**(**T**) *of all left-hand sides of rules in A, and R^+* **TGraphP**(**T**) *be the coproduct of all right-hand sides of rules in A. Then the* ***parallel applicable rule*** *is a pair (r^+, m^+) where $r^+ : L^+ \to R^+$*

and $m^+ : L^+ \to C$ are the universal morphisms induced by the coproduct generating the left-hand side L^+.

$$\begin{tikzcd}
 & & Li^T \arrow[rrr, "ri^T"] \arrow[ddl, hook, "li^T"'] \arrow[dddl, "mi^T"] & & & Ri^T \arrow[ddl, hook', "si^T"'] \\
L1^T \arrow[rrr, "r1^T"] \arrow[urr, dotted, dash] \arrow[dr, hook, "l1^T"'] \arrow[ddr, bend right, "m1^T"'] & & & R1^T \arrow[urr, dotted, dash] \arrow[dr, hook, "s1^T"] & & \\
 & L^+ \arrow[rrr, "r^+"] \arrow[d, "m^+"] & & & R^+ & \\
 & C^T & & & &
\end{tikzcd}$$

☺

Definition 6.4 (Unfolding of depth d) *The (finite) unfolding $U^d(GG)$ of depth d of a (typed) graph grammar $GG = (T, I^T, N, n)$ is obtained inductively as follows:*

Ind. Basis: *Unfolding of depth 0:*

$$U^0(GG) = (I^T, I^{I \nearrow T}, \emptyset, \emptyset)$$

Ind. Hyp.: *Let $U^i = (C_i^T, IN^{C_i \nearrow T}, N_i, n^i)$ be an unfolding of depth i.*

Ind. Step: *Unfolding of depth $i+1$:*
The unfolding of depth $i+1$, $U^{i+1} = (C_{i+1}^T, IN^{C_{i+1} \nearrow T}, N_{i+1}, n^{i+1})$ is constructed in 4 steps:

1. *Construct the set of applicable rules $ApplRules^{i+1}$:*

$$ApplRules^{i+1} = ApplRules(U^i, GG) - Applied(U^i)$$

2. *Construct the new core graph C^{i+1} and inclusion c_{i+1}^T:*
Let (r^+, m^+) be the parallel applicable rule of $ApplRules^{i+1}$. Then the core graph of step $i+1$ is constructed as the pushout object of diagram (1) below (pushout in **TGraphP(T)** *– see Construction B.14), and the corresponding core graph inclusion is given by the pushout morphism $c_{i+1}^T : C_i^T \to C_{i+1}^T$.*

$$\begin{tikzcd}
L^+ \arrow[dr, "m^+"'] & dom(r^+) \arrow[l, hook', "r^{+\blacktriangledown}"'] \arrow[r, mapsto, "r^{+!}"] \arrow[d, "m'"] \arrow[dl, phantom, "="] \arrow[dr, phantom, "(1)"] & R^+ \arrow[d, "m'^{\bullet}"] \\
 & C_i^T \arrow[r, hook, "c_{i+1}^T"'] & C_{i+1}^T
\end{tikzcd}$$

3. *Add the rules in $ApplRules^{i+1}$ to $U^{i+1}(GG)$:*
For all $(nr, r^T, m^T) \in ApplRules^{i+1}$, $(nr, a) \in N_{i+1}$, where $r_a = r^T$, $pre_a =$

$c_{i+1}^T \circ m^T$, $post_a = m'^\bullet \circ s^T$ (see also Def. 6.3)

$$\begin{tikzcd}
L^T \arrow[r, "r^T"] \arrow[dd, mapsto, "m^T"'] \arrow[ddr, mapsto, "pre_a"] & R^T \arrow[dd, mapsto, "post_a"] \arrow[dr, hook, "s^T"] & \\
\arrow[r, phantom, "="] & \arrow[r, phantom, "="] & R^+ \arrow[dl, mapsto, "m'^\bullet"] \\
C_i^T \arrow[r, hook, "c_{i+1}^T"'] & C_{i+1}^T &
\end{tikzcd}$$

4. *Add the actions in N_i to $U^{i+1}(GG)$:*
   For all $(nr, b) \in N_i$, $(nr, a) \in N_{i+1}$, where $r_a = r_b$, $pre_a = c_{i+1}^T \circ pre_b$, $post_a = c_{i+1}^T \circ post_b$.

$$\begin{tikzcd}
L_a = L_b \arrow[r, "r_a = r_b"] \arrow[d, mapsto, "pre_b"'] \arrow[dr, mapsto, "pre_a" near end] & R_a = R_b \arrow[d, mapsto, "post_a"] \arrow[dl, mapsto, "post_b"' near end] \\
C_i^T \arrow[r, hook, "c_{i+1}^T"'] & C_{i+1}^T
\end{tikzcd}$$

☺

Remark.

1. *Note that between unfoldings U^i and U^{i+1} there is a corresponding inclusion (prefix morphism) induced by the inclusion of core graphs c_{i+1}^T.*

2. *As the construction of colimits and pushouts is only unique up to isomorphism, the definition above is not deterministic, but all results are isomorphic (a coproduct construction was used to get the parallel applicable rule and a pushout to obtain the core graph of a step $i + 1$).*

☺

Before we show that the unfolding is well-defined, we will give an example of its construction.

Example **6.5 (Unfolding Construction)** Consider the graph grammar GG depicted in Figure 6.1. Rule r deletes the looping edge and preserves the $\bullet_1$-vertex. Rule s deletes the edge and its source vertex, preserves the target vertex and creates a looping edge. All these vertices have the same type ($\bullet$). The indices just indicate that they are different occurrences of the same type element. By definition, the unfolding of depth 0 is an occurrence grammar consisting only of the initial graph and a core graph. This unfolding $U^0(GG)$ is shown in Figure 6.2.

All subgraphs of C^0 are trivially concurrent graphs. Rule r can not be applied at C^0 because there is no looping edge in this graph. For rule s, there are three possible matches. The set of applicable rules of step 1 is shown in Figure 6.3, together with their corresponding parallel applicable rule $s^+ : L^+ \to R^+$. The next step is to apply this parallel rule to the core graph C^0. This is also shown in Figure 6.3. Then we have to prolongate the matches of each rule to the new core graph (C^1). The result U^1 of this step is shown in Figure 6.2, where the rules are

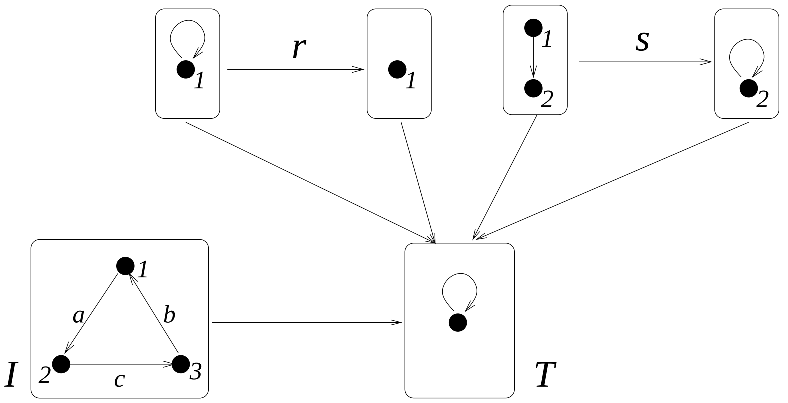


Figure 6.1: Grammar GG

not explicitely represented, just the core graph and the pre- and post-conditions of the rules. The rules are numbered for convenience (such that it is easier to talk about them). If some edge is in the range of a pre- or post-condition of a rule, the corresponding source and target vertices must obviously also be in the image of this morphism (and therefore this is not drawn).

Now let us analyze the relationships between these actions and elements of the core graph. All actions are causally independent, and the weak conflict relation is $s3 \xrightarrow{\#} s1$, $s1 \xrightarrow{\#} s2$ and $s2 \xrightarrow{\#} s3$. There are no conflicts. Between elements of the core graph, we have the following dependencies: $\bullet_1 \leq \overset{e}{\to}$, $\overset{a}{\to} \leq \overset{e}{\to}$, $\bullet_2 \leq \overset{f}{\to}$, $\overset{c}{\to} \leq \overset{f}{\to}$, $\bullet_3 \leq \overset{d}{\to}$, $\overset{b}{\to} \leq \overset{d}{\to}$. The weak conflicts between types are $\overset{d}{\to} \xrightarrow{\#} \overset{e}{\to}$, $\overset{e}{\to} \xrightarrow{\#} \overset{f}{\to}$ and $\overset{f}{\to} \xrightarrow{\#} \overset{d}{\to}$.

The graphs $G1$ to $G4$ in Figure 6.4 are concurrent graphs of C^1. The graph $G5$ is not a concurrent graph because $\bullet_3 \leq \overset{d}{\to}$. If we choose subgraph $G1$ we can apply rules r and s. The same holds for the subgraphs $G2$ and $G3$. By considering other subgraphs, we can only get rules that have already been applied. Thus we can build the parallel applicable rule using these six rules, do the retyping and obtain the unfolding of depth 2. Note that in this unfolding, the newly created edges causally depend on their vertices (due to the conflict between deletion and preservation). Therefore, these edges can not belong to any concurrent graph. From now on, there can not be any rule that is applicable that was not applied yet. Therefore, all other unfoldings of depth greater than two will yield the same result as this step. ☺

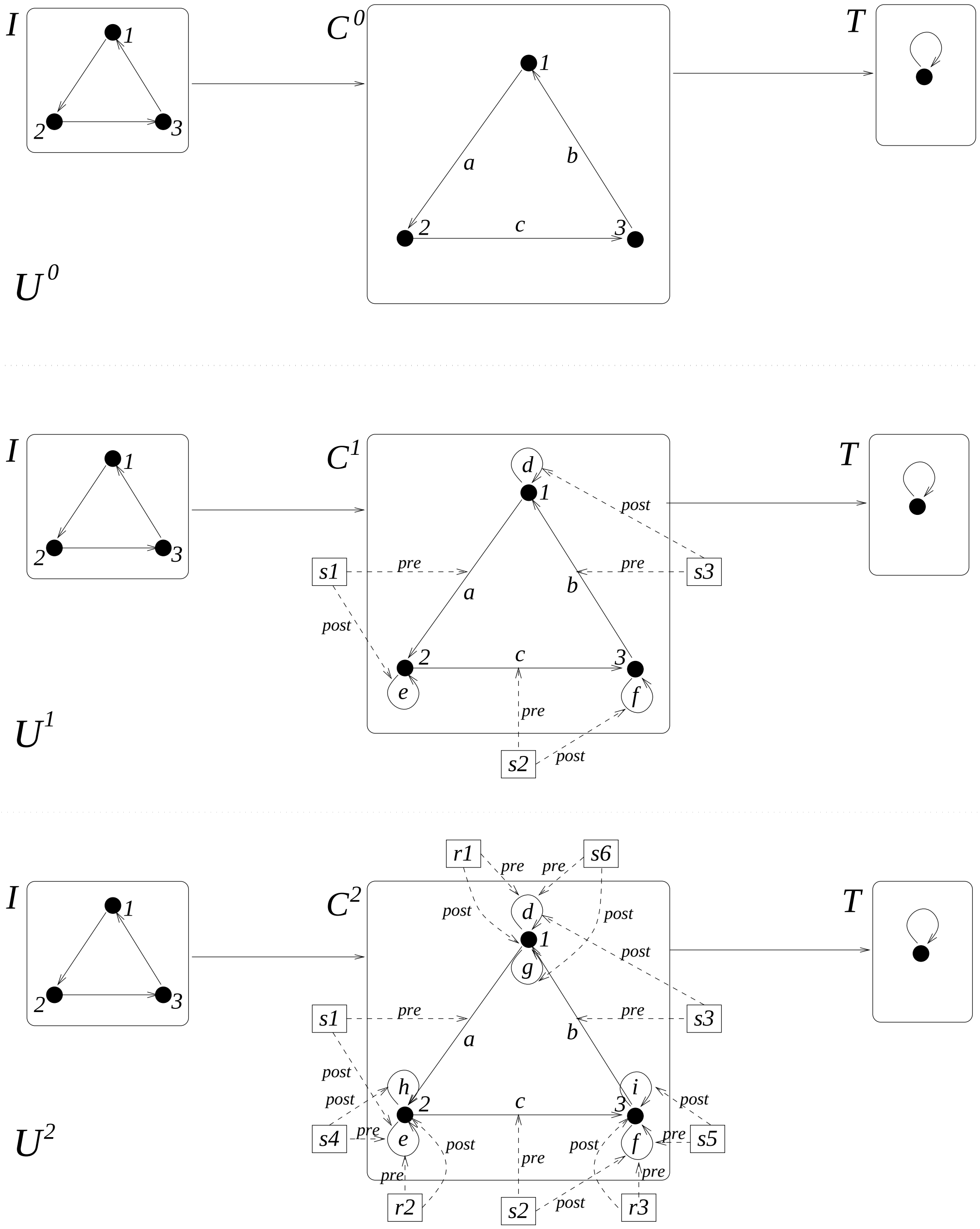


Figure 6.2: Finite Unfoldings of *GG*

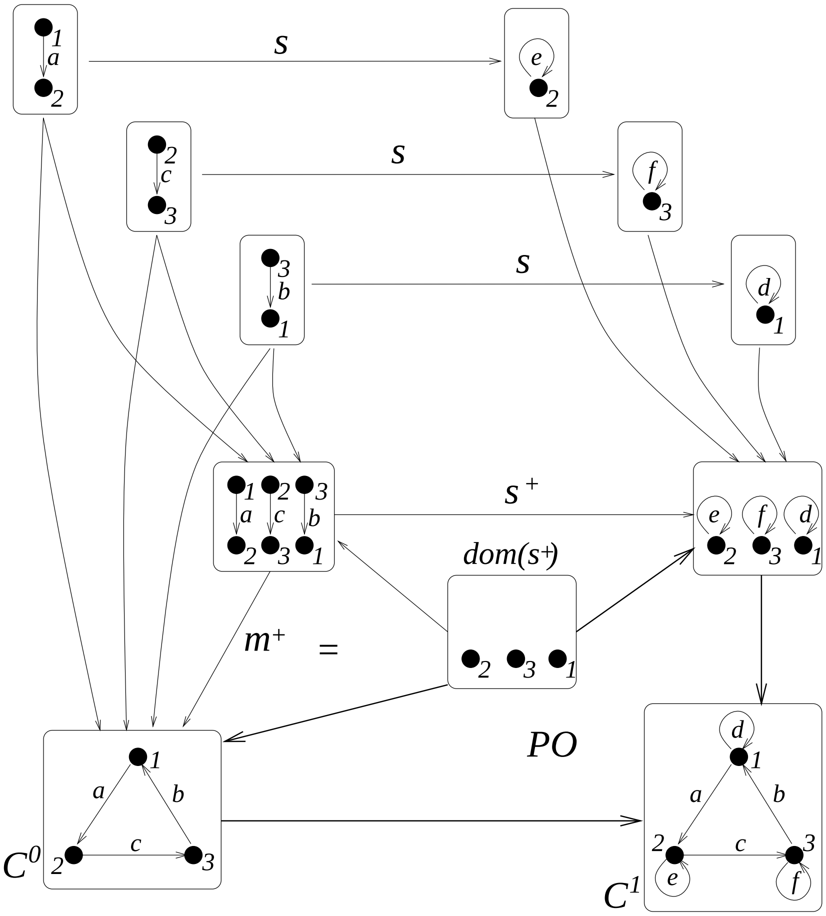


Figure 6.3: Applicable Rules and New Core Graph

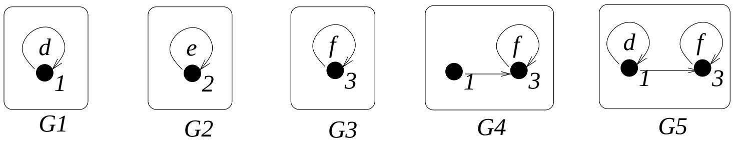


Figure 6.4: Subgraphs of C^1

Proposition 6.6 *The unfolding of depth d is well-defined and is an occurrence graph grammar, for all $d \in \mathbb{N}$.* ☺

Proof. U^d is a doubly-typed graph grammar with type C_d^T. The weak commutativity requirement for the rules is satisfied because each rule r^T of GG satisfies this condition (and therefore also r^+) and by the construction of the typing morphism *pre* and *post* (the rule r^+ is weakly commuting with type graph c_{i+1}^T by the pushout construction (1) of step 2. of the unfolding construction). Thus it remains to show that U^d satisfies the requirements of occurrence graph grammars (see Def. 5.29). This will be proven by induction on the depth d.

Ind. Basis (Depth 0) : In this case $U^0(GG) = (I^T, I^{I\nearrow T}, \emptyset, \emptyset)$ by definition. As there are no actions, the relations $\leq$, $\leq^{\#}$ and $\overset{\#}{\Longleftrightarrow}$ are empty, what implies that axioms 1., 2., 3., 5. and 6. are trivially satisfied. As $in_{U^0(GG)} = id_{I^T}$, axiom 5. is also satisfied.

Ind. Hyp. (Depth i) : The unfolding of depth i is well-defined and an occurrence grammar.

Ind. Step (Depth $i+1$) : Let $p = (p_T^{OP}, p_N) : U^i(GG) \to U^{i+1}(GG)$ be the prefix morphism induced by the core graph inclusion of step $i+1$.

1. Acyclic local occurrence relations: For each $a \in N_{i+1}$, the relation $\leq_a^{\#}$ is antisymmetric because by definition of unfolding, $pre_a(L_a)$ is a concurrent graph and by axiom 3. of concurrent graphs.

2. No sefl-conflicts: Assume that there is $a \in N_{i+1}$ such that $a \overset{\#}{\Longleftrightarrow} a$. By definition of the conflict relations (Def. 5.25) this means that $a \overset{\#}{\Longrightarrow} a$. i.e., there is an action $b \in N_{i+1}$ such that $a \overset{\#}{\longrightarrow} b$ and $b \leq a$ (Def. 5.25 of inherited weak conflict relation). As $a \overset{\#}{\longrightarrow} b$, b deletes some item, say x, that is needed by a, and as $b \leq a$, b creates some item, say y, that is needed by a. This imples, by Def. 5.20 of dependency relation, that $x \leq y$. Therefore, as $x \in pre_a(L_a)$ and $b \leq a$ (i.e., $b \in Pre^{\leq}a)$, $pre_a(L_a)$ can not be a concurrent graph because condition 1. of concurrent graphs (Def. 5.33) is violated. But by the construction of the unfolding, $pre_a(L_a)$ is a concurrent graph (see step 1. of the construction). Therefore we conclude that $a \overset{\#}{\not\longrightarrow} b$, and this implies that $a \overset{\#}{\not\Longrightarrow} a$ and that $a \overset{\#}{\not\Longleftrightarrow} a$.

3. Finite causes: Assume that there is $a \in N_{i+1}$ such that $Pre^{\leq}(a)$ is infinite. Then either there is an infinite sequence of causally dependent actions that are causes of a or a depends on infinitely many actions that are that are independent from each other. Assume that there is an infinite decreasing sequence of actions $s = \ldots \trianglelefteq a2 \trianglelefteq a1 \trianglelefteq a$. As $a1 \trianglelefteq a$, $a1$ must belong to $rng(p_N)$. As $U^i(GG)$ is an occurrence graph grammar by hypothesis, p is a prefix morphism and prefix morphisms preserv causal dependencies (Prop. 5.41), $Pre^{\leq}(a1))$ is finite. As prefix morphisms reflect $\leq$ (Prop. 5.41), s must also be finite. Now assume that a depends on infinitely many actions that are independent. Let $S = \{a' \in N_{i+1} \mid a' \trianglelefteq a\}$. S is infinite by assumption, thus $Pre^{\leq|_S}(a)$ is infinite. This means that there are infinitely many actions that create items that are needed by a. But by the definition of the applicable rules (Def. 6.1), $pre_a(L_a)$ can not be a possible match in this case

because $pre_a(L_a)$ is not a concurrent graph (it violates axiom 4. of Def. 5.33). Therefore we conclude that $Pre^{\leq}(a)$ is finite.

4. Core graph: The fact that each item of C^T_{i+1} is created only once and that there are no items that are not in the image of some right-hand side of rule or initial graph is assured by the pushout construction of step 2. of the construction of the unfolding and by the fact that U^i satisfies this condition.
5. The construction of $ApplRuled^{i+1}$ in step 1. of the unfolding as a set assures that the same triple consisting of a rule name, a rule and a match are not selected twice to be applied in the same unfolding step. Moreover, all triples of already applied rules/matches are not in this set, what implies that the same triple can never be applied twice in two different unfolding steps.
6. The last axiom is satisfied by the construction of the unfolding and by the fact that each rule name in GG is associated to only one rule pattern.

√

Definition 6.7 (Unfolding Semantics of a Graph Grammar) *The (infinite) unfolding $Unf(GG)$ of a graph grammar GG is obtained as the colimit in* **OccGG** *of the diagram consisting of all finite unfoldings of GG and the corresponding inclusions (prefix morphisms).*

*The **unfolding semantics** of a graph grammar GG is its unfolding $Unf(GG)$.* ☺

Proposition 6.8 *The unfolding semantics of a graph grammar is well-defined.* ☺

Proof. We have to prove that the colimit of the diagram of finite unfoldings and prefix morphisms exists in the category **OccGG**. The following construction of this colimit is analogous to the construction of colimits of directed diagrams of concurrent derivations presented in [Kor96]. The colimit object $U = (C^T_U, I^{C_U \nearrow T}, N_U, n^U)$ and the colimit morphisms $qi = (qi^{OP}_T, qi_N) : U^i \to U$, for each finite unfolding U^i of GG, are constructed as follows:

- C^T_U is the colimit in **TGraphP(T)** of the diagram $C^T_0 \overset{c1^T}{\hookrightarrow} C^T_1 \overset{c2^T}{\hookrightarrow} C^T_2 \cdots$ of all core graphs and core graph inclusions of the unfoldings of depth $i \in \mathbb{N}$. This colimit exists and as all morphisms in this diagram are total and injective, the colimit morphisms $di^T : C^T_j \to C^T_U$ are also total and injective (see [Kor96] for a proof).
- $I^{C_U \nearrow T} = (I^T, t^{I^T}, C^T_U)$, where $t^{I^T} = d^T_0$
- N_U: Let S be the colimit object in **SetP** of the diagram $N_1 \overset{p2^N}{\hookrightarrow} N_2 \overset{p3^N}{\hookrightarrow} N_3 \cdots$ of all sets of action names and action names components of the prefix morphisms $pi = (pi^{OP}_T, pi_N) : U^{i-1} \to U^i$ induced by the unfolding construction. Let $si : N_i \to S$ be the colimit morphisms. Then $N_U = \{(n_a, a) \mid \exists x = si(n_a, a') \in S, r_a = r_{a'}, pre_a = di^T \circ pre_{a'}, post_a = di^T \circ post_{a'}\}$.
- $n^U(n_a, a) = a$, for all $(n_a, a) \in N_U$
- $qi_T = (di^T)^{-1}$ and $qi_N(n_a, a') = (n_a, a)$, where $r_a = r_{a'}$, $pre_a = di^T \circ pre_{a'}$, $post_a = di^T \circ post_{a'}$

Note that N_U is isomorphic to S (and thus also a colimit of the diagram of action names) because for each element $x \in S$, all $si(n_a, a') = x$ use the same rule and the pre- and postconditions of a' have the same image in the core graph C_U^T (all pi are prefix morphisms).

U is obviously a doubly-typed graph grammar with double-type C_U^T and the conditions of occurrence graph grammars are satisfied because all U^i are occurrence graph grammars and the grammar U is just the gluing of all these grammars (where neither actions nor elements of the core graphs are glued together from some U^i into U). Moreover, it is easy to check that the pairs qi are prefix morphisms.

The commutativity requirement and universal property of colimits follows from the construction of U and all qi based on the colimits of their core graph and set of rule names components. $\surd$

6.2 Unfoldings and Concurrent Semantics

The next theorem establishes a connection between the unfolding and the concurrent semantics of a graph grammar. This relationship is

> The unfolding semantics $Unf(GG)$ represents exactly the "gluing" of all concurrent derivations of a graph grammar GG along the concurrent derivation morphisms.

This gluing construction as a definition for the unfolding of a graph grammar was introduced in [KR95], where non-deterministic processes for graph grammars were investigated. Now we prove that the constructive way of obtaining the unfolding given in Defs. 6.4 and 6.7 and the "categorical" way via a colimit construction of a diagram of concurrent derivations yield the same result. As the concurrent derivations represent all sequential derivations of a grammar (see [Kor96] for a proof), the next theorem implies that the unfolding semantics describes also exactly all sequential derivations of a grammar.

Theorem 6.9 *Let GG be a graph grammar and $Unf(GG)$ be its unfolding. Then $Unf(GG)$ is isomorphic to the colimit of the diagram consisting of all concurrent derivation in $CDer_{GG}$ and all prefix morphisms between them.* ☺

Proof. The proof will be done for all finite unfoldings and concurrent derivations by induction on their depth. Then as the unfolding is the colimit of all finite unfoldings and infinite derivations are also colimits of their finite prefixes, we conclude that $Unf(GG)$ is the desired colimit.

Ind. Basis (Depth 0) : The unfolding of depth 0 is defined as $U^0(GG) = (I^T, I^{I\nearrow T}, \emptyset, \emptyset)$. All concurrent derivations of depth 0 have also an isomorphism as the inclusion of the initial graph into the core graph and an empty set of actions. All prefix morphisms between these concurrent derivations are isomorphisms. Therefore we conclude that $U^0(GG)$ is the desired colimit.

Ind. Hyp. (Depth d) : The unfolding $U^d(GG) = (C_d^T, IN^{C_d\nearrow T}, N_d, n^d)$ is the colimit of the diagram $\mathtt{D}^d$ consisting of all concurrent derivations of depth $j \leq d$ and all prefix morphisms between them, where the colimit morphisms are $c_d : \kappa \rightarrow U^d(GG)$ for each $\kappa \in \mathtt{D}^d$. Moreover, the colimit morphisms are prefix morphisms.

Ind. Step (Depth d+1) : Let $U^{d+1}(GG) = (C^T_{d+1}, IN^{C_{d+1} \nearrow T}, N_{d+1}, n_{d+1})$ be the unfolding of GG of depth $d+1$ and $i : U^d(GG) \to U^{d+1}(GG)$ be the corresponding inclusion. Let $\mathtt{D}^{d+1}$ be the diagram of all concurrent derivations of depth $j \leq d+1$ and all prefix morphisms between them. We have to find, for each $\kappa j_{d+1} \in \mathtt{D}^{d+1}$, a prefix morphism $cj_{d+1} : \kappa j_{d+1} \to U^{d+1}(GG)$ that commutes with all morphisms in $\mathtt{D}^{d+1}$, and show that $U^{d+1}(GG)$ has the universal property of colimits.

$$\begin{tikzcd}
\kappa j_d \arrow[r, hook, "cj_d"] \arrow[d, hook, "pj"'] \arrow[dr, phantom, "(1)"] & U^d(GG) \arrow[d, hook, "i"] \\
\kappa j_{d+1} \arrow[r, hook, dashed, "cj_{d+1}"'] & U^{d+1}(GG)
\end{tikzcd}$$

Obviously we have that $\mathtt{D}^d \subseteq \mathtt{D}^{d+1}$. Let $\kappa j_{d+1} \in \mathtt{D}^{d+1}$ and $depth(\kappa j_{d+1}) < d+1$. Then $\kappa j_{d+1} \in \mathtt{D}^d$. In this case we define the colimit morphism $cj_{d+1} : \kappa j_{d+1} \to U^{d+1}(GG)$ by $cj_{d+1} = i \circ cj_d$. Now let $\kappa j_{d+1} \in \mathtt{D}^{d+1} - \mathtt{D}^d$. Obviously we have that $depth(\kappa j_{d+1}) = d+1$. Let $\mathtt{D}^d_{\kappa j_{d+1}}$ be the diagram consisting of all concurrent derivations κ'_d in $\mathtt{D}^d$ such that there are prefix morphisms $pi : \kappa'_d \to \kappa j_{d+1}$. This diagram must contain at least one concurrent derivation with depth d because $depth(\kappa j_{d+1}) = d+1$. Let κj_d be the maximal prefix derivation of depth d of the diagram consisting of $\mathtt{D}^d)_{\kappa j_{d+1}}$, κj_{d+1} and all prefix morphisms between them, and $pj : \kappa j_d \to \kappa j_{d+1}$ be the corresponding maximal prefix morphism (see Def. 5.55). As $depth(\kappa j_d) = d$ there is a colimit morphism $cj_d = (c1^{OP}_T, c1_N) : \kappa j_d \to U^d(GG)$. Then the morphism $cj_{d+1} = (c2^{OP}_T, c2_N) : cdj_{d+1} \to U^{d+1}(GG)$ is constructed as follows:

- Action component: $\forall (n_a, a) \in N_{d+1}$:

$$c2_N(n_a, a) = \begin{cases} i_N \circ c1_N(n_a, a'), \text{ if } = pj_N(n_a, a') = (n_a, a \\ (n_a, a''), \text{ otherwise} \end{cases}$$

where $r_{a''} = r_a$, $pre_{a''} = (i_T)^{-1} \circ (c1_T)^{-1} \circ pj_T \circ pre_a$.

$$\begin{tikzcd}
N\kappa j_d \arrow[r, hook, "c1_N"] \arrow[d, hook, "pj_N"'] \arrow[dr, phantom, "="] & N_d \arrow[d, hook, "i_N"] \\
N\kappa j_{d+1} \arrow[r, hook, dashed, "c2_N"'] & N_{d+1}
\end{tikzcd}
\qquad
\begin{tikzcd}
C\kappa j_d & & C_d \arrow[ll, "c1_T"'] \\
C\kappa j_{d+1} \arrow[u, "pj_T"] \arrow[urr, phantom, "="] & & C_{d+1} \arrow[u, "i_T"'] \arrow[ll, dashed, "c2_T"] \\
& L_a \arrow[ul, "pre_a"] \arrow[ur, "pre_{a''}"'] \arrow[u, phantom, "="] &
\end{tikzcd}$$

The pre-condition $pre_{a''}$ must be total because κj_d is the maximal prefix derivation of depth d (see Prop. 5.57). Moreover, $pre_{a''}(L_a)$ is a concurrent graph because $pre_a(L_a)$ is a concurrent graph and prefix morphisms preserve and reflect concurrent graphs (Prop. 5.35). Therefore the triple $(n_{a''}, r_{a''}, pre_{a''})$ is a possible applicable rule and must be in $ApplRules^{d+1}$, what implies that the corresponding action must be in N_{d+1}. As $U^{d+1}(GG)$ is an occurrence graph grammar, axiom 5. of Def. 5.29 assures that there is only one possibility of mapping a'' to an action of N_{d+1} ($post_{a''}$ is uniquely determined by $r_{a''}$ and $pre_{a''}$).

- Type component: $\forall x \in C_{d+1}$:

$$c2_T(x) = \begin{cases} c1_T \circ i_T(x'), \text{ if } = pj_T(x') = x \\ x'', \text{ otherwise} \end{cases}$$

where x is created by $(n_{a''}, a'')$, $post_{a''}(y) = x$, $c2_N(n_a, a) = (n_{a''}, a'')$ and $post_a(y) = x''$. (Remind that each element of a core graph that is not in the initial graph is created by only one rule.)

As each of the rules of $N\kappa j_{d+1}$ is mapped to a rule with identical pattern, all rules are mapped and the initial graphs are equal, cj_{d+1} is a graph grammar morphism. It is injective because it is defined as a composition of injective morphisms on the items that are in the image of pj and because $post_{a''}$ is injective on the created items. For analogous reasons, $c2_T$ is surjective. The type component of the morphism $c2_T$ is the identity of T because all involved morphisms have id_T as the corresponding component. Therefore requirement 2. of occurrence graph grammars is satisfied. The first requirement is satisfied because pj, cj_d and i are occurrence graph grammars morphisms. Moreover, as these are prefix morphisms, the construction of cj_{d+1} assures that cj_{d+1} is a prefix morphism.

$$\begin{tikzcd}
 & U^d(GG) \arrow[d, hook, "i"'] \arrow[ddr, bend left, "u_d"] & \\
\kappa j_{d+1} \arrow[r, hook, "cj_{d+1}"] \arrow[drr, bend right, "x_{d+1}"'] & U^{d+1}(GG) \arrow[dr, dashed, "u_{d+1}"] & = \\
 & = & X
\end{tikzcd}$$

By definition, cj_{d+1} makes diagram (1) commute. Thus it remains to show the universal property of the colimit. Let X be an occurrence graph grammar and $xj = (xj_T^{OP}, xj_N) : \kappa j_{d+1} \to X$ be occurrence graph grammar morphisms that make all diagrams with morphisms of $\mathtt{D}^{d+1}$ commute. In particular this object and morphisms commute with $\mathtt{D}^d$ because this diagram is contained in $\mathtt{D}^{d+1}$. As $U^d(GG)$ is the colimit of $\mathtt{D}^d$, there is a universal morphism $u_d : U^d(GG) \to X$. Thus let us define $u_{d+1} = (u_T^{OP}, u_N) : U^{d+1}(GG) \to X$ as follows:

$$u_N(n_a, a) = \begin{cases} u_{dN}(n_{a'}, a'), \text{ if } = i_N(n_a, a) = (n_{a'}, a') \\ xj_N(n_{a''}, a''), \text{ if } c2_N(n_a, a) = (n_{a''}, a'') \end{cases}$$

By Prop. 5.51 each action of N_{d+1} must be in the image of at least one morphism cj_{d+1} (the concurrent derivation obtained by restricting the unfolding of the causes of this action must be a concurrent derivation of GG). Moreover, it can not be the case that to actions of different concurrent derivations in $\mathtt{D}^{d+1}$ are mapped to the same action in κj_{d+1} and to different actions in X because the morphisms xj must commute with all prefix morphisms in $\mathtt{D}^{d+1}$. The mapping u_T of type graphs can be derived from the mapping u_N (analogously to the construction of cj_{d+1} above). Based on the facts that u_{dN} and xj_N are occurrence graph grammar morphisms, we conclude that u_{d+1} is also an occurrence graph grammar morphism. Uniqueness of u_{d+1} follows from the fact that all morphisms cj_{d+1} are together surjective on $U^{d+1}(GG)$.

$\surd$

6.3 Relationship between Typed-Graph Grammars and Their Unfolding Semantics

In Sect. 6.1 we showed how to construct an unfolding of a graph grammar. Then one can ask whether this construction can be extended to a functor yielding for each graph grammar its unfolding semantics and for each graph grammar morphism an occurrence graph grammar morphism. Such a functorial relationship is interesting because it means that, if two grammars are related by a morphism, their semantics are also related by a corresponding morphism, that is, syntactical relationship between grammars induce semantical ones. The first problem we have to face towards the definition of such a functor is that the unfolding construction defined in the last section is only unique up to isomorphism. One way to solve this problem would be to make a suitable "choice of occurrence grammars" and then take the unfolding from this distinguished class (a suitable choice should be based on prefix-isomorphism classes of occurrence nets – see below). The second problem is how to map a graph grammar morphism to an occurrence graph grammar morphism. In general, there are many choices of morphisms that are possible to be associated with a concrete graph grammar morphism via an unfolding functor. This is because of the graph grammar morphisms that were chosen. We wanted to have a quite flexible relationship between two graph grammars, and therefore there is no tight connection between the rules and the initial graphs of two grammar that are connected by a morphism. We know that there exists suitable morphisms mapping left- and right-hand sides of the rules of one grammar into the rules of the other grammar, but we do not know exactly how these morphisms look like. Depending on the morphisms that are chosen, the core graphs will be mapped in one or another way. This will be illustrated in the next examples.

Example **6.10 (Morphisms between Occurrence Grammars)** In the examples we will present here we will always assume that the morphism mapping the type graphs of the occurrence grammars (not the core graphs) is given, as well as the mapping of rule names. This is because these are the component that we have in a graph grammar morphism, and we want now to investigate which would be the possible occurrence grammar morphisms that could be associated to this given morphism via an unfolding functor. Consider the occurrence grammars $Occ1$ and $Occ2$ shown in Figure 6.5. There we can see that there are two ways of mapping the core graph $C2$ to the core graph $C1$ that are compatible with the initial graph and with the mapping of the rule r. The only requirement for such a morphism for initial graph is that the initial graph of $Occ2$ is isomorphic to the retyping of the initial graph of $Occ1$. Therefore, if we choose the isomorphism mapping $\bullet_1 \to \blacksquare_1$ and $\bullet_2 \to \blacksquare_2$ we will find out the the core component given by the solid arrows yields a graph grammar morphism. If we instead choose the isomorphism $\bullet_1 \to \blacksquare_2$ and $\bullet_2 \to \blacksquare_1$, we will obtain that the core mapping given by the dashed arrows yields an occurrence graph grammar morphism.

Now consider the occurrence graph grammars depicted in Figure 6.6. Again we have a similar situation as in the first case, just that there is only one way to map the initial graph. But again we have two different morphisms because the vertices created by action $a1$ are not directly related to the vertices created by action $b1$ (only by this "there exists one morphism" relation). Note that, for each morphism between $R1$ and $R2$ that is chosen, there exists then only one way to map the actions $a2$ and $a3$. The aim of this last example is to show that occurrence graph grammar morphisms describe very close relationships between two occurrence grammars. Figure 6.7 shows two occurrence graph grammars where there is no occurrence graph grammar morphism from

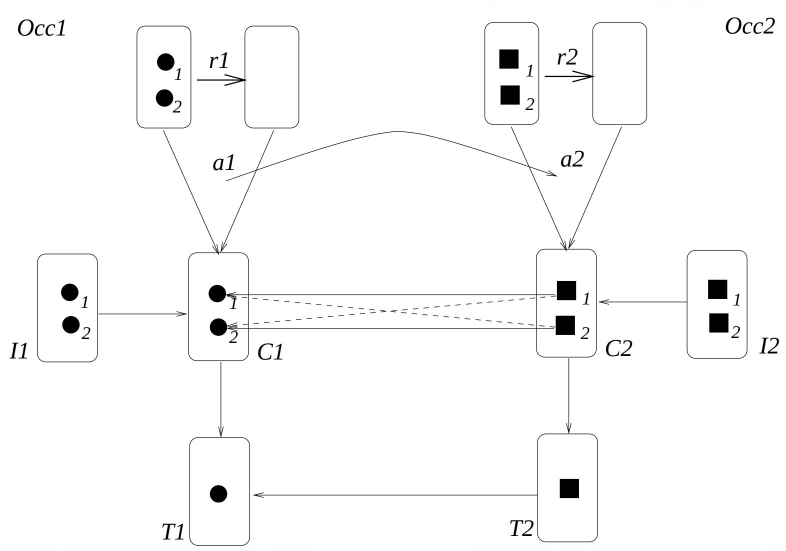


Figure 6.5:

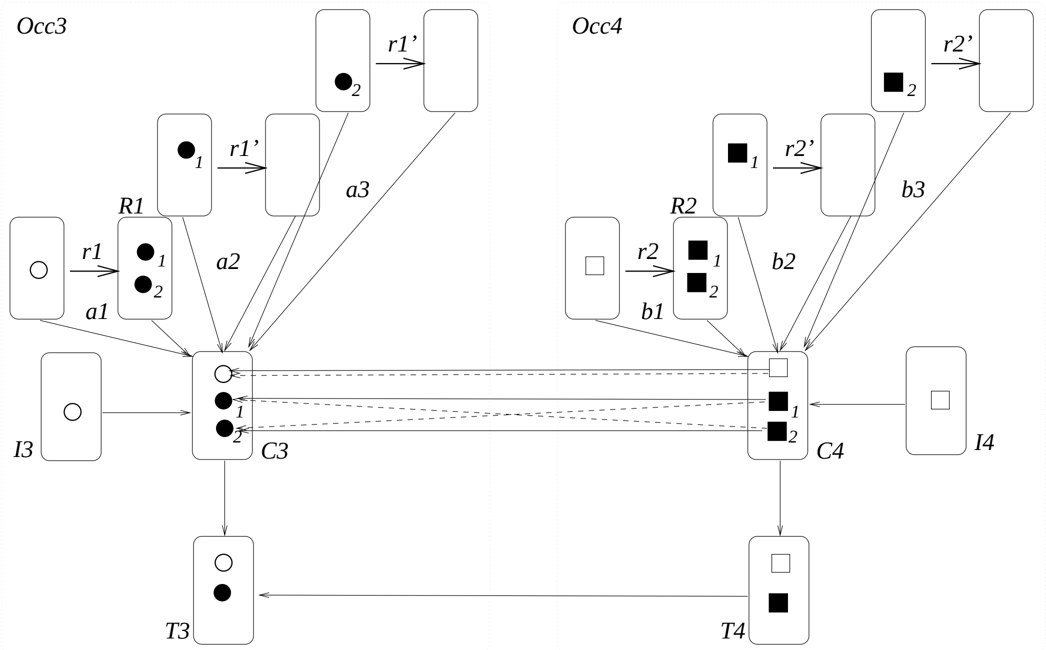


Figure 6.6:

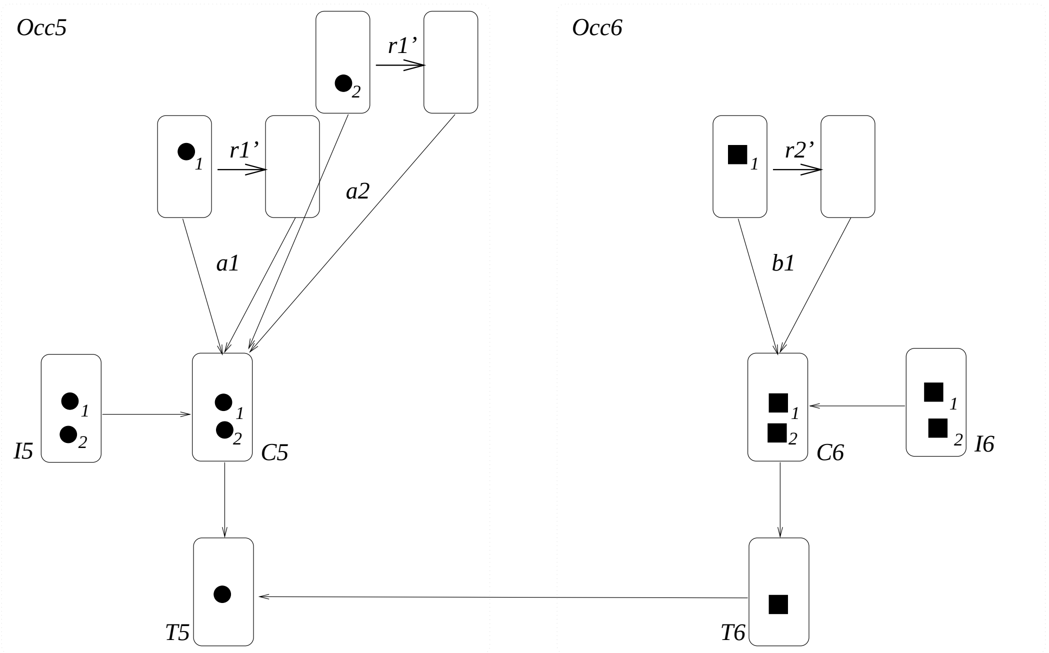


Figure 6.7:

$Occ5$ to $Occ6$. First of all, the two actions $a1$ and $a2$ can not be both mapped to the action $b1$ because this mapping of the core graph would either not commute with $a1$ or with $a2$. As partial morphisms are allowed, we could try not to map one of these actions. But this trial would also fail because this would violate condition 2. of occurrence graph grammar morphisms (Def. 5.31): the diagram using the domain restriction of such a core graph component would not yield a pullback. Note that $Occ2$ is not an unfolding because if the rule $r2$ is a rule of the grammar, then it should have been applied at the other existing match to become an unfolding. ☺

These examples show that, when there is an occurrence grammar morphism between two grammars, the other occurrence grammar morphisms that may exist using the same type graph morphism and same mapping of rules describe practically the same relationship. This is not accidentally this way, but is basically a consequence of the second requirement of occurrence graph grammars (Def. 5.31). This requirement is quite reasonable because it assures that all instances in the core graph of some item of the type graph are mapped compatibly, where compatibly here means via a pullback (because the translation of graphs is made via pullbacks). The different choices for pullbacks that can be made for the domain restriction yield different occurrence graph grammar morphisms.

After this deeper investigation of occurrence grammar morphisms, we will now define a category of *abstract occurrence graph grammars* and *abstract occurrence grammar morphisms*, and, by using this category and a suitable modification of the folding functor defined in Sect. 5.5, we will be able to define an adjunction between the categories of graph grammars and abstract occurrence grammars.

The first step towards the category of abstract occurrence grammars is to define a suitable notion of an abstract occurrence grammar. This will be based on prefix (iso)morphisms. A prefix-isomorphism class of occurrence grammars is an isomorphism class which is determined by prefix morphisms. Prefix-isomorphisms assure that all elements of this class use exactly the same rules and initial graph, but the core graph is up to isomorphism. As the core graph represents the intermediate graphs obtained during the application of rules of a grammar, the fact that we choose it here only up to isomorphism corresponds to the idea of derivation of the algebraic approach to graph grammars: the result of a derivation is unique up to isomorphism, and thus it is natural to say that the "results" of all derivation steps shall also be unique up to isomorphism only.

Definition 6.11 (Prefix-Equivalence Classes) *A* ***prefix-equivalence*** *class of occurrence graph grammars is an equivalence class defined as follows:*

$$\overline{Occ} = \{Occ' | p : Occ' \to Occ \text{ is an isomorphism and a prefix-morphism}\}$$

☺

Remark. *Prefix isomorphisms can be seen as a kind of "standard isomorphism" between occurrence grammars [CEL$^+$94b]. Standard isomorphisms are obtained as a choice of isomorphisms which includes identities and is compatible with composition. As identities are prefix morphisms, prefix morphisms compose and there is at most one prefix morphism between two occurrence graph grammar, prefix morphism can almost be considered as standard isomorphisms. "Almost" because prefix isomorphisms only exist between occurrence graph grammars having the same initial graph, same type and using the same rules (only the core graphs may be different).* ☺

Now we will define abstract occurrence graph grammars as prefix equivalence classes of occurrence grammars. The abstract morphisms are, as discussed above, determined by the mapping of the type graph and the rule names. Therefore, we consider two morphisms as belonging to the same class if they have the same corresponding components (this is formalized by using the folding functor).

Definition 6.12 (Abstract Occurrence Graph Grammars) *An* ***abstract occurrence graph grammar*** *is a prefix-equivalence class of occurrence graph grammars. An* ***abstract occurrence graph grammar morphism*** $\overline{f} : \overline{Occ} \to \overline{Occ1}$ *is defined as* $\overline{f} = \{g \mid \mathcal{F}(g) = \mathcal{F}(f)\}$*, where* $f, g : Occ1 \to Occ2$ *are morphisms in* **OccGG**.

The category of abstract occurrence graph grammars and abstract occurrence graph grammar morphisms is denoted by $\overline{\mathbf{OccGG}}$*, where identities are defined by* $\overline{id_{Occ}}$ *and composition by* $\overline{g} \circ \overline{f} = \overline{g \circ f}$*.* ☺

Remark. *In an abstract occurrence graph grammar, only the core graph is up to isomorphism, and is an abstract occurrence graph grammar morphisms, only the mapping of core graphs is up to isomorphism (assured by the occurrence graph grammar morphism conditions – Def. 5.31 and Def. 5.15).* ☺

Proposition 6.13 *The category $\overline{\mathbf{OccGG}}$ is well-defined.* ☺

Proof.

1. *Identities are well-defined:* By the definition of $\overline{Occ}$ as a prefix-equivalence class and the definition of prefix morphisms (Def. 5.39), all occurrence grammars in $\overline{Occ}$ have the same rules and the same type. Therefore we must have that for each $Occ1, Occ2 \in \overline{Occ}$, $\mathcal{F}id_{Occ1} = \mathcal{F}id_{Occ2}$.

2. *Composition is well-defined:* This is basically due to the fact that between two occurrence grammars $Occ1, Occ2$ in $\overline{Occ}$ there can be only one prefix (iso)morphism (Prop. 5.43). Let $\overline{f} : \overline{Occ1} \to \overline{Occ2}$ and $\overline{g} : \overline{Occ2} \to \overline{Occ3}$ be abstract occurrence grammar morphisms, $(f : Occ1 \to Occ2), (f' : Occ1' \to Occ2') \in \overline{f}$ and $(g : Occ2 \to Occ3, (g' : Occ2' \to Occ3') \in \overline{g}$. We obviously have that $\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$ because the composition of morphisms is defined componentwise. The same holds for $g' \circ f'$. It remains to show that $\mathcal{F}(g' \circ f') = \mathcal{F}(g \circ f)$. This holds because the type component and the rules mapping between the objects $Occi$ and $Occi'$, for $i = 1..3$ are identities.

√

Definition 6.14 (Folding Functor II) *The* abstract folding functor $\overline{\mathcal{F}} : \overline{\mathbf{OccGG}} \to GG$ *is defined as follows:*

Objects *: $\overline{\mathcal{F}}(\overline{Occ}) = \mathcal{F}(Occ)$, for all $\overline{Occ} \in \overline{\mathbf{OccGG}}$, $Occ \in \overline{Occ}$*

Morphisms *: $\overline{\mathcal{F}}(\overline{f}) = \mathcal{F}(f)$ for all morphism $\overline{f} \in \overline{\mathbf{OccGG}}$, $f \in \overline{f}$*

☺

Remarks. *Well-definedness of the abstract folding functor follows from the fact that the objects of $\overline{\mathbf{OccGG}}$ are equivalence classes of occurrence grammars that have the same initial graph and use the same rules, and that morphisms of $\overline{\mathbf{OccGG}}$ are equivalence classes of morphisms that have the same components mapping type graphs and rules.* ☺

Now we will show that the unfolding of graph grammars is a co-free construction with respect to the (abstract) folding functor. This means that there is an unfolding functor that is a right-adjoint to the (abstract) folding functor. The co-unit of this construction is the inclusion $i : \overline{\mathcal{F}}(Unf(GG)) \to GG$. The co-unit is in general not the identity because there may be rules of GG that are never used, and thus are not in $Unf(GG)$. As $Unf(GG)$ is constructed using the initial graph and the rules of GG, i can be defined as an inclusion.

Theorem 6.15 *Given a graph grammar GG. Then its unfolding $Unf(GG)$ together with the inclusion morphism $i : \overline{\mathcal{F}}(Unf(GG)) \to GG$ is a co-free construction with respect to the folding functor $\overline{\mathcal{F}}$.* ☺

Proof. Let $GG = (T, I^T, N_{GG}, n^{GG})$ be a graph grammar and $Unf(GG) = (C^T, I^{C \nearrow T}, N, n)$ be the unfolding of GG. Then we have to show that the inclusion $i : \overline{\mathcal{F}}(Unf(GG)) \to GG$ is universal from any other folding to GG, i.e., for any (abstract) occurrence graph grammar

$U = (CU^{TU}, IU^{CU \nearrow TU}, NU, nu)$ and any graph grammar morphism $k : \overline{\mathcal{F}}(U) \to GG$ there exists a unique $\overline{h} : U \to Unf(GG)$ in $\overline{\mathbf{OccGG}}$ such that $k = u \circ \overline{\mathcal{F}}(\overline{h})$. That is

$$\begin{tikzcd}
GG & : & Unf(GG) & \text{such that} & \overline{\mathcal{F}}(Unf(GG)) \arrow[r, "i"] & GG \\
\overline{\mathcal{F}}(U) \arrow[u, "\forall k"] & & U \arrow[u, "\exists!\overline{h}"] & & \overline{\mathcal{F}}(U) \arrow[u, "\overline{\mathcal{F}}(\overline{h})"] \arrow[ur, "k"'] \arrow[ur, phantom, "(1)", near end, shift left=3] &
\end{tikzcd}$$

We will make this proof by induction on the depth of $Unf(GG)$ proving that for all approximations n of $Unf(GG)$ there is a unique $\overline{h}^n : U^n(U) \to U^n(GG)$ such that $(1)^n$ commutes. As $Unf(GG)$ and U are the colimits of their approximations then there must be a unique $\overline{h}$ such that (1) commutes. It is enough to find one morphism that makes (1) commute, uniqueness follow from the definition of $\overline{\mathbf{OccGG}}$(if there are two morphisms that make (1) commute, they must be the same).

n=0 : By degfinition of the unfolding of depth 0, we have that $C_0^T = I^T$ and $CU_0^{TU} = IU^{TU}$. Let $h^0 = ((h_{c0}^{k_T})^{OP}, \emptyset)$, where h_{c0} is the composition of the universal morphism induced by the retyping of IU and the corresponding pullback morphism of the retyping (see Def. 5.5). Diagram (2) is a pullback and (3) commutes because k is a graph grammar morphism. Thus, h^0 is an occurrence graph grammar morphism. Let $h^0 \in \overline{h}^0$. Then $\overline{\mathcal{F}}(\overline{h}^0) = (k_{T^{OP}}, \emptyset)$. Thus $(1)^0$ trivially commutes and $\overline{h}^0$ is the only morphism such that this happens (by definition of $\overline{\mathbf{OccGG}}$).

$$\begin{tikzcd}
CU = IU \arrow[r, maps to] & TU \\
dom(h_{c0}) = C \arrow[u, maps to, "h_{c0}!=h_{c0}"] \arrow[r, maps to] \arrow[d, hook, "c^{\blacktriangledown}=id"'] \arrow[ur, phantom, "(2)"] \arrow[dr, phantom, "(3)"] & dom(k_T) \arrow[u, maps to, "k_T!"'] \arrow[d, hook, "k_T^{\blacktriangledown}"] \\
C = I \arrow[r, maps to] & T
\end{tikzcd}$$

n+1 : By induction hypothesis we have that there is a unique $\overline{h}^n$ that makes $(1)^n$ commute. Let $h^n = ((h_{cn}^{k_T})^{OP}, h_{nn}) \in \overline{h}^n$, $\overline{iU} : U^n(U) \to U^{n+1}(U)$ and $\overline{i} : U^n(GG) \to U^{n+1}(GG)$ be the unfolding inclusions, $iU \in \overline{iU}$ and $i \in \overline{i}$.

$$\begin{tikzcd}
U^n(U) \arrow[r, "h^n"] \arrow[d, hook, "iU"'] \arrow[dr, phantom, "="] & U^n(GG) \arrow[d, hook, "i"] \\
U^{n+1}(U) \arrow[r, hook, dashed, "h^{n+1}"] & U^{n+1}(GG)
\end{tikzcd}$$

Because h^n is an occurrence graph grammar morphism, diagram (4) is a pullback and (5) commutes. Let (6) be the pullback of morphisms $t^{CU_{n+1}}$ and $k_T!$ with pullback object P. As iU is a graph grammar morphism we have that $t^{CU_{n+1}} = t^{CU_n} \circ iU_T$. This implies that there is a universal morphism $u : P \to dom(h_{cn})$ induced by pullback (4). Moreover u makes (7) commute and thus (7) is pullback (because (4) and (6) are pullbacks). Let $\mathcal{T}_{k_T}(L_a)$ be the retyping of L_a with respect to the morphism k_T (that is $\mathcal{T}_{k_T}(L_a)$ is the pullback object of morphisms $k_T!$ and t^{L_a}). Diagram (8) commutes because $U^{n+1}(U)$ is a doubly-typed graph grammar. Thus, analogously to the obtention of u, we obtain an universal morphism $p2$ that makes (9) be a pullback. Let $h_{nn+1} : NU_{n+1} \rightarrow N_{n+1}$ be defined as follows: $\forall(n_a, a) \in NU_{n+1}$:

$$h_{nn+1}(n_a, a) = \begin{cases} \texttt{undef}, \text{ if } n_a \notin dom(k_N) \\ i_N \circ h_{nn}(n_{a'}, a'), \text{ if } = iU_N(n_{a'}, a') \\ (n_{a''}, a''), \text{ otherwise} \end{cases}$$

where $n_{a''} = k_N(n_a)$, $r_{a''} = n^{GG} \circ k_N(n_a)$, $pre_{a''} = (i_T)^{-1} \circ h^{\blacktriangledown}_{cn} \circ u \circ p2 \circ i^{L_{a''}}$ and $post_{a''}$ is the post-condition of the action having $pre_{a''}$ as pre-condition and $r_{a''}$ as rule. The morphism $i^{L_{a''}}$ is the left-hand side component of the subrule inclusion of L_a (that must exist because k is a graph grammar morphism). This inclusion is not unique, but all choices will lead to the same morphism in $\overline{\mathbf{OccGG}}$(because the same rule and type graph are used).

$$\begin{tikzcd}
 & TU & dom(k_T) \arrow[l, mapsto, "k_T!"'] \arrow[r, hook, "k_T^{\blacktriangledown}"] \arrow[dr, phantom, "(5)"] & T \\
 & CU_n \arrow[u, mapsto, "t^{CU_n}"] \arrow[ur, phantom, "(4)"] & dom(h_{cn}) \arrow[l, mapsto, "h_{cn}!"'] \arrow[u, mapsto] \arrow[r, hook, "h_{cn}^{\blacktriangledown}"] & C_n \arrow[u, mapsto] \\
(8) & CU_{n+1} \arrow[u, two heads, "iU_T"] \arrow[uu, mapsto, bend left, "t^{CU_{n+1}}"] \arrow[ur, phantom, "(7)"] & P \arrow[l, mapsto, "p3"'] \arrow[u, two heads, "u"] \arrow[uu, mapsto, bend left] & C_{n+1} \arrow[u, two heads, "i_T"'] \arrow[uu, mapsto, bend right] \\
L_a \arrow[uuur, mapsto, bend left, "t^{L_a}"] \arrow[ur, mapsto, "pre_a"] & \mathcal{T}_{k_T}(L_a) \arrow[l, mapsto, "p1"] \arrow[ur, mapsto, "p2"'] \arrow[uuur, mapsto, bend left] \arrow[u, phantom, "(9)"] & & L_{a'} \arrow[ll, hook', "i^{L_{a''}}"] \arrow[u, "pre_{a''}"']
\end{tikzcd}$$

Well-definedness of h_{nn+1}: assume that $pre_{a''}$ is not total. By construction, this can only happen if there is $x \in p2 \circ i^{L_{a''}}(L_{a''})$ and $x \notin dom(u)$. As (7) commutes, this means that $p3(x)$ is created is $U^{n+1}(U)$ (that is, $p3(x) \notin dom(iU_T)$), what is not possible because $p3(x) \in rng(pre_a)$ (because (9) commutes). As pre_a is a concurrent graph, occurrence graph grammar morphisms preserve concurrent graphs and prefix morphisms reflect concurrent graphs, $pre_{a''}$ is also a concurrent graph. Therefore, by the definition of the unfolding, there must be a unique action a'' of $U^{n+1}(GG)$ using rule $r_{a''}$ and match $pre_{a''}$.

The type component of h^{n+1} is completely determined by h_{nn+1} and h_{cn} because all items of the core graph must have been either in the initial graph present or created by exactly one rule. The items that have been created by actions that are not in the image of h_{nn+1} can not be mapped and the items that have been created by actions that are in the image o h_{nn+1}, otherwise h^{n+1} would not become a graph grammar morphism. Let $p : P \rightarrow C_n$ be defined as follows:

$$p(x) = \begin{cases} (i_T)^{-1} \circ h^{\blacktriangledown}_{cn} \circ u(x), \text{ if } = x \in dom(u) \\ post_{a''}(y), \text{ otherwise} \end{cases}$$

where a'' is the action in Unf^n that creates x, and $p2' \circ i^{R_{a''}}(x) = y$ (again, this is non-deterministic, but any choice will lead to an isomorphic result). The morphism p is total by definition, and injective because i_T, $h^{\blacktriangledown}_{cn}$ and u are injective and $p2'$ is injective on items that are created. Thus we define h_{cn+1} as being the morphism where $dom(h_{cn+1}) \cong P$ and the span representation is isomorphic to $(p, p3)$.

Analogously to the proof of Theo. 6.9, this pair is an occurrence graph grammar morphism. Therefore $\overline{h}^{n+1}$ is in $\overline{\textbf{OccGG}}$ and makes (1) commute.

$\surd$

The existence of an adjunction in which the unfolding functor is the right-adjoint assures that all (deterministic and non-deterministic) computations of a graph grammar GG, described by (abstract) occurrence graph grammars $\overline{Occ}$ such that there is an inclusion $i : \overline{\mathcal{F}}(\overline{Occ}) \rightarrow GG$, are included in the unfolding of GG. This will be shown in the next theorem.

Theorem 6.16 *The unfolding semantics $Unf(GG)$ describes all deterministic and non-deterministic computations of a graph grammar GG, that is, for all (abstract) occurrence graph grammars Occ such that there is an inclusion $i : \overline{\mathcal{F}}(\overline{Occ}) \rightarrow GG$, there is a prefix morphism $p : Occ \rightarrow Unf$, where $Occ \in \overline{Occ}$ and $Unf \in Unf(GG)$.* ☺

Proof. Due to Theo. 6.15, there is a morphism $p : Occ \rightarrow Unf(GG)$ that is compatible with the type and rules mappings. The construction of p is described in the proof of Theo. 6.15. By the fact that k is an inclusion, the construction of p yields a prefix morphism. $\surd$

6.4 Parallel Composition and Unfolding of Graph Grammars

Similar to the composition of graph grammars in general there should be also a composition of occurrence graph grammars. Occurrence graph grammars are graph grammars based on more ellaborated graphs and that satisfy some additional conditions. In other formalisms, like transition systems and Petri nets, the composition of their unfoldings was defined in terms of limit constructions [WN94, MMS94]. For these reasons, analogous to the synctactical case, a composition based on limits (or more concretely, pullbacks) seems to be a good candidate for the composition of unfoldings. Therefore, we will take advantage of our categorical setting (in which some operations like pullbacks are described abstractly) and directly define it as a pullback (having products as special cases). In Theo. 6.18 and Example 6.19 we will see that this choice is really adequate, and stresses the fact that the explicit construction of this parallel composition can be made analogous to the one of parallel composition of typed graph grammars.

Definition 6.17 (Occ-Cooperative Parallel Composition) *Let $\overline{Occi}$, for $i = 0, 1, 2$ be abstract occurrence graph grammars, and $\overline{s1} : \overline{Occ1} \rightarrow \overline{Occ0}$ and $\overline{s2} : \overline{Occ2} \rightarrow \overline{Occ0}$ be abstract occurrence graph grammar morphisms. Then the* ***occ-cooperative parallel composition*** *$\overline{Occ1}||_{\overline{Occ0}}\overline{Occ2}$ of $\overline{Occ1}$ and $\overline{Occ2}$ with respect to $\overline{Occ0}$ using $\overline{s1}$ and $\overline{s2}$ is the pullback of $\overline{s1}$ and $\overline{s2}$ in the category $\overline{\textbf{OccGG}}$, if it exists.* ☺

Remark. *From this definition it follows that not every occurrence grammars can be composed. But this is also the case for typed graph grammars, where the composition was only defined for special morphisms and interface grammar. A deeper investigation on the existence of* *arbitrary pullbacks in* $\overline{\mathbf{OccGG}}$*, as well as in* **GG***, and their interpretation is left for future work. The pullbacks that were shown to exist and have an interpretation in terms of parallel composition in* **GG** *will have a correspondence in* $\overline{\mathbf{OccGG}}$ *(Theo. 6.18).* ☺

Having defined syntactical and semantical composition operators immediately raises the questionabout their relation. The next theorem shows that the parallel composition operators of graph grammars defined in Chap. 4 are compatible with the unfolding semantics. This result allows for specifications of a system based on the specifications of its components:when merging the components to get the whole system, the semantics can be obtained be a suitable merge of the semantics of the components. Thus, analogously to the theory of modules developed in [EM90] for algebraic specifications, we believe that this composition operators and the unfolding semantics can be used as a basis for a similar module concept for graph grammars (but using limits instead of colimits as in [EM90]).

Theorem 6.18 *The unfolding semantics is compatible with the parallel composition of grammars, that is given graph grammars* GGi*, for* $i = 0, 1, 2$ *with* $GG1||_{GG0}GG2$ *being their cooperative parallel composition with respect to morphisms* $s1$ *and* $s2$ *we have that*

$$Unf(GG1||_{GG0}GG) = Unf(GG1)||_{Unf(GG0)}Unf(GG1).$$

☺

Proof. The functors $\mathcal{U}$ and $\overline{\mathcal{F}}$ form an adjunction in which $\mathcal{U}$ is the right-adjoint (Theo. 6.15). The fact that right-adjoints preserve limits and that the composition of graph grammars and of unfoldings are based on limits (Theos. 4.16, 4.24 and Def. 6.17) yields the desired compatibility. $\surd$

Example **6.19** Figure 2.9 shows part of the unfolding of the PBX system (actions $a1$, $a2$ and $a5$ are obtained in the first unfolding step, and action $a6$ in the second unfolding step). The rules that are used in these actions are already in the interface grammar AGV. Thus, the unfolding of AGV will have also actions corresponding to these ones (all elements needed by these AGV-rules are in the initial graph of the abstract view). Analogously, we may find unfoldings for PLV and CLV and these actions will also be in these unfoldings. In fact, the unfolding of AGV may contain many more actions than $Unf(CGV)$ because the rules of AGV have less elements in the left-hand sides that have to be present. Thus, in the case of this example, where the specialization morphisms are total on the rules component, the unfolding $Unf(CGV)$ will have less actions than the unfoldings of the other grammars. It can be considered as an intersection of the actions that may occur in both components (this means, each action of $Unf(CGV)$ represents a synchronization of actions of $Unf(PLV)$ and $Unf(CLV)$). ☺

7

Related Work

The purpose of this chapter is to discuss in more details the relationships between the concepts introduced in this thesis and the already existing ones in the literature.

The main aims of this thesis are to present parallel composition operators for graph grammars and an unfolding semantics that is compatible with these operators and is particularly well-suited to concurrent systems. Therefore, the relationships to other approaches will be done in two parts: relationships between the parallel composition of graph grammars and other composition operators, and relationship between unfolding semantics and other semantics. In Sect. 7.1 other kinds of composition of graph grammars are considered and in Sect. 7.2 other kinds of concurrent semantics for graph grammars are discussed.

A lot of inspiration for the concepts developed here for graph grammars came from the area of Petri nets. Therefore, some of the main results, like unfolding semantics and pure parallel composition, are (non-trivial) generalizations of corresponding concepts for Petri nets. In Sect. 7.3 we discuss concepts of Petri nets that are related to this work. Moreover, we will discuss how the concept of cooperative parallel composition can be seen as a reasonable new composition operator for nets. This means that the specialization of some concepts presented here may enrich the theory of Petri nets.

7.1 Other Approaches to the Composition of Graph Grammars

Parallel and Distributed Graph Grammars: Specially in the algebraic approach to graph grammars, the suitability of graph grammars to concurrent and distributed systems was an aspect of major interest. This led to the definition of parallel and distributed graph grammars. In [Tae96] an overview of different approaches (not only the algebraic one) to parallelism and distribution in graph grammars is given, and corresponding concepts of parallel and distributed graph grammars in the DPO approach are investigated in more detail. The basic idea of parallel graph grammars is to allow not only one rule to be applied at a time but many. A set of applicable rules is summarized into a single rule via parallel rule and/or amalgamation constructions and this composed rule is then applied to the actual graph. Distributed grammars are based on distributed graphs, that are diagrams of graphs. This idea was originally defined for the DPO approach in [EBHL88] and was later on adapted for the SPO approach in [EL93b]. Each distributed graph represents a global state of one system, and this state can be decomposed into local states. Then, rules may be applied separately to the local states and the results may be joined together to build again a global state. The rules of such grammars can then be local to one component or global, where global rules are obtained by parallel/amalgamated composition of local rules. Both concepts of parallel and distributed graph grammars give the impression that a system is a collection of smaller subsystems. The operational semantics of these grammars is based on parallel/amalgamated derivations. But, although one can see some structuring in the states/rules, no operators for constructing the graph grammar representing the whole system based on graph grammars representing the components is given. The aim of this kind of grammars is different than the aim of the parallel composition of graph grammars presented in this thesis. Parallel/Distributed graph grammars are meant to describe a parallel/distributed system as a whole and give it a semantics that takes into account this structuring (therefore, usually a distributed state is not "flattened" to a simple graph). The parallel composition of graph grammars has as main aim to allow the construction of a system based on smaller components. In other words, a parallel/distributed graph grammar is one grammar in which rules may be applied in parallel (or synchronously via amalgamated rules) and a parallel composition of grammars is a grammar whose rules are based on the rules of the component grammars. Obviously, rules of a parallelly composed grammar may also be applied in parallel. This is assured because the semantics given to such a grammar (the unfolding semantics) is a true concurrency semantics. In the case of parallel/distributed graph grammars, parallelism is expressed at the semantical level by explicitly including derivations using parallel/amalgamated rules whereas in a parallelly composed grammar, parallelism is expressed by the absence of causal relationships between corresponding actions (synchronization is expressed by the application of parallel/amalgamated rules).

DIEGO: In [TS95] a concept for building graph grammars from smaller components was introduced. This concept is called DIEGO and is based on (hierarchically) distributed graph grammars. According to the idea of distributed graph grammars, the DIEGO components are not glued together, but they just execute together (the semantics of a DIEGO specification is based on a distributed graph grammar). Relationships between components are given by suitable graph grammar morphisms.For DIEGO, the issue of compatibility of the composition of DIEGO-modules with respect to some semantical model was not yet investigated. However, it should be said that DIEGO is still under development and provides a promising concept of encapsulation and distribution of data for graph grammars. Moreover, it can be considered a s an instance of a general module concept for graph transformations introduced in [EE95].

Classes: In [Kor94, Kor96] first ideas about a way of combining graph grammars (within the SPO approach) that represented classes of a class-based system were sketched. There, a class based system was defined to be a doubly-typed graph grammar (using our terminology), where the double-type graph was called class graph. The idea suggested (but not formally defined) for the composition of such classes was that the composition of different classes along a common subclass should be induced by the gluing of the class graphs. This idea is implemented in the definition of cooperative parallel composition (Sect. 4.2.2) and therefore we believe that this composition can be used to formalize the intuitive composition of classes of [Kor96].

TROLL light: In [WG96] a graph grammar description of the object description language TROLL light [CGH92] was presented. The SPO approach used in this work is an extension of the SPO approach of [Löw90] by considering attributes (as in [LKW93]) and partial algebras. Each template (kind of object) of a TROLL light specification becomes a kind of vertex and relationships between vertices become edges. The event of a TROLL light specification are described by (valuation) rules. The interesting aspect of this approach is that the execution of events that involve more than one object is done via amalgamated rules. These rules are used to force the synchronization of events that necessarily must occur together. In [WG96] amalgamated rules were used just at the semantical level to explain the operational behaviour of TROLL light. As the cooperative parallel composition of graph grammars is based on amalgamated rules, one can think of another possibility to describe the semantics of TROLL light: each template of TROLL is described by one graph grammar and the synchronization points are collected in a common interface grammar. Then the behaviour of such an specification can be given by the unfolding of the corresponding composed grammar (obtained by the cooperative parallel composition). Obviously, to realize this the concepts defined in this thesis should be extended with the necessary notions of attributed and partial graphs.

Composition of Graph Transformation Systems: A notion of composition of graph transformation systems (graph grammars without initial graphs) was introduced in [CH95, HCEL96]. This composition corresponds to a union (gluing) of graph transformation systems and is realized by colimit constructions on graph transformation systems. The (interleaving and concrete) semantics of graph transformation system based on graph transition systems is compositional with respect to this union operator. It was noticed in [HCEL96] that for the definition of the union operator it is a necessary condition that initial graphs are not considered. A significant difference between the union and the parallel composition operators are that for the union a rule of the interface can only be specialized in by one rule in each component, whereas in the cooperative parallel composition the same rule of the interface may be specialized by many rules of each component. The latter allows the interpretation of the components as specializations (or refinements) of the interface. The rules of a parallelly composed system are based on the composition of the rules of the component systems (using parallel and amalgamated rules). Moreover, the parallel composition is compatible with the unfolding semantics, that is truly concurrent and representation independent.

ESM-Systems: ESM systems [Jan93] are a kind of graph rewriting systems that originated from actor grammars [JR89]. For these systems, a kind of composition based on gluing of computation structures on a common subpart have been defined. Formally, this gluing is obtained by a pushout construction. In [Jan93, Jan96] it is shown that this composition is compatible with a true concurrency semantics for ESM systems based on processes. As ESM systems do not have an initial state, this composition corresponds to the one of graph transformation systems and the comparison to the parallel composition introduced in this thesis is analogous to that case.

GRACE: The main aim of the language GRACE [Kre95, KK96] is to provide an approach to build large graph rewriting systems from smaller ones independently from a concrete graph grammar approach. The idea is that GRACE provides a kind of abstract interface between different kinds of graph grammar components. The basic units of GRACE are called *transformation units*. Such a transformation unit can be seen as a graph grammar with application conditions and that may use (in the sense of include) other transformation units.

The semantics of a transformation unit is given by the input/output relations induced by the application of rules of the transformation unit (that is, the semantics abstracts out from the transformation process). As one of the main aims of GRACE is to build a big system by composing smaller ones, composition operators are of great importance. In particular, the semantics of the union operator shall be the union of the relations describing the semantics of the components (see [KK96]). The union operator describes a kind of union like of graph transformation systems in the DPO-approach discussed before. It is an interesting topic of research to find out whether the parallel composition operators presented here can be also a reasonable operators for GRACE.

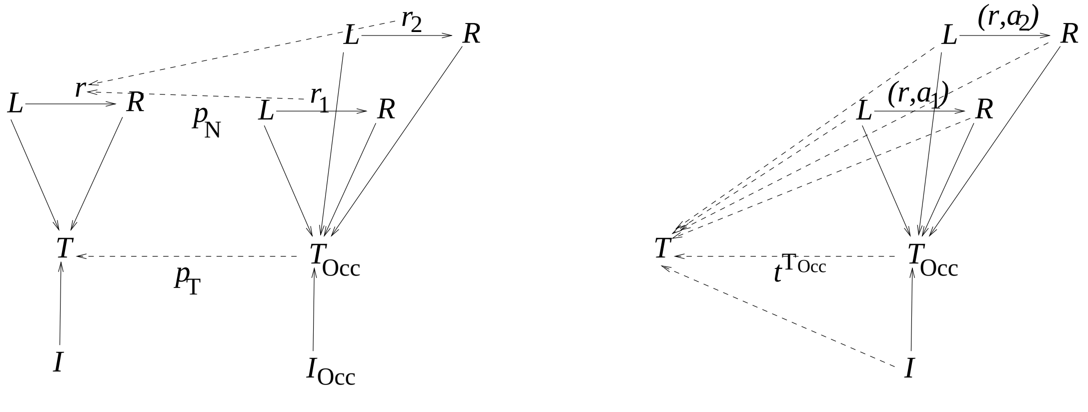
7.2 Other Approaches to Concurrent Semantics of Graph Grammars

Concurrent Derivations: The introduction of the unfolding semantics for graph grammars in Chap. 6 had the main aim to provide a semantics for graph grammars in which the aspects of concurrency and compositionality are the most important ones. Other semantics for graph grammars that also describe suitably concurrent aspects are the concurrent semantics (Def. 3.46) and the processes semantics defined in [KR95]. These two semantics describe the behaviour of a graph grammar by a category of (deterministic or non-deterministic in the case of [KR95]) concurrent derivations. Thus, both include some redundancy because it can be that the same action is represented in different objects. In the unfolding semantics, each action is represented only once, and all actions are described together in the same structure. This makes the analysis of relationships between them easier. Moreover, it is not investigated whether the concurrency and the process semantics are compositional with respect to the parallel composition of graph grammars. The formal relationship between the concurrent and unfolding semantics is given by Theo. 6.9.

Actor Grammars: In [JR91] a structure called *computation graph* was introduced as a representation of rewriting processes in actor grammars [JR89]. These graphs are labeled, bipartite, directed, acyclic graphs in which one kind of nodes represent actions (or events) and the other kind represent nodes of states. Computation graphs can be seen as the counterparts for actor grammars of (deterministic) occurrence nets in net theory. The correspondence between computation graphs and occurrence grammars as defined in Def. 5.29 is roughly as follows: the nodes representing states of a computation graph are the vertices of the core graph of the occurrence grammar, and the node representing actions correspond to actions of an occurrence grammar, where the pre- and post-morphisms correspond to the incoming and outcoming edges connecting the action-nodes to their corresponding state-nodes.

Event Structure Semantics: A true concurrent, branching structure semantics for graph grammars is the event structure semantics [Sch94, CEL$^+$94a, Kor95, CEL$^+$96b]. In [Cor95] there is a discussion about the adequacy of event structures as a semantics for graph grammars. Graph grammars allow a very elaborated description of states and states changes, whereas the relationships between different rules are not explicitly described (but implicitly through the overlappings in the type graph). In an event structure semantics, states are not represented, but only events and relationships among them. Therefore, an event structure semantics for graph grammars puts emphasis on the relationships between derivation steps. Instead, an unfolding semantics also represents the states of a system.

Graph Grammar Processes: There are different notions of graph grammar processes. In [Kre83, KW86] and [KR95] graph grammar processes are described as partial order of derivation steps (called derivation processes and concurrent derivations, respectively). An occurrence graph grammar here represents a process following this idea. The unfolding semantics is thus the "biggest" process of a graph grammar (it includes all other processes). In [CMR96a] another approach to graph grammar processes was presented, mainly inspired by Petri nets processes: a graph process is a morphism from an occurrence grammar to the original grammar. Thus, this work introduces occurrence grammars, and therefore we will make a more detailed comparison to our definition of occurrence graph grammars. We will refer to the occurrence grammars defined in [CMR96a] as DPO-occurrence grammars in the following. A DPO-occurrence grammar is a (simply) typed graph grammar, whereas an occurrence graph grammar is a doubly-typed graph grammar. Therefore, to describe a computation of a graph grammar GG using a DPO-occurrence grammar Occ requires a mapping $p : Occ \to GG$ that indicates in which way the rules (and items of the type graphs) of Occ are related to the ones of GG. This mapping p is then called a graph process. For example, a graph process p of a graph grammar $GG = (T, I, \{r\}, n)$, where $n(r) = (L \to R)$ (here we abstract from the fact that DPO rules are spans of morphisms), can be the one illustrated in diagram (1) of Figure 7.1 (where I and I_{Occ} are isomorphic). [1] The same computation of GG can be described by using an occurrence graph grammar DGG shown in diagram (2) of Figure 7.1. The basic difference is that the mapping $p : Occ \to GG$ is "internalized" in the occurrence graph grammar DGG: the component p_T became the typing morphism of T_{Occ} and the mapping of rules is restricted to identities (by using a mapping P_N, a rule $r_1 : L \to R$ of Occ may be mapped to an isomorphic one $r : L' \to R'$ of GG).



(1) (2)

Figure 7.1: (1) Graph Process p (2) Occurrence Grammar DGG

Now we will compare in more detail the axioms of DPO-occurrence grammars with the axioms of occurrence grammars. As discussed in the introduction of Chap. 5, the causal order of DPO-occurrence grammars corresponds here to the occurrence order. DPO-occurrence grammars are finite (the initial graph is finite and the set of rules is finite) and deterministic

[1] Note that, to be able to define a graph grammar process in this way, it is necessary that graph grammar morphisms allow the identification of elements of the type graph. This is not possible using the kind of morphisms defined here, but these, in turn, allow for splitting of types, that is a necessary condition for the universality of the parallel composition of graph grammars.

(there are no conflicts). Therefore, DPO-occurrence grammars correspond to deterministic occurrence grammars (Def. 5.44). In a DPO-occurrence grammar, each item of the core graph is created and consumed by at most one production. This is assured in deterministic occurrence grammars by axiom 4. (the type graph is a core graph). This axiom also assures the strong safety requirement of DPO-occurrence grammars and (together with axiom 1.) that the minimal elements with respect to the occurrence order correspond to the initial graph. The fact that the causal relation of a DPO-occurrence grammar is a partial order fulfills axiom 1. Axiom 3. is trivially satisfied by DPO-occurrence grammars because they are finite. Axiom 5. is satisfied by DPO-occurrence grammars because they have no conflicts. However there is not a bijective correspondence between DPO-occurrence grammars and deterministic occurrence grammars. In DPO-occurrence grammars it is required that the pre-conditions of the actions satisfy the gluing condition of DPO-grammars, and such an axiom is not present in deterministic occurrence grammars (because there is no gluing condition in the SPO-approach). For deterministic grammars, a special structure is imposed on the names of rules (through axiom 6.), and this is not the case for DPO-occurrence grammars.

7.3 Petri Nets

Petri nets [Pet62] have been used since the early 70's as a formalism to describe concurrent systems ([Rei85] provides a good introduction to Petri nets). The main reasons for the success of Petri nets is that they rely on a simple concept of states and transitions and that they provide a graphical representation of the system, what makes the understanding of the model and its behaviour easier even for non specialists. Moreover, Petri nets have a rich theory of concurrency and a large number of specification and analysis tools.

There is a series of works concerned about the relation between Petri nets and graph grammars (some of them are [Kre81, Rei81, KW86, CEL$^+$94a, KR96]). In [Sch96] a survey of different ways of syntactically representing Petri nets with graph grammars, and in [Cor95] the semantical aspects of such relationship are considered in more details. The basic idea of the different ways of representing nets with grammars is always the same: the markings of a net are represented by graphs and the transitions are represented by rules. The differences occur in the choice of the graphs that represent the markings. In spite of the different representations for states, practically all ways of translating nets into graph grammars yield analogous theoretical results. These results are related to suitability of the translation: for example, one of the desired results is that whenever a transition is enabled by a marking of a net, then the corresponding rule is applicable at the translated match.

For the purposes of the comparison done in this thesis, we will stick to the 'minimal way' of modeling Petri nets by algebraic graph grammars as given in [CEL$^+$94a, KR96]. In this approach, Petri nets appear as a very simple kind of graph grammars, namely graph grammars over discrete graphs (graphs without edges) where all rules are completely partial (nothing is preserved from the left- to the right-hand side). In more detail, we have that the places of the net become vertices in a type graph T, the tokens are vertices in a graph typed over T and the transitions are rules.

Example **7.1** Consider the net N on the left-side of Figure 7.2. A corresponding graph grammar GG is shown on the right. For each transition of N there is a corresponding rule in GG. The places of N become are described by vertices of the type graph T of GG. Formally, GG is a tuple

(T, I^T, N, n) where[2]

- $T = (\{a, b\}, \emptyset, \emptyset, \emptyset\}$ (a discrete graph having two vertices)
- $I^T = (I \overset{t^G}{\rightarrow} T)$, where $G = (\{a\}, \emptyset, \emptyset, \emptyset\}$ (a discrete graph having only one vertex) and $t^G(a) = a$ (the type of a is a);
- $N = \{t, s\}$;
- $n(t)$ and $n(s)$ are depicted in Figure 7.2, where the types of the vertices of the involved graphs indicate their corresponding names.

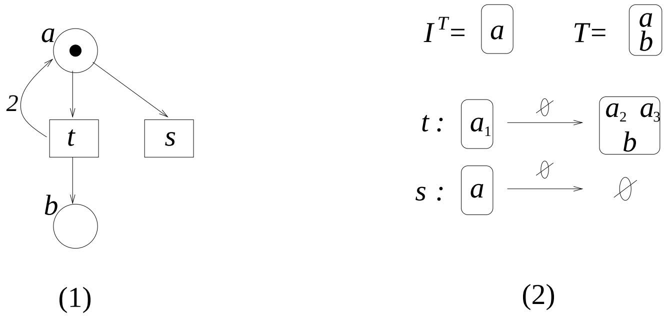


Figure 7.2: (1) P/T-Net N (2) (Typed-)Graph Grammar GG

☺

Roughly speaking, the relationship between Petri nets and (SPO)-graph grammars can be summarized as in the following table. [3] To get this correspondence we have to restrict the graph grammar matches to injective ones (because in Petri nets tokens can not be "identified"). Note that most of the concepts of Petri nets relate to "abstract" ones for graph grammars. This is because the tokens in the nets have no individuality by assumption. Only in the process semantics tokens gain individuality and thus the correspondence is more immediate (here we consider net processes for P/T-nets as given in [MMS94, MMS96], that are a a refinement of the non-sequential processes of [GR83]). In the paper [KR96] the semantical relationship between nets and grammars was defined having trees as semantical domain (reachability trees in the case of nets and sequential derivation trees in the case of graph grammars). There it is was shown that the semantics is only compatible if we consider abstract graph grammar semantics (that is, derivations up to isomorphism). This matter is explained in more details in [Cor95].

[2] For the formal definitions of the translations shown in the examples of this section see [KR96].

[3] Here we will only compare Petri nets with SPO graph grammars. In the case of grammars representing Petri nets, the DPO and SPO approaches yield corresponding derivations sequences.

	Petri Nets	Graph Grammars
syntax	net	graph grammar
state	set of tokens	(discrete abstract) graph
state change	switching of a transition	(abstract) derivation step
sequences of changes	switching sequence	(abstract) sequential derivation
seq. semantics	set of switching sequences	(abstract) sequential semantics
concurrent history	net process	concurrent derivation
conc. semantics	process semantics	category $\mathbf{ADer_{GG}}$
	unfolding semantics	unfolding semantics

7.3.1 Petri Nets and Parallel Composition

The following example shows the result of applying the pure parallel composition of graph grammars to graph grammars that can be considered as translation of Petri nets. After the example, this composition will be compared with other parallel composition operators of the area of Petri nets.

Example **7.2** Consider nets $N1$ and $N2$ and their corresponding graph grammars $GG1$ and $GG2$ in Figure 7.3. The pure parallel composition of $GG1$ and $GG2$ is the grammar $GG3$, that corresponds to the net $N3$. ☺

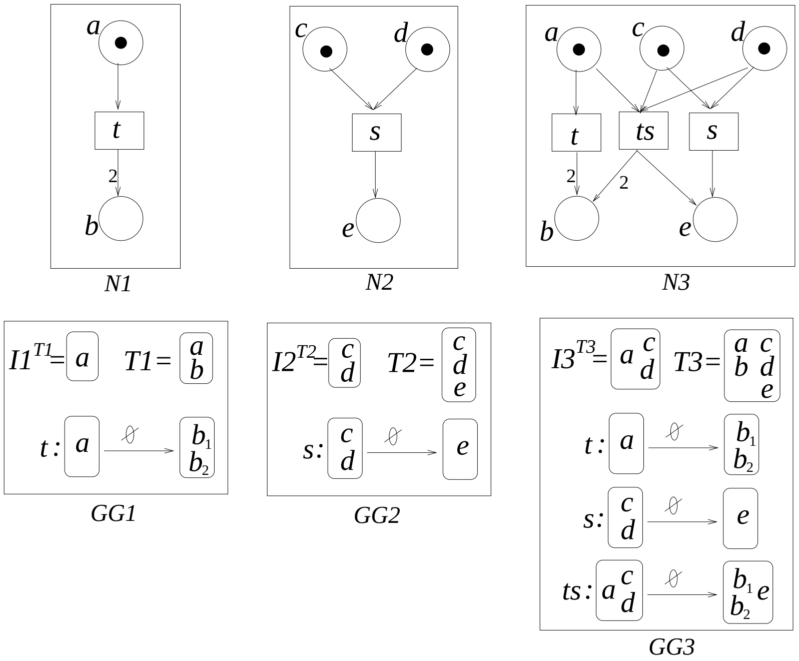


Figure 7.3: Pure Parallel Composition of Petri Nets

This kind of parallel composition corresponds to the parallel composition of nets described (using different frameworks) in [Win87b, WN94, MMS94, Men94]. we will call this composition of Petri nets as net parallel composition. To restrict the possible synchronizations between the transitions of the nets in their net parallel composition, a restriction operator was used in [Win87b]. The restriction operator is a unary operator whose application deletes some transitions of the original net. One can obtain different kinds of compositions of nets based on the net parallel composition followed by an application of the restriction operator (see [Win87b] for more details). Similar constructions for restriction (in a different framework) were presented in [Vog92]. The cooperative parallel composition can be used to perform these two operations (net parallel composition and restriction) in one step, and has an important advantage with respect to the net parallel composition with restriction of [Win87b, Vog92], namely that places (and also initial markings) may be shared. A composition of nets sharing a common subnet will be illustrated in the following example.

Example **7.3** To build a parallel composition of nets with restriction using the cooperative parallel composition, the transitions that shall necessarily be synchronized must be in the interface net. In this example, the interface net is net $N0$ of Figure 7.4 and the transition t of nets $N1$ and $N2$ shall be synchronized in the parallel composition. Moreover, places a of these two nets shall be glued together, and therefore it is also part of $N0$. In this example, we will not draw explicitly the grammars that correspond to the involved nets, although they are used as basis to construct the composition described in this example. Net $N3$ shows the result of the cooperative parallel composition of nets $N1$ and $N2$ with respect to the net $N0$, where the morphisms are indicated by same names for places and transitions. ☺

Analogously to graph grammars, the interface net can be considered as an abstract global description of a system that is specialized in the component nets. The advantages of the cooperative parallel composition for nets are thus the same as for graph grammars: this composition allows sharing of substructures and is compatible with the unfolding semantics (that is also a suitable concurrent semantics for Petri nets – see Sect. 7.3.2).

There are other kinds of composition of nets based on gluing places [Vog92, Val94, PER95] or even subnets [PER95, Men94]. Using arbitrary nets, these composition operators are usually not compatible with semantic models of nets that take the initial marking into account. In [Vog92] and [Men94], strong conditions are imposed to the nets to assure a compositional semantics. In [PER95] nets without initial markings are considered (and then we have a case analogous to graph transformation systems).

7.3.2 Petri Nets and Unfolding Semantics

Petri net processes [GR83, BD87, MMS94] define the deterministic, concurrent behaviours of a net. They are the concurrent correspondent to the switching sequences. The unfolding semantics for nets was introduced in [NPW81] and was based on a kind of Petri nets called occurrence nets, which are, roughly speaking, acyclic, safe nets without backwards conflicts. Now we will discuss the relationship between the notion of occurrence grammar of this thesis (specialized for the case of nets) and occurrence nets. For the case of graph grammars representing Petri nets, the weak conflict relation $\xrightarrow{\#}$ (Def. 5.23) is always symmetric because no items are preserved. This implies that $\overset{\#}{\Longleftrightarrow}=\overset{\#}{\Longrightarrow}$ (Def. 5.25). Let GG be an occurrence

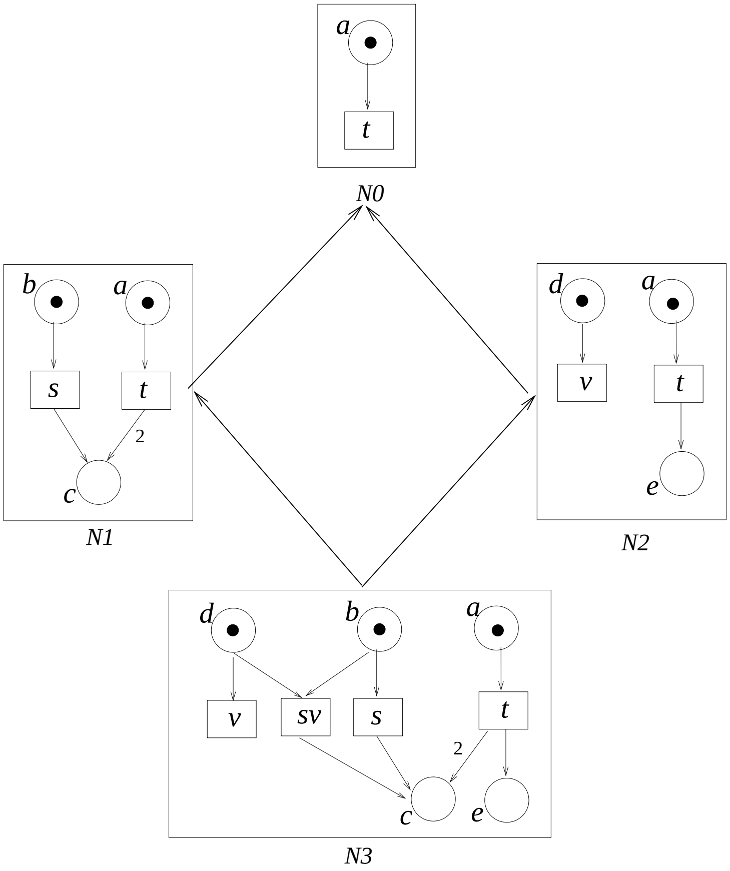


Figure 7.4: Cooperative Parallel Composition of Petri Nets

grammar and N be a net. Then the fact that N is acyclic follows from axioms 1. and 2. of occurrence grammars (Def. 5.29). Safeness and the absence of backward conflicts follows from axiom 4. Axioms 3. and 4. assure that each transition of N has a finite number of causes and that N is free of auto-concurrency, respectively. Axiom 6. imposes an specific shape for the names of places of the occurrence net. This last axiom is originally not an axiom of occurrence nets, but it is also reasonable for the case of nets. It assures that the names of the transitions of an occurrence net are the name of a transition of the original net indexed by the occurrence of this transition. This allows us to relate the occurrence of a transition with the original transition without needing a morphism. The opposite direction, namely that an occurrence net yields via a translation an occurrence graph grammar, holds analogously (provided the names of the transitions have the special form required by axiom 6.).

The construction of the unfolding of a net usually starts with an inductive definition:

1. The initial state of the net is represented by places in the unfolding net. Multiple tokens in one place are represented by multiple places.
2. Add each switchable transition to the actual unfolding net, together with the corresponding pre- and post-conditions (again, multiplicities are represented by multiple places).

At each iteration step we obtain via this construction an approximation of the expected result. Then, if we take the set of all (maybe infinitely many) iteration steps and do a directed colimit, we obtain the (possibly infinite) unfolding of a net. This unfolding is by construction an occurrence net. The unfolding described above was defined formally in [MMS94], where unfolding semantics for arbitrary place/transition nets was introduced. One should notice that in this semantics, tokens are distinguishable. Therefore, the unfolding semantics of graph grammars introduced in this thesis seems to deliver, for the case of grammars representing Petri nets and if only injective matches are considered, the same result as the unfolding of nets defined in [MMS94] This will be illustrated in the following example.

Example **7.4** The unfolding $\mathcal{U}(N)$ of the net N shown in Figure 7.2 is shown in Figure 7.5.

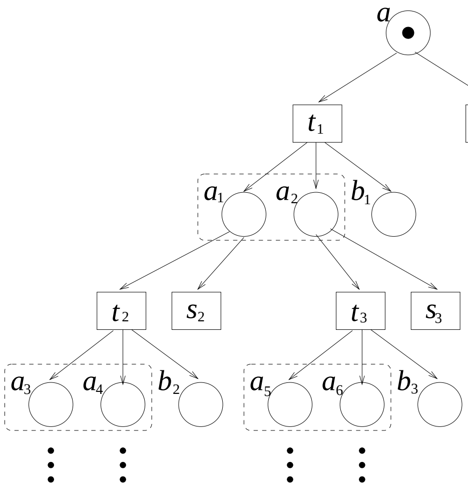
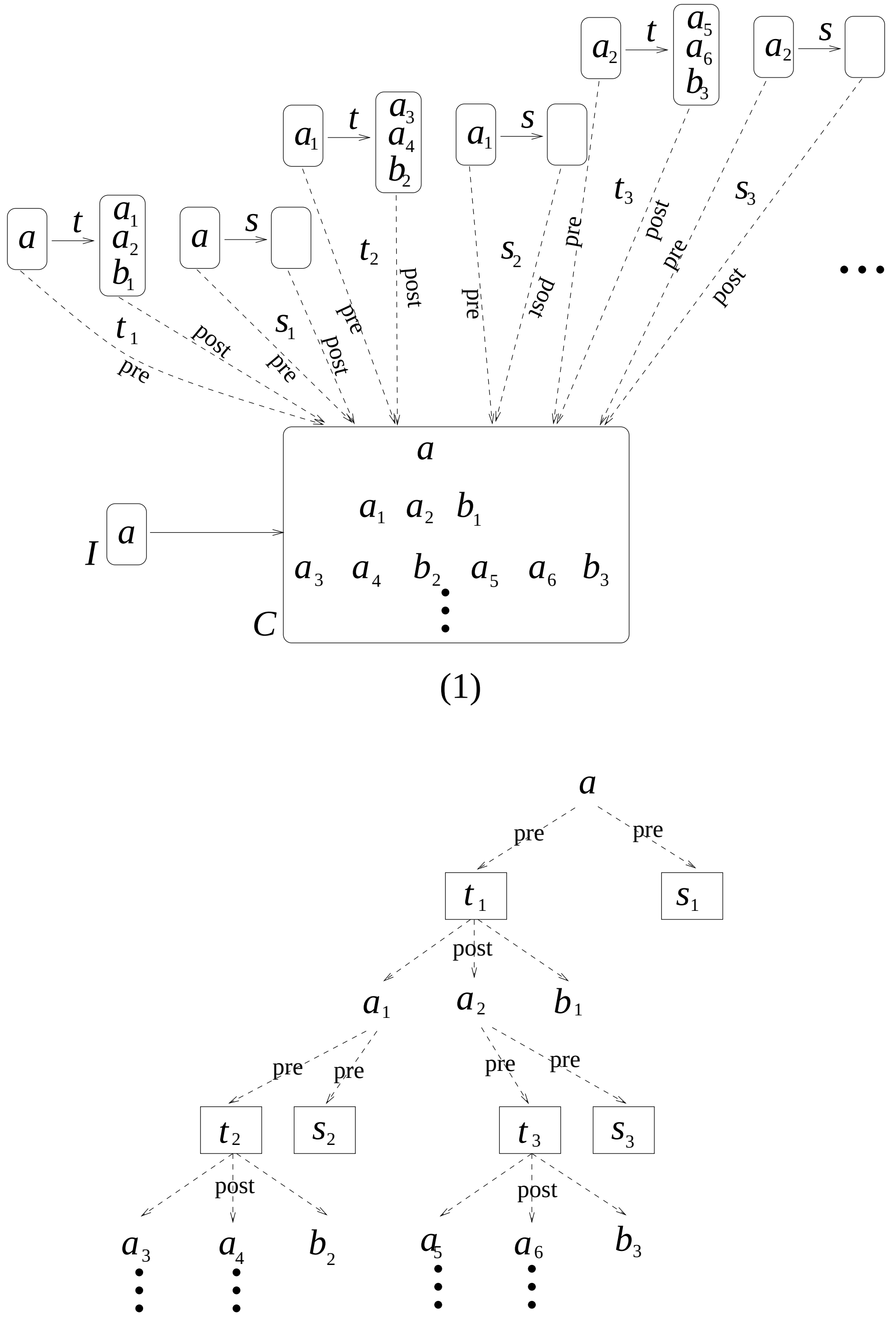


Figure 7.5: Unfolding of N

The unfolding of the graph grammar GG is given in Figure 7.6. The usual graphical representation of the graph grammar unfolding looks like (1): the unfolding consists of an initial graph and set of actions, all typed over the core graph C. The dependencies between the actions are, as in the case of nets, given by the post/pre relationships between them. For example, the action s_2 depends on t_1 because t_1 generates something (in its post-condition) that is needed by s_2 (in its pre-condition). In the case of grammars representing nets, one can give a more explicit graphical representation of these dependencies, as shown in (2), where the actions are drawn "within the core graph". ☺

Besides the cooperative parallel composition for place/transition nets, another possible application of the concepts and results obtained here in the area of Petri nets is by considering the specialization of graph grammars into contextual nets [MR95] (that are nets that allow the preservation of tokens by a transition). It would be interesting to check whether the unfolding semantics and the parallel composition operators yield suitable notions for this kind of nets.



(1)

(2)

Figure 7.6: Unfolding of GG

8

Conclusion

In this thesis we introduced parallel composition and unfolding semantics for graph grammars. The next three points say how these concepts contribute to fulfill the aim of this thesis of providing an approach to parallel composition and unfolding semantics for graph grammars in which the aspects of concurrency and compositionality play a central role.

- *Parallel composition* (Defs. 4.12 and 4.20) allows for the composition of different graph grammars with or without sharing of subparts. The rules of the composed grammar are obtained as (parallel/amalgamated) compositions of the rules of the component grammars, and the initial graph of the composed grammar is obtained by gluing the initial graphs of the components along the initial graph of the interface grammar. Theorems 4.16 and 4.24 assure that the composed graph grammar is syntactically related (via morphisms) to the component grammars, and Theo. 4.11 then yields the result that all derivations of the composed grammar can be translated into derivations of the component grammars, that is, a semantical relationship between the composed grammar and the component grammars is established.

- The *unfolding* of a graph grammar (Def. 6.4 and 6.7) is constructed incrementally and yields a true concurrent, branching structure semantics in which the states as well as the changes of states are explicitly represented. Besides the incremental construction, the unfolding can also be constructed as a suitable gluing of all concurrent derivations of a graph grammar. This result is shown in Theo. 6.9 and gives the relationship between the unfolding and the concurrent semantics of a graph grammar. This result implies also that the unfolding represents exactly all sequential derivations of a graph grammar (because the concurrent derivations represent exactly all sequential derivations). The unfolding of a graph grammar is a special kind of graph grammars, an *occurrence graph grammar* (Def. 5.29 and Prop. 6.6). Therefore, a number of relationships between the actions of the unfolding may be used to reason about the computations described by this unfolding. These relationships are also defined between elements of the core graph of an unfolding, and this allows us to reason about reachable states (graphs) described in this unfolding. The existence of an adjunction (Theo. 6.15) induced by the unfolding construction relating the categories of graph grammar and of (abstract) occurrence graph grammars can be used to show that any (deterministic or non-deterministic) computation of a grammar expressed in terms of an occurrence graph grammar is contained in the unfolding semantics of this grammar. Moreover, it assures that each syntactical relationship between two graph grammars expressed through a graph grammar morphism induces a semantical relationship between the corresponding unfoldings.

- The *compatibility* of the unfolding semantics with the parallel composition operators is given by Theo. 6.18, and is based on the facts that the parallel composition operators correspond to the product and pullback in the category of graph grammars (Theo.4.16 and Theo. 4.24) and that these categorical constructions are preserved by the unfolding functor (assured by Theo. 6.15 and the fact that the unfolding is a right-adjoint). This means that we obtained a framework for the specification of concurrent and reactive systems in which the semantics of the whole system can be obtained from the semantics of its components using suitable syntactical and semantical composition operators.

The concepts of parallel composition and unfolding of graph grammars seem to be very promising. A number of interesting extensions/generalizations that can be done having these concepts as starting points. In the following we list some of them. First, we will consider possibilities to weaken some of the requirements that have been made in this thesis. Topics 2. to 7. are concerned with extensions of this work by using different graph/derivation concepts, structuring and practical applicational aspects. Topic 8. discusses the embedding of this work on the general theory of concurrency. Topics 9. and 10. provide ideas related to Petri nets, and 11. discusses the possibility to define corresponding concepts for DPO grammars as introduced here for SPO grammars.

1. Weaker Requirements

   For constructing and obtaining the results concerned with parallel composition and unfolding, a number of restrictions had to be made. For the well-definedness of the cooperative parallel composition, we considered a safe interface grammar and specialization morphisms. It would be probably possible to drop these requirements if we consider more concrete graph grammar morphisms including the subrule relationships explicitly. For this kind of morphisms, an associative choice of pullbacks may be needed.

   For the unfolding semantics, injective and consuming rules were required. If these requirements are dropped, the axiomatization of occurrence grammars would have to be changed (these axioms are needed to find the concurrent graphs to be used in the next step of an unfolding). Moreover, we used doubly-typed graphs as occurrence graph grammars. It would be possible to use (practically) the same axioms (without the last one) to define a notion of occurrence graph grammars using (simply) typed grammars. However, to construct the unfolding the information about the second type is needed for identifying the concurrent graphs. Maybe it is possible to code this type information into the names of the vertices/edges. This would lead to an unfolding that is closer to the Petri nets unfolding. For graph grammars, the use of the core graph as a typed graph seems to be more natural than to code this type information into the vertices/edges. It would be interesting to find out which are the advantages and disadvantages of each approach.

2. Attributed Graph Grammars

   For the specification of real systems, usually just using typed graphs gives raise to quite complex description of states because everything (including the data types involved in the system) has to be coded graphically. A higher-level representation is obtained in graph grammars by using *attributed graphs* [LKW93], that are graphs equipped with algebraic data types, called attributes. This allows to use variables (and terms) in the rules and therefore reduces considerably the number of rules that are necessary to describe a system. Moreover, this makes the rules easier to understand. In [KR96] it was shown that attributed graph grammars can be transformed into labeled graph grammars.

As attributed graph grammars are particularly interesting for practical applications, it is important to develop theory for this kind of grammars. Therefore, it is a subject of further research to lift the concepts of parallel composition and unfolding defined in this thesis for this kind of grammars.

3. Other Graph Structures

   Not only attributed graph grammars have many practical applications. Other kinds of graphs structures, like hypergraphs, labeled and attributed hypergraphs, labeled graphs, typed attributed graphs, seem to have their own application fields. Therefore it should be interesting to investigate if it is possible to describe the concepts introduced in this thesis in a more general way such that all these kinds of graphs become instances of this general framework. This work can be based on the concept *generalized graph structures* [Kor93, Kor96] or a combination of this with *high-level replacement systems* [EHKP91, EL93a].

4. Module Concept

   The concept of cooperative parallel composition hints on the fact that it is possible to glue two different grammars with respect to a third one in such a way that there is a semantical compatibility. This can be used to define a module concept like the one for algebraic specification developed in [EM90]. For the case of graph grammars, it seems that, instead of using composition operators based on colimits as it is the case for algebraic specifications, composition operators of graph grammar modules shall be defined by limit constructions. In this field there is still a lot of work to be done. First efforts towards a module concept for graph grammars have been done in [EE95]. One of the important extensions that are necessary towards a suitable notion of a module concept is a more general notion of a graph grammar morphism, that allows us to map a rule to a suitable refinement on the target grammar. This kind of morphism can be useful for the actualization of formal parameters, as well as for hiding informations from the export to the body of a module. Probably then a more abstract semantics based on some kind of observability concept for derivations of graph grammars will be adequate, that is, rather than requiring that the formal parameter and the actual parameter have the same semantics we can require that they have the same observable semantics.

5. Methodology for Practical Applications

   The work together with the company Nutec for the development of the specification of the telephone system using graph grammars stressed the lack of a methodology to build a graph grammar specification of a problem. Such a methodology shall contain a comprehensive description of graph grammars and their behaviour (for non-specialists) and a series of steps that shall be followed to solve a problem using graph grammars. This is of great importance such that graph grammars become a well accepted specification method in the industry.

6. Analysis of Graph Grammars

   Another important aspect that shall increase the acceptance of graph grammars as a feasible specification formalism is the existence of analysis methods. In the area of Petri nets, a lot of work have been made in the area of analysis. Basically, there are two kind of analysis: statical and dynamic. Statical analysis is for example the invariant analysis. For graph grammars, statical aspects may follow from the relationships between rules (induced by their overlappings on the type graph). Dynamic analysis is usually based on reachability graphs or unfoldings. By the inductive construction of the unfolding, we believe that many properties can be also shown based on the unfolding for the case of graph grammars. For proving some properties of a Petri net automatically, it is important to reduce the possibly infinite semantical graph to an equivalent finite one (via coverability methods). For graph grammars, such analysis would need a corresponding notion of coverability to make the unfolding of a grammar finite.

7. Application Conditions

   Sometimes, the specification of complex operations require the use of some mechanism to restrict the application of rules, besides the existence of some match. This can be done in graph grammars through *application conditions* [Hec95, HHT96] and *consistency conditions* [Kor94]. An application condition is a condition that has to be satisfied by a morphism from the left-hand side of a rule to an actual graph to become a match. Thus, application conditions restrict the possible matches of a rule to an actual graph. This has an influence at the semantics of a grammar because some derivations that are possible without application conditions become impossible with them. To construct an unfolding semantics for a grammar with application conditions, this conditions have to be checked in step 1. of the construction of the unfolding (Def. 6.7) to find the possible matches. Moreover, if negative conditions shall be checked, the notion of concurrent graph have also to be changed. This will be explained in more details in item 11, where we discuss unfoldings for DPO grammars (that have an inherent application condition, the gluing condition).

8. Structure of Concurrency

   In the introduction of this thesis we already discussed shortly the fact that only causality and conflict relationships may not be enough to tackle adequately with the description of concurrency in some frameworks. In [JK93] an axiomatic model was introduced in order to capture more precisely the concurrent aspects of a system. Like event structures, this model is based on a set of events and relations between them that have to satisfy some conditions. The important relations in this model are a precedence and a weak causality relations. This is done in an abstract way, that is, for a concrete specification formalism like graph grammars, we have to find out what the precedence and weak causality relations mean. Here, we have defined a number of relationships between actions (and also between types) of an occurrence grammar. It is still open in which way these relationships can be seen as suitable interpretations (or instantiations) of the precedence and weak causality relations.

9. Relationships to Petri Nets

   We already discussed in Sect. 7.3 some relationships between graph grammars and Petri nets. It is a subject of future work to establish a formal relationship between the concepts of occurrence grammars and occurrence nets, and unfoldings of grammars and nets. However, we should say here that, due to our choice of morphisms (that in turn was made to allow for parallel composition in the framework of graph grammars), the adjunction we have between the categories of graph grammars and occurrence graph grammars does not specialize to the one described in [MMS94] for Petri nets. Nevertheless, it seems that analogous concepts as developed in this thesis for graph grammars can be interesting also for the case of Petri nets.

10. Algebraic High-Level Nets

   In [KR96] it was shown that algebraic high-level nets [EPR94] can be seen as a special case of attributed graph grammars in the same way that place/transition nets can be seen as a special case of labeled (or typed) graph grammars. Therefore, if the concepts of parallel composition and unfolding can be lifted to attributed graph grammars (see point 2) it is probably possible to use them also for algebraic high-level nets. These parallel composition operators would than allow for a composition of nets with an initial marking that is compatible with a true-concurrency semantics.

11. Parallel Composition and Unfolding of DPO Grammars

   The concept of parallel composition can be probably without problems defined also for DPO graph grammars. But for unfoldings the situation is more complex because of the gluing condition. This has an effect in the constructions of the matches for each step of the unfolding (step 1. of Def. 6.7). In our approach, the it was only necessary to check whether the image of a “match candidate” was a concurrent graph, that is, a subgraph of a reachable graph. This is enough in the SPO approach because there are no extra conditions that have to be checked with respect to the items of the actual graph that are not in the image of the match. For the DPO approach, it would have to be checked whether the “match candidate” fulfills the gluing condition and that its image is a reachable graph (and not a subgraph of it). This means that some extra conditions would be needed in the definition of concurrent graph (Def. 5.33).

A

Notation

Usually the following conventions will be used:

Symbol	**: Meaning**
$dom(f)$	: domain of f
$rng(f)$	: range of f
$f^{\blacktriangledown}$	: domain inclusion
$f!$	: domain restriction
$(f)^{-1}$	: inverse
f^{OP}	: dual morphism to the morphism f
$\rightarrow$	: partial morphism
$\mapsto$	: total morphism
$\rightarrowtail$	: injective morphism
$\twoheadrightarrow$	: surjective morphism
$\hookrightarrow$	: total and injective morphism
`undef`	: undefined value
G^T	: typed graph (G, T^G, T), $t^G : G \mapsto T$
f^T	: typed graph morphism (f, id_T)
$card(S)$	: cardinality of S $(card(S) \in \mathbb{N} \cup \{\omega\})$
A^{∞}	: lists over A
A^*	: finite lists over A
$L1 \bullet L2$	: concatenation of lists
λ	: empty list
σ_i	: ith element of the list σ
$\|\sigma\|$	: number of elements of σ $(\|\sigma\| \in \mathbb{N} \cup \{\omega\})$
$[r]$	: equivalence class
$x \in G$	: $x \in V_G \cup E_G$, if G is a graph

B

Categorical Constructions

B.1 Basic Concepts of Category Theory

- *Category*: A ***category*** $\mathbf{Cat}$ consists of a class $Obj_{\mathbf{Cat}}$ of objects, for each pair $A, B \in Obj_{\mathbf{Cat}}$ a set $Mor_{\mathbf{Cat}}(A, B)$ of morphisms, written $f : A \to B$ for each $f \in Mor_{\mathbf{Cat}}(A, B)$, and a composition $g \circ f : A \to C$ for each pair of morphisms $f : A \to B$ and $g : B \to C$ such that we have

  1. $(h \circ g) \circ f = h \circ (g \circ f)$ for all morphisms f, g, h if at least one side is defined, and
  2. for each object A of $\mathbf{Cat}$ there is a distinguished identity morphism id_A with $f \circ id_A = f$ and $id_A \circ g = g$ whenever the left-hand sides are defined.

- *Isomorphism*: A morphism $f : A \to B$ in a category $\mathbf{Cat}$ is called an ***isomorphism*** provided there exists a morphism $g : B \to A$ such that $g \circ f = id_A$ and $f \circ g = id_B$.

- *Dual Category*: The ***dual or opposite category*** $\mathbf{Cat}^{OP}$ of $\mathbf{Cat}$ has the same class of objects, for each $f : A \to B$ in $\mathbf{Cat}$ we have a dual morphism $f^{OP} : B \to A$ in $\mathbf{Cat}^{OP}$, and the composition in $\mathbf{Cat}^{OP}$ is defined by $f^{OP} \circ g^{OP} = (g \circ f)^{OP}$. For each property $PROP$ in the category $\mathbf{Cat}$, the dual property $COPROP$ is obtained by reversing the arrows of all morphisms.

- *Product*: The (binary) ***product*** of a pair $A1, A2$ of objects in $\mathbf{Cat}$ is an object P together with morphisms $pi : P \to Ai$, for $i = 1, 2$, such that for all objects X and all morphisms $xi : X \to Ai$ in $\mathbf{Cat}$ there is a unique $u : X \to P$ with $pi \circ u = xi$, for $i = 1, 2$.

$$\begin{tikzcd}
A1 & & A2 \\
& P \arrow[ul, "p1"'] \arrow[ur, "p2"] & \\
& X \arrow[u, "u"'] \arrow[uul, bend left, "x1"] \arrow[uur, bend right, "x2"'] &
\end{tikzcd}$$

- *(Binary) Coproduct*: The (binary) ***coproduct*** of a pair $A1, A2$ of objects in **Cat** is an object C together with morphisms $gi : Ai \rightarrow C$, for $i = 1, 2$, such that for all objects X and all morphisms $xi : Ai \rightarrow X$ in **Cat** there is a unique $u : C \rightarrow X$ with $u \circ gi = xi$, for $i = 1, 2$.

$$\begin{tikzcd}
A1 \arrow[dr, "g1"] \arrow[ddr, bend right, "x1"'] & & A2 \arrow[dl, "g2"'] \arrow[ddl, bend left, "x2"] \\
& C \arrow[d, "u"'] & \\
& X &
\end{tikzcd}$$

- *Pullback*: The ***pullback*** of a pair of morphisms $f1 : A1 \rightarrow A0$ and $f2 : A2 \rightarrow A0$ in **Cat** is an object PB together with morphisms $f2^\bullet : PB \rightarrow A1$ and $f1^\bullet : PB \rightarrow A2$ such that for all objects X and all morphisms $xi : X \rightarrow Ai$ in **Cat** with $f1 \circ x1 = f2 \circ x2$ there is a unique $u : X \rightarrow PB$ with $fi^\bullet \circ u = xi$, for $i = 1, 2$.

$$\begin{tikzcd}
A0 & A1 \arrow[l, "f1"'] \arrow[dl, phantom, "(PB)"] & \\
A2 \arrow[u, "f2"] & PB \arrow[l, "f1^\bullet"] \arrow[u, "f2^\bullet"'] & \\
& & X \arrow[ul, "u"'] \arrow[uul, bend right, "x2"'] \arrow[ull, bend left, "x1"] \arrow[u, phantom, "="] \arrow[l, phantom, "="]
\end{tikzcd}$$

- *Pushout*: The ***pushout*** of a pair of morphisms $f1 : A0 \rightarrow A1$ and $f2 : A0 \rightarrow A2$ in **Cat** is an object PO together with morphisms $f2^\bullet : A1 \rightarrow PO$ and $f1^\bullet : A2 \rightarrow PO$ such that for all objects X and all morphisms $xi : Ai \rightarrow X$ in **Cat** with $x1 \circ f1 = x2 \circ f2$ there is a unique $u : PO \rightarrow X$ with $u \circ fi^\bullet = xi$, for $i = 1, 2$.

$$\begin{tikzcd}
A0 \arrow[r, "f1"] \arrow[d, "f2"'] \arrow[dr, phantom, "(PO)"] & A1 \arrow[d, "f2^\bullet"] \arrow[ddr, bend left, "x2"] & \\
A2 \arrow[r, "f1^\bullet"'] \arrow[drr, bend right, "x1"'] & PO \arrow[dr, "u"] & \\
& & X \arrow[u, phantom, "="] \arrow[l, phantom, "="]
\end{tikzcd}$$

- *Functor*: A ***functor*** F : **Cat1** $\rightarrow$ **Cat2** between categories **Cat1** and **Cat2** assigns to each object $A1$ in **Cat1** an object $F(A1)$ in **Cat2** and to each morphism $f : A1 \rightarrow B1$ in **Cat1** a morphism $F(f)$ in **Cat2** such that we have

  1. $F(g \circ f) = F(g) \circ F(f)$, for all $g \circ f$ in **Cat1**, and
  2. $F(id_A) = id_{F(A)}$, for all objects A in **Cat1**

Notation: As in category theory usually the result of a construction consists not only of one object, but also of some morphisms, we will use in th following the notation: if the resulting object is called C, the resulting morphisms are listed in $\rightarrow^C$.

B.2 Categories of Sets

Definition B.1 (Categories Set and SetP) *The categories* **Set** *and* **SetP** *have all sets as objects, and total resp. partial functions as morphisms. Identities are the identity functions and composition is the composition of functions.* ☺

Definition B.2 (Constructions in Set) *Consider the sets Ai, for $i = 1..10$, and the total functions fj, for $j = 1..4$ and $\perp \notin Ai$, for $i = 1..10$.*

$$\begin{tikzcd}
A1 & & A2 \\
& Prod \arrow[ul, mapsto, "p1"'] \arrow[ur, mapsto, "p2"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
A3 \arrow[dr, mapsto, "i1"] & & A4 \arrow[dl, mapsto, "i2"'] \\
& Coprod &
\end{tikzcd}$$

$$\begin{tikzcd}
A5 & A6 \arrow[l, mapsto, "f1"'] \\
A7 \arrow[u, mapsto, "f2"] \arrow[ur, phantom, "="] & PB \arrow[l, mapsto, "f1^\bullet"] \arrow[u, mapsto, "f2^\bullet"']
\end{tikzcd}
\qquad
\begin{tikzcd}
A8 \arrow[r, mapsto, "f3"] \arrow[d, mapsto, "f4"'] \arrow[dr, phantom, "="] & A9 \arrow[d, mapsto, "f4^\bullet"] \\
A10 \arrow[r, mapsto, "f3^\bullet"'] & PO
\end{tikzcd}$$

Then we define the following constructions:

- $P = Prod^{\mathbf{Set}}(A1, A2) = A1 \times A2$ *(cartesian product of sets)*
  $\rightarrow^P = (p1, p2)$, *where* $p1(a, b) = a$ *and* $p2(a, b) = b$ *(projection functions)*

- $C = Coprod^{\mathbf{Set}}(A3, A4) = (A3 \times \{\perp\}) \cup (\{\perp\} \times A4)$ *(disjoint union of sets)*
  $\rightarrow^C = (i1, i2)$, *where* $i1(a) = (a, \perp)$ *and* $i2(b) = (\perp, b)$

- $B = PB^{\mathbf{Set}}(A7 \overset{f2}{\rightarrow} A5 \overset{f1}{\leftarrow} A6) = \{(a, b) \in A7 \times A6 | f2(a) = f1(b)\}$
  $\rightarrow^B = (f2^\bullet, f1^\bullet)$, *where* $f2^\bullet(a, b) = a$ *and* $f1^\bullet(a, b) = b$

- $O = PO^{\mathbf{Set}}(B10 \overset{g4}{\leftarrow} B8 \overset{g3}{\rightarrow} B9) = \{Q \subseteq (B10 \times \{\perp\}) \cup (\{\perp\} \times B9) \mid (a, \perp) \in Q, a = g4(x)$ *implies* $(\perp, g3(x)) \in Q$ *and* $(\perp, b) \in Q, b = g3(x)$ *implies* $(g4(x), \perp) \in Q\}$
  $\rightarrow^O = (g4^\bullet, g3^\bullet)$, *where* $g3^\bullet(a) = \{Q \mid (a, \perp) \in Q\}$ *and* $g4^\bullet(b) = \{Q \mid (\perp, b) \in Q\}$

☺

Theorem B.3 *The constructions defined in Def. B.2 correspond to product, coproduct, pullback and pushout constructions in the category* **Set**, *respectively.* ☺

Proof. See [AHS90]. √

Definition B.4 (Constructions in SetP) *Consider the sets Bi, for $i = 1..10$, and the functions gj, for $j = 1..4$ and $\perp$ is not an element of any of these sets.*

$$\begin{tikzcd}
B1 & & B2 \\
& Prod \arrow[ul, "p1"'] \arrow[ur, "p2"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
B3 \arrow[dr, mapsto, "i1"] & & B4 \arrow[dl, mapsto, "i2"'] \\
& Coprod &
\end{tikzcd}$$

$$\begin{tikzcd}
B5 & B6 \arrow[l, "g1"'] \\
B7 \arrow[u, "g2"] \arrow[ur, phantom, "="] & PB \arrow[l, "g1^\bullet"] \arrow[u, "g2^\bullet"']
\end{tikzcd}
\qquad
\begin{tikzcd}
B8 \arrow[r, "g3"] \arrow[d, "g4"'] \arrow[dr, phantom, "="] & B9 \arrow[d, "g4^\bullet"] \\
B10 \arrow[r, "g3^\bullet"'] & PO
\end{tikzcd}$$

Then we define the following constructions:

- $P = Prod^{\mathbf{SetP}}(B1, B2) = (B1 \times B2) \cup (B1 \times \{\bot\}) \cup (\{\bot\} \times B2)$
  $\rightarrow^P = (p1, p2)$, *where* $p1(a,b) = \begin{cases} a, & \textit{if } a \neq \bot \\ undef, & otherwise \end{cases}$ *and* $p2(a,b) = \begin{cases} b, & \textit{if } b \neq \bot \\ undef, & otherwise \end{cases}$

- $C = Coprod^{\mathbf{SetP}}(B3, B4) = (B3 \times \{\bot\}) \cup (\{\bot\} \times B4)$ *(disjoint union of sets)*
  $\rightarrow^C = (i1, i2)$, *where* $i1(a) = (a, \bot)$ *and* $i2(b) = (\bot, b)$

- $B = PB^{\mathbf{SetP}}(B7 \overset{g2}{\rightarrow} B5 \overset{g1}{\leftarrow} B6) = \{(a,b) \in dom(g2) \times dom(g1) | g2(a) = g1(b)\} \cup$
  $\{(a,b) | a \in B7 \textit{ and } a \notin dom(g2) \textit{ and } b \in B6 \textit{ and } b \notin dom(g1)\} \cup$
  $\{(a,\bot) | a \in B7 \textit{ and } a \notin dom(g2)\} \cup \{(\bot,b) | b \in B6 \textit{ and } b \notin dom(g1)\}$
  $\rightarrow^B = (g2^\bullet, g1^\bullet)$, *where* $g1^\bullet(a,b) = \begin{cases} a, & \textit{if } a \neq \bot \\ \texttt{undef}, & otherwise \end{cases}$ *and* $g2^\bullet(a,b) = \begin{cases} b, & \textit{if } b \neq \bot \\ \texttt{undef}, & otherwise \end{cases}$

- $O = PO^{\mathbf{SetP}}(B10 \overset{g4}{\leftarrow} B8 \overset{g3}{\rightarrow} B9) = \{Q \subseteq (B10 \times \{\bot\}) \cup (\{\bot\} \times B9) \mid (a,\bot) \in Q, a = g4(x) \textit{ implies } (\bot, g3(x)) \in Q \textit{ and } (\bot, b) \in Q, b = g3(x) \textit{ implies } (g4(x), \bot) \in Q\}$
  $\rightarrow^O = (g4^\bullet, g3^\bullet)$, *where* $g3^\bullet(a) = \begin{cases} [(a,\bot)], & \textit{if } [(a,\bot)] \in O \\ \texttt{undef}, & otherwise \end{cases}$
  and
  $g4^\bullet(b) = \begin{cases} [(\bot,b)], & \textit{if } [(\bot,b)] \in O \\ \texttt{undef}, & otherwise \end{cases}$

☺

Theorem B.5 *The constructions defined in Def. B.4 correspond to product, coproduct, pullback and pushout constructions in the category* **SetP**, *respectively.* ☺

Proof. The category **SetP** considered here is a special case of the graph structures defined in [Löw90], namely as a graph structure with respect to a signature containing one sort s and one operation $f : s \rightarrow s$. The constructions defined above are specializations of the constructions in [Löw90] for this special case (see also [Kor93] for set-theoretical characterizations). √

B.3 Categories of Graphs

The constructions on the categories of graphs are based on the constructions of corresponding categories of sets.

Definition B.6 (Constructions in Graph) *Consider the graphs* $Gi = (V_{Gi}, E_{Gi}, s^{Gi}, t^{Gi})$, *for* $i = 1..8$, *and the total graph morphisms* fj, *for* $j = 1..4$ *and* $\bot$ *is not an element of any set of vertices or edges of these graphs.*

$$\begin{tikzcd}
G1 \arrow[dr, "i1"] & & G2 \arrow[dl, "i2"'] \\
& Coprod &
\end{tikzcd}$$

$$\begin{tikzcd}
G3 & G4 \arrow[l, |->, "f1"'] \\
G5 \arrow[u, |->, "f2"] \arrow[ur, phantom, "="] & PB \arrow[l, |->, "f1^\bullet"] \arrow[u, |->, "f2^\bullet"']
\end{tikzcd}
\qquad
\begin{tikzcd}
G6 \arrow[r, |->, "f3"] \arrow[d, |->, "f4"'] \arrow[dr, phantom, "="] & G7 \arrow[d, "f4^\bullet"] \\
G8 \arrow[r, |->, "f3^\bullet"'] & PO
\end{tikzcd}$$

Then we define the following constructions:

- $C = Coprod^{\mathbf{Graph}}(G1, G2) = (Coprod^{\mathbf{Set}}(V_{G1}, V_{G2}), Coprod^{\mathbf{Set}}(E_{G1}, E_{G2}), source, target)$, *where* $source(e1, e2) = (source^{G1}(e1), source^{G2}(e2))$ *and* $target(e1, e2) = (target^{G1}(e1), target^{G2}(e2))$
  $\rightarrow^{C} = (i1, i2)$, *where* $i1 = (i1_V, i1_E)$ *and* $i2 = (i2_V, i2_E)$, *where the components are the inclusions defined by the coproducts in* **Set**.

- $B = PB^{\mathbf{Graph}}(G5 \overset{f2}{\rightarrow} G3 \overset{f1}{\leftarrow} G4) = (PB^{\mathbf{Set}}(V_{G5} \overset{f2_V}{\rightarrow} V_{G3} \overset{f1_V}{\leftarrow} V_{G4}), PB^{\mathbf{Set}}(E_{G5} \overset{f2_E}{\rightarrow} E_{G3} \overset{f1_E}{\leftarrow} E_{G4}), source, target)$,
  where $source(e5, e4) = (source^{G5}(e5), source^{G4}(e4))$ *and* $target(e5, e4) = (target^{G5}(e5), target^{G4}(e4))$
  $\rightarrow^{B} = (f2^\bullet, f1^\bullet)$, *where* $f1^\bullet = (f1^\bullet_V, f1^\bullet_E)$ *and* $f2^\bullet = (f2^\bullet_V, f2^\bullet_E)$

- $O = PO^{\mathbf{Graph}}(G8 \overset{f4}{\leftarrow} G6 \overset{f3}{\rightarrow} G7) = (PO^{\mathbf{Set}}(V_{G8} \overset{f4_V}{\rightarrow} V_{G6} \overset{f3_V}{\leftarrow} V_{G7}), PO^{\mathbf{Set}}(E_{G8} \overset{f4_E}{\rightarrow} E_{G6} \overset{f3_E}{\leftarrow} E_{G7}), source, target)$,
  where $source : E_{PO} \rightarrow V_{PO}$ *is obtained as the unique functions such that* $f3^\bullet_V \circ source^{G8} = source \circ f3^\bullet_E$ *and* $f4^\bullet_V \circ source^{G7} = source \circ f4^\bullet_E$ *and* $target : E_{PO} \rightarrow V_{PO}$ *is obtained analogously.* $\rightarrow^{O} = (f4^\bullet, f3^\bullet)$, *where* $f3^\bullet = (f3^\bullet_V, f3^\bullet_E)$ *and* $f4^\bullet = (f4^\bullet_V, f4^\bullet_E)$
  ☺

Theorem B.7 *The constructions defined in Def. 5.33 correspond to coproduct, pullback and pushout constructions in the category* **Graph**, *respectively.* ☺

Proof. These constructions are special cases from the next ones, and thus can be considered as graph structures, for which, in the case of total morphisms, limits and colimits are constructed componentwise. √

Definition B.8 (Constructions in GraphP) *Consider the graphs* Hi, *for* $i = 1..5$, *and the functions* gj, *for* $j = 1..2$ *and* $\perp$ *is not an element of any of these sets.*

$$\begin{tikzcd}
H1 \arrow[dr, "i1"] & & H2 \arrow[dl, "i2"'] \\
& Coprod &
\end{tikzcd}
\qquad
\begin{tikzcd}
H3 \arrow[r, "g1"] \arrow[d, "g2"'] \arrow[dr, phantom, "="] & H4 \arrow[d, "g2^\bullet"] \\
H5 \arrow[r, "g1^\bullet"'] & PO
\end{tikzcd}$$

Then we define the following constructions:

- $C = Coprod^{\mathbf{GraphP}}(H1, H2) = Coprod^{\mathbf{Graph}}(H1, H2)$
  $\rightarrow^{C} = (i1, i2)$, *where* $i1 = (i1_V, i1_E)$ *and* $i2 = (i2_V, i2_E)$, *where the components are the inclusions defined by the coproducts in* **Set**.

- $O = PO^{\mathbf{GraphP}}(H5 \overset{g2}{\leftarrow} H3 \overset{g1}{\rightarrow} H4) = (V, E, source, target)$,
  where $V = PO^{\mathbf{SetP}}(V_{H5} \overset{g2_V}{\rightarrow} V_{H3} \overset{f1_V}{\leftarrow} V_{H4})$, $E = PO^{\mathbf{SetP}}(E_{H5} \overset{g2_E}{\rightarrow} E_{G3} \overset{g1_E}{\leftarrow} E_{H4}) - \{[(e5, e4)] | (source^{G5}(e5) \notin V \text{ or } target^{G5}(e5) \notin V \text{ or } e5 = \bot) \text{ and } (source^{G4}(e4) \notin V \text{ or } target^{G4}(e4) \notin V \text{ or } e4 = \bot)\}$,
  $source : E_{PO} \rightarrow V_{PO}$ *is obtained as the unique total functions such that* $g2^{\bullet}_V \circ source^{H4} = source \circ g2^{\bullet}_E$ *and* $g1^{\bullet}_V \circ source^{H5} = source \circ g1^{\bullet}_E$ *and* $target : E_{PO} \rightarrow V_{PO}$ *is obtained analogously (such a function exists by the construction of* E *and is unique because of the pushout properties of* V_{PO}*).* $\rightarrow^{O} = (g2^{\bullet}, g1^{\bullet})$, *where* $g1^{\bullet} = (g1^{\bullet}_V, g1^{\bullet}_E)$ *and* $g2^{\bullet} = (g2^{\bullet}_V, g2^{\bullet}_E)$

☺

Theorem B.9 *The constructions defined in Def. B.8 correspond to coproduct and pushout constructions in the category* **GraphP**. ☺

Proof. Analogously to **SetP**, **GraphP** is a special graph structure category, namely to the signature having two sorts V and E and two operations $s : E \rightarrow V$ and $t : E \rightarrow V$. Thus, the constructions here are specializations of the constructions of [Löw90]. √

B.4 Categories of Typed Graphs

Definition B.10 (Constructions in TGraph) *Consider the typed graphs* Gi^{Ti}, *for* $i = 1..8$, *and the morphisms* fj^{tj}, *for* $j = 1..4$ *and* $\bot$ *is not an element of any of the involved sets.*

$$\begin{tikzcd}
G1^{T1} \arrow[dr, maps to, "i1^{t1}"] & & G2^{T2} \arrow[dl, maps to, "i2^{t2}"'] \\
& Coprod &
\end{tikzcd}$$

$$\begin{tikzcd}
G3^{T3} & G4^{T4} \arrow[l, maps to, "f1^{t1}"'] \\
G5^{T5} \arrow[u, maps to, "f2^{t2}"] \arrow[ur, phantom, "="] & PB \arrow[l, maps to, "f1^{\bullet t2^{\bullet}}"] \arrow[u, maps to, "f2^{\bullet t1^{\bullet}}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
G6^{T6} \arrow[r, maps to, "f3^{t3}"] \arrow[d, maps to, "f4^{t4}"'] \arrow[dr, phantom, "="] & G7^{T7} \arrow[d, maps to, "f4^{\bullet t4^{\bullet}}"] \\
G8^{T8} \arrow[r, maps to, "f3^{\bullet t3^{\bullet}}"'] & PO^{T0}
\end{tikzcd}$$

Then we define the following constructions:

- $C = Coprod^{\mathbf{TGraph}}(G1^{T1}, G2^{T2}) = (Coprod^{\mathbf{Graph}}(G1, G2), t, Coprod^{\mathbf{Graph}}(T1, T2))$,
  where $t(x) = \begin{cases} t1 \circ t^{G1}(x), & \text{if } x \in G1 \\ t2 \circ t^{G2}(x), & \text{if } x \in G2 \end{cases}$
  $\rightarrow^{C} = (i1^{t1}, i2^{t2})$, *where* $i1^{t1} = (i1, t1)$ *and* $i2^{t2} = (i2, t2)$, *where* $i1$, $i2$, $t1$ *and* $t2$ *are the inclusions defined by the corresponding coproducts in* **Graph**.

- $B = PB^{\mathbf{TGraph}}(G5^{T5} \overset{f2^{t2}}{\rightarrow} G3^{T3} \overset{f1^{t1}}{\leftarrow} G4^{T4}) = (PB^{\mathbf{Graph}}(G5 \overset{f2}{\rightarrow} G3 \overset{f1}{\leftarrow} G4), t, PB^{\mathbf{Graph}}(T5 \overset{t2}{\rightarrow} T3 \overset{t1}{\leftarrow} T4))$,
  where $t(x) = (t^{G5} \circ f1^{\bullet}(x), t^{G4} \circ f2^{\bullet}(x))$

$\rightarrow^{B}= (f2^{\bullet t2^{\bullet}}, f1^{\bullet t1^{\bullet}})$, where $f1^{\bullet t1^{\bullet}} = (f1^{\bullet}, t1^{\bullet})$ and $f2^{\bullet t2^{\bullet}} = (f2^{\bullet}, t2^{\bullet})$ are the morphisms of pullbacks of the component graphs.

- *$O = PO^{\mathbf{TGraph}}(G8^{T8} \overset{f4^{t4}}{\leftarrow} G6^{T6} \overset{f3^{t3}}{\rightarrow} G7^{T7}) = (PO^{\mathbf{Graph}}(G8 \overset{f4}{\rightarrow} G6 \overset{f3}{\leftarrow} G7), t, PO^{\mathbf{Graph}}(T8 \overset{t4}{\rightarrow} T6 \overset{t3}{\leftarrow} T7))$,*
  where $t : PO \rightarrow TO$ is the unique total graph morphism such that $t \circ f3^{\bullet} = t3^{\bullet} \circ t^{G8}$ and $t \circ f4^{\bullet} = t4^{\bullet} \circ t^{G7}$.

  $\rightarrow^{O}= (f4^{\bullet t4^{\bullet}}, f3^{\bullet t3^{\bullet}})$, where $f3^{\bullet t3^{\bullet}} = (f3^{\bullet}, t3^{\bullet})$ and $f4^{\bullet t4^{\bullet}} = (f4^{\bullet}, t4^{\bullet})$

☺

Theorem B.11 *The constructions defined in Def. B.10 correspond to coproduct, pullback and pushout constructions in the category* **TGraph**, *respectively.* ☺

Proof. This category is in fact a comma category constructed using twice the functor $\mathcal{I}d_{\mathbf{Graph}}$. Constructions in comma categories are done componentwise, provided that the underlying categories allow these constructions and that the corresponding functors preserve them. This is the case for the category **TGraph**, what gives raise to the constructions defined above. √

Definition B.12 (Constructions in TGraphP) *Consider the typed graphs Hi^{Ti}, for $i = 1..5$, and the morphisms gj^{tj}, for $j = 1..2$ and $\perp$ is not an element of any of the involved sets.*

$$\begin{tikzcd}
H1^{T1} \arrow[dr, mapsto, "i1^{t1}"] & & H2^{T2} \arrow[dl, mapsto, "i2^{t2}"'] \\
& Coprod &
\end{tikzcd}
\qquad
\begin{tikzcd}
H3^{T3} \arrow[r, "g1^{t1}"] \arrow[d, "g2^{t2}"'] \arrow[dr, phantom, "="] & H4^{T4} \arrow[d, "g2^{\bullet t2^{\bullet}}"] \\
H5^{T5} \arrow[r, "g1^{\bullet t1^{\bullet}}"'] & PO^{TO}
\end{tikzcd}$$

Then we define the following constructions:

- *$C = Coprod^{\mathbf{TGraphP}}(H1^{T1}, H2^{T2}) = Coprod^{\mathbf{TGraph}}(H1^{T1}, H2^{T2})$*
  $\rightarrow^{C}= (i1^{t1}, i2^{t2})$, where $i1^{t1} = (i1, t1)$ and $i2^{t2} = (i2, t2)$, where $i1$, $i2$, $t1$ and $t2$ are the inclusions defined by the corresponding coproducts in **Graph**.

- *$O = PO^{\mathbf{TGraphP}}(H5^{T5} \overset{g2^{t2}}{\leftarrow} H3^{T3} \overset{g1^{t1}}{\rightarrow} H4^{T4}) = (H, t, T)$,*
  where $T = PO^{\mathbf{GraphP}}(T5 \overset{t2}{\rightarrow} T3 \overset{t1}{\leftarrow} T4)$, H is the biggest subgraph of $PO^{GP}(H5 \overset{g2}{\rightarrow} H3 \overset{g1}{\leftarrow} H4)$ that is completely typed in T and $t : PO \rightarrow TO$ is the unique total graph morphism such that $t \circ g1^{\bullet} = t1^{\bullet} \circ t^{H5}$ and $t \circ g2^{\bullet} = t2^{\bullet} \circ t^{H4}$ (this morphism exists because of the construction of H and is unique due to pushout properties of T).

  $\rightarrow^{O}= (g2^{\bullet t2^{\bullet}}, g1^{\bullet t1^{\bullet}})$, where $g1^{\bullet t1^{\bullet}} = (g1^{\bullet}, t1^{\bullet})$ and $g2^{\bullet t2^{\bullet}} = (g2^{\bullet}, t2^{\bullet})$

☺

Theorem B.13 *The constructions defined in Def. B.12 correspond to coproduct and pushout constructions in the category* **TGraphP**. ☺

Proof. Similar to **TGraph**, constructions in **TGraphP** can be obtained componentwise, followed by a "totalization construction" in the case of pushouts. For example, in the pushout construction, first we constructed the pushout objects H and T, and then took everything from H that didn't have a type in T (because types may have been deleted by the pushout of types). This way of constructing colimits is due to the fact that this category can be defined as a generalized graph structure category (GGS category) [Kor96]. These categories are a generalization of comma categories in view of partial morphisms. The fact that these constructions exist in **TGraphP(T)** is based on the fact that they exist in **GraphP** and that the functor $\mathcal{I}d_{\mathbf{GraphP}}$ preserves them. $\surd$

and the inclusion functor Besides the constructions given above, the construction of pushouts preserving the type component are very important because the computation units of (typed) graph grammars will be defined based on them. Thus, although these pushouts can be considered as special cases of the constructions above, we'll give them explicitly in the next definition.

Definition B.14 (Pushouts in TGraphP(T)) *Consider the typed graphs Gi^T, for $i = 1..3$ and the partial (typed) graph morphisms $f1^T$ and $f2^T$.*

$$\begin{tikzcd}
G1^T \arrow[r, "f1^T"] \arrow[d, "f2^T"'] \arrow[dr, phantom, "="] & G2^T \arrow[d, "f2^{\bullet T}"] \\
G3^T \arrow[r, "f1^{\bullet T}"'] & PO^T
\end{tikzcd}$$

Then we define the following construction

- $O = PO^{\mathbf{TGraphP(T)}}(G3^T \overset{f2^T}{\leftarrow} G1^T \overset{f1^T}{\rightarrow} G2^T) = (PO^{\mathbf{GraphP}}(G3 \overset{f2}{\rightarrow} G1 \overset{f1}{\leftarrow} G2), t, T)$*, where* $t : PO \rightarrow T$ *is the unique total graph morphism such that* $t \circ f1^\bullet = t^{G3}$ *and* $t \circ f2^\bullet = t^{G2}$.

  $\rightarrow^O = (f2^{\bullet T}, f1^{\bullet T})$*, where* $f1^{\bullet T} = (f1^\bullet, id_T)$ *and* $f2^{\bullet T} = (f2^\bullet, id_T)$

☺

Remark. *Pushouts in* **TGraph(T)** *are special cases of pushouts in* **TGraphP(T)**. ☺

Proposition B.15 *The construction defined in Def. B.14 is the pushout in the category* **TGraphP(T)**. ☺

Proof. The category **TGraphP(T)** can be defined as a GGS category using the functors $\mathcal{I}d_{\mathbf{GraphP}}$ and $\mathcal{I}nc_{TP} : \mathbf{T} \rightarrow \mathbf{GraphP}$. As the only morphism in **T** is total, pushouts here are also defined componentwise (the totalization construction is not necessary because no types are deleted). $\surd$

B.5 Categories of Double-Typed Graphs

Definition B.16 (Pushouts in $\mathbf{DTGraphP}(\mathbf{TG^T})$) *Consider the double-typed graphs $Gi^{TG\nearrow T}$, for $i = 1..3$, and the double-typed graph morphisms $g1^{TG\nearrow T}$ and $g2^{TG\nearrow T}$. Then we define the following construction:*

$$O = PO^{\mathbf{DTGraphP}(\mathbf{T})}(G3^{TG\nearrow T} \overset{g2^{TG\nearrow T}}{\longleftarrow} G1^{TG\nearrow T} \overset{g1^{TG\nearrow T}}{\longrightarrow} G2^{TG\nearrow T})$$

$= (PO^{\mathbf{TGraphP}(\mathbf{T})}(G3^T \overset{g2^T}{\rightarrow} G4^T \overset{g1^T}{\leftarrow} G2^T), t, TG^T)$, where $t : PO^T \rightarrow TG^T$ is the unique total graph morphism such that $t \circ g1^{\bullet T} = t^{G3}$ and $t \circ g2^{\bullet T} = t^{G2}$. $\rightarrow^O = (g2^{\bullet TG\nearrow T}, g1^{\bullet TG\nearrow T})$, where $g1^{\bullet TG\nearrow T} = (g1^{\bullet T}, id_{TG^T})$ and $g2^{\bullet TG\nearrow T} = (g2^{\bullet T}, id_{TG^T})$ ☺

Proposition B.17 *The construction defined in Def. B.16 is the pushout in the category* $\mathbf{DTGraphP}(\mathbf{T})$. ☺

Proof. Analogous to the proof of Prop. B.15. $\surd$

C

Proofs

Lemma C.1 *Let $g^{T1} : G1^{T1} \rightarrow H1^{T1}$ be total (injective, isomorphism). Then $\mathcal{T}_f(g^{T1}) = g'^{T2}$ is also total (injective, isomorphism).* ☺

Proof. For the diagrams of this proof see the proof of Prop. 3.10. Remind that $g2 = g2! \circ (g2^{\blacktriangledown})^{-1}$, $g2^{\blacktriangledown}$ is total and injective and $g2!$ is total.

1. *Let $g1^{T1}$ be total:* In this case, $dom(g1) = G1$ and $g1^{\blacktriangledown} = id_{G1}$. The pullback construction (2) yields $dom(g2) = G2$ and $g2^{\blacktriangledown} = id_{G2}$. Therefore, $g2$ must be total.

2. *Let $g1^{T1}$ be injective:* In this case $g1!$ is injective. By Prop. 3.10 we have that (3) is a pullback. Injectivity is inherited by pullbacks of total graph morphisms and therefore $g2!$ is also injective. This implies that $g2$ is injective.

3. *Let g^{T1} be an isomorphism:* In this case $g1^{\blacktriangledown}$ and $g1!$ are isomorphisms. As (2) and (3) are pullbacks, $g2^{\blacktriangledown}$ and $g2!$ are also isomorphisms. This implies that $g2$ is an isomorphism. $\surd$

Lemma C.2 *Let $f : T2 \rightarrow T1$ be an injective (typed) graph morphism, $G1^{T1} \in$* **TGraphP(T1)***, and $G2^{T2} = \mathcal{T}_f(G1^{T1})$. Then if $rng(t^{G1}) \subseteq rng(f)$ then $G1 \cong G2$, and the same holds for morphisms.* ☺

Proof. If f is injective, $f! : dom(f) \rightarrow T1$ is also injective. Pullback of total graph morphisms inherit injectivity and thus $f!^G$ is also injective. It is total because it is a pullback morphism of total morphisms. By hypothesis, $rng(t^{G1}) \subseteq rng(f)$. As $f!$ is the domain restriction of f, $rng(f) = rng(f!)$ and therefore $rng(t^{G1}) \subseteq rng(f!)$. This means that for all $x \in G1$ there is $t \in dom(f!)$ such that $t^{G1}(x) = f!(t)$. Thus by pullback construction of $G2$, there is $y \in G2$ such that $f!^G(y) = x$ and $t^{G1\bullet}(y) = t$, what implies that $f!^G$ is surjective and thus an isomorphism.

Let g be a morphism with type $T1$ and $\mathcal{T}_f(g) = g'^{T2}$. Above we proved that the retyping of the source and target objects of g with respect to the morphism f are isomorphic to the original ones. As the retyped morphism g' commute with g using the retyping morphisms of the objects (that are isomorphisms), g and g' are isomorphic. $\surd$

Proposition 3.12

Proof. Let $\mathcal{T}_f(x1^{T1}) = x2^{T2}$, for $x \in \{L, R, G, H, r, m, r^\bullet, m^\bullet\}$. Pushouts in the category **TGraphP(T1)** are constructed componentwise in **GraphP** and $\mathbf{Id_{T1}}$ (see Def. B.14). Therefore (3) is a pushout in **GraphP**. As pushouts in **TGraphP(T2)** are also constructed componentwise in **GraphP** and $\mathbf{Id_{T2}}$, it is enough to prove that (4) is a pushout (because $T2$ is trivially the pushout of the second component). By assumption we have that $r1$ is injective and $m1$ is total. As (3) is a pushout in **GraphP**, $r1^\bullet$ is also injective (but $m1^\bullet$ is not necessarily total). Lemma C.1 yields that $r2$ and $r2^\bullet$ are injective and $m2$ is total.

Let P together with morphisms $x1 : G2 \to P$ and $x2 : R2 \to P$ be the pushout of $r2$ and $m2$ in **GraphP**. We have to prove that there exists an isomorphism $u : P \to H2$ such that (5) and (6) commute. As $\mathcal{T}_f$ is a functor, diagram (4) commutes. Therefore, there is a universal morphism $u : P \to H2$ such that (5) and (6) commute. It remains to show that u is an isomorphism:

$$\begin{tikzcd}
L1 \arrow[r, tail, "r1"] \arrow[d, "m1"'] \arrow[dr, phantom, "(3)"] & R1 \arrow[d, "m1^\bullet"] \\
G1 \arrow[r, tail, "r1^\bullet"'] & H1
\end{tikzcd}$$

$$\begin{tikzcd}
L2 \arrow[r, tail, "r2"] \arrow[d, "m2"'] \arrow[dr, phantom, "(4)"] & R2 \arrow[d, "m2^\bullet"] \arrow[ddr, bend left, "x2"] & \\
G2 \arrow[r, tail, "r2^\bullet"'] \arrow[drr, bend right, "x1"'] & H2 \arrow[dr, phantom, "(6)", near start, shift left=3ex] & \\
 & \arrow[u, phantom, "(5)"] & P \arrow[ul, "u"']
\end{tikzcd}$$

$$\begin{tikzcd}
L1 \arrow[rr, tail, "r1"] \arrow[dd, "m1"'] & & R1 \arrow[dd, "m1^\bullet"] & \\
 & L2 \arrow[ul, "f!^L"] \arrow[rr, tail, "r2"] \arrow[dd, "m2"] \arrow[ur, phantom, "(9)"] \arrow[l, phantom, "(7)"] & & R2 \arrow[ul, "f!^R"'] \arrow[dd, "m2^\bullet"] \arrow[dl, phantom, "(10)"] \\
G1 \arrow[rr, tail, "r1^\bullet"] & & H1 & \\
 & G2 \arrow[ul, "f!^G"] \arrow[rr, tail, "r2^\bullet"'] \arrow[ur, phantom, "(8)"] & & H2 \arrow[ul, "f!^H"']
\end{tikzcd}$$

1. Total: Let $e \in P$. As P is a pushout object, there must be a pre-image for e in $G2$, $R2$ or both. Assume there is $e1 \in G2$ and $x1(e1) = e$. If $e1 \in rng(m2)$ then there exists $e2 \in rng(r2)$ such that $x2(e2) = e$ because P is a pushout object. Let $e0 \in L2$ such that $m2(e0 = e1)$ and $r2(e0) = e2$. As (4), (5) and (6) commute, if $r2^\bullet(e1)$ is defined then $m2^{bullet}(e2) = r2^\bullet(e1)$. In this case, $e \in dom(u)$ because (5) and (6) commute. If $r2^\bullet(e1) = \mathtt{undef}$ then $m2^\bullet(e2) = \mathtt{undef}$ as well (because (4), (5) and (6) commute), and this would imply that $e \notin dom(u)$. But this is not possible because (8) and (10) commute and (3) is a pushout. If $e1 \notin rng(m2)$ then there is no $a0 \in L1$ such that $m1(a0) = f!^G(e1)$ because (7) is a pullback. As (3) is a pushout in **GraphP**, either $(f!^G(e1) \in dom(r1^\bullet))$ or $(f!^G(e1) \notin dom(r1^\bullet)$ and $a0$ is an edge having $v0 \in G1$ as source or target vertex and $v0 \in rng(m1)$ and $v0 \notin dom(r1\bullet)$. In the first case $(f!^G(e1) \in dom(r1^\bullet))$, as (8) commutes, $e1 \in dom(r2^\bullet)$. Then commutativity of (5) yields that $r2^\bullet(e1) = u(e)$. In the second case, as $dom(r2^\bullet)$ is obtained as pullback of $r1^\blacktriangledown$ and $f!^G$, we must have that $e1 \notin dom(r2^\bullet)$. Let $v1$ be the pre-image of $v0$ in $G2$, i.e., $f!^G(v1) = v0$. As (7) is a pullback, $v1 \in rng(m2)$. As (9) commutes, $v1 \notin dom(r2)$. Therefore, as P is a pushout object, $v1 \notin dom(x1)$, what implies that $e1 \notin dom(x1)$. But this means that there is no $e \in P$such that $x1(e1) = e$, what contradicts the hypothesis.

   Now assume that $e2 \in R2$ and $x2(e2) = e$. Analogously to the first case, if $e2 \in rng(r2)$ then $e \in dom(u)$ because (4), (5), (6), (8) and (10) commute and (3) is a pushout. If $e2 \notin rng(r2)$ then there is no $a1 \in dom(r1)$ such that $r1!(a1) = f!^{H\bullet}(e2)$ because $dom(r2)$ is a pullback of $r1!$ and $f!^H$ (see Prop. 3.10). As (3) is a pushout, either

$e1 \in dom(m1^\bullet)$ or ($e1 \notin dom(m1^\bullet)$ and $a1$ is an edge whose target or source vertex $v1 \in dom(r1)$ is in deletion conflict with another vertex $v1'$, i.e., $m1(v1) = m1(v1')$ and $v1' \notin dom(r1)$). In the first case ($e1 \in dom(m1^\bullet)$) commutativity of (10) and (6) assure that $u(e) = m2^\bullet(e2)$. In the second case, we obtain that there must be $v2, v2' \in L2$ such that $f!^G(v2) = v1$ and $f!^G(v2') = v1'$ because $v1$ and $v1'$ must have the same type in the type graph (as they are identified by $m1$) and $L2$ is obtained as a pullback with respect to the type morphism (that must map the type of $v1$ and $v1'$ because the type of $e1$ is mapped). As (7) is a pullback, $m2(v1') = \texttt{undef}$. As P is a pushout of $m2$ and $r2$, we have that $e \notin P$, what contradicts the hypothesis.

2. Injective: Let $e, e' \in P$, $e \neq e'$. Assume per absurd that $u(e) = u(e')$. As (5) and (6) commute, we have 3 cases:

   (a) $e, e' \in rng(x1)$: As $x1$ is injective, this would mean that there are two different items in $G2$ that are identified via $r2^\bullet$, and this is not possible because $r2^\bullet$ is injective.

   (b) $e, e' \in rng(x2) - rng(x1)$: In this case as P is a pushout object we must have that there are $e2, e2' \in R2$ such that $x2(e2) = e$ and $x2(e2') = e'$ (obviously in this case we have that $e2 \neq e2'$). Moreover, as (6) commutes and $u(e) = u(e')$ by assumption we must have that $m2^\bullet(e2) = m2^\bullet(e2')$. This implies that the types of $e2$ and $e2'$ must be the same (because they are identified by a morphism an morphisms preserve types). As P is a pushout and $e, e' \notin rng(x1)$ we conclude that $e2, e2' \notin rng(r2)$, what implies that $e2, e2' \notin rng(r2!)$. As the $dom(r2)$ is constructed as a pullback of $f!^R$ and $dom(r1)$, we conclude that there are no $f!^H(e2), f!^H(e2') \notin rng(r1!)$. As (3) is a pushout and (10) commutes, $f!^H(e2), f!^H(e2') \in dom(m1^\bullet)$. If $f!^H(e2) = f!^H(e2')$ then $e2$ and $e2'$ must have different types because $R2$ is obtained as a pullback of t^{R1} and $f!$. But this contradicts the fact that they have the same type (see above). If $f!^H(e2) \neq f!^H(e2')$ then we must have that $m1^\bullet(f!^H(e2)) \neq m1^\bullet(f!^H(e2'))$ because (3) is a pushout. As (10) commutes, we must have that $m2^\bullet(e2) \neq m2^\bullet(e2')$, and this contradicts the hypothesis. Thus, commutativity of (6) yields that $u(e) \neq u(e')$.

   (c) $e \in rng(x1)$,$e' \in rng(x2) - rng(x1)$: In this case, we must have that the pre-image $e2'$ of e' under $x2$ in not in the range of $r2$, i.e., $e2' \notin rng(r2)$. Then analogously to the previous case, we have that $f!^R(e2') \notin rng(r1)$ and as (3) is a pushout, $m1^\bullet \circ f!^R(e2') \notin rng(r1)$ and this element must be different from any other in the image of $r1^\bullet$. Commutativity of (8) yields then that there can be no element $e1 \in G2$ such that $r2^\bullet(e1) = m2^\bullet(e2)$. This implies that $u(e) \neq u(e')$.

3. Surjective: Let $e3 \in H2$. Assume that $e3 \notin rng(u)$. As (3) is a pushout, either $f!^H(e3) \in rng(m1^\bullet)$ or $f!^H(e3) \in rng(r1^\bullet)$ or both. Analogously to the previous cases, we then conclude that $e3 \in rng(m2^\bullet)$ or $e3 \in rng(r2^\bullet)$ or both. As (5) and (6) commute, we conclude that $e3 \in rng(u)$.

$\surd$

Proposition 3.13

Proof. Let $f : T2 \rightarrow T1$ and $g : T3 \rightarrow T2$ be morphisms in **GraphP**. We have to show that $\mathcal{T}_{g\circ f} \cong \mathcal{T}_g \circ \mathcal{T}_f$. This means that for each object $I1^{T1} \in$ **TGraphP(T1)**, there is an isomorphism $u_I^{T3} : \mathcal{T}_{g\circ f}(I1^{T1}) \rightarrow \mathcal{T}_g \circ \mathcal{T}_f(I1^{T1})$ that is compatible with the application of the corresponding functors to morphisms (i.e., there is a natural isomorphism between these functors). We will first define u_I^{T3} and then show that the compatibility requirement i satisfied.

u_I^{T3} : Let $I1^{T1} \in$ **TGraphP(T1)**, $\mathcal{T}_f(I1^{T1}) = I2^{T2}$, $\mathcal{T}_g \circ \mathcal{T}_f(I1^{T1}) = I3^{T3}$ and $\mathcal{T}_{g\circ f}(I1^{T1}) = I3'^{T3}$ (the morphisms obtained via the corresponding retyping constructions are shown in the diagrams below). As f and g are morphisms, the diagrams (1) and (2) below are pullbacks (by definition of the retyping construction). Diagram (3) is a pullback by construction (the intersection of domains of partial morphisms – see [Ken91]). The retyping construction of $I2$ also assures that $t2 = f^\blacktriangledown \circ t1^\bullet$. Thus, there is a universal morphism $u : I3 \rightarrow dom(g \circ f)$ induced by pullback (3) such that $t2^\bullet = y \circ u$ and $t1^\bullet \circ i2 = x \circ u$. Let (1) be the square defined by the morphisms $i2$,$t1^\bullet$, u and x. Obviously, (4) commutes. As (2) and (3) are pullbacks and (4) commutes, (4) is also a pullback. As (1) and (4) are pullbacks, (1)+(4) is also a pullback. As $I3$ and $I3'$ are pullback objects of the same diagram, we conclude that $I3 \cong I3'$, $t1'^\bullet \cong t2^\bullet$ and $i3 \cong i1 \circ i2$. Therefore, $I3^{T3} \cong I3'^{T3}$. Let $u_I^{T3} : I3^{T3} \rightarrow I3'^{T3}$ be the unique isomorphism such that $i1 \circ i2 \circ u_I = i3$ and $t3 \circ u_I = t3'$

$$\begin{tikzcd}
 & & & & T1 \\
I1 \arrow[urrrr, "t1"] & & & & \\
 & & (1) & & dom(f) \arrow[uu, "f!"'] \arrow[dll, "f^\blacktriangledown"] \\
I2 \arrow[uu, "i1"] \arrow[rr, "t2"'] \arrow[urrrr, "t1^\bullet"] & & T2 & & \\
 & (2) & (4) & (3) & dom(g \circ f) \arrow[uu, "x"'] \arrow[dll, "y"] \\
I3 \arrow[uu, "i2"] \arrow[rr, "t2^\bullet"'] \arrow[urrrr, "u"] & & dom(g) \arrow[uu, "g!"] & &
\end{tikzcd}$$

Compatibility requirement: Let $r1^{T1} : L1^{T1} \rightarrow I1^{T1}$ be a morphism in **TGraphP(T1)**, $\mathcal{T}_f(r1^{T1}) = r2^{T2}$, $\mathcal{T}_g \circ \mathcal{T}_f(r1^{T1}) = r3^{T3}$ and $\mathcal{T}_{g\circ f}(r1^{T1}) = r3'^{T3}$. We have to show that the a pair of isomorphisms $u_L : L3 \rightarrow L3'$ and $u_I : I3 \rightarrow I3'$ makes (5) commute. Let $f!^{r1}$ and $f!^{r2}$ be the pullback morphisms obtained by the domain constructions of $r1$ and $r2$, respectively (see Def. 3.8).

$$\begin{tikzcd}
L1 \arrow[r, "r1"] \arrow[dr, phantom, "(5)"] & I1 \\
L2 \arrow[u, mapsto, "u^L"] \arrow[r, "r2"'] & I2 \arrow[u, mapsto, "u^I"']
\end{tikzcd}
\qquad
\begin{tikzcd}
L1 \arrow[dr, mapsto, "f!^{L1}"] \arrow[drr, phantom, "(8)"] & & dom(r1) \arrow[ll, hook, "r1^{\blacktriangledown}"'] \arrow[rr, mapsto, "r1!"] \arrow[d, hook, "f!^{r1}"] \arrow[drr, phantom, "(9)"] & & I1 \arrow[dl, mapsto, "f!^{I1}"'] \\
\arrow[r, phantom, "(10)"] & L & dom(r) \arrow[l, hook, "r^{\blacktriangledown}"] \arrow[r, mapsto, "r!"'] & I & \arrow[l, phantom, "(11)"] \\
L2 \arrow[uu, mapsto, "u^L"] \arrow[ur, mapsto, "f!^{L2}"'] \arrow[urr, phantom, "(6)"] & & dom(r2) \arrow[ll, hook, "r2^{\blacktriangledown}"] \arrow[rr, mapsto, "r2!"'] \arrow[u, mapsto, "f!^{r2}"'] \arrow[urr, phantom, "(7)"] & & I2 \arrow[uu, mapsto, "u^I"'] \arrow[ul, mapsto, "f!^{I2}"]
\end{tikzcd}$$

Let diagram (14) be the pullback of $f!$ and t^L with pullback object $L2$ and diagram (15) be the pullback of $f!$ and t^R with pullback object $R2$. Assume that (5) doesn't commute. Then we have three cases:

1. There is $e \in L2$such that $r1 \circ u_L(e) = e3$ and $u_I \circ r2(e) = \texttt{undef}$:
   As u_I is total, $r2(e) = \texttt{undef}$,i.e., $e \notin dom(r2)$. As (6) is a pullback we have that $f!^{L2}(e) \notin dom(r)$. As (10) commutes and (8) is a pullback we conclude that $u_L(e) \notin dom(r1)$. this means that $r1 \circ u_I(e) = \texttt{undef}$. But this contradicts our hypothesis.
2. There is $e \in L2$such that $r1 \circ u_L(e) = \texttt{undef}$ and $u_I \circ r2(e) = e4$:
   Analogous to the first case.
3. There is $e \in L2$such that $r1 \circ u_L(e) = e3$, $u_I \circ r2(e) = e4$ and $e3 \neq e4$:
   Let $e1 = r1^{\blacktriangledown} \circ u_L(e)$ and $e2 = r2^{\blacktriangledown}(e)$. Due to the commutativity of (6)–(11) we have that $f!^{I1} \circ r1 \circ u^L(e) = f!^{I1} \circ u^I \circ r2(e)$. As $f!^{I1}$ is total and $r1 \circ u^L(e) \neq u^I \circ r2(2)$ by assumption, the only possibility is that $f!^{I1}(e3) = f!^{I1}(e4)$. The graph $I1$ is obtained as a retyping of I with respect to$f$ (pullback (13)). Therefore the only possibility to have items identified by f^{I1} is that these items have different types in $dom(f)$, and these types are identified by $f!$. As $I2$ is also the retyping of I with respect to f, there must be also two items in $I2$ with different types that are identified in $f!^{I2}$. As u^I is an isomorphism, there can be only one way to map these items to $I1$ such that this mapping is compatible with the type $T2$. This implies that $e3 = e4$ (because if they are different, they must have different types). But this contradicts our hypothesis that $e3 \neq e4$.

Thus, we conclude that (5) commutes.

$$\begin{tikzcd}
& L \arrow[r, mapsto, "t^L"] \arrow[dr, phantom, "(12)"] & T1 \arrow[dr, phantom, "(13)"] & I \arrow[l, mapsto, "t^I"'] & \\
\arrow[r, phantom, "(5)"] & L1 \arrow[u, mapsto, "f!^{L1}"] \arrow[r, mapsto, "t^{L1}"] & dom(f) \arrow[u, mapsto, "f!"'] & I1 \arrow[u, mapsto, "f!^{I1}"'] \arrow[l, mapsto, "t^{I1}"'] & \arrow[l, phantom, "(6)"] \\
L2 \arrow[uur, mapsto, bend left, "f!^{L2}"] \arrow[ur, mapsto, "u_L"] \arrow[urr, mapsto, bend right, "t^{L2}"'] \arrow[urr, phantom, "(16)"] & & & & I2 \arrow[uul, mapsto, bend right, "f!^{I2}"'] \arrow[ul, mapsto, "u_I"'] \arrow[ull, mapsto, bend left, "t^{I2}"] \arrow[ull, phantom, "(17)"]
\end{tikzcd}$$

$\surd$

Lemma C.3 *Let $T1 \overset{i1}{\to} T12 \overset{i2}{\leftarrow} T2$ be a coproduct in* **GraphP**, *(1) and (2) commute in* **Graph**. *Then $L1 \overset{l1}{\to} L \overset{l2}{\leftarrow} L2$ is a coproduct in* **GraphP** *iff (1) and (2) are pullbacks in*

Graph.

$$\begin{tikzcd}
L1 \arrow[drr, hook, "l1"] \arrow[ddd, "t1"'] & & & & L2 \arrow[dll, hook', "l2"'] \arrow[ddd, "t2"] \\
& & L \arrow[ddd, "t12"'] & & \\
& (1) & & (2) & \\
T1 \arrow[drr, hook, "i1"'] & & & & T2 \arrow[dll, hook', "i2"] \\
& & T12 & &
\end{tikzcd}$$

☺

Proof. ⇐: *Let (1) and (2) be pullbacks.*

L is a coproduct iff $l1$ and $l2$ are total, injective, jointly surjective and $rng(l1) \cap rng(l2) = \emptyset$.

1. As (1) and (2) are pullbacks in **Graph** and $i1$ and $i2$ are total and injective, $l1$ and $l2$ are total and injective.
2. Assume that $l1$ and $l2$ are not jointly surjective. Then there must be an element $z \in L$such that $x \notin rng(l1)$ and $x \notin rng(l2)$. As t is total, $t(x) \in T12$. As $i1$ and $i2$ are jointly surjective, there is $x1 \in T1$such that $i1(x1) = t(x)$ or $x2 \in T2$such that $i2(x2) = t(x)$. Assume we have the first case ($x1$). As (1) is a pullback and $i1(x1) = t(x)$, there must be $e1 \in L1$such that $l1(e1) = x$ and $t1(e1) = x1$. Thus, $x \in rng(l1)$. Analogously, if we use $x2$ we obtain that $x \in rng(l2)$. Therefore, $l1$ and $l2$ are jointly surjective.
3. Assume that $rng(l1) \cap rng(l2) \neq \emptyset$. Then there is $x \in L$such that $l1(x1) = x = l2(x2)$. This implies that $t \circ l1(x1) = x = t \circ l2(x2)$. Thus $i1 \circ t1(x1) = i2 \circ t2(x2)$ (because (1) and (2) commute). $i1$ and $i2$ are coproduct morphisms and thus $rng(i1) \cap rng(i2) = \emptyset$. As $t1$ and $t2$ are total, there can not be any $x1$ and $x2$ such that $i1 \circ t1(x1) = i2 \circ t2(x2)$. As t is total, this implies that $l1(x1)$ must be different from $l2(x2)$. But this contradicts the hypothesis, and thus, $rng(l1) \cap rng(l2) = \emptyset$.

⇒: *Let L be a coproduct.*

$$\begin{tikzcd}
L1 \arrow[d, "u"] \arrow[drr, hook, "l1"] \arrow[ddd, bend right, "t1"'] & & & L2 \arrow[dl, hook', "l2"'] \\
PB \arrow[rr, "x1"'] \arrow[dd, "x2"] & & L \arrow[ll, "u'"'] \arrow[ddd, "t12"] & \\
& (3) & & \\
T1 \arrow[drr, hook, "i1"'] & & & \\
& & T12 &
\end{tikzcd}$$

Assume that $L1$ is not the desired pullback object, but PB together with morphisms $x1 : PB \to T1$ and $x2 : PB \to L$ (diagram (3)). As (1) commutes, there is an universal morphism $u : L1 \to PB$ such that $l1 = x1 \circ u$ and $t1 = x2 \circ u$. As $i1$ is injective, $x1$ is also injective and can be inverted. Thus we get a partial morphism $(x1)^{-1} \circ l2 : L2 \rightarrow PB$ that is also injective and is surjective and $(x1)^{-1} \circ x1 = id_L$ (the inverse is not true because $(x1)^{-1}$ may be partial). As L is a coproduct in **GraphP**, there is and universal morphism $u' : L \rightarrow PB$ such that $(x1)^{-1} \circ l2 = u' \circ l2$ and $u = u' \circ l1$. From $x1 \circ u = l1$ we derive that $(x1)^{-1} \circ x1 \circ u = (x1)^{-1} \circ l1$ and thus $u = (x1)^{-1} \circ l1$. Obviously, $(x1)^{-1} \circ l2 = (x1)^{-1} \circ l2$, and thus by uniqueness of universal morphisms we conclude that $(x1)^{-1} = u'$. Totality of u follows from the fact that it is the universal morphism induced by a pullback in **Graph**. As $u = (x1)^{-1} \circ l1$, and $l1$ and $(x1)^{-1}$ are injective, u is also injective. As $(x1)^{-1}$ is surjective, u is surjective. Thus u is an isomorphism and the square (1) is a pullback.

That square (2) is also a pullback can be shown analogously.

√

Lemma C.4 *Let $T1 \overset{i2^\bullet}{\rightarrow} T3 \overset{i1^\bullet}{\leftarrow} T2$ be the pushout in* **GraphP***of $T1 \overset{i21}{\leftarrow} T0 \overset{i2}{\rightarrow} T2$, and $i1$ and $i2$ be total and injective. Let (5)-(6) be pullbacks in* **Graph** *and (2)-(3) commute in* **Graph**. *Then (6) is a pushout in* **GraphP** *iff (2)-(3) are pullbacks in* **Graph**. ☺

$$\begin{tikzcd}
 & T0 \arrow[dl, "i1"'] \arrow[dr, "i2"] & \\
T1 \arrow[dr, "i2^\bullet"'] & (1) & T2 \arrow[dl, "i1^\bullet"] \\
 & T3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L0 \arrow[dl, "l1"'] \arrow[dr, "l2"] \arrow[ddd, bend left, "t0"] & \\
L1 \arrow[dr, "l2^\bullet"] \arrow[ddd, "t1"'] & & L2 \arrow[dl, "l1^\bullet"'] \arrow[ddd, "t2"] \\
(4) & L3 \arrow[ddd, bend right, "t3"'] & (5) \\
(2) & T0 \arrow[dl, "i1"] \arrow[dr, "i2"'] & (3) \\
T1 \arrow[dr, "i2^\bullet"'] & & T2 \arrow[dl, "i1^\bullet"] \\
 & T3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & L0 \arrow[dl, "l1"'] \arrow[dr, "l2"] & \\
L1 \arrow[dr, "l2^\bullet"'] & (6) & L2 \arrow[dl, "l1^\bullet"] \\
 & L3 &
\end{tikzcd}$$

Proof. ⇐: *Let (2)-(3) be pullbacks.*

Let (7) be a pushout with pushout object P. Then we have to show that there is an isomorphism $u : P \rightarrow L3$ and that (8) and (9) commute.

$$\begin{tikzcd}
 & L0 \arrow[dl, "l1"'] \arrow[dr, "l2"] & \\
L1 \arrow[dr, "x1"] \arrow[ddr, bend right, "l1^\bullet"'] & (7) & L2 \arrow[dl, "x2"'] \arrow[ddl, bend left, "l2^\bullet"] \\
(8) & P \arrow[d, "u"'] & (9) \\
 & L3 &
\end{tikzcd}$$

The square (6) commutes because all other squares (1)–(5) commute. As (6) commutes and (7) is a pushout, there is an universal morphism $u : P \to L3$ such that (8) and (9) commute. $l1^\bullet$ and $l2^\bullet$ are total and injective because $i1^\bullet$ and $i2^\bullet$ are total and injective and (2) and (3) are pullbacks. Analogously, $l1$ and $l2$ are total and injective because $i1$ and $i2$ are injective and (5) and (6) are pullbacks. As (7) is a pushout, $x1$ and $x2$ are also total and injective. As (8) and (9) commute, $x1$, $x2$, $l1^\bullet$ and $l2^\bullet$ are total and injective, and $x1$ and $x2$ are pushout morphisms (and thus together surjective), u must be total and injective. Thus it remains to show that u is surjective.

Assume that u is not surjective. Then there is $x \in L3$ such that $x \notin rng(u)$. This implies that $x \notin rng(l1^\bullet)$ and $x \notin rng(l2^\bullet)$ because (8) and (9) commute. Let $t3(x) = t'''$. As $L2$ is a pullback, there is no $t'' \in T2$ such that (i)$i1^\bullet(t'') = t3(x)$. As $L1$ is a pullback, there is no $t' \in T1$ such that (ii)$i2^\bullet(t') = t3(x)$. As $i1^\bullet$ and $i2^\bullet$ are pushout morphisms, they are together surjective. Thus as $t''' \in T3$, either there is $t'' \in T2$ such that $i1^\bullet(t'') = t'''$ or there is $t' \in T1$ such that $i1^\bullet(t') = t'''$. But these two cases contradict (i) and (ii) resp. Thus $x \in rng(u)$, what implies that u is surjective, and thus an isomorphism.

$\Rightarrow$: *Let (6) be a pushout.*
Analogously to the proof of the corresponding part of Lemma C.3 we obtain that (2) and (3) are pullbacks (one just have to substitute coproduct by pushout in the proof). $\checkmark$

Proposition 4.8

Proof. This proof will be done in 2 steps: first we prove that $norm(l) \in Steps^\infty_{GG}$ (if the corresponding requirements are fulfilled), and then that $norm(l) \in SDer_{GG}$ if $l \in SDerI_{GG}$.

1. Let $l \in StepsI^\infty_{GG}$ and $l = \lambda$ or $OUT_{i-1} = IN_i$ for all $i = 2..|l|$. The proof that $norm(l) \in Steps^\infty_{GG}$ will be done by induction on the length of l:

   Induction Basis: $|l| = 0$
   In this case, $norm(l) = \lambda$, that is a trivial sequence of GG.

   Induction Basis: $|l| = 1$
   Let $l = s1$. Here we may have two cases:

   (a) r_{s1} is iso: Then $norm(l) = \lambda$ and thus $norm(l) \in Steps^\infty_{GG}$.

   (b) r_{s1} is not iso: Then by definition of $StepsI_{GG}$, $s1 \in Steps_{GG}$. As $norm(l) = s1$, we conclude that $norm(l) \in Steps_{GG} \subseteq Steps^\infty_{GG}$.

   Induction Hypothesis: For l such that $|l| \leq n$, $norm(l) \in Steps^\infty_{GG}$ and $OUT_{i-1} = IN_i$, for all $i = 2..|norm(l)|$.

   Induction Step: $|l| = n + 1$
   Let $l = s1 \bullet l1$. We have two cases:

   (a) r_{s1} is not iso: In this case, $norm(l) = s1 \bullet norm(l1)$. As r_{s1} is not iso, $s1 \in Steps_{GG}$. By induction hypothesis $norm(l1) \in Steps^\infty_{GG}$ because $|l1| = |l| - 1 = n + 1 - 1 = n$. Thus by definition of $Steps^\infty_{GG}$, $norm(l) = s1 \bullet l1 \in Steps^\infty_{GG}$. Moreover, $l1$ satisfies the condition: $norm(l1) = \lambda$ or $OUT_i - 1 = IN_i$, for all $i = 2..|norm(l1)|$. If $norm(l1) = \lambda$, OUT_{s1} is the last derivation step of the normalized sequence, and then we are ready. If $norm(l1) \neq \lambda$ we have to show that $OUT_{s1} = IN_{norm(l1)}$. By hypothesis, $OUT_{s1} = IN_{s2}$ where $l1 = s2 \bullet l2$. Here we have 2 cases:

i. r_{s2} is not iso: Then $norm(l1) = s2 \bullet norm(l2)$, what means that $norm(l) = s1 \bullet s2 \bullet norm(l2)$. Then we obviously have that $OUT_{s1} = IN_{norm(l1)} = IN_{s2}$.

ii. r_{s2} is iso: Then $norm(l1) = norm(s3' \bullet l3)$, where $l2 = s3 \bullet l3$. By definition of $s3'$, we have that $IN_{s3'} = IN_{s2}$, and thus $IN_{norm(l1)} = IN_{s2} = OUT_{s1}$.

(b) r_{s1} is iso: We have 3 cases:

i. $l1 = \lambda$: Then $norm(l) = \lambda \in Steps^{\infty}_{GG}$.

ii. $l1 = s2 \bullet l2$ and $l2 = \lambda$: Then $norm(l1) = norm(s2')$. Assume that $s2' \in StepsI_{GG}$. Then $norm(s2') \in Steps^{\infty}_{GG}$ because $|s2'| = 1$. Thus it remains to show that $s2' \in StepsI_{GG}$. As $r_{s2'} = r_{s2}$ by definition, $r_{s2'}$ is either an isomorphism or a rule of GG. By definition, the derivation step of $s2'$ is $S2'$ =(1)+(2) as shown below. As (2) is the derivation step of $s2$, it is a pushout. (1) is a trivial pushout: it commutes by definition of $m_{s2'}$ and id and $r^{\bullet}_{s1}$ are isomorphisms. Thus (1)+(2)= $S2'$ is also a pushout. As $m_{s2'}$ is total by definition, we conclude that $s2' \in StepsI_{GG}$.

$$\begin{tikzcd}
L_{s2} \arrow[r, maps to, "id"] \arrow[d, maps to, "m_{s2'}"'] \arrow[dr, phantom, "(1)"] & L_{s2} \arrow[r, "r_{s2}"] \arrow[d, maps to, "m_{s2}"] \arrow[dr, phantom, "(2)"] & R_{s2} \arrow[d] \\
IN_{s1} \arrow[r, maps to, "r^{\bullet}_{s1}"'] & OUT_{s1} = IN_{s2} \arrow[r] & OUT_{s2}
\end{tikzcd}$$

iii. $l1 = s2 \bullet l2$ and $l2 \neq \lambda$: Then $norm(l) = norm(s2' \bullet l2)$. Analogously to the previous case, we conclude that $s2' \in StepsI_{GG}$. Thus, $s2' \bullet l2 = l1' \in StepsI_{GG}$. By construction, $l1'$ fulfills the condition that $OUT_i = IN_{i+1}$ ($l2$ fulfills this requirement and $OUT_{s2'} = OUT_{s2} = IN_{l2}$). Moreover, $|l1'| = |l| - 1$ because the step $s1$ was removed. Thus by induction hypothesis we have that $norm(l1') \in StepsI^{\infty}_{GG}$, what implies that $norm(l) \in StepsI^{\infty}_{GG}$.

2. Let $l \in SDerI_{GG}$. We have to show that $norm(l) \in SDer_{GG}$:

(a) $norm(l) = \lambda$: Trivial.

(b) $norm(l) \neq \lambda$: Here we have to show that the initial graph is the one of GG and output graphs of steps are the same as the input graph of the subsequent steps.

$$\begin{tikzcd}
L_{s1} \arrow[r, "r_{s1}"] \arrow[d, maps to, "m_{s1}"'] & R_{s1} \arrow[dr] & L_{s2} \arrow[r, "r_{s2'}"] \arrow[d, maps to] \arrow[dll, maps to, bend right, "m_{s2'}"'] & R_{s2} \arrow[d, maps to] & & L_{si} \arrow[r, "r_{si'}"] \arrow[d, maps to] \arrow[dlllll, maps to, bend right=15, "m_{si'}"'] & R_{si} \arrow[d, maps to] \\
IN_{s1} \arrow[rr] & & OUT_{s1} = IN_{s2} \arrow[r] & OUT_{s2} & \cdots & OUT_{si-1} = IN_{si} \arrow[r] & OUT_{si}
\end{tikzcd}$$

Initial Graph : By hypothesis, $l \in SDerI_{GG}$, what implies that $IN_l = IN_{GG}$. Let si be the first step of l such that r_{si} is not an isomorphism. Then, by def. of $norm$, $norm(l) = norm(s2' \bullet l2) = norm(s3' \bullet l3) = \ldots = norm(si' \bullet li)$. By definition of $norm$, $IN_{s2'} = IN_l$, $IN_{s3'} = IN_{s2'}$ and so on. Thus $IN_{si'} = IN_l$. As r_{si} is not iso, $norm(si' \bullet li) = si' \bullet norm(li)$. Therefore we conclude that $IN_l = IN_{GG}$ is the initial graph of $norm(l)$.

Intermediate Graphs : By hypothesis, $OUT_{i-1} = OUT_i$ for all $i = 2..|l|$. This proof may be reduced to the proof that the output graph OUT_{k-1} of an already normalized part of l is equal to the initial graph of the next normalized step. This can be shown analogously to the first item, and thus we conclude that $OUT_{j-1} = IN_j$ for all $j = 2..|norm(l)|$.

$$\begin{tikzcd}
L_{sk} \arrow[r, "r_{sk}"] \arrow[d, |->, "m_{sk}"'] & R_{sk} \arrow[dr] & L_{sk+1} \arrow[r, "r_{s2k+1'}"] \arrow[d, |->] & R_{sk+1} \arrow[d, |->] & & L_{si} \arrow[r, "r_{si'}"] \arrow[d, |->] \arrow[dlll, |->, bend right=15, "m_{si'}"'] & R_{si} \arrow[d, |->] \\
IN_{sk} \arrow[rr] & & OUT_{sk} = IN_{sk+1} \arrow[r] & OUT_{sk} & \cdots & OUT_{si-1} = IN_{si} \arrow[r] & OUT_{si}
\end{tikzcd}$$

√

Lemma C.5 *Let $I1 \hookrightarrow I12 \hookleftarrow I2$ and $T1 \hookrightarrow T12 \hookleftarrow T2$ below be coproducts in* **GraphP**, *and squares (1)–(4) be pullbacks in* **Graph**. *Then square (5) is also a pullback in* **Graph**, *where u_T and u_I are the universal morphisms induced by the coproducts $T12$ and $I12$, respectively.*

$$\begin{tikzcd}
 & & I1 \arrow[dll, hook', "i1"'] \arrow[drr, |->, "x1"] \arrow[ddd, |->, bend left=15, "t^{I1}"] & & \\
I12 \arrow[ddd, |->, "t^{I12}"'] & & & & IX \arrow[ddd, |->, "t^{IX}"] \\
 & (3) & I2 \arrow[ull, hook, "i2"'] \arrow[urr, |->, "x2"'] \arrow[ddd, |->, bend right=15, "t^{I2}"'] & (4) & \\
 & (1) & T1 \arrow[dll, hook', "t1"] \arrow[drr, |->, "y1"'] & (2) & \\
T12 & & & & TX \\
 & & T2 \arrow[ull, hook, "t2"] \arrow[urr, |->, "y2"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
I12 \arrow[r, |->, "u_I"] \arrow[d, |->, "t^{I12}"'] \arrow[dr, phantom, "(5)"] & IX \arrow[d, |->, "t^{IX}"] \\
T12 \arrow[r, |->, "u_T"'] & TX
\end{tikzcd}$$

☺

Proof. As coproducts are special pushouts in **GraphP**, this is a special case of Lemma C.6 that will be proved next. √

Lemma C.6 *Let squares (1) and (2) be pushouts in* **GraphP**, *where i1, i2,t1 and t2 are total and injective, and squares (3)–(8) be pullbacks in* **Graph**. *Then square (9) is also a pullback in* **Graph**, *where u_T and u_I are the universal morphisms induced by the pushouts $T3$ and $I3$, respectively.*

$$\begin{tikzcd}
 & & I1 \arrow[drr, hook, "i3"] \arrow[drrrr, mapsto, "x1"] \arrow[ddd, mapsto, bend left, "t^{I1}"] & & & & \\
I0 \arrow[urr, hook, "i1"] \arrow[drr, hook, "i2"] \arrow[ddd, mapsto, "t^{I0}"] & & & & I3 \arrow[rr, mapsto, two heads, "u_I"] \arrow[ddd, mapsto, "t^{I3}"] & & IX \arrow[ddd, mapsto, "t^{IX}"] \\
 & (5) & I2 \arrow[urr, hook, "i4"] \arrow[urrrr, mapsto, "x2"] \arrow[ddd, mapsto, bend right, "t^{I2}"'] & (6) & & (8) & \\
 & (3) & T1 \arrow[drr, hook, "t3"] \arrow[drrrr, mapsto, "y1"] & (4) & & (7) & \\
T0 \arrow[urr, hook, "t1"] \arrow[drr, hook, "t2"] & & & & T3 \arrow[rr, mapsto, two heads, "u_T"'] & & TX \\
 & & T2 \arrow[urr, hook, "t4"] \arrow[urrrr, mapsto, "y2"'] & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
I3 \arrow[rr, mapsto, "u_I"] \arrow[dd, mapsto, "t^{I3}"'] & & IX \arrow[dd, mapsto, "t^{IX}"] \\
 & (9) & \\
T3 \arrow[rr, mapsto, "u_T"'] & & TX
\end{tikzcd}$$

☺

Proof. We have to show that (9) commutes and has the universal property.

Commutativity : Follows from the commutativity of (5)–(8), the universal properties of u_I and u_T and the fact that $i3$ and $i4$ (resp. $t3$ and $t4$) are jointly surjective (because (1) and (2) are pushouts).

Universal Property : Let (10) below be a pullback in **Graph**. As (9) commutes and u_I and t^{I3} are total, there is an universal morphism $u : I3 \to PB$ such that (11) and (12) commute. We'll show that u is an isomorphism in **Graph**, i.e., total,injective and surjective. Totality follows from the fact that it is an universal morphism in **Graph**.

$$\begin{tikzcd}
I3 \arrow[dr, mapsto, "u_T"] \arrow[drr, mapsto, bend left, "u_I"] \arrow[ddr, mapsto, bend right, "t^{I3}"'] & & \\
 & PB \arrow[r, mapsto, "z1"'] \arrow[d, mapsto, "z2"'] \arrow[ul, phantom, "(12)"] \arrow[dl, phantom, "(11)"] & IX \arrow[d, mapsto, "t^{IX}"] \\
 & T3 \arrow[r, mapsto, "u_T"'] \arrow[ur, phantom, "(10)"] & TX
\end{tikzcd}$$

Injectivity : Let $a1, a2 \in I3$, $a1 \neq a2$. As $I3$ is a pushout, there are 3 possibilities to be considered (the others may be derived from these ones):

1. $a1, a2 \in rng(i3)$: In these case, there are $b1, b2 \in I1$ such that $i3(b1) = a1$ and $i3(b2) = a2$. Obviously, $b1 \neq b2$. As (8) is a pullback, $b1$ and $b2$ must be mapped to different items of IX or $T1 \hookrightarrow T3$ or both. These 3 cases lead to the conclusion that there must be $d1, d2 \in PB$, $d1 \neq d2$ such that $u(a1) = d1$ and $u(a2) = d2$ (using the facts that (6) is a pullback and (11) and (12) must commute).
2. $a1, a2 \in rng(i4)$: Analogous to the first case.
3. ($a1 \in rng(i3)$ and $a1 \notin rng(i4)$) and ($a2 \in rng(i4)$ and $a2 \notin rng(i3)$) : In these case, there are $b \in I1, c \in I2$ such that $i3(b) = a1$ and $i4(c) = a2$. As (5) and (6) are pullbacks we conclude that $t^{I3}(a1) \neq t^{I3}(a2)$. Therefore, there

are $d1, d2 \in PB$, $d1 \neq d2$ such that $z2(d1) = t^{I3}(a1)$ and $z2(d2) = t^{I3}(a2)$. As (11) commutes, we conclude that $u(a1) \neq u(a2)$.

4. $a1, a2 \in rng(i3)$ and $a1, a2 \in rng(i4)$:

Subjectivity : Let $d \in PB$ and $d \notin rng(u)$. As (11) commutes, $z2(d) \neq rng(t^{I3})$, what implies that $z2(d) \neq rng(t^{I3} \circ i3) = rng(t3 \circ t^{I2}$. Analogously we obtain that $z2(d) \neq rng(t4 \circ t^{I1})$. But this means that $z2(d) \neq rng(t3)$ and $z2(d) \neq rng(t4)$ what is a contradiction because $t3$ and $t4$ are pushout morphisms and thus together surjective. Therefore we conclude that $d \in rng(u)$, i.e., u is surjective.

$\surd$

Theorem 4.24

Proof. The proof that u is a graph grammar morphism is analogous to the corresponding proof for products. However there is an additional case where one rule $nrx \in NX$ is mapped to rules $nr1 \in N1, n2 \in N2$. Then there must be a common subrule $nr0 \in N0$, $nr0 = s1_N(nr1)$ and $nr0 = s2_N(nr2)$. By construction of the cooperative parallel composition, in this case there is a rule $nr3 \in N3$ such that $nr1 = p1_N(nr3)$ and $nr2 = p2_N(nr3)$. Let the rule morphisms corresponding to $nr0, nr1, nr2, nr3, nrx$ be $r0, r1, r2, r3, rx$, respectively. By construction $r3$ is the amalgamation of $r1$ and $r2$ with respect to $r0$.We must show now that $r3$ is a subrule of a rx.

All these rules are related by a kind of subrule relation which are formally captured by composing the subrule morphism with the corresponding retyping morphism. Recall that, in general, the resulting morphism is not unique. However specialization morphisms $s1, s2$ assure that there are unique subrule relations $s1^r = (s1L, s1R) : r0 \to r1$ and $s2^r = (s2L, s2R) : r0 \to r2$. The morphisms $pi^r = (piL, piR) : ri \to r3$, for $i = 1, 2$, are uniquely given by the amalgamation construction. In the following, many properties will only be shown for one of the components (usually the left hand side) the other then holds by analogy.

By explicitly referring also to the left and right hand components of these morphisms we obtain the commuting squares depicted in the following diagram in the category **TGraphP**.

$$\begin{tikzcd}
L0 \arrow[rrr, tail, "r0"] \arrow[dr, hook, "s2L"] \arrow[dd, hook, "s1L"'] & & & R0 \arrow[dr, hook, "s2R"] \arrow[dd, hook, "s1R"'] & & \\
& L2 \arrow[rrr, tail, "r2"] \arrow[dd, hook, "p2L"] \arrow[dddr, hook, bend left, "x2L"] & & & R2 \arrow[dd, hook, "p2R"] \arrow[dddr, hook, bend left, "x2R"] & \\
L1 \arrow[rrr, tail, "r1"] \arrow[dr, hook, "p1L"] \arrow[ddrr, hook, bend right, "x1L"'] & & & R1 \arrow[dr, hook, "p1R"] \arrow[ddrr, hook, bend right, "x1R"'] & & \\
& L3 \arrow[rrr, tail, "r3"'] \arrow[dr, dashed, "uL"] & (1) & & R3 \arrow[dr, dashed, "uR"] & \\
& & LX \arrow[rrr, tail, "rx"'] & & & RX
\end{tikzcd}$$

First we will show that the mapping of rule names induces a commuting diagram

In order to see this, consider an element $o0 \in L0$ for which there are two different images $ox = x1L \circ s1L(o0)$ and $zx = x2L \circ s2L(o0)$ in LX, i.e., $ox \neq zx$. Let $to0, tox, tzx$ be their types. By construction $x1_T \circ s1_T = x2_T \circ s2_T$ which yields a common type $tox = x1_T \circ s1_T(to0) = tzx \in TX$. Thus the retyping of LX with respect to $x1_T \circ s1_T$ will yield two different items $ox0$ and $zx0$ in $\mathcal{T}_{x1 \circ s1}(LX)$. By construction $\mathcal{T}_{x1 \circ s1}(rx)$ is a subrule of $r0$ which contradictory requires that both items $ox0$ and $zx0$ have different preimages in $L0$. This implies that $x1L \circ s1L = x2L \circ s2L$ and analogous for right-hand sides.

By using this commutativity and the fact that the squares with tips in $L3$ and $R3$ have been constructed as pushouts, we infer the existence of a pair of universal morphisms $uL : L3 \to LX$ and $uR : L3 \to LX$ which uniquely satisfy $uL \circ p1L = x1L$, $uL \circ p2L = x2L$, $uR \circ p1R = x1R$ and $uR \circ p2R = x2R$. As $p1L$ and $p2L$ are total, injective and jointly surjective we conclude that (1) commutes.

This leaves to show that (1) gives indeed raise to a subrule diagram, i.e., diagram (1) can be splitted into two subdiagrams (2) and (3) such that (3) is a retyping of rx with respect to u_T and (2) is a subrule relation between $r3$ and the retyped rule $\mathcal{T}_{u_T}(rx)$.

$$\begin{tikzcd}
L3 \arrow[r, tail, "r3"] \arrow[d, hook, "suL"'] \arrow[dr, phantom, "(2)"] & R3 \arrow[d, hook, "suR"] \\
\mathcal{T}_{u_T}(LX) \arrow[r, tail, "\mathcal{T}_{u_T}(rx"] \arrow[d, mapsto, "TuL"'] \arrow[dr, phantom, "(3)"] & \mathcal{T}_{u_T}(RX) \arrow[d, mapsto, "TuR"] \\
LX \arrow[r, tail, "rx"'] & RX
\end{tikzcd}$$

So we first perform the retyping of rx with respect to u_T which yields the commuting diagram (3) with the retyped rule $\mathcal{T}_{u_T}(rx) : \mathcal{T}_{u_T}(LX) \to \mathcal{T}_{u_T}(RX)$ and total morphisms $TuL : \mathcal{T}_{u_T}(LX) \to LX$ and $TuR : \mathcal{T}_{u_T}(RX) \to RX$. The retyping is essentially obtained from constructing the pullback of $(t^{LX}, u_T!)$ and symmetrically of $(t^{RX}, u_T!)$. Due to the pullback property of $\mathcal{T}_{u_T}(LX)$ we obtain a unique morphism suL such that $(u_T^{\blacktriangledown})^{-1} \circ t^{L3} = t^{LX\bullet} \circ suL$ and $uL = TuL \circ suL$. We proceed by showing that suL is total and injective. The corresponding arguments for suR are analogous. We therefore omit them.

Total: Consider an element $o3 \in L3$ with type $to3 = t^{L3}(o3)$. Each type element in $T3$ has a preimage in $T1$ or $T2$ ($p1_T$ and $p2_T$ are jointly surjective by pushout construction). Assume that $to3$ has a preimage $tol \in T1$, i.e., $u_T(tol) = to3$. By construction of $r3$ there is a preimage $o1 \in L1$ of $o3$ with type tol. Moreover, by the definition of subrules it is required that $o1$ has an image in the retyping $\mathcal{T}_{x1_T}(LX)$ and thus $o1 \in dom(x1L)$ because retyping morphisms are total. Let $x1L(o1) = ox$. As uL is a universal morphism it satisfies $x1L = uL \circ p1L$. So we infer $o3 \in dom(uL)$ which directly implies that suL must be total. Analogously, we obtain the same result if we assume that $to3$ has a preimage in $T2$.

Injective: Let $o3$ and $z3$ be two different elements in $L3$. Assume that they are identified by suL, i.e., $ox3 = suL(o3) = suL(z3)$, which requires that all these elements have the same type $to3$. Since retyping morphisms are total, there is a single image $ox = TuL \circ suL(o3)$ and hence only a single type tox. Retyping of $L3$ with respect to $p1_T$ provides two different preimages $o31$ and $z31$ of $o3$ and $z3$; The subrule relation then ensures that each of this has a distinguished preimage $o1$ and $z1$ in $L1$. Consequently $o1$ and $z1$ have a common type $to1$. But retyping of LX with respect to $x1_T$ leads to a single preimage of ox which contradicts the subrule condition.

Now, we are going to proof that (2) is a commuting diagram. Therefore consider an element $o3 \in L3$ with type $to3$ and an element $zx3 \in RX$ with type $tox3$ for which we will show the following facts:

1. $zx3 = \mathcal{T}_{u_T}(rx) \circ suL(o3)$ implies $zx3 = suR \circ r3(o3)$ and
2. $zx3 = suR \circ r3(o3)$ implies $zx3 = \mathcal{T}_{u_T}(rx) \circ suL(o3)$.

For the first task recall that suL is total and let $ox3 = suL(o3)$ be the image of $o3$ in $TuL(LX)$. The retyping construction requires the existence of different images $ox = TuL(ox3)$ and $zx = TuR(zx3)$ with type $tox = u_T(to3)$ and related via $\mathcal{T}_{u_T}(rx)$. Assume there is a preimage $to1$ of $to3$ with respect to $p1_T$ (again, due to the fact that $p1_T$ and $p2_T$ are jointly surjective we will show the property for $p1_T$, the other case is analogous). The definition of subrules provides two different elements $o1 \in L1$ and $z1 \in R1$ in being preimages of ox and zx respectively and related via the rule $r1$. The subrule relation carried by $p1$ finally ensures the existence of an item $z3 \in R3$ and the missing link between $o3$ and $z3$ provided by $r3$.

In order to see the second property recall (again) that suL is total and thus there is an element $ox3 = suL(o3)$. This leaves to show that $zx3 = \mathcal{T}_{u_T}(rx)(ox3)$. Totality of retyping morphisms provides items $TuL(ox3)$ and $TuR(zx3)$ in rx. Due to the fact that the diagram (2)+(3) commutes by construction, we infer that $zx = rx(ox)$. Now the desired connection $zx3 = \mathcal{T}_{u_T}(rx)(ox3)$ follows from the retyping construction.

The next step will prove that (2) is indeed a pushout in $\mathbf{TGraphP}(\mathbf{T3})$. As pushouts in this category are constructed componentwise in **GraphP** and $\mathbf{Id}_{T3}$, we will consider only the first component (the pushout in $\mathbf{Id}_{T3}$ is trivially satisfied by all morphisms in $\mathbf{TGraphP}(\mathbf{T3})$). The pushout construction in **GraphP** (see Appendix B.3) ensures that a commuting diagram as that above where all morphisms are injective and one pair (suL, suR) is total is a pushout provided that the following conditions are satisfied:

1a) a vertex $vx3$ is preserved iff there is no preimage to be deleted, i.e., $vx3 \in dom(\mathcal{T}_{u_T}(rx))$ iff $(\nexists v3 \in L3 - dom(r3)$ such that $vx3 = suL(v3))$.

1b) an edge $ex3$ is preserved iff there is no preimage to be deleted and it does not have a source or target vertex to be deleted, i.e., $ex3 \in dom(\mathcal{T}_{u_T}(rx))$ iff $(\nexists e3 \in L3 - dom(r3)$: $ex3 = suL(e3)$ and $source(ex3), target(ex3) \in dom(\mathcal{T}_{u_T}(rx)))$.

2) $\mathcal{T}_{u_T}(rx)$ and suR are jointly surjective.

1a): Consider a vertex $vx3 \in \mathcal{T}_{u_T}(LX)$ with type $tvx3 \in T3$.

Assume that there is a preimage $v3 \in L3 - dom(r3)$. Since diagram (2) commutes this immediately ensures $vx3 \notin dom(\mathcal{T}_{u_T}(rx))$. Assume that there is no preimage $v3 \in L3$ but $vx3 \notin dom(\mathcal{T}_{u_T}(rx))$. Its type $tvx3$ must have a preimage in $T1$ (or $T2$). Retyping rx with respect to $x1_T$ yields a preimage $vx1 \in \mathcal{T}_{x1_T}(LX)$ of $TuL(vx)$. which in turn requires a preimage $v1 \in L1$ by definition of subrules (and analogously for $x2_T$). The amalgamation construction then ensures that the image of $v1$ must be a preimage of $vx3$. Assume that there is a preserved preimage $v3 \in dom(r3)$. Since suR is total, and diagram (2) commutes this clearly implies that $vx \in dom(\mathcal{T}_{u_T}(rx))$.

1b): Consider an edge $ex \in LX$. Basically the arguments are the same as for that concerned about vertices above. We discuss the remaining cases where $source(ex) \notin dom(\mathcal{T}_{u_T}(rx))$ which analogously holds for $target(ex) \notin dom(\mathcal{T}_{u_T}(rx))$ then. More precisely we additionally have to show that $source(ex) \in dom(\mathcal{T}_{u_T}(rx))$ is implied if either (i) the edge is preserved $ex \in dom(\mathcal{T}_{u_T}(rx))$ or (ii) the edge is deleted and there is a preserved preimage, i.e., $ex \in suL(dom(r3)) - dom(\mathcal{T}_{u_T}(rx))$ Case (i) is a straight consequence of the fact that $\mathcal{T}_{u_T}(rx)$ is a (typed) graph morphism. Case (ii) is already contradicted by the fact that the diagram (2) commutes and suR is total which require that ex and its preimage have an image in $\mathcal{T}_{u_T}(RX)$.

2) Let $zx \in \mathcal{T}_{u_T}(RX)$ be an element which neither has a preimage with respect to suR nor with respect to $\mathcal{T}_{u_T}(rx)$. Its type $tzx = t^{L3}(zx)$ must have a preimage in $T1$ (or in $T2$). Retyping rx with respect to $x1_T$ yields a preimage $zx1 \in \mathcal{T}_{X1_T}(RX)$ of $TuR(zx)$. Since the subrule diagram corresponding to $x1_T$ is a pushout, $zx1$ must either have a preimage with respect to $sx1R$ or with respect to $\mathcal{T}_{x1_T}(rx)$. In both cases the amalgamation construction yields a preimage of zx: the first with respect to suR and the second with respect to $\mathcal{T}_{u_T}(rx)$.

Finally, we will show that $su = (suL, suR)$ fulfills the safety property of subrules. So consider some $ox \in LX$ such that there is a type $to3 \in T3$ with $u_T(to3) = tox = t^{LX}(ox)$. Assume that $to3$ has a preimage $to1 \in T1$ with respect to $p1_T$ (the case for $T2$ is analogous). Retyping of LX with respect to $x1_T$ yields a preimage $o1 \in L1$ then because $x1_T = u_T \circ p1_T$ and due to the definition of subrules. This however provides an element $o3 \in L3$ which must be an image of $o1$ and the desired preimage of ox.

$\surd$

Bibliography

[AHS90] J. Adamek, H. Herrlich, and G. Strecker, *Abstract and concrete categories*, Series in Pure and Applied Mathematics, John Wiley and Sons, 1990.

[BD87] E. Best and R. Devillers, *Sequential and concurrent behaviour in Petri net theory*, Theoretical Computer Science **55** (1987), 87–136.

[BFH87] P. Boehm, H.-R. Fonio, and A. Habel, *Amalgamation of graph transformations: a synchronization mechanism*, Journal of Computer and System Science **34** (1987), 377–408.

[BW90] M. Barr and C. Wells, *Category theory for computing science*, Series in Computer Science, Prentice Hall International, London, 1990.

[CEL$^+$94a] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi, *An event structure semantics for safe graph grammars*, Programming Concepts, Methods and Calculi (E.-R. Olderog, ed.), North-Holland, 1994, IFIP Transactions A-56.

[CEL$^+$94b] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi, *Note on standard representation of graphs and graph derivations*, Proc. Graph Grammar Workshop Dagstuhl'93, Springer Verlag, 1994, Lecture Notes in Computer Science 776, pp. 104–118.

[CEL$^+$96a] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg, *The category of typed graph grammars and their adjunction with categories of derivations*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[CEL$^+$96b] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi, *An event structure semantics for graph grammars with parallel productions*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[CGH92] S. Conrad, M. Gogolla, and R. Herzig, *TROLL light: A core language for specifying objects*, Technical Report 92-02, Technical University of Braunschweig, 1992.

[CH95] A. Corradini and R. Heckel, *A compositional approach to structuring and refinement of typed graph grammars*, Electronic Notes in Theoretical Conmputer Science **2** (1995), 167–176, Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation.

[CMR96a] A. Corradini, U. Montanari, and F. Rossi, *Graph processes*, Fundamenta Informaticae, to appear, 1996.

[CMR$^+$96b] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe, *Algebraic approaches to graph transformation I: Basic concepts and double pushout approach*, The Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1996, to appear.

[Cor95] A. Corradini, *Concurrent computing: from Petri nets to graph grammars*, Eletronic Notes in Theoretical Computer Science **2** (1995), 245–260, Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation.

[CR96] A. Corradini and F. Rossi, *Synchronized composition of graph grammar productions*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[EBHL88] H. Ehrig, P. Boehm, U. Hummert, and M. Löwe, *Distributed parallelism of graph transformation*, 13th Int. Workshop on Graph Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 314 (Berlin), Springer Verlag, 1988, pp. 1–19.

[EE95] H. Ehrig and G. Engels, *Towards a module concept for graph transformation systems: The software perspective*, Proc. Colloquium on Graph Transformation and its Application in Computer Science (G. Valiente Feruglio and F. Rosello Llompart, eds.), Technical Report B-19, Universitat de les Illes Balears, 1995.

[EHK$^+$96] 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*, The Handbook of Graph Grammars, Volume 1: Foundations, World Scientific, 1996, to appear.

[EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce, *Parallelism and concurrency in high-level replacement systems*, Math. Struct. in Comp. Science **1** (1991), 361–404.

[Ehr79] H. Ehrig, *Introduction to the algebraic theory of graph grammars*, 1st Graph Grammar Workshop, Lecture Notes in Computer Science 73 (V. Claus, H. Ehrig, and G. Rozenberg, eds.), Springer Verlag, 1979, pp. 1–69.

[EL93a] H. Ehrig and M. Löwe, *Categorical principles, techniques and results for high-level replacement systems in computer science*, Applied Categorical Structures **1** (1993), no. 1, 21–50.

[EL93b] H. Ehrig and M. Löwe, *Parallel and distributed derivations in the single pushout approach*, Theoretical Computer Science **109** (1993), 123 – 143.

[EM85] H. Ehrig and B. Mahr, *Fundamentals of algebraic specification 1: Equations and initial semantics*, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer, Berlin, 1985.

[EM90] H. Ehrig and B. Mahr, *Fundamentals of algebraic specification 2: Module specifications and constraints*, EATCS Monographs on Theoretical Computer Science, vol. 21, Springer, Berlin, 1990.

[EPR94] H. Ehrig, J. Padberg, and L. Ribeiro, *Algebraic high-level nets: Petri nets revisited*, Recent Trends in Data Type Specification, Springer Verlag, 1994, Lecture Notes in Computer Science 785, pp. 188–206.

[GR83] U. Goltz and W. Reisig, *The non-sequential behaviour of Petri nets*, Information and Computation (1983), no. 57, 125–147.

[Gro96] M. Große–Rhode, *Transition specifications for dynamic abstract data types*, Applied Categorical Structures, to appear, 1996.

[GW91] P. Godefroid and P. Wolper, *Using partial orders for the efficient verification of deadlock freedom and safety properties*, Computer Aided Verification – CAV'91, 1991, Lecture Notes in Computer Science 575, pp. 332–342.

[Hab92] A. Habel, *Hyperedge replacement: Grammars and languages*, Lecture Notes in Computer Science, vol. 643, Springer Verlag, Berlin, 1992.

[HCEL96] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe, *Horizontal and vertical structuring of typed graph transformation systems*, Mathematical Structures in Computer Science, to appear, 1996.

[Hec95] R. Heckel, *Algebraic graph transformations with application conditions*, Master's thesis, Technical University of Berlin, 1995.

[Hen88] M. Hennessy (ed.), *Algebraic theory of processes*, The MIT Press, Combridge, Massachussets, 1988.

[HHT96] A. Habel, R. Heckel, and G. Taentzer, *Graph grammars with negative application conditions*, Fundamenta Informaticae, to appear, 1996.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Jan93] D. Janssens, *ESM systems and the composition of their computations*, Proc. Graph Grammar Workshop Dagstuhl 93, Springer Verlag, 1993, Lecture Notes in Computer Science 776, pp. 203–217.

[Jan96] D. Janssens, *The decomposition of ESM computations*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[Jen92] K. Jensen, *Coloured Petri nets. basic concepts, analysis methods and practical use*, Springer-Verlag, Berlin, 1992.

[JK91] R. Janicki and M. Koutny, *Invariant semantics of nets with inhibitor arcs*, CONCUR'91, Springer Verlag, 1991, Lecture Notes in Computer Science 527.

[JK93] R. Janicki and M. Koutny, *Structure of concurrency*, Theoretical Computer Science **112** (1993), 5–52.

[JR89] D. Janssens and G. Rozenberg, *Actor grammars*, Mathematical Systems Theory **22** (1989), 75–107.

[JR91] D. Janssens and G. Rozenberg, *Structured transformations and computation graphs for actor grammars*, 4th Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 532 (H. Ehrig, H.-J. Kreowski, and G. Rozenberg, eds.), Sringer Verlag, 1991, pp. 446–460.

[Ken91] R. Kennaway, *Graph rewriting in some categories of partial maps*, 4th Int. Workshop on Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 532 (H. Ehrig, H.-J. Kreowski, and G. Rozenberg, eds.), Springer Verlag, 1991, pp. 475–489.

[KK96] H.-J. Kreowski and S. Kuske, *On the interleaving semantics of transformation units - a step into GRACE*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[Kor93] M. Korff, *Single pushout transformations of generalized graph structures*, Tech. Report RP 220, Federal University of Rio Grande do Sul, Porto Alegre, Brazil, 1993.

[Kor94] M. Korff, *Graph-interpreted graph transformations for concurrent object-oriented systems*, Extended abstract for the 5th International Workshop on Graph Grammars and their Application to Computer Science, 1994.

[Kor95] M. Korff, *True concurrency semantics for single pushout graph transformations with applications to actor systems*, Information Systems - Correctness and Reusability (R. J. Wieringa and R. B. Feenstra, eds.), World Scientific, 1995, pp. 33–50.

[Kor96] M. Korff, *Generalized graph structure grammars with applications to concurrent object-oriented systems*, Ph.D. thesis, Technical University of Berlin, 1996.

[KR95] M. Korff and L. Ribeiro, *Concurrent derivations as single pushout graph grammar processes*, Electronic Notes in Theoretical Computer Science, **2** (1995), 113–122, Proc. of the SEGRAGRA'95 Workshop on Graph Rewriting and Computation.

[KR96] M. Korff and L. Ribeiro, *Formal relationship between graph grammars and Petri nets*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[Kre77] H. J. Kreowski, *Transformation of derivation sequences in GraGra*, Lecture Notes in Computer Science 56, Springer Verlag, 1977, pp. 275–286.

[Kre81] H.-J. Kreowski, *A comparison between Petri-nets and graph grammars*, Lecture Notes in Computer Science 100, Springer Verlag, 1981, pp. 1–19.

[Kre83] H. J. Kreowski, *Graph grammar derivation processes*, Proc. International Workshop WG'83, Trauner Verlag, 1983, pp. 136–150.

[Kre95] H.-J. Kreowski, *Graph grammars for software specification and programming: an eulogy in praise of GRACE*, Proc. Colloquium on Graph Transformation and its Application in Computer Science, Tech. Report B-19, Universitat de lesIlles Balears, 1995.

[KW86] H.-J. Kreowski and A. Wilharm, *Net processes correspond to derivation processes in graph grammars*, Theoretical Computer Science **44** (1986), 275–305.

[LD94] M. Löwe and J. Dingel, *Parallelism in single-pushout graph rewriting*, Lecture Notes in Computer Science 776 (1994), 234–247.

[LKW93] M. Löwe, M. Korff, and A. Wagner, *An algebraic framework for the transformation of attributed graphs*, Term Graph Rewriting: Theory and Practice (M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, eds.), John Wiley & Sons Ltd, 1993, pp. 185–199.

[Löw90] M. Löwe, *Extended algebraic graph transformations*, Ph.D. thesis, Technical University of Berlin, 1990.

[Löw93] M. Löwe, *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science **109** (1993), 181–224.

[McM92] K. L. McMillan, *Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits*, Computer Aided Verification – CAV'92, 1992, Lecture Notes in Computer Science 663, pp. 164–177.

[McM95] K. L. McMillan, *Trace theoretic verification of asynchrounous circuits using unfoldings*, Computer Aided Verification – CAV'95, 1995, Lecture Notes in Computer Science 939, pp. 180–195.

[Men94] Paulo Menezes, *Compositional reification of Petri nets*, Tech. Report 25/94, Politechnical University of Lisbon, 1994.

[Mes92] J. Meseguer, *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), 73–155.

[Mil89] R. Milner, *Communication and concurrency*, International Series in Computer Science, Prentice Hall, London, 1989.

[MM90] J. Meseguer and U. Montanari, *Petri nets are monoids*, Information and Computation **88** (1990), no. 2, 105–155.

[MMS94] J. Meseguer, U. Montanari, and V. Sassone, *On the model of computation of place/transition Petri nets*, Application and Theory of Petri Nets'94, Springer, 1994, Lecture Notes in Computer Science 815, pp. 16–38.

[MMS96] J. Meseguer, U. Montanari, and V. Sassone, *On the semantics of place/transition Petri nets*, Mathmatical Structures in Computer Science, to appear, 1996.

[MP92] D.E. Monarchi and G.I. Puhr, *A research typology for object-oriented analisys and design*, Communications of the ACM **35** (1992), no. 9, 35–47.

[MR95] U. Montanari and F. Rossi, *Contextual nets*, Acta Informatica, vol. 32, 1995.

[Mue95] J. Mueller, *Foundations of relational graph rewriting systems*, Master's thesis, Technical University of Berlin, Dep. of Comp. Sci., 1995.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel, *Petri nets, event structures and domains: part 1*, Theoretical Computer Sciencei **13** (1981), 85–108.

[PER95] J. Padberg, H. Ehrig, and L. Ribeiro, *Algebraic high-level net transformation systems*, Mathematical Structures in Computer Science **5** (1995), 217–256.

[Pet62] C.A. Petri, *Kommunikation mit Automaten*, Ph.D. thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.

[Pet77] J. L. Peterson, *Petri nets*, Computing Surveys **9** (1977), no. 3, 223–252.

[Pet80] C. A. Petri, *Introduction to general net theory*, Net Theory and Applications, Springer, 1980, Lecture Notes in Computer Science 84, pp. 1–9.

[PL91] D. Probst and F. Hon Li, *Partial-order model checking: a guide for the perplexed*, Computer Aided Verification – CAV'91, 1991, Lecture Notes in Computer Science 575, pp. 322–331.

[PP96] F. Parisi-Presicce, *Transformation of graph grammars*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[Pra86] V. Pratt, *Modelling concurrency with partial orders*, Int. Journal on Parallel Programming **15** (1986), 33–71.

[Rei81] W. Reisig, *A graph grammar representation of nonsequential processes*, Graph-theoretic Concepts in Computer Science. Bad Honnef 1980. (H. Noltemeier, ed.), Lecture Notes in Computer Science, vol. 100, Springer Verlag, 1981, pp. 318 – 325.

[Rei85] W. Reisig, *Petri nets*, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer-Verlag, 1985.

[Rib96] L. Ribeiro, *A telephone system's specification using graph grammars*, Tech. Report 96-23, Technical University of Berlin, 1996.

[Sas94] V. Sassone, *On the semantics of Petri nets: processes, unfoldings and infinite computations*, Ph.D. thesis, Pisa University, 1994.

[Sch91] A. Schürr, *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, Deutscher Universitätsverlag GmbH, Wiesbaden, 1991.

[Sch94] G. Schied, *On relating rewriting systems and graph grammars to event structures*, Graph Transformations in Computer Science (H.-J. Schneider and H. Ehrig, eds.), Springer Verlag, 1994, Lecture Notes in Computer Science 776, pp. 326–340.

[Sch96] H.J. Schneider, *Graph grammars as a tool to define the behaviour of processes: from Petri nets to Linda*, 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg'94, Lecture Notes in Computer Science, 1996, to appear.

[Tae96] G. Taentzer, *Parallel and distributed graph transformation: Formal description and application to communication-based systems*, Ph.D. thesis, Technical University of Berlin, 1996.

[TS95] G. Taentzer and A. Schürr, *DIEGO, another step towards a module concept for graph transformation systems*, Electronic Notes in Theoretical Computer Science **2** (1995), 85–94, Proc. of the SEGRAGRA'95 Workshop on Graph rewriting and Computation.

[Val94] A. Valmari, *Compositional analysis with place-bordered subnets*, Applications and Theory of Petri Nets'94, 1994, Lecture Notes in Computer Science 815, pp. 531–547.

[Vog92] W. Vogler, *Modular construction and partial order semantics of Petri nets*, Springer Verlag, 1992, Lecture Notes in Computer Science 625.

[Wag93] A. Wagner, *Vergleich von High-Level-Replacement-Systemen basierend auf dem Double- bzw. Single-Pushout-Ansatz*, Master's thesis, Technical University of Berlin, 1993.

[WG96] A. Wagner and M. Gogolla, *Defining operational behavior of object specifications by attributed graph transformations*, Fundamenta Informaticae, to appear, 1996.

[Win87a] G. Winskel, *Event structures*, Proc. of the Advanced Course on Petri Nets, Springer Verlag, 1987, Lecture Notes in Computer Science 255, pp. 325–392.

[Win87b] G. Winskel, *Petri nets, algebras, morphisms, and compositionality*, Information and Computation **72** (1987), 197–238.

[Win89] G. Winskel, *An introduction to event structures*, In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Springer Verlag, 1989, pp. 364–397.

[WN94] G. Winskel and M. Nielsen, *Models for concurrency*, Tech. Report BRICS RS-94-12, University of Aarhus, 1994.

Index

Lebenslauf

1968 geboren in Porto Alegre (Brasilien).

1981 Beendigung der Grundschule – *Escola de 1º Grau Santa Rosa de Lima* (Porto Alegre).

1984 Beendigung der Oberschule – *Escola de 1º e 2º Graus Nossa Senhora do Rosário* (Porto Alegre).

1985 Aufnahme an der Universität *Universidade Federal do Rio Grande do Sul* (UFRGS) in Fach Verfahrenstechnik und an der Universität *Pontifícia Universidade Católica do Rio Grande do Sul* (PUC/RS) in Fach Informatik.

1986 Fortsetzung des Studiums an der Universität UFRGS in Fach Informatik.

Englischzertifikat an der Sprachschule *Centro de Cultura Anglo-Americano* (CCAA).

1988 B.Sc.-Diplom in Informatik – Diplomarbeit: *"The VDM Method and its application to the formal specification of an admission system"*.

Mitarbeiterin in Projekt AMPLO an der Universität UFRGS.

1989 Beginn des M.Sc.-Studiums an der Universität UFRGS.

1990 Gastwissenschaftlerin bei *IBM Brasil* (Scientific Center – Rio de Janeiro).

1991 M.Sc.-Diplom. Masterthesis: *"Integration in PROSOFT of correct environments obtained from algebraic specifications and executed by rewriting systems"*.

Lehrtätigkeit in Fächern Informatik und Mathematik (8 SWS + 4 SWS).

Deutschzertifikat *"Deutsch als Fremdsprache"* am Goetheinstitut (Porto Alegre).

1992 Doktorandenstipendium zum Studium an der TU Berlin.

Deutschzertifikat "Mittelstufe I" an der Hartnackschule (Berlin).

1994 Mitarbeiterin im GRAPHIT Projekt

1996 Abschluß der Promotion an der TU Berlin mit der Dissertation *"Parallel composition and unfolding semantics of graph grammars"*