Chapter 3

CONCURRENT SEMANTICS OF ALGEBRAIC GRAPH TRANSFORMATIONS

PAOLO BALDAN, ANDREA CORRADINI
UGO MONTANARI, FRANCESCA ROSSI
University of Pisa, Computer Science Department,
Corso Italia 40, 56125 Pisa, Italy
{*baldan,andrea,ugo,rossi*}*@di.unipi.it*

HARTMUT EHRIG, MICHAEL LÖWE
Techincal University of Berlin, Computer Science Department,
Franklinstrasse 28/29, 10587 Berlin, Germany.
{*ehrig,loewe*}*@cs.tu-berlin.de*

Graph transformation systems are widely recognized as a powerful formalism for the specification of concurrent and distributed systems. Therefore, the need emerges naturally of developing formal concurrent semantics for graph transformation systems allowing for a suitable description and analysis of their computational properties. The aim of this chapter is to review and compare various concurrent semantics for the *double pushout (DPO) algebraic approach* to graph transformation, using different mathematical structures and describing computations at different levels of abstraction. We first present a *trace* semantics, based on the classical shift equivalence on graph derivations. Next we introduce *graph processes*, which lift to the graph transformation framework the notion of non-sequential process for Petri nets. Trace and process semantics are shown to be equivalent, in the sense that given a graph transformation system, the corresponding category of derivation traces and that of (concatenable) processes turns out to be isomorphic. Finally, a more abstract description of graph transformation systems computations is given by defining a semantics based on Winskel's *event structures*.

Contents

3.1 Introduction

Graph transformation systems (or *graph grammars*) have been deeply studied along the classical lines of the theory of formal languages, namely focusing on the properties of the generated graph languages and on their decidability; briefly, on the *results* of the generation process. However, quite early, graph transformation systems have been recognized as a powerful tool for the specification of concurrent and distributed systems. The basic idea is that the state of many distributed systems can be represented naturally (at a suitable level of abstraction) as a graph, and (local) transformations of the state can be expressed as production applications. Thus a stream of research has grown, concentrating on the *rewriting process itself* seen as a modelling of system computations, studying properties of derivation sequences, their transformations and equivalences.
The appropriateness of graph grammars as models of concurrency is confirmed by their relationship with another classical model of concurrent and distributed systems, namely, Petri nets [1,2]. Basically a Petri net can be viewed as a graph rewriting system that acts on a restricted kind of graphs, namely discrete, labelled graphs (that can be considered as sets of tokens labelled by places), the productions being the transitions of the net. In this view, general graph transformation systems are a *proper* extension of Petri nets in two dimensions:

1. they allow for a *more structured description of the state*, that is an arbitrary, possibly non-discrete, graph;

2. they allow for the specification of *context-dependent rewritings*, where part of the state is required for the rewriting step to take place, but is not affected by the step.

Several approaches to graph transformation have been considered in the literature. The first volume of the Handbook of Graph Grammars and Computing by Graph Transformation [3] provides a comprehensive introduction to the subject. The basic idea common to all these approaches is very simple: a graph transformation system consists of a set of rules, called *graph productions*; each production has the form $q : L \rightsquigarrow R$ and specifies that, under certain conditions, once an occurrence (a *match*) of the left-hand side L in a graph G has been detected, it can be replaced by the right-hand side R. The form of graph productions, the notion of match and in general the mechanisms stating how a production can be applied to a graph and what the resulting graph is, depend on the specific graph transformation formalism.
This presentation is concerned with the so-called *algebraic approach* [4,5], where the basic notions of production and direct derivation are defined in

terms of constructions and diagrams in a category. The resulting theory is very general and flexible, easily adaptable to a very wide range of structures, simply by changing the underlying category of objects which are rewritten. More precisely we concentrate on the *double-pushout (DPO)* approach to graph transformation, historically the first of the algebraic approaches, proposed in the early seventies by H. Ehrig, M. Pfender and H.J. Schneider. The name DPO is related to the fact that the application of a production to a graph is defined via two pushout diagrams in the category of graphs and total graph morphisms. Roughly speaking, the first pushout is intended to model the removal of the left-hand side of the production from the graph to be rewritten, and the second one the addition of the right-hand side.
Although not explicitly treated in this chapter, it is worth recalling the existence of an alternative algebraic approach to graph transformation, called the *single-pushout (SPO)* approach [6,7], where a basic rewriting step is defined via a single pushout diagram in the category of graphs and *partial* morphisms.

Semantics of graph transformation
For sequential systems it is often sufficient to consider an input/output semantics and thus the appropriate semantic domain is usually a suitable class of functions from the input to the output domains. When concurrent or distributed features are involved, instead, typically more information about the actual computation of the system has to be recorded in the semantic domain. For instance, one may want to know which steps of computation are independent (concurrent), which are causally related and which are the (non-deterministic) choice points. This information is necessary, for example, if one wants to have a compositional semantics, allowing to reduce the complexity of the analysis of concurrent systems built from smaller parts, or if one wants to allocate a computation on a distributed architecture. Roughly speaking, *non-determinism* can be represented either by collecting all the possible different computations in a set, or by merging the different computations in a unique *branching* structure where the choice points are explicitly represented. On the other hand, *concurrent* aspects can be represented by using a *truly concurrent* approach, where the causal dependencies among events are described directly in the semantics using a partially ordered structure. Alternatively, an *interleaving* approach can be adopted, where concurrency is reduced to non-determinism, in the sense that the concurrent execution of events is represented as the non-deterministic choice among the possible interleavings of such events.
Along the years, Petri nets have been equipped with rich, formal computation-based semantics, including both interleaving and truly concurrent models (see, among others, [8,9]). In many cases such semantics have been defined by using

well-established categorical techniques, often involving adjunctions between suitable categories of nets and corresponding categories of models [10,11,12,13]. To propose graph transformation systems as a suitable formalism for the specification of concurrent/distributed systems that generalizes Petri nets, we are naturally led to the attempt of equipping them with a satisfactory semantic framework, where the truly concurrent behaviour of grammars can be suitably described and analyzed. After the seminal work of Kreowski [14], during the last years many original contributions to the theory of concurrency for algebraic, both DPO and SPO, graph transformation systems have been proposed, most of them inspired by the above outlined correspondence with Petri nets. In particular, for the DPO approach to graph transformation a *trace semantics* has been proposed in [15,5] developing some basic ideas of [14]. Furthermore, *graph processes* [16,17] and *event structure semantics* [18,19] have been introduced. The goal of this chapter is to review such semantics and to discuss the formal relationships existing among them.

Trace semantics
Historically the first truly concurrent semantics proposed for graph grammars has been the *derivation trace* semantics [14,15,5]. *Derivation traces* are defined as equivalence classes of derivations with respect to the least equivalence containing the *abstraction equivalence* and the *shift equivalence* relations, which is called the *concatenable truly-concurrent (ctc) equivalence*. *Abstraction equivalence* [15] is a suitable refinement of the isomorphism relation on derivations (needed to guarantee the concatenability of traces), which allows to abstract from what we can call "representation details", namely from the concrete identity of the items of the graphs involved in a derivation. *Shift equivalence* [14,20,21] relates derivations that differ only for the order in which independent direct derivations are performed, and thus it is aimed at extracting the truly concurrent behaviour of the grammar. Being concatenable, the derivation traces of a grammar $\mathcal{G}$ can be seen as arrows of a category, having abstract graphs as objects. Such category, denoted $\mathbf{Tr}[\mathcal{G}]$ and called the *category of derivation traces* (or the *abstract truly concurrent model of computation*), is the most abstract model of computation for a grammar in a hierarchy of models originally presented in [5], and it can be reasonably assumed to express the truly concurrent behaviour of a grammar at an adequate level of abstraction.

Process semantics
In the theory of Petri nets, (non-sequential deterministic Goltz-Reisig) processes [9] are nets satisfying suitable acyclicity and conflict-freeness requirements. Processes include also a mapping to the original net, specifying how

transitions in the process can be identified with firings of transitions of the original net. This notion has been lifted to graph grammars in [16], where a *graph process* of a grammar $\mathcal{G}$ is defined as a special kind of grammar $\mathcal{O}$, called "occurrence grammar", equipped with a mapping to the original grammar $\mathcal{G}$. This mapping is used to associate to the derivations of $\mathcal{O}$ corresponding derivations of $\mathcal{G}$. The basic property of a graph process is that the derivations of $\mathcal{G}$ which are in the range of such mapping constitute a full class of *shift*-equivalent derivations. Therefore the process can be regarded as an abstract representation of such a class and plays a rôle similar to a *canonical derivation* [14].
The theory of Petri nets comprises also a very interesting notion of *concatenable* process [22,23]: Goltz-Reisig processes, enriched with an ordering of the minimal and maximal places, can be composed sequentially. In this way they can be seen as arrows of a category, which can be shown to capture the truly concurrent behaviour of a net. Following the same idea, [17] introduces *concatenable graph processes*, which enrich graph processes with the additional information needed to concatenate them, keeping track of the flow of causality. (Abstract) concatenable processes of a grammar $\mathcal{G}$ form the arrows of a category $\mathbf{CP}[\mathcal{G}]$, where objects are abstract graphs, called the *category of concatenable processes* of the grammar $\mathcal{G}$.

Event structure semantics
Graph processes, as well as derivation traces, represent single deterministic computations of a grammar, taking care only of the concurrent nature of such computations. The intrinsic non-determinism of a grammar is implicitly represented by the existence of several different "non confluent" processes (or traces) having the same source. When working with concurrent non-deterministic systems, a typical alternative approach consists of representing in a unique branching structure all the possible computations of the system. This structure expresses not only the causal ordering between the events, but also gives an explicit representation of the branching (choice) points of the computations. One of the most widely used models of this kind are Winskel's *event structures* [10,24,25], a simple event-based semantic model where events are considered as atomic and instantaneous steps, which can appear only once in a computation. An event can occur only after some other events (its causes) have taken place and the execution of an event can inhibit the execution of other events.
An event structure semantics for graph grammars can be derived from the category of derivation traces via a comma category construction [19,18]. Starting from the assumption that (concatenable) derivation traces are an adequate abstract representation of the deterministic truly concurrent computations of a

grammar, the category of objects of $\mathbf{Tr}[\mathcal{G}]$ under the start graph of the grammar $\mathcal{G}$ provides a synthetic representation of all the possible computations of the grammar beginning from the start graph. For *consuming* grammars (i.e., such that every production deletes something) the induced partial order $\mathbf{Dom}[\mathcal{G}]$ turns out to be prime algebraic, finitely coherent and finitary, and thus, by well known results [24], it determines an event structure. Actually, the algebraic properties of the domain are proved indirectly, by providing an explicit construction of an event structure $\mathbf{ES}[\mathcal{G}]$ and then by showing that its finite configurations are in one to one correspondence with the elements of the domain $\mathbf{Dom}[\mathcal{G}]$. Thus $\mathbf{ES}[\mathcal{G}]$ and $\mathbf{Dom}[\mathcal{G}]$ are conceptually equivalent, in the sense that one can be recovered from the other. It is worth noticing that the event structure semantics differs from the previously described trace and process semantics, not only because it represents in a unique structure both the concurrent and non deterministic aspects of grammar computations, but also because it is more abstract, since it completely forgets the structure of the states of the system.

Relations among the semantics

Processes and derivation traces are based on the same fundamental ideas, namely abstraction from representation details and true concurrency, but, as stressed above, they have concretely a rather different nature. Derivation traces provide a semantics for grammars where the independence of events is represented implicitly by the fact that derivations in which the events appear in different orders are in the same trace. Processes, instead, provide a partial order semantics, where the events and the relationships among them are represented explicitly. The natural question asking whether a precise relationship can be established between the two semantics is answered positively. The category $\mathbf{CP}[\mathcal{G}]$ of concatenable processes can be shown to be isomorphic to the category of derivation traces $\mathbf{Tr}[\mathcal{G}]$ of the grammar. The process corresponding to a trace can be obtained elegantly by performing a colimit construction on any derivation belonging to the trace. Viceversa, given a process, a trace can be obtained by considering the equivalence class of all derivations which apply the productions of the process in any order compatible with the causal ordering.

The isomorphism result is based on a suitable characterization of the ctc-equivalence (used to define traces), which plays also a basic rôle in the definition of the event structure $\mathbf{ES}[\mathcal{G}]$ associated to a grammar [18]. This fact, together with the close relationship existing between traces and processes, allows us to give an intuitive characterization of the events and configurations of $\mathbf{ES}[\mathcal{G}]$ in terms of processes. Basically, configurations are shown to be in one-to-one

correspondence with processes which have as source graph the start graph of the grammar. Events are one-to-one with a subclass of such processes having a production which is the maximum w.r.t. the causal ordering. This completes the series of "unifying" results presented in the chapter.

The chapter is structured as follows. Section 3.2 reviews the basics of the double pushout approach to graph transformation. The presentation slightly differs from the classical one, since we consider *typed* graph grammars, a generalization of usual graph grammars where a more sophisticated labelling technique for graphs is considered. Section 3.3 reviews the definition of the category of (concatenable) derivation traces $\mathbf{Tr}[\mathcal{G}]$ for a grammar $\mathcal{G}$. Section 3.4 discusses the graph process semantics and introduces the category $\mathbf{CP}[\mathcal{G}]$ of concatenable graph processes. In Section 3.5 the isomorphism between the category $\mathbf{Tr}[\mathcal{G}]$ of derivation traces and the category $\mathbf{CP}[\mathcal{G}]$ of concatenable processes of a grammar $\mathcal{G}$ is proved. Section 3.6 presents the construction of the event structure for a graph grammar and its characterization in terms of processes. Section 3.7 compares the results of the chapter with other related works in the literature.
The chapter is closed by an Appendix presenting an effective construction of canonical graphs, which are used for decorating traces and processes.

Chapter 3 in this book will show with some concrete examples how graph rewriting can be used to model the behaviour of concurrent and distributed systems. In particular, the work developed in the present chapter will be applied to give concurrent semantics to formalisms such as concurrent constraint programming and the π-calculus.

3.2 Typed Graph Grammars in the DPO approach

In this section we review the basics of the *algebraic approach* to graph transformation based on the *double-pushout (DPO)* construction [20]. In the last subsection we will also give some more insights on the relation between DPO graph grammar and Petri nets outlined in the Introduction.
The presentation slightly differs from the classical one since we consider *typed* graph grammars [16], a generalization of usual graph grammars where a more sophisticated labelling technique for graphs is considered: each graph is typed on a structure that is itself a graph (called the *graph of types*) and the labelling function is required to be a graph morphism, i.e., it must preserve the graphical structure. This gives, in a sense, more control on the labelling mechanism. Working with typed graph grammars just means changing the category over which the rewriting takes place, from labelled graphs to typed graphs. From

the formal side, to move from labelled to typed graphs we just replace the category of labelled graphs with that of typed graphs in the definitions and results of the DPO approach to graph transformation.

Definition 3.2.1 (graph)
A *(directed, unlabelled) graph* is a tuple $G = \langle N, E, s, t\rangle$, where N is a set of *nodes*, E is a set of *arcs*, and $s, t : E \to N$ are the *source* and *target* functions. A graph is *discrete* if it has no arcs. Sometimes we will denote by N_G and E_G the set of nodes and arcs of a graph G and by s_G, t_G its source and target functions.
A *graph morphism* $f : G \to G'$ is a pair of functions $f = \langle f_N : N \to N', f_E : E \to E'\rangle$ which preserve sources and targets, i.e., such that $f_N \circ s = s' \circ f_E$ and $f_N \circ t = t' \circ f_E$; it is an *isomorphism* if both f_N and f_E are bijections; moreover, an *abstract graph* $[G]$ is an isomorphism class of graphs, i.e., $[G] = \{H \mid H \simeq G\}$. An *automorphism* of G is an isomorphism $h : G \to G$; it is *non-trivial* if $h \neq id_G$. The category having graphs as objects and graph morphisms as arrows is called **Graph**.

Formally, the category of typed graphs can be defined as the category of *graphs over the graph of types*. Let us recall this categorical notion first, together with the symmetrical one.

Definition 3.2.2 (category of objects over/under a given object [26])
Let $\mathbf{C}$ be a category and let x be an object of $\mathbf{C}$. The *category!of objects (of $\mathbf{C}$) over x*, denoted $(\mathbf{C} \downarrow x)$, is defined as follows. The objects of $(\mathbf{C} \downarrow x)$ are pairs $\langle y, f\rangle$ where $f : y \to x$ is an arrow of $\mathbf{C}$. Furthermore, $k : \langle y, f\rangle \to \langle z, g\rangle$ is an arrow of $(\mathbf{C} \downarrow x)$ if $k : y \to z$ is an arrow of $\mathbf{C}$ and $f = g \circ k$.
Symmetrically, the *category of objects (of $\mathbf{C}$) under x*, denoted $(x \downarrow \mathbf{C})$, has pairs like $\langle f, y\rangle$ as objects, where $f : x \to y$ is an arrow in $\mathbf{C}$; and $k : \langle f, y\rangle \to \langle g, z\rangle$ is an arrow of $(x \downarrow \mathbf{C})$ if $k : y \to z$ is an arrow of $\mathbf{C}$ and $k \circ f = g$.

Definition 3.2.3 (category of typed graphs)
Given a graph TG, the category **TG-Graph** of *TG-typed graphs* is defined as the category of graphs over TG, i.e., $(\mathbf{Graph} \downarrow TG)$. Objects of **TG-Graph**, i.e., pairs like $\langle G, m\rangle$ where G is a graph and $m : G \to TG$ is a graph morphism, will be called simply *typed graphs* if TG is clear from the context; similarly, arrows will be called *typed graph morphisms*.

It is worth stressing that the typing mechanism subsumes the usual labelling technique, where there are two colour alphabets Ω_N for nodes and Ω_E for arcs, and, correspondingly, two labelling functions. In fact, consider the graph of

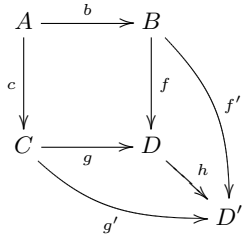


Figure 3.1: Pushout diagram.

types $TG_\Omega = \langle \Omega_N, \Omega_E \times \Omega_N \times \Omega_N, s, t \rangle$, with $s(e, n_1, n_2) = n_1$ and $t(e, n_1, n_2) = n_2$. It is easily seen that there is an isomorphism between the category of labelled graphs over $\langle \Omega_N, \Omega_E \rangle$ and the category of TG_Ω-typed graphs.
The very basic idea in the algebraic approach to graph transformation is that of *gluing of graphs*, expressed in categorical term as a *pushout construction* [26].

Definition 3.2.4 (pushout)
Let **C** be a category and let $b : A \to B$, $c : A \to C$ be a pair of arrows of **C**. A triple $\langle D, f : B \to D, g : C \to D \rangle$ as in Figure 3.1 is called a *pushout* of $\langle b, c \rangle$ if the following conditions are satisfied:

[Commutativity]
$f \circ b = g \circ c$;

[Universal Property]
for any object D' and arrows $f' : B \to D'$ and $g' : C \to D'$, with $f' \circ b = g' \circ c$, there exists a unique arrow $h : D \to D'$ such that $h \circ f = f'$ and $h \circ g = g'$.

In this situation, D is called a *pushout object* of $\langle b, c \rangle$. Moreover, given arrows $b : A \to B$ and $f : B \to D$, a *pushout complement*indexpushout complement of $\langle b, f \rangle$ is a triple $\langle C, c : A \to C, g : C \to D \rangle$ such that $\langle D, f, g \rangle$ is a pushout of b and c. In this case C is called a *pushout complement object* of $\langle b, f \rangle$.

In the category **Set** of sets and (total) functions the pushout object can be characterized as $D = (B + C)/_\equiv$, where $\equiv$ is the least equivalence on $B + C = \{\langle 0, x \rangle \mid x \in B\} \cup \{\langle 1, y \rangle \mid y \in C\}$ such that, for each $a \in A$, $\langle 0, b(a) \rangle \equiv \langle 1, c(a) \rangle$, or in words, D is the disjoint union of B and C, where the images of A through b and c are equated. The morphisms f and g are the obvious

embeddings. One can see that, analogously, in the category of graphs and (total) graph morphisms the pushout object can be thought of as the *gluing* of B and C, obtained by identifying the images of A via b and c. According to this interpretation, the pushout complement object C of b and f, is obtained by removing from D the elements of $f(B)$ which are not images of $b(A)$.
A typed production in the double-pushout approach is a *span*, i.e., a pair of typed graph morphisms with common source. Moreover each production has an associated name which allows one to distinguish between productions with the same associated span. Such name plays no rôle when the production is applied to a graph, but it is relevant in certain transformations of derivations and when relating different derivations.

Definition 3.2.5 (typed production)
A *(TG-typed graph) production* $(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ is a pair of injective typed graph morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$. It is called *consuming* if morphism $l : K \rightarrow L$ is not surjective. The typed graphs L, K, and R are called the *left-hand side*, the *interface*, and the *right-hand side* of the production, respectively.

Although sometimes we will consider derivations starting from a generic typed graph, a typed graph grammar comes equipped with a start graph, playing the same rôle of the initial symbol in string grammars, or of the initial marking for Petri nets. Conceptually, it represents the initial state of the system modeled by the grammar.

Definition 3.2.6 (typed graph grammar)
A *(TG-typed) graph grammar* $\mathcal{G}$ is a tuple $\langle TG, G_s, P, \pi \rangle$, where G_s is the *start (typed) graph*, P is a set of *production names*, and π a function mapping each production name in P to a graph production. Sometimes we shall write $q : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ for $\pi(q) = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$. Grammar $\mathcal{G}$ is called *consuming* if all its productions are consuming, and *finite* if the set of productions P is finite.

Example 3.1 (client-server systems)
As a running example we will use the simple typed graph grammar shown in Figure 3.2, which models the evolution of client-server systems (this is a little modification of an example from [5]). The typing morphisms from the involved graphs to the graph of types TG are not depicted explicitly, but are encoded by attaching to each item its type, i.e., its image in TG, separated by a colon. We use natural numbers for nodes, and underlined numbers for edges. For example, the node 4 of the start graph G_0 is typed over node C of TG.

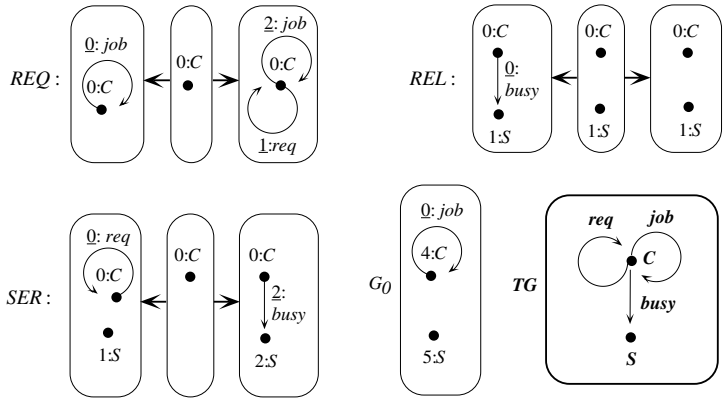


Figure 3.2: Productions, start graph and graph of types of the grammar $\mathcal{C}$-$\mathcal{S}$ modeling client-server systems.

The typed graphs represent possible configurations containing servers and clients (represented by nodes of types S and C, respectively), which can be in various states, as indicated by edges. A loop of type *job* on a client means that the client is performing some internal activity, while a loop of type *req* means that the client issued a request. An edge of type *busy* from a client to a server means that the server is processing a request issued by the client.
Production *REQ* models the issuing of a request by a client. After producing the request, the client continues its internal activity (*job*), while the request is served asynchronously; thus a request can be issued at any time, even if other requests are pending or if the client is being served by a server. Production *SER* connects a client that issued a request with a server through a *busy* edge, modeling the beginning of the service. Since the production deletes a node of type S and creates a new one, the *dangling condition* (see below) ensures that it will be applied only if the server has no incoming edges, i.e., if it is not busy. Production *REL* (for *release*) disconnects the client from the server (modeling the end of the service). Notice that in all rules the graph morphisms are inclusions.
The grammar is called $\mathcal{C}$-$\mathcal{S}$ (for *client-server*), and it is formally defined as $\mathcal{C}\text{-}\mathcal{S} = \langle TG, G_0, \{REQ,\ SER,\ REL\}, \pi \rangle$, where π maps the production names to the corresponding production spans depicted in Figure 3.2. □

Since in this chapter we work only with typed notions, when clear from the context we omit the word "typed" and do not indicate explicitly the typing morphisms.

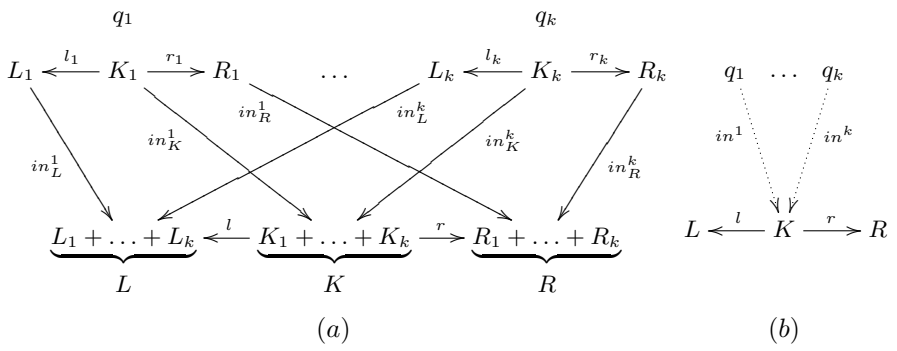


Figure 3.3: (a) The parallel production $\langle (q_1, in^1), \ldots, (q_k, in^k) \rangle : (L \xleftarrow{l} K \xrightarrow{r} R)$ and (b) its compact representation.

The application of a production produces a *local* transformation in the rewritten graph, hence the idea of allowing for the concurrent application of more than one production naturally emerges. The idea of "parallel composition" of productions is naturally formalized in the categorical setting by the notion of parallel production.

Definition 3.2.7 ((typed) parallel productions)
A *parallel production* (over a given typed graph grammar $\mathcal{G}$) has the form $\langle (q_1, in^1), \ldots, (q_k, in^k) \rangle : (L \xleftarrow{l} K \xrightarrow{r} R)$ (see Figure 3.3), where $k \geq 0$, $q_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ is a production of $\mathcal{G}$ for each $i \in \underline{k}$,[a] L is a coproduct object of the typed graphs in $\langle L_1, \ldots, L_k \rangle$, and similarly R and K are coproduct objects of $\langle R_1, \ldots, R_k \rangle$ and $\langle K_1, \ldots, K_k \rangle$, respectively. Moreover, l and r are uniquely determined, using the universal property of coproducts, by the families of arrows $\{l_i\}_{i \in \underline{k}}$ and $\{r_i\}_{i \in \underline{k}}$, respectively. Finally, for each $i \in \underline{k}$, in^i denotes the triple of injections $\langle in^i_L : L_i \to L, in^i_K : K_i \to K, in^i_R : R_i \to R \rangle$. The *empty* production is the (only) parallel production with $k = 0$, having the empty graph $\emptyset$ (initial object in **TG-Graph**) as left-hand side, right-hand side and interface, and it is denoted by $\emptyset$.

In the rest of the chapter we will often denote the parallel production of Figure 3.3 (a) simply as $q_1 + q_2 + \ldots + q_k : (L \xleftarrow{l} K \xrightarrow{r} R)$; note however that the "+" operator is not assumed to be commutative. We will also use the

[a] For each $n \in \mathbb{N}$, by $\underline{n}$ we denote the set $\{1, 2, \ldots, n\}$ (thus $\underline{0} = \emptyset$).

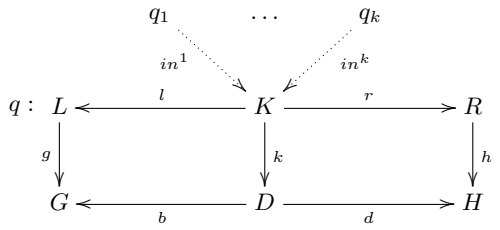


Figure 3.4: (Parallel) direct derivation as double-pushout construction.

more compact drawing of Figure 3.3 (b) to depict the same parallel production. Furthermore, we will freely identify a production q of $\mathcal{G}$ with the parallel production $\langle\langle q, \langle id_L, id_K, id_R\rangle\rangle\rangle$; thus, by default productions will be parallel in the rest of the presentation.
The rewriting procedure involves two pushout diagrams in the category of graphs, hence the name of double-pushout approach.

Definition 3.2.8 ((parallel) direct derivation)
Given a typed graph G, a parallel production $q = q_1 + \ldots + q_k : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, and a *match* (i.e., a graph morphism) $g : L \rightarrow G$, a *(parallel) direct derivation δ from G to H using q (based on g)* exists if and only if the diagram in Figure 3.4 can be constructed, where both squares are required to be pushouts in **TG-Graph**. In this case, D is called the *context* graph, and we write $\delta : G \Rightarrow_q H$, or also $\delta : G \Rightarrow_{q,g} H$; only seldom we shall write $\delta : G \stackrel{\langle g,k,h,b,d\rangle}{\Longrightarrow_q} H$, indicating explicitly all the morphisms of the double-pushout. If $q = \emptyset$, i.e., q is the empty production, then $G \Rightarrow_\emptyset H$ is called an *empty direct derivation*.

To have an informal understanding of the notion of direct derivation, one should recall the interpretation of the pushout as gluing of graphs. According to this interpretation, the rewriting procedure removes from the graph G the images via g of the items of the left-hand side which are not in the image of the interface, namely $g(L - l(K))$, producing in this way the graph D. Then the items in the right-hand side, which are not in the image of the interface, namely $R - r(K)$, are added to D, obtaining the final graph H. Thus the interface K (common part of L and R) specifies what is preserved. For what regards the *applicability* of a production to a given match, it is possible to prove that the situation in the category **TG-Graph** is exactly the same as in **Graph**, namely pushouts always exist, while for the existence of the pushout

complement some conditions have to be imposed, called *gluing conditions* [4], which consist of two parts:

[Dangling condition]
No edge $e \in G - g(L)$ is incident to any node in $g(L - l(K))$;

[Identification condition]
There is no $x, y \in L$, $x \neq y$, such that $g(x) = g(y)$ and $y \notin l(K)$.

Nicely, the gluing conditions have a very intuitive interpretation: the dangling condition avoids the deletion of a node if some arc is still pointing to it, and thus ensures the absence of dangling arcs in D. The identification condition requires each item of G which is deleted by the application of q, to be the image of only one item of L. Among other things, this ensures that the application of a production cannot specify simultaneously both the preservation and the deletion of an item (see [5] for a broader discussion). Uniqueness of the pushout complement, up to isomorphism, follows from the injectivity of l.

Remark:
If $G \overset{\langle g,k,h,b,d\rangle}{\Rightarrow_\emptyset} H$ is an *empty direct derivation*, then morphisms g, k, and h are necessarily the only morphisms from the empty (typed) graph (since $\langle\emptyset,\emptyset\rangle$ is initial in **TG-Graph**), while b and d must be isomorphisms. Morphism $d \circ b^{-1} : G \to H$ is called the *isomorphism induced* by the empty direct derivation. Moreover, for any pair of isomorphic graphs $G \simeq H$, there is one empty direct derivation $G \overset{\langle\emptyset,\emptyset,\emptyset,b,d\rangle}{\Rightarrow_\emptyset} H$ for each triple $\langle D, b : D \to G, d : D \to H\rangle$, where b and d are isomorphisms. An empty direct derivation $G \overset{\langle\emptyset,\emptyset,\emptyset,b,d\rangle}{\Rightarrow_\emptyset} H$ will be also briefly denoted as $G \overset{\langle b,d\rangle}{\Rightarrow_\emptyset} H$. □
A parallel derivation can be seen as a sequence of single steps of the system, each one consisting of the concurrent execution of a set of independent basic actions of the system, analogously to step sequences of Petri nets.

Definition 3.2.9 ((parallel) derivation)
A *(parallel) derivation (over $\mathcal{G}$)* is either a graph G (called an *identity derivation*, and denoted by $G : G \Rightarrow^* G$), or a sequence of (parallel) direct derivations $\rho = \{G_{i-1} \Rightarrow_{q_i} G_i\}_{i \in \underline{n}}$ such that $q_i = q_{i1} + \ldots + q_{ik_i}$ is a (parallel) production over $\mathcal{G}$ for all $i \in \underline{n}$ (as in Figure 3.6). In the last case, the derivation is written $\rho : G_0 \Rightarrow^*_{\mathcal{G}} G_n$ or simply $\rho : G_0 \Rightarrow^* G_n$. If $\rho : G \Rightarrow^* H$ is a (possibly identity) derivation, then graphs G and H are called the *source* and the *target graphs* of ρ, and will be denoted by $\sigma(\rho)$ and $\tau(\rho)$, respectively. The *length* of a derivation ρ, denoted by $|\rho|$, is the number of direct derivations in ρ, if it is not an identity, and 0 otherwise. The *order* of ρ, denoted by $\#\rho$, is the total

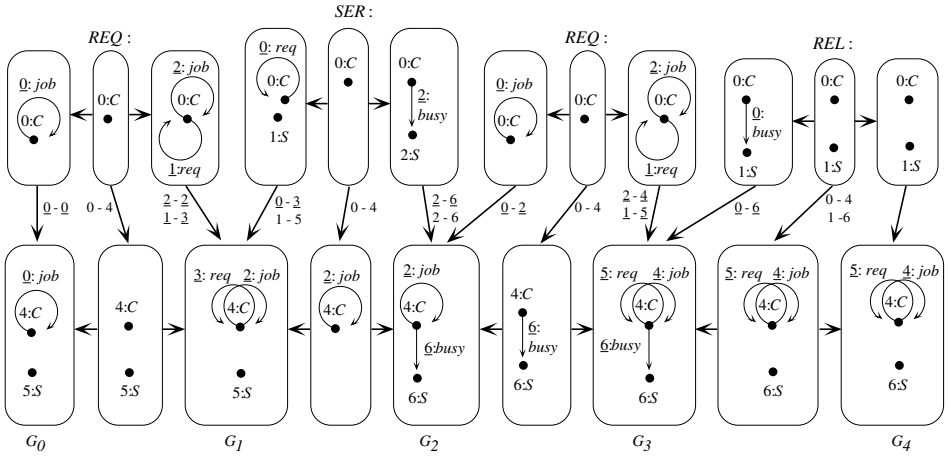


Figure 3.5: A derivation of grammar $\mathcal{C}$-$\mathcal{S}$ starting from graph G_0.

number of elementary productions used in ρ, i.e., $\#\rho = \sum_{r=1}^{n} k_r$; moreover, $prod(\rho) : \underline{\#\rho} \to P$ is the function returning for each j the name of the j-th production applied in ρ—formally, $prod(\rho)(j) = q_{is}$ if $j = \left(\sum_{r=1}^{i} k_r\right) + s$. The *sequential composition* of two derivations ρ and ρ' is defined if and only if $\tau(\rho) = \sigma(\rho')$; in this case it is denoted $\rho\,;\,\rho' : \sigma(\rho) \Rightarrow^* \tau(\rho')$, and it is the diagram obtained by identifying $\tau(\rho)$ with $\sigma(\rho')$ (thus if $\rho : G \Rightarrow^* H$, then $G\,;\,\rho = \rho\,;\,H$, where G and H are the identity derivations).

Example 3.2 (derivation)
Figure 3.5 shows a derivation ρ using grammar $\mathcal{C}$-$\mathcal{S}$ and starting from the start graph G_0. The derivation models the situation where a request is issued by the client, and while it is handled by the server, a new request is issued. All the horizontal morphisms are inclusions, while the vertical ones are annotated by the relation on graph items they induce. □

3.2.1 Relation with Petri nets

The basic notions introduced so far allow us to give a more precise account of the relation between Petri nets and graph grammars in the double pushout approach. Being Petri nets one of the most widely accepted formalisms for the representation of concurrent and distributed systems, people working on the

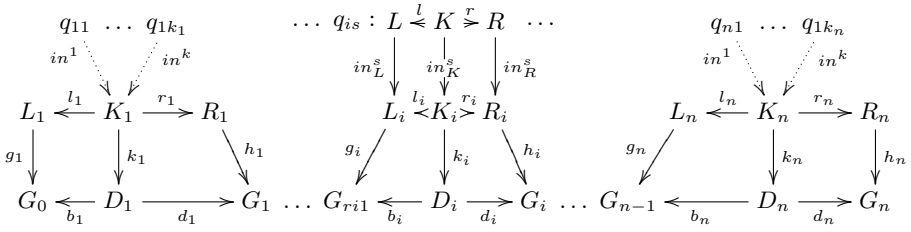


Figure 3.6: A (parallel) derivation, with explicit drawing of the s-th production of the i-th direct derivation.

concurrency theory of graph grammars have been naturally led to compare their formalisms with nets. Therefore various encodings of nets into graph graph grammars have been proposed along the years, all allowing to have some correspondences between net-related notions and graph-grammars ones. Some encodings involving the DPO approach can be found in [27,28,29], while for a complete survey the reader can consult [30]. All these papers represent a net as a grammar by explicitly encoding the topological structure of the net as well as the initial marking in the start graph. A slightly simpler encoding comes from the simple observation that a Petri net is essentially a rewriting system on multisets, and that, given a set A, a multiset of A can be represented as a discrete graph typed over A. In this view a P/T Petri net can be seen as a graph grammar acting on discrete graphs typed over the set of places, the productions being (some encoding of) the net transitions: a marking is represented by a set of nodes (tokens) labelled by the place where they are, and, for example, the unique transition t of the net in Fig. 3.7.(a) is represented by the graph production in the top row of Fig. 3.7.(b): such production consumes nodes corresponding to two tokens in A and one token in B and produces new nodes corresponding to one token in C and one token in D. The interface is empty since nothing is explicitly preserved by a net transition. Notice that in this encoding the topological structure of the net is not represented at all: it is only recorded in the productions corresponding to the transitions.

It is easy to check that this representation satisfies the properties one would expect: a production can be applied to a given marking if and only if the corresponding transition is enabled, and the double pushout construction produces the same marking as the firing of the transition. For instance, the firing of transition t, leading from the marking $3A + 2B$ to the marking $A + B + C + D$ in Figure 3.7.(a) becomes the double pushout diagram of Figure 3.7.(b).

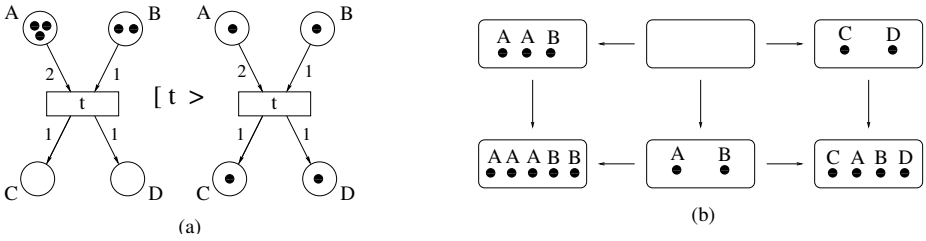


Figure 3.7: Firing of a transition and corresponding DPO derivation.

The considered encoding of nets into grammars further enlightens the dimensions in which graph grammars properly extends nets. First of all grammars allow for a more structured state, that is a general graph rather than a multiset (discrete graph). Even if (multi)sets are sufficient for many purposes, it is easy to believe that in more complex situations the state of a distributed system cannot be faithfully described just as a *set* of components, because also *relationships* among components should be represented. Thus graphs turn out to be more natural for representing distributed states. As a consequence, graph rewriting appears as a suitable formalism for describing the evolution of a wide class of systems, whose states have a natural distributed and interconnected nature.
Perhaps more interestingly, graph grammars allow for productions where the interface graph may not be empty, thus specifying a "context" consisting of items that have to be present for the productions to be applied, but are not affected by the application. In this respect, graph grammars are closer to some generalizations of nets in the literature, called nets with read (test) arcs or contextual nets (see e.g. [31,32,33]), which generalize classical nets by adding the possibility of checking for the presence of tokens which are not consumed.

3.3 Derivation trace semantics

Historically, the first truly concurrent semantics for graph transformation systems proposed in the literature has been the derivation trace semantics. It is based on the idea of defining suitable equivalences on concrete derivations, equating those derivations which should be considered undistinguishable according to the following two criteria:

- *irrelevance of representation details*, namely of the concrete identity of the items in the graphs involved in a derivation, and

- *true concurrency*, namely the irrelevance of the order in which independent productions are applied in a derivation.

The corresponding equivalences, called respectively *abstraction equivalence* and *shift equivalence*, are presented below. Concatenable derivation traces are then defined as equivalence classes of concrete derivations with respect to the least equivalence containing both the abstraction and the shift equivalences. Due to an appropriate choice of the abstraction equivalence, the obvious notion of sequential composition of concrete derivations induces an operation of sequential composition at the abstract level. Thus, as suggested by their name, concatenable derivation traces can be sequentially composed and therefore they can be seen as arrows of a category. Such category, called here the *category of concatenable derivation traces*, coincides with the *abstract truly concurrent model of computation* of a grammar presented in [5], namely the most abstract model in a hierarchy of models of computation for a graph grammar.

3.3.1 Abstraction equivalence and abstract derivations

Almost invariably, two isomorphic graphs are considered as representing the *same* system state, being such a state determined only by the topological structure of the graph and by the typing. This is extremely natural in the algebraic approach to graph transformation, where the result of the rewriting procedure is defined in terms of categorical constructions and thus determined only up to isomorphism.[b] A natural solution to reason in terms of *abstract graphs* and *abstract derivations* consists of considering two derivations as equivalent if the corresponding diagrams are isomorphic. Unfortunately, if one wants to have a meaningful notion of sequential composition between abstract derivations this approach does not work. For an extensive treatment of this problem we refer the reader to [34,15]. Roughly speaking, the difficulty can be described as follows. Two isomorphic graphs, in general, are related by more than one isomorphism, but if we want to concatenate derivations keeping track of the flow of causality we must specify in some way how the items of two isomorphic graphs have to be identified. The solution we propose is suggested by the theory of Petri nets, and in particular by the notion of concatenable net process [22,23], and borrows a technique of [35]. We choose for each class of isomorphic typed graphs a specific graph, called *canonical graph*, and we decorate the source and target graphs of a derivation with a pair of isomorphisms from the corresponding canonical graphs to such graphs. In such a way we are able to

[b] At the concrete level, the fact that the pushout and pushout complement constructions are defined only up to isomorphism generates an undesirable and scarcely intuitive *unbounded non-determinism* for each production application.

distinguish "equivalent"[c] elements in the source and target graphs of derivations and we can safely define their sequential composition. An alternative equivalent solution has been proposed in [34,15], making use of a distinguished class of *standard isomorphisms*. We refer the reader to Section 3.7 for a discussion about the relationship between the two techniques.

Definition 3.3.1 (canonical graphs)
We denote by Can the operation that associates to each (TG-typed) graph its *canonical graph*, satisfying the following properties:

1. $Can(G) \simeq G$;
2. if $G \simeq G'$ then $Can(G) = Can(G')$.

The construction of the canonical graph can be performed by adapting to our slightly different framework the ideas of [35] and a similar technique can be used to single out a class of standard isomorphisms in the sense of [34,15]. Working with finite graphs the constructions are effective. The Appendix gives a detailed description of a concrete construction of the canonical graph.

Definition 3.3.2 (decorated derivation)
A *decorated derivation* $\psi : G_0 \Rightarrow^* G_n$ is a triple $\langle m, \rho, M \rangle$, where $\rho : G_0 \Rightarrow^* G_n$ is a derivation and $m : Can(G_0) \to G_0$, $M : Can(G_n) \to G_n$ are isomorphisms. If ρ is an identity derivation then ψ is called *discrete*.

In the following we denote the components of a decorated derivation ψ by m_ψ, ρ_ψ and M_ψ. For a decorated derivation ψ, we write $\sigma(\psi)$, $\tau(\psi)$, $\#\psi$, $|\psi|$, $prod(\psi)$ to refer to the results of the same operations applied to the underlying derivation ρ_ψ.

Definition 3.3.3 (sequential composition)
Let ψ and ψ' be two decorated derivations such that $\tau(\psi) = \sigma(\psi')$ and $M_\psi = m_{\psi'}$. Their *sequential composition*, denoted by $\psi \,;\, \psi'$, is defined as follows:

$$\langle m_\psi, \rho_\psi \,;\, \rho_{\psi'}, M_{\psi'} \rangle.$$

One could have expected sequential composition of decorated derivations ψ and ψ' to be defined whenever $\tau(\psi) \simeq \sigma(\psi')$, regardless of the concrete identity of the items in the two graphs. We decided to adopt a more concrete notion of concatenation since it is technically simpler and it induces, like the more general one, the desired notion of sequential composition at the abstract level.

[c] With "equivalent" we mean here two items related by an automorphism of the graph, that are, in absence of further informations, indistinguishable.

The abstraction equivalence identifies derivations that differ only for representation details. As announced it is a suitable refinement of the natural notion of diagram isomorphism.

Definition 3.3.4 (abstraction equivalence)
Let ψ and ψ' be two decorated derivations, with $\rho_\psi : G_0 \Rightarrow^* G_n$ and $\rho_{\psi'} : G'_0 \Rightarrow^* G'_{n'}$ (whose i^{th} steps are depicted in the low rows of Figure 3.8). Suppose that $q_i = q_{i1} + \ldots + q_{ik_i}$ for each $i \in \underline{n}$, and $q'_j = q'_{j1} + \ldots + q'_{jk'_j}$ for each $j \in \underline{n'}$. Then they are *abstraction equivalent*, written $\psi \equiv^{abs} \psi'$, if

1. $n = n'$, i.e., they have the same length;
2. for each $i \in \underline{n}$, $k_i = k'_i$ and for all $s \in \underline{k_i}$, $q_{is} = q'_{is}$; i.e., the productions applied in parallel at each direct derivation are the same and they are composed in the same order; in particular $\#\psi = \#\psi'$;
3. there exists a family of isomorphisms

$$\{\theta_{X_i} : X_i \to X'_i \mid X \in \{L, K, R, G, D\},\ i \in \underline{n}\} \cup \{\theta_{G_0}\}$$

between corresponding graphs appearing in the two derivations such that

   (a) the isomorphisms relating the source and target graphs commute with the decorations, i.e., $\theta_{G_0} \circ m = m'$ and $\theta_{G_n} \circ M = M'$;
   (b) the resulting diagram commutes (the middle part of Figure 3.8 represents the portion of the diagram relative to step i, indicating only the s^{th} of the k_i productions applied in parallel with the corresponding injections).[d]

Notice that two derivations are abstraction equivalent if, not only they have the same length and apply the same productions in the same order, but also, in a sense, productions are applied to "corresponding" items (Condition 3). In other words abstraction equivalence identifies two decorated derivations if one can be obtained from the other by uniformly renaming the items appearing in the involved graphs.
Relation $\equiv^{abs}$ is clearly an equivalence relation. Equivalence classes of decorated derivations with respect to $\equiv^{abs}$ are called *abstract derivations* and are denoted by $[\psi]_{abs}$, where ψ is an element of the class.
It is easy to prove that if $\psi \equiv^{abs} \psi'$ and $\psi_1 \equiv^{abs} \psi'_1$ then, if defined, $\psi; \psi_1 \equiv^{abs} \psi'; \psi'_1$. Therefore sequential composition of decorated derivations lifts to composition of abstract derivations.

[d] Notice that the isomorphisms θ_{X_i} for $X \in \{L, K, R\}$, relating corresponding parallel productions, are uniquely determined by the properties of coproducts.

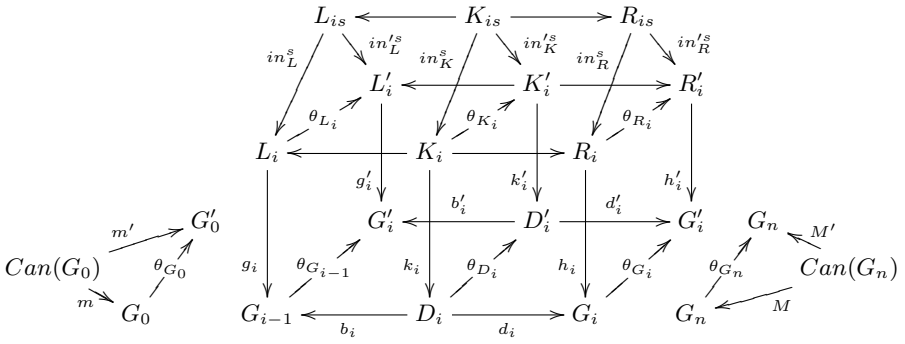


Figure 3.8: Abstraction equivalence of decorated derivations (the arrows in productions spans are not labelled).

Definition 3.3.5 (category of abstract derivations)
The *category of abstract derivations* of a grammar $\mathcal{G}$, denoted by $\mathbf{Abs}[\mathcal{G}]$, has abstract graphs as objects, and abstract derivations as arrows. In particular, if $\psi : G \Rightarrow^* H$, then $[\psi]_{abs}$ is an arrow from $[G]$ to $[H]$. The identity arrow on $[G]$ is the abs-equivalence class of a discrete derivation $\langle i, G, i\rangle$, where $i : Can(G) \to G$ is any isomorphism, and the composition of arrows $[\psi]_{abs} : [G] \to [H]$ and $[\psi']_{abs} : [H] \to [X]$ is defined as $[\psi\,;\,\psi'']_{abs} : [G] \to [X]$, where $\psi'' \in [\psi']_{abs}$ is such that the composition is defined.

It is worth stressing that, whenever $\tau(\psi) \simeq \sigma(\psi')$, we can always rename the items in the graphs of ψ', in order to obtain a derivation ψ'', abs-equivalent to ψ' and composable with ψ, namely such that $\tau(\psi) = \sigma(\psi'')$ and $M_\psi = m_{\psi''}$. Basically, it suffices to substitute in the derivation ψ' each item x in $\sigma(\psi')$ with $M_\psi(m^{-1}_{\psi'}(x))$.

3.3.2 Shift equivalence and derivation traces

From a truly concurrent perspective two derivations should be considered as equivalent when they apply the same productions to the "same" subgraph of a certain graph, even if the order in which the productions are applied may be different. The basic idea of equating derivations which differ only for the order of independent production applications is formalized in the literature through the notion of *shift equivalence* [14,36,4]. The shift equivalence is based on the possibility of sequentializing a parallel direct derivation (the *analysis*

construction) and on the inverse construction (*synthesis*), which is possible only in the case of *sequential independence*. The union of the shift and abstraction equivalences will yield the *(concatenable) truly concurrent* equivalence, whose equivalence classes are the *(concatenable) derivation traces*.

Let us start by defining the key notion of sequential independence.

Definition 3.3.6 (sequential independence)
A derivations $\delta_1;\delta_2$, consisting of two direct derivations $\delta_1 : G \Rightarrow_{q',g_1} X$ and $\delta_2 : X \Rightarrow_{q'',g_2} H$ (as in Figure 3.9) is *sequentially independent* if $g_2(L_2) \cap h_1(R_1) \subseteq g_2(l_2(K_2)) \cap h_1(r_1(K_1))$; in words, if the images in X of the left-hand side of q'' and of the right-hand side of q' overlap only on items that are preserved by both derivation steps. In categorical terms, this condition can be expressed by requiring the existence of two arrows $s : L_2 \to D_1$ and $u : R_1 \to D_2$ such that $d_1 \circ s = g_2$ and $b_2 \circ u = h_1$.

Example 3.3 (sequential independence)
Consider the derivation of Figure 3.5. The first two direct derivations are not sequential independent; in fact, the edge $\underline{3}:req$ of graph G_1 is in the image of both the right-hand side of the first production and the left-hand side of the second one, but it is in the context of neither the first nor the second direct derivation. On the contrary, in the same figure both the derivation from G_1 to G_3 and that from G_2 to G_4 are sequential independent. □

Notice that, differently from what happens for other formalisms, such as Petri nets or term rewriting, two rewriting steps δ_1 and δ_2 do not need to be applied at completely disjoint matches to be independent. The graphs to which δ_1 and δ_2 are applied can indeed overlap on something that is preserved by both rewriting steps. According to the interpretation of preserved items as read-only resources, this fact can be expressed by saying that graph rewriting allows for the *concurrent access to read resources*.
The next well-known result states that every parallel direct derivation can be sequentialized in an arbitrary way as the sequential application of the component productions, and, conversely, that every sequential independent derivation can be transformed into a parallel direct derivation. These constructions, in general non-deterministic, are used to define suitable relations among derivations (see, e.g., [14,37,5]). Unlike the usual definition of analysis and synthesis, we explicitly keep track of the permutation of the applied productions induced by the constructions. Therefore we first introduce some notation for permutations.

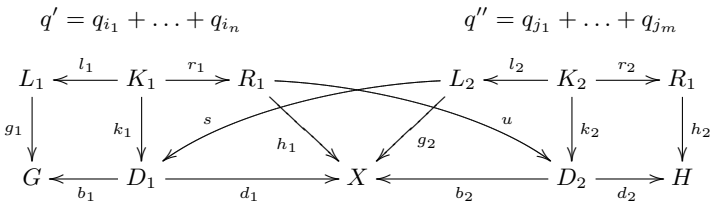


Figure 3.9: Sequential independent derivations.

Definition 3.3.7 (permutations)
A *permutation* on the set $\underline{n} = \{1, 2, \ldots, n\}$ is a bijective mapping $\Pi : \underline{n} \to \underline{n}$. The *identity permutation* on $\underline{n}$ is denoted by Π^n_{id}. The *composition* of two permutations Π_1 and Π_2 on $\underline{n}$, denoted by $\Pi_1 \circ \Pi_2$, is their composition as functions, while the *concatenation* of two permutations Π_1 on $\underline{n_1}$ and Π_2 on $\underline{n_2}$, denoted by $\Pi_1 \mid \Pi_2$, is the permutation on $\underline{n_1 + n_2}$ defined as

$$\Pi_1 \mid \Pi_2(x) = \begin{cases} \Pi_1(x) & \text{if } 1 \leq x \leq n_1 \\ \Pi_2(x - n_1) + n_1 & \text{if } n_1 < x \leq n_2 \end{cases}$$

Concatenation and composition of permutations are clearly associative.

Proposition 3.3.8 (analysis and synthesis)
Let $\rho : G \Rightarrow_q H$ be a parallel direct derivation using $q = q_1 + \ldots + q_k$: $(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$. Then for each partition $\langle I = \{i_1, \ldots, i_n\}, J = \{j_1, \ldots, j_m\}\rangle$ of $\underline{k}$ (i.e., $I \cup J = \underline{k}$ and $I \cap J = \emptyset$) there is a constructive way—in general non-deterministic—to obtain a sequential independent derivation $\rho' : G \Rightarrow_{q'} X \Rightarrow_{q''} H$, called an analysis *of ρ, where $q' = q_{i_1} + \ldots + q_{i_n}$, and $q'' = q_{j_1} + \ldots + q_{j_m}$ as in Figure 3.9. If ρ and ρ' are as above, we shall write $\rho \equiv^{an}_{\Pi} \rho'$, where Π is the permutation on $\underline{k}$ defined as $\Pi(i_x) = x$ for $x \in \underline{n}$, and $\Pi(j_x) = n + x$ for $x \in \underline{m}$.*
Conversely, let $\rho : G \Rightarrow_{q'} X \Rightarrow_{q''} H$ be a sequential independent derivation. Then there is a constructive way to obtain a parallel direct derivation $\rho' = G \Rightarrow_{q'+q''} H$, called a synthesis *of ρ. In this case, we shall write $\rho \equiv^{syn}_{\Pi} \rho'$, where $\Pi = \Pi^{\#\rho}_{id}$.* □

Informally, two derivations are shift equivalent if one can be obtained from the other by repeatedly applying the analysis and synthesis constructions. The next definition emphasizes the fact that the sets of productions applied in two

shift equivalent derivations are related by a permutation which is constructed inductively starting from the permutations introduced by analysis and synthesis.

Definition 3.3.9 (shift equivalence)
Derivations ρ and ρ' are *shift equivalent via permutation* Π, written $\rho \equiv^{sh}_{\Pi} \rho'$, if this can be deduced by the following inference rules:

$$(\mathsf{SH-id})\ \frac{}{\rho \equiv^{sh}_{\Pi^{\#\rho}_{id}} \rho} \qquad (\mathsf{SH-\emptyset})\ \frac{d \circ b^{-1} = id_G}{G \equiv^{sh}_{\emptyset} G \overset{\langle \emptyset,\emptyset,\emptyset,b,d\rangle}{\Rightarrow}_{\emptyset} G} \qquad (\mathsf{SH-an})\ \frac{\rho \equiv^{an}_{\Pi} \rho'}{\rho \equiv^{sh}_{\Pi} \rho'}$$

$$(\mathsf{SH-syn})\ \frac{\rho \equiv^{syn}_{\Pi} \rho'}{\rho \equiv^{sh}_{\Pi} \rho'} \quad (\mathsf{SH-sym})\ \frac{\rho \equiv^{sh}_{\Pi} \rho'}{\rho' \equiv^{sh}_{\Pi^{-1}} \rho} \quad (\mathsf{SH-trans})\ \frac{\rho \equiv^{sh}_{\Pi} \rho',\ \rho' \equiv^{sh}_{\Pi'} \rho''}{\rho \equiv^{sh}_{\Pi' \circ \Pi} \rho''}$$

$$(\mathsf{SH-comp})\ \frac{\rho_1 \equiv^{sh}_{\Pi_1} \rho_1',\ \rho_2 \equiv^{sh}_{\Pi_2} \rho_2',\ \tau(\rho_1) = \sigma(\rho_2)}{\rho_1 \,;\, \rho_2 \equiv^{sh}_{\Pi_1 | \Pi_2} \rho_1' \,;\, \rho_2'}$$

Note that by $(\mathsf{SH} - \emptyset)$ an empty direct derivation is shift equivalent to the identity derivation G if and only if the induced isomorphism is the identity. The *shift equivalence* is the equivalence relation $\equiv^{sh}$ defined as $\rho \equiv^{sh} \rho'$ iff $\rho \equiv^{sh}_{\Pi} \rho'$ for some permutation Π.

It is worth stressing that the shift equivalence abstracts both from the order in which productions are composed inside a single direct parallel step and from the order in which independent productions are applied at *different* direct derivations.

Example 3.4 (shift equivalence)
Figure 3.10 shows a derivation ρ' which is shift equivalent to derivation ρ of Figure 3.5. It is obtained by applying the synthesis construction to the subderivation of ρ from G_1 to G_3. □

Despite the unusual definition, borrowed from [18], it is easy to check that the shift equivalence just introduced is the same as in [14,4,15]. From the definitions of the shift equivalence and of the analysis and synthesis constructions, it follows that $\rho \equiv^{sh} \rho'$ implies that ρ and ρ' have the same order and the same source and target graphs (i.e., $\#\rho = \#\rho'$, $\sigma(\rho) = \sigma(\rho')$, and $\tau(\rho) = \tau(\rho')$; by the way, this guarantees that rules $(\mathsf{SH} - \mathsf{comp})$ and $(\mathsf{SH} - \mathsf{trans})$ are well-defined. The shift equivalence can be extended in a natural way to decorated derivations.

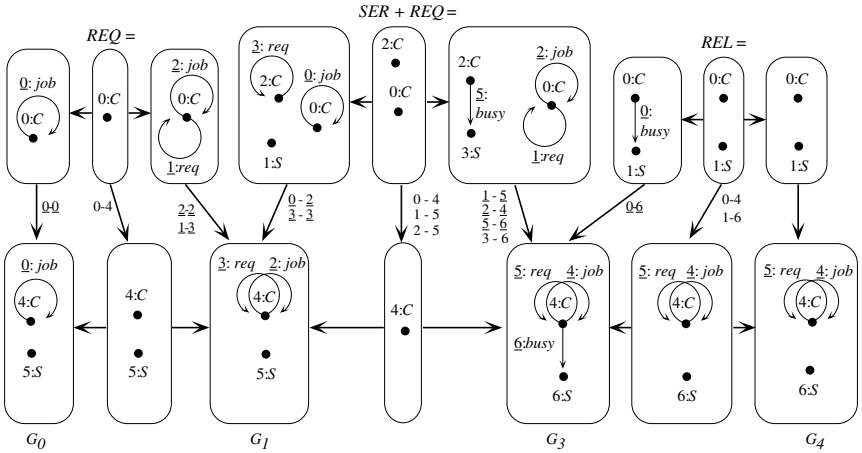


Figure 3.10: A derivation ρ' in grammar $\mathcal{C}$-$\mathcal{S}$, shift-equivalent to derivation ρ of Figure 3.5.

Definition 3.3.10
The *shift equivalence* on decorated derivations, denoted with the same symbol $\equiv^{sh}$, is defined by $\langle m, \rho, M\rangle \equiv^{sh} \langle m, \rho', M\rangle$ if $\rho \equiv^{sh} \rho'$.

Equivalence $\equiv^{sh}$ does not subsume abstraction equivalence, since, for example, it cannot relate derivations starting from different but isomorphic graphs. We introduce a further equivalence on decorated derivations, obtained simply as the union of $\equiv^{abs}$ and $\equiv^{sh}$, and we call it *truly-concurrent* (or *tc-*) equivalence, since in our view it correctly equates all derivations which are not distinguishable from a true concurrency perspective, at an adequate degree of abstraction from representation details. A small variation of this equivalence is introduced as well, called *ctc-equivalence*, where the first "c" stays for "concatenable". Equivalence classes of (c)tc-equivalent decorated derivations are called *(concatenable) derivation traces*, a name borrowed from [38].

Definition 3.3.11 (truly-concurrent equivalences and traces)
Two decorated derivations ψ and ψ' are *ctc-equivalent via permutation* Π, written $\psi \equiv^{c}_{\Pi} \psi'$, if this can be deduced by the following inference rules:

$$(\mathsf{CTC{-}abs})\ \frac{\psi \equiv^{abs} \psi'}{\psi \equiv^{c}_{\Pi^{\#_\psi}_{id}} \psi'} \quad (\mathsf{CTC{-}sh})\ \frac{\psi \equiv^{sh}_{\Pi} \psi'}{\psi \equiv^{c}_{\Pi} \psi'} \quad (\mathsf{CTC{-}trans})\ \frac{\psi \equiv^{c}_{\Pi} \psi' \quad \psi' \equiv^{c}_{\Pi'} \psi''}{\psi \equiv^{c}_{\Pi' \circ \Pi} \psi''}$$

Equivalence $\equiv^{c}$, defined as $\psi \equiv^{c} \psi'$ iff $\psi \equiv^{c}_{\Pi} \psi'$ for some permutation Π, is called the *concatenable truly concurrent (ctc-) equivalence*. Equivalence classes

of derivations with respect to $\equiv^c$ are denoted as $[\psi]_c$ and are called *concatenable derivation traces*. A *derivation trace* is an equivalence class of derivations with respect to the *truly-concurrent (tc-) equivalence* $\equiv$ defined by the following rules:

$$(\mathsf{TC-ctc})\ \frac{\psi \equiv^c_\Pi \psi'}{\psi \equiv_\Pi \psi'} \quad (\mathsf{TC-iso})\ \frac{\psi \equiv_\Pi \psi' \quad \alpha \text{ discrete decor. deriv. s.t. } \psi' ; \alpha \text{ is defined}}{\psi \equiv_\Pi \psi' ; \alpha}$$

Equivalently, we could have defined tc-equivalence as $\psi \equiv_\Pi \psi'$ if and only if $\psi \equiv^c_\Pi \langle m_{\psi'}, \rho_{\psi'}, M' \rangle$, for some isomorphism $M' : Can(\tau(\psi')) \to \tau(\psi')$. Informally, differently from ctc-equivalence, tc-equivalence does not take into account the decorations of the target graphs of derivations, that is their ending interface, and this is the reason why derivation traces are not concatenable. Derivation traces will be used in Section 3.6 to define an event structure semantics for a graph grammar.
Concatenable derivation traces, instead, are naturally equipped with an operation of sequential composition, inherited from concrete decorated derivations, which allows us to see them as arrows of a category having abstract graphs as objects. Such category is called the category of concatenable derivation traces (or the abstract truly concurrent model of computation) of the grammar.

Definition 3.3.12 (category of concatenable derivation traces)
The *category of concatenable derivation traces* of a grammar $\mathcal{G}$, denoted by $\mathbf{Tr}[\mathcal{G}]$, is the category having abstract graphs as objects, and concatenable derivation traces as arrows. In particular, if $\psi : G \Rightarrow^*_{\mathcal{G}} H$ then $[\psi]_c$ is an arrow from $[G]$ to $[H]$. The identity arrow on $[G]$ is the ctc-equivalence class of a discrete derivation $\langle i, G, i \rangle$, where i is any isomorphism from $Can(G)$ to G. The composition of arrows $[\psi]_c : [G] \to [H]$ and $[\psi']_c : [H] \to [X]$ is defined as $[\psi ; \psi'']_c : [G] \to [X]$, where $\psi'' \in [\psi']_c$ is a decorated derivation such that $\psi ; \psi''$ is defined.

Category $\mathbf{Tr}[\mathcal{G}]$ is well-defined because so is the sequential composition of arrows: in fact, if $\psi_1 \equiv^c \psi_2$ and $\psi'_1 \equiv^c \psi'_2$ then (if defined) $\psi_1 ; \psi'_1 \equiv^c \psi_2 ; \psi'_2$ (hence the attribution "concatenable"). As for abstract derivations, whenever $\tau(\psi) \simeq \sigma(\psi')$, it is always possible to concatenate the corresponding traces, namely one can always find a derivation $\psi'' \in [\psi']_c$ such that $\psi ; \psi''$ is defined.

3.4 Process Semantics

Derivation traces represent (truly concurrent) deterministic computations of a grammar, but they do not provide an explicit characterization of the events

occurring in computations and of the causal relationships among them. Graph processes arise from the idea of equipping graph grammars with a semantics which on the one hand explicitly represent events and relationships among them, and on the other hand uses graph grammars themselves as semantic domain.

Analogously to what happens for Petri nets, a *graph process* [16] of a graph grammar $\mathcal{G}$ is defined as an "occurrence grammar" $\mathcal{O}$, i.e., a grammar satisfying suitable acyclicity and conflict freeness constraints, equipped with a mapping from $\mathcal{O}$ to $\mathcal{G}$. This mapping is used to associate to the derivations in $\mathcal{O}$ corresponding derivations in $\mathcal{G}$. The basic property of a graph process is that the derivations in $\mathcal{G}$ which are in the range of such mapping constitute a full class of *shift*-equivalent derivations. Therefore the process can be regarded as an abstract representation of such a class and plays a rôle similar to a *canonical derivation* [14].

Graph processes are not naturally endowed with a notion of sequential composition, essentially because of the same problem described for abstract derivations: if the target graph of a process is isomorphic to the source graph of a second process, the naïve idea of composing the two processes by gluing the two graphs according to an isomorphism does not work. In fact, in general we can find several distinct isomorphisms relating two graphs, which may induce sequential compositions of the two processes which substantially differ from the point of view of causality. Using the same technique adopted for derivations, *concatenable graph processes* [17] are defined as graph processes enriched with the additional information (a decoration of the source and target graphs) needed to concatenate them.

3.4.1 Graph Processes

Classically, in the theory of Petri net, processes have been considered only for T-restricted nets, where transitions have non-empty precondition and postcondition, or for consuming nets, where at least the precondition of each transition is non-empty. Analogously, the theory of graph processes for DPO grammars [16,17] has been developed just for consuming grammars. As for nets, such a choice is motivated by the fact that in the non-consuming case some technical and conceptual difficulties arise, essentially related to the possibility of applying in parallel several (causally) indistinguishable copies of the same production (*autoconcurrency*). However, as we will see in Section 3.6, the problem becomes serious only when trying to define an event structure semantics, since the event structure generated for a non consuming grammar is no more a meaningful representation of the behaviour of the grammar. To

make clear which kind of problems arise and to stress where the hypothesis of having consuming grammars is strictly necessary, in this presentation we take a more liberal view, deferring the restriction to consuming grammars as long as possible.
In the sequel it will be useful to consider graphs as unstructured sets of nodes and arcs. The following definition provides some necessary notation.

Definition 3.4.1 (items of a graph)
For a given graph G, with $Items(G)$ we denote the disjoint union of nodes and edges of G; for simplicity, we will often assume that all involved sets are disjoint, to be allowed to see $Items(G)$ as a set-theoretical union.
Given a set $S \subseteq Items(G)$ of items of G we denote by N_S the subset of S containing only nodes, namely $S \cap N_G$. Similarly E_S denotes the subset of arcs in S, namely $S \cap E_G$. Finally $Gr(S)$ denotes the structure $\langle N_S, E_S, s_{|E_S}, t_{|E_S} \rangle$, where s and t are the source and target functions of G. Notice that this is a well-defined graph only if $s(E_S) \cup t(E_S) \subseteq N_S$.

In the following, by L_q (resp. K_q, R_q) we will sometimes denote the graph L (resp., K, R) of a production $q : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$. Moreover, when we want to specify the typing of L_q, K_q, and R_q, we will assume that $L_q = \langle LG_q, tl_q \rangle$, $K_q = \langle KG_q, tk_q \rangle$, and $R_q = \langle RG_q, tr_q \rangle$.
The basic ingredient for the definition of graph processes is the notion of occurrence grammar, which is a special kind graph grammar satisfying suitable acyclicity and conflict freeness constraints. Each production of an occurrence grammar underlying a process is intended to represent a precise occurrence of an application of a production in the original grammar. Therefore the "meaningful" derivations of an occurrence grammar, representing computations of the original grammar, are only those where each production is executed exactly once. In the consuming case the fact that each production of an occurrence grammar is applied just once is ensured by the structure of the grammar itself, while in the possibly non-consuming case, this restriction must be "imposed" explicitly. Consequently the grammar underlying a graph process will be required to be "safe" only with respect to derivations which apply each production at most once. The usual notion of strong safety and the weaker one needed for non-consuming grammars are formalized as follows.

Definition 3.4.2 (safety properties for graph grammars)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a graph grammar. A TG-typed graph $\langle G, m \rangle$ is *injective* if the typing morphism m is injective. The grammar $\mathcal{G}$ is called *quasi safe* if for all derivations $G_s \Rightarrow^* H$ using each production at most once,

the graph H is injective. The grammar $\mathcal{G}$ is called *strongly safe* if the above condition holds for general derivations.

The definition is easily understood by observing that if we think of nodes and arcs of the type graph as a generalization of places in Petri nets, then the condition of safety for a marking (each place of the net is filled with at most one token) is naturally generalized to typed graphs by requiring the injectivity of the typing morphism.
We recall that if the gluing conditions are satisfied, then the application of a production to a graph G via the match g (which may be non-injective) can be thought of, at a concrete level, as removing $g(L - l(K))$ from G and then adding $h(R - r(K))$ (see Figure 3.4 to understand the meaning of g and h) [4]. Intuitively, if we consider injective morphisms as inclusions then all graphs reachable in a (quasi) safe grammar (by applying each production at most once) can be seen as subgraphs of the graph of types. As a consequence, we will say that a production, according to his typing, *creates*, *preserves* or *consumes* an item of the type graph. Using a (contextual) net-like language, we speak then of *pre-set* ${}^\bullet q$, *context* $\underline{q}$ and *post-set* $q^\bullet$ of a production q. The notions of pre-set, post-set and context of a production, as well as the notion of causality have a clear interpretation only under some safety condition. However for technical reasons it is preferable to define them for general graph grammars.

Definition 3.4.3 (pre-set, post-set, context)
Let $\mathcal{G}$ be a graph grammar. For any $q \in P$ we define

$${}^\bullet q = \mathit{Items}(tl_q(LG_q - l_q(KG_q))) \qquad q^\bullet = \mathit{Items}(tr_q(RG_q - r_q(KG_q)))$$
$$\underline{q} = \mathit{Items}(tk_q(KG_q))$$

and we say that q *consumes*, *creates* and *preserves* items in ${}^\bullet q$, $q^\bullet$ and $\underline{q}$, respectively. Similarly for a node or arc $x \in \mathit{Items}(TG)$ we write ${}^\bullet x$, $x^\bullet$ and $\underline{x}$ to denote the sets of productions which create, consume and preserve x, respectively.

Then a notion of causality between items of the type graph and productions can be defined naturally as follows:

Definition 3.4.4 (causal relation)
The *causal relation* of a grammar $\mathcal{G}$ is given by the structure $\langle \mathit{Elem}(\mathcal{G}), < \rangle$, where $\mathit{Elem}(\mathcal{G})$ is the set $\mathit{Items}(TG) \cup P$ and $<$ is the transitive closure of the relation $\prec$ defined by the following clauses: for any node or arc x in TG, and for productions $q_1, q_2 \in P$

1. if $x \in {}^\bullet q_1$ then $x \prec q_1$;

2. if $x \in q_1{}^\bullet$ then $q_1 \prec x$;

3. if $(q_1{}^\bullet \cap \underline{q_2}) \cup (\underline{q_1} \cap {}^\bullet q_2) \neq \emptyset$ then $q_1 \prec q_2$;

As usual $\leq$ is the reflexive closure of $<$. Pairs of elements of $Elem(\mathcal{G})$ which are not related by the causal relation are said to be *concurrent*.

Thinking of $\mathcal{G}$ as (some kind of) safe grammar, the first two clauses of the definition of relation $\leq$ are obvious: each production depends on all the items in its left-hand side (but not in its interface graph) and causes all the items in its right-hand side (but not in the interface graph). The third one formalizes the fact that if an item is generated by q_1 and preserved by q_2, then q_2 cannot be applied before q_1, and, symmetrically, that if an item is preserved by q_1 and consumed by q_2, then in a computation where all productions are applied q_1 must precede q_2.

A (deterministic) occurrence grammar is a graph grammar where the causal relation is a partial order (thus there are no cycles), each item of the type graph is created (consumed) by at most one production, and all productions consume and produce items of the type graph with "multiplicity" one (i.e., the typing morphism is injective on the produced and consumed part). As a consequence the grammar is quasi safe and all its production are applicable.

Definition 3.4.5 (occurrence grammar)
An *occurrence grammar* is a graph grammar $\mathcal{O} = \langle TG, G_s, P, \pi \rangle$ such that

1. its causal relation $<$ is irreflexive and its reflexive closure $\leq$ is a partial order;

2. consider the sets Min of minimal elements and Max of maximal elements of $\langle Items(TG), \leq \rangle$; then $Min(\mathcal{O}) = Gr(Min)$ and $Max(\mathcal{O}) = Gr(Max)$ are well-defined subgraphs of TG; moreover G_s coincides with $Min(\mathcal{O})$, typed by the inclusion;

3. for all productions $q \in P$, the typing tl_q is injective on $LG_q - l_q(KG_q)$, and similarly tr_q is injective on $RG_q - r_q(KG_q)$.

4. for all $x \in Items(TG)$, x is consumed by at most one production in P, and it is created by at most one production in P (i.e., $|{}^\bullet x|, |x^\bullet| \leq 1$).

Since the start graph of an occurrence grammar $\mathcal{O}$ is determined by $Min(\mathcal{O})$, we often do not mention it explicitly.

Intuitively, the above conditions rephrase in the framework of graph grammars the analogous conditions of deterministic occurrence nets. Condition (1) requires causality to be acyclic. In particular, irreflexivity of $<$ disallows an item of the type graph to appear both in the pre-set and in the context (post-set) of the same production. Condition (2) forces the set of minimal items of the type graph to be a well-defined subgraph of the type graph TG, coinciding with the start graph of the grammar. Also the set of maximal items of the type graph is asked to form a subgraph of TG. It can be shown that this condition implies that for each $n \in N_{TG}$, $e \in E_{TG}$ such that $n \in \{s(e), t(e)\}$, and for any production $q \in P$, if $q \leq n$ then $q \leq e$, and if $n \leq q$ then $e \leq q$. Consequently the dangling condition is satisfied in every derivation consistent with causal dependency. Condition (3), instead, is closely related to safety and requires that each production consumes and produces items with multiplicity one. Together with irreflexivity of $<$, it disallows the presence of some productions which surely could never be applied, because they fail to satisfy the identification condition with respect to the typing morphism. Finally, Condition (4) requires the absence of backward and forward conflicts.

Proposition 3.4.6 (occurrence and safe grammars)
Each occurrence grammar $\mathcal{O}$ is quasi safe. Moreover if $\mathcal{O}$ is consuming then $\mathcal{O}$ is strongly safe.

Proof
Let $\mathcal{O} = \langle TG, P, \pi \rangle$ be an occurrence grammar and let $\rho : G_s \Rightarrow^* G_n$ be a derivation in $\mathcal{O}$, using each production of $\mathcal{G}$ at most once. To fix notation suppose $\rho = \{G_{i-1} \Rightarrow_{q_i} G_i\}_{i \in \underline{n}}$. Without loss of generality we can assume that each direct derivation of ρ applies a single production.
We show by induction on the order n of the derivation that the graph G_n is injective. For $n = 0$ just recall that G_s is a subgraph of TG, typed by the inclusion. If $n > 0$, by inductive hypothesis G_{n-1} is injective. Moreover the typing of q_n is injective on $RG_{q_n} - r_{q_n}(KG_{q_n})$. This observation, together with the fact that the items of the start graph have empty pre-set (they are minimal with respect to causality) and each item of the type graph is produced by at most one production implies that the graph G_n, which can be expressed as

$$G_{n-1} - (LG_{q_n} - l_{q_n}(KG_{q_n})) \cup (RG_{q_n} - r_{q_n}(KG_{q_n}))$$

is injective.
If the grammar is consuming, since each production has a non empty pre-set and causality $\leq$ is a partial order (there are no "cycles"), every derivation in the grammar starting from the start graph can apply each production at most

once. Thus for each such derivation the above reasoning applies and therefore the grammar is strongly safe. □

Traditionally, occurrence grammars have been defined only in the consuming case, and as strongly safe grammars satisfying Conditions (1)-(4) of Definition 3.4.5. The last proposition ensures that our notion of occurrence grammar extends the classical one in a consistent way, in the sense that they coincide when restricted to consuming grammars.

An occurrence grammar is a fully fledged graph grammar, thus one may "execute" it by considering the possible derivations beginning from the start graph $Min(\mathcal{O})$. However, it is natural to consider only those derivations which are consistent with the causal ordering. In fact for any finite subset of productions $P' \subseteq P$, closed with respect to causality, *all* productions in P' can be applied in a derivation with source in $Min(\mathcal{O})$, in any order compatible with $\leq$ (i.e., if $q_1 < q_2$ then q_1 is applied first). Furthermore, all subgraphs of the type graph reached in such derivations are made of concurrent nodes and edges, and can be characterized in an elegant way using the causal relation. Here is a summary of these results.

Definition 3.4.7 (reachable sets)
Let $\mathcal{O} = \langle TG, P, \pi \rangle$ be an occurrence grammar, and let $\langle P, \leq \rangle$ be the restriction of the causal relation to the productions of $\mathcal{O}$. For any finite left-closed $P' \subseteq P$,[e] the *reachable set* associated to P' is the set of nodes and arcs $S_{P'} \subseteq Items(TG)$ defined as

$$x \in S_{P'} \quad \text{iff} \quad \forall q \in P \,.\, (x \leq q \Rightarrow q \notin P') \wedge (x \geq q \Rightarrow q \in P').$$

Theorem 3.4.8
Let $\mathcal{O} = \langle TG, P, \pi \rangle$ be an occurrence grammar, and let $S_{P'}$ be a reachable set for some finite left-closed $P' \subseteq P$. Then

1. *The elements of $S_{P'}$ are pairwise concurrent, and $Gr(S_{P'})$ is a graph.*

2. *The graph $Gr(S_{P'})$ is reachable from the start graph $Min(\mathcal{O})$; more precisely, $Min(\mathcal{O}) \Rightarrow^*_{P'} Gr(S_{P'})$ with a derivation which applies exactly once every production in P', in any order consistent with $\leq$.* □

Notice that in particular $Min(\mathcal{O}) = Gr(S_\emptyset)$ and $Max(\mathcal{O}) = Gr(S_P)$. Thus, by the above theorem, if $\mathcal{O}$ is a finite occurrence grammar

$$Min(\mathcal{O}) \Rightarrow^*_P Max(\mathcal{O})$$

[e] A subset Y of an ordered set $\langle X, \leq \rangle$ is *left-closed* if for any $x \in X$ and $y \in Y$, $x \leq y$ implies $x \in Y$.

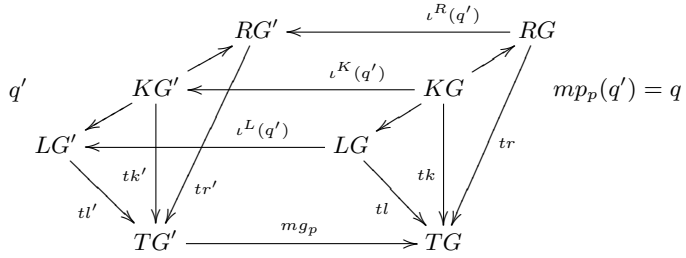


Figure 3.11: Mapping in a graph process.

using all productions in P exactly once, in any order consistent with $\leq$. This makes clear why a graph process of a grammar $\mathcal{G}$, that we are going to define as a finite occurrence grammar plus a mapping to the original grammar, can be seen as representative of a set of derivations of $\mathcal{G}$ where independent steps may be switched.

Definition 3.4.9 (graph process)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar. A *(finite) process* p for $\mathcal{G}$ is a pair $\langle \mathcal{O}_p, \phi_p \rangle$, where $\mathcal{O}_p = \langle TG_p, P_p, \pi_p \rangle$ is a finite occurrence grammar and $\phi_p = \langle mg_p, mp_p, \iota_p \rangle$, where

1. $mg_p : TG_p \to TG$ is a graph morphism;

2. $mp_p : P_p \to P$ is a function mapping each production $q' : (L' \leftarrow K' \rightarrow R')$ in P_p to an isomorphic production $q = mp_p(q') : (L \leftarrow K \rightarrow R)$ in P and

3. ι_p is a function mapping each production $q' \in P_p$ to a triple of isomorphisms $\iota_p(q') = \langle \iota_p^L(q') : L \to L', \iota_p^K(q') : K \to K', \iota_p^R(q') : R \to R' \rangle$, making the diagram in Figure 3.11 commute.

The graphs $Min(\mathcal{O})$ and $Max(\mathcal{O})$ typed over TG by the corresponding restrictions of mg_p, are denoted by $Min(p)$ and $Max(p)$ and called, respectively, the *source* and *target* graphs of the process.

Example 3.5 (process)
Figure 3.12 shows a process for grammar $\mathcal{C}$-$\mathcal{S}$ of Example 3.1. The typing morphisms from the productions of the process to TG_p are inclusions, and the

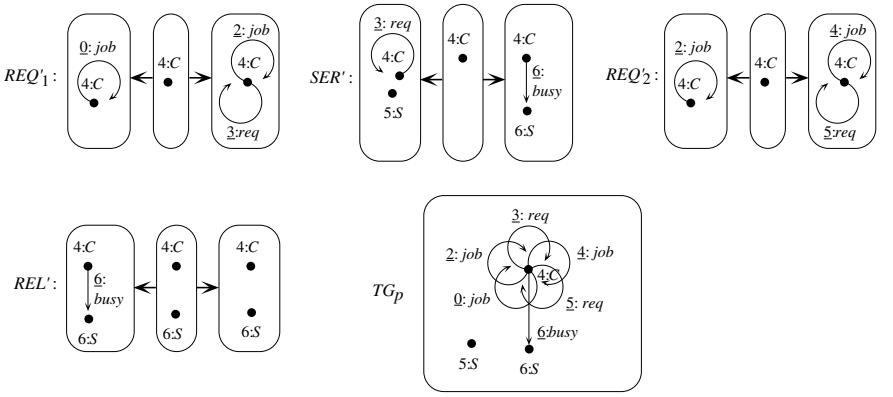


Figure 3.12: A graph process of grammar $\mathcal{C}$-$\mathcal{S}$.

start graph is the subgraph of TG_p containing items $5 : S$, $\underline{0} : job$ and $4 : C$ (thus exactly the start graph G_0 of $\mathcal{C}$-$\mathcal{S}$), because these are the only items which are not generated by any production.
The morphism $mg_p : TG_p \to TG$ is shown as usual by typing the items of TG_p with their image in TG, and the mapping from the productions of the process to that of $\mathcal{C}$-$\mathcal{S}$ is the obvious one. It is easy to check that all the conditions of Definition 3.4.5 are satisfied. □

Definition 3.4.10 (isomorphism of processes)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar. An *isomorphism* f between two processes p_1 and p_2 of $\mathcal{G}$ is a pair $\langle fg, fp \rangle : p_1 \to p_2$ such that

1. $fg : \langle TG_{p_1}, mg_{p_1} \rangle \to \langle TG_{p_2}, mg_{p_2} \rangle$ is an isomorphism (of TG-typed graphs);

2. $fp : P_{p_1} \to P_{p_2}$ is a bijection such that $mp_{p_1} = mp_{p_2} \circ fp$;

3. for each $q_1 : (L_1 \leftarrow K_1 \rightarrow R_1)$ in P_{p_1}, $q_2 = fp(q_1) : (L_2 \leftarrow K_2 \rightarrow R_2)$ in P_{p_2}, if $q = mp_{p_1}(q_1) = mp_{p_2}(q_2) : (L \leftarrow K \rightarrow R)$ in P, the diagram in Figure 3.13 commutes.

To indicate that p_1 and p_2 are *isomorphic* we write $p_1 \cong p_2$.

As one could expect a process isomorphism from p_1 to p_2 induces an isomorphism of ordered structures between $Elem(\mathcal{O}_{p_1})$ and $Elem(\mathcal{O}_{p_2})$.

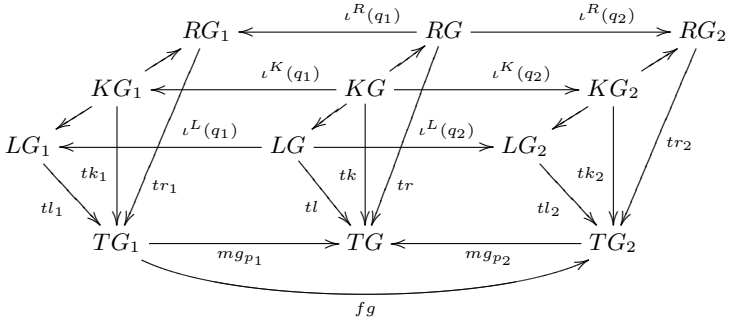


Figure 3.13: Graph process isomorphism.

Proposition 3.4.11
Let p_1 and p_2 be two processes of a grammar $\mathcal{G}$, and let $f = \langle fg, fp \rangle : p_1 \to p_2$ be an isomorphism. Then $fg \cup fp : \langle Elem(\mathcal{O}_{p_1}), \leq_{p_1} \rangle \to \langle Elem(\mathcal{O}_{p_2}), \leq_{p_2} \rangle$ is an isomorphism of partial orders.

Proof
By definition $fg \cup fp$ is a bijection. Moreover it is monotone. In fact let us look at the basic cases defining the causal relation $\leq_{p_1}$ (see Definition 3.4.4). For instance, for the first case suppose that $x, q \in Elem(\mathcal{O}_{p_1})$, with $x < q$ because $x \in tl_q(LG_q - l_q(KG_q))$; then we have $fg(x) \in fg(tl_q(LG_q - l_q(KG_q))) = tl_{fp(q)}(LG_{fp(q)} - KG_{fp(q)})$, by definition of process isomorphism. Therefore $fg(x) \leq_{p_2} fp(q)$. Also in the other three cases we proceed straightforwardly and thus we conclude that $fg \cup fg$ is monotone. In the same way $(fg \cup fp)^{-1}$ is monotone. Thus it is an isomorphism of partial orders. □

Corollary 3.4.12
Let p_1 and p_2 be two processes of a grammar $\mathcal{G}$. If $p_1 \cong p_2$ then $Min(p_1) \simeq Min(p_2)$ and $Max(p_1) \simeq Max(p_2)$, the isomorphisms being established by the corresponding restrictions of fg. □

3.4.2 Concatenable Graph Processes

A concatenable graph process is a graph process equipped with two isomorphisms relating its source and target graphs to the corresponding canonical graphs. The two isomorphisms allow us to define a deterministic operation of sequential composition of processes consistent with causal dependencies, which

is then exploited to define a category $\mathbf{CP}[\mathcal{G}]$ having abstract graphs as objects and concatenable processes as arrows.

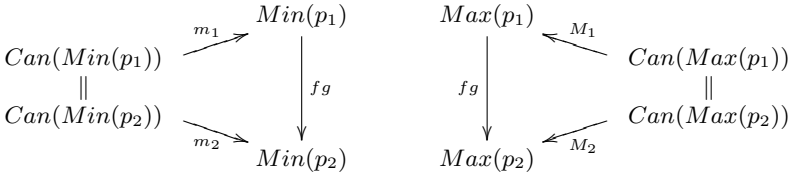
Definition 3.4.13 (concatenable process)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar. A *concatenable process (c-process)* for $\mathcal{G}$ is a triple

$$cp = \langle m, p, M \rangle$$

where p is a process and $m : Can(Min(p)) \to Min(p)$, $M : Can(Max(p)) \to Max(p)$ are isomorphisms (of TG-typed graphs). We denote with $Min(cp)$ and $Max(cp)$ the graphs $Min(p)$ and $Max(p)$.
If the occurrence grammar $\mathcal{O}_p$ associated to the process has an empty set of productions (and thus $Min(p) = Max(p) = TG_p$), the c-process cp is called a *discrete process* and denoted as $Sym_{\mathcal{G}}(m, \langle TG_p, mg_p \rangle, M)$.[f]

It should be noted that the two isomorphisms to the source and target graphs of a process play the same rôle of the ordering on maximal and minimal places of *concatenable processes* in Petri net theory [22]. From this point of view, *concatenable* graph processes are related to the graph processes of [16] in the same way as the concatenable processes of [22] are related to the classical Goltz-Reisig processes for P/T nets [9].
The notion of isomorphism between c-processes is the natural generalization of that of graph processes isomorphism, namely the mapping between the type graphs of the two processes is required to be "consistent" with the decorations.

Definition 3.4.14 (isomorphism of c-processes)
Let $cp_1 = \langle m_1, p_1, M_1 \rangle$ and $cp_2 = \langle m_2, p_2, M_2 \rangle$ be two c-processes of a grammar $\mathcal{G}$. An *isomorphism* between cp_1 and cp_2 is an isomorphism of processes $\langle fg, fp \rangle : p_1 \to p_2$ such that the following diagrams commute:



where $fg : Min(p_1) \to Min(p_2)$ and $fg : Max(p_1) \to Max(p_2)$ denote the restrictions of fg to the corresponding graphs.[g] If there exists an isomorphism

[f] We omit the typing morphism mg and denote the discrete process as $Sym_{\mathcal{G}}(m, TG_p, M)$, when this cannot generate confusion.

[g] As shown in Corollary 3.4.12, for each pair of isomorphic processes p_1 and p_2 the corresponding *Min* and *Max* graphs are isomorphic as well, i.e., $Min(p_1) \simeq Min(p_2)$ and $Max(p_1) \simeq Max(p_2)$

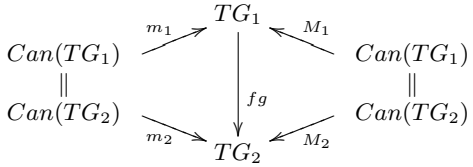
$f : cp_1 \to cp_2$ we say that cp_1 and cp_2 are *isomorphic* and we write $cp_1 \cong cp_2$.

Definition 3.4.15 (abstract c-process)
An *abstract c-process* is an isomorphism class of c-processes. It is denoted by $[cp]$, where cp is a member of that class.

Proposition 3.4.16
Let $\mathcal{G}$ be a graph grammar and let $Sym_{\mathcal{G}}(m_j, TG_j, M_j)$, for $j = 1, 2$, be two discrete processes, with $TG_1 \simeq TG_2$. Then they are isomorphic if and only if

$$m_1^{-1} \circ M_1 = m_2^{-1} \circ M_2.$$

Proof
First of all let us notice that in the case of discrete processes, since the sets of productions are empty and $Min(\mathcal{O}_j) = Max(\mathcal{O}_j) = TG_j$, using the fact that $TG_1 \simeq TG_2$, the isomorphism conditions reduce to the existence of a (typed) graph isomorphism $fg : \langle TG_1, mg_1 \rangle \to \langle TG_2, mg_2 \rangle$ such that the following diagram commutes:



Now if $m_1^{-1} \circ M_1 = m_2^{-1} \circ M_2$ then we can define $fg = M_2 \circ M_1^{-1} = m_2 \circ m_1^{-1}$. Viceversa if the discrete processes are isomorphic then, by commutativity of the above diagram, we conclude that $m_1^{-1} \circ M_1 = m_2^{-1} \circ M_2$. □
By the previous proposition, an abstract discrete process $[Sym_{\mathcal{G}}(m, TG, M)]$ can be characterized as:

$$\{Sym_{\mathcal{G}}(m', TG, M') \mid TG \simeq TG',\ m'^{-1} \circ M' = m^{-1} \circ M\}.$$

The isomorphism $m^{-1} \circ M$ is called the *automorphism on $Can(TG)$ induced by the (abstract) discrete process.*
Given two concatenable processes cp_1 and cp_2 such that the target graph of the first one and the source graph of the second one, as well as the corresponding decorations, coincide, we can concatenate them by gluing the type graphs along the common part.

Definition 3.4.17 (sequential composition)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar and let $cp_1 = \langle m_1, p_1, M_1 \rangle$ and $cp_2 = \langle m_2, p_2, M_2 \rangle$ be two c-processes for $\mathcal{G}$, such that $Max(cp_1) =$

$Min(cp_2)$ and $M_1 = m_2$. Suppose moreover that the type graphs of cp_1 and cp_2 overlap only on $Max(cp_1) = Min(cp_2)$ and suppose also P_{p_1} and P_{p_2} disjoint. Then the *concrete sequential composition* of cp_1 and cp_2, denoted by $cp_1 ; cp_2$ is defined as

$$cp = \langle m_1, p, M_2 \rangle,$$

where p is the (componentwise) union of the processes p_1 and p_2. More precisely the type graph TG_p is

$$TG_p = \langle N_{TG_{p_1}} \cup N_{TG_{p_2}}, E_{TG_{p_1}} \cup E_{TG_{p_2}}, s_1 \cup s_2, t_1 \cup t_2 \rangle,$$

where s_i and t_i are the source and target functions of the graph TG_{p_i} for $i \in \{1, 2\}$, and analogously the typing morphism $mg_p = mg_{p_1} \cup mg_{p_2}$. Similarly $P_p = P_{p_1} \cup P_{p_2}$, $\pi_p = \pi_{p_1} \cup \pi_{p_2}$, $mp_p = mp_{p_1} \cup mp_{p_2}$ and $\iota_p = \iota_{p_1} \cup \iota_{p_2}$. Finally, the start graph is $G'_s = Min(cp_1)$.

It is not difficult to check that the sequential composition $cp_1 ; cp_2$ is a well-defined c-process. First of all $\mathcal{O}_p$ is an occurrence grammar. In fact, TG_{p_1} and TG_{p_2} overlap only on $Max(cp_1) = Min(cp_2)$ and thus it is immediate to see that the causal relation $\leq_p$ on $\mathcal{O}_p$ is a partial order. Such relation can be expressed as the transitive closure of the union of the causal orders of the two processes with a "connection relation" r_c, suitably relating productions of cp_1 which use items of $Max(cp_1)$ with productions of cp_2 using the corresponding items of $Min(cp_2)$. Formally, $\leq_p = (\leq_{p_1} \cup \leq_{p_2} \cup r_c)^*$, where relation $r_c \subseteq P_{p_1} \times P_{p_2}$ is defined by

$$r_c = \left\{ \langle q_1, q_2 \rangle \mid \begin{array}{l} \exists x \in Max(cp_1) = Min(cp_2). \\ x \in (q_1{}^\bullet \cap {}^\bullet q_2) \cup (\underline{q_1} \cap {}^\bullet q_2) \cup (q_1{}^\bullet \cap \underline{q_2}) \end{array} \right\}.$$

A routine checking allows us to conclude that Conditions $1-4$ of the definition of occurrence grammar are satisfied. In particular $Min(cp') = Min(cp_1) = G'_s$ and each element in $Items(TG_p)$ is created (consumed) by at most one productions since elements "shared" by the original processes can be created only in cp_1 and consumed only in cp_2. Finally, the validity of conditions regarding ϕ_p (see Definition 3.4.9) easily follows from the way it is defined using ϕ_{p_1} and ϕ_{p_2}.

The reader could be surprised and somehow displeased by the restrictiveness of the conditions which have to be satisfied in order to be able to compose two concrete processes. To understand our restriction one should keep in mind that the notion of sequential composition on concrete processes is not interesting in itself, but it is just introduced to be lifted to a meaningful notion of sequential composition on abstract processes. Since the restricted definition fulfills this

aim, we found it better to avoid a technically more involved (although more general) definition, leading to a non-deterministic result. As in the case of derivations, in fact, processes can be seen up to renaming of the items in their components and thus, if $Max(cp_1) \simeq Min(cp_2)$, we can always rename the items of cp_2 to make the sequential composition defined.

Proposition 3.4.18
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar and let $cp_1 \cong cp'_1$ and $cp_2 \cong cp'_2$ be c-processes for $\mathcal{G}$. Then (if defined) $cp_1 ; cp_2 \cong cp'_1 ; cp'_2$.

Proof
Just notice that if $f_j = \langle fg_j, fp_j \rangle : cp_j \to cp'_j$ $(j = 1, 2)$ are c-process isomorphisms, then the isomorphism between $cp_1 ; cp_2$ and $cp'_1 ; cp'_2$ can be obtained as $\langle fg_1 \cup fg_2, fp_1 \cup fp_2 \rangle$. □
By the previous proposition we can lift the concrete operation of sequential composition of c-processes to abstract processes:

$$[cp_1] ; [cp_2] = [cp'_1 ; cp'_2]$$

for $cp'_1 \in [cp_1]$ and $cp'_2 \in [cp_2]$ such that the concrete composition is defined.

Definition 3.4.19 (category of concatenable processes)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar. We denote with $\mathbf{CP}[\mathcal{G}]$ the category of *(abstract) c-processes* having abstract graphs typed over TG as objects and abstract c-processes as arrows. An abstract c-process $[cp]$ is an arrow from $[Min(cp)]$ to $[Max(cp)]$. The identity on an abstract graph $[G]$ is the discrete process $[Sym_{\mathcal{G}}(i, G, i)]$ (where $i : Can(G) \to G$ is any isomorphism), whose induced automorphism is identity.

A routine checking proves that the operation o5f sequential composition on c-processes is associative and that $[Sym_{\mathcal{G}}(i, G, i)]$ satisfies the identity axioms.

3.5 Relating derivation traces and processes

Although based on the same fundamental ideas, namely abstraction from representation details and true concurrency, processes and derivation traces have concretely a rather different nature. Derivation traces provide a semantics for grammars where the independence of events is represented implicitly by collecting in the same trace derivations in which the events appear in different orders. Processes, instead, provide a partial order semantics which represents explicitly the events and their relationships. In this section we show that there

exists a close relationship between the trace and the process semantics introduced in the last two sections. More precisely we prove that the category $\mathbf{Tr}[\mathcal{G}]$ of concatenable (parallel) derivation traces (Definition 3.3.12) is isomorphic to the category of concatenable (abstract) processes $\mathbf{CP}[\mathcal{G}]$.

3.5.1 Characterization of the ctc-equivalence

The isomorphism result relies on a characterization of the ctc-equivalence on decorated derivations which essentially expresses the invariant of a derivation trace. Roughly speaking, such characterization formalizes the intuition that two derivations are (c)tc-equivalent whenever it is possible to establish a correspondence between the productions that they apply and between the graph items in the two derivations, in such a way that "corresponding" productions consume and produce "corresponding" graph items. The correspondence between the graph items has to be compatible with the decorations on the source (and target) graphs. These notions and results will also play a fundamental rôle, in the next section, for the definition of an event structure associated to a graph grammar.
The basic notions used in the characterization result presented below are those of consistent four-tuple and five-tuple.

Definition 3.5.1 (consistent four-tuples and five-tuples)
If $\rho : G_0 \Rightarrow^* G_n$ is the derivation depicted in Figure 3.6, let $\approx_\rho$ be the smallest equivalence relation on $\bigcup_{i=0}^{n} \mathit{Items}(G_i)$ containing relation $\sim_\rho$, defined as

$$\begin{aligned} x \sim_\rho y \quad \Leftrightarrow \quad & \exists r \in \underline{n} \,.\, x \in \mathit{Items}(G_{r-1}) \wedge y \in \mathit{Items}(G_r) \wedge \\ & \wedge \exists z \in \mathit{Items}(D_r) \,.\, b_r(z) = x \wedge d_r(z) = y. \end{aligned}$$

Denote by $\mathit{Items}(\rho)$ the set of equivalence classes of $\approx_\rho$, and by $[x]_\rho$ the class containing item x.[h] For a decorated derivation ψ, we will often write $\mathit{Items}(\psi)$ to denote $\mathit{Items}(\rho_\psi)$.
A four-tuple $\langle \rho, h_\sigma, f, \rho' \rangle$ is called *consistent* if ρ and ρ' are derivations, $h_\sigma : \sigma(\rho) \to \sigma(\rho')$ is a graph isomorphism between their source graphs, $f : \underline{\#\rho} \to \underline{\#\rho'}$ is an injective function such that $\mathit{prod}(\rho) = \mathit{prod}(\rho') \circ f$, and there exists a total function $\xi : \mathit{Items}(\rho) \to \mathit{Items}(\rho')$ satisfying the following conditions:

- $\forall x \in \mathit{Items}(\sigma(\rho)) \,.\, \xi([x]_\rho) = [h_\sigma(x)]_{\rho'}$, i.e., ξ must be consistent with isomorphism h_σ;

[h] Without loss of generality we assume here that the sets of items of the involved graphs are pairwise disjoint.

- for each $j \in \underline{\#\rho}$, let i and s be determined by $j = \left(\sum_{r=1}^{i} k_r\right) + s$ (i.e., the j-th production of ρ is the s-th production of its i-th parallel direct derivation), and similarly let s' and i' satisfy $f(j) = \left(\sum_{r=1}^{i'} k'_r\right) + s'$. Then for each item x "consumed" by production $prod(\rho)(j) : \left(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R\right)$, i.e., $x \in L - l(K)$, it must hold $\xi([g_i(in_L^s(x))]_\rho) = [g'_{i'}(in_L^{s'}(x))]_{\rho'}$. In words, ξ must relate the items consumed by corresponding production applications (according to f);

- A similar condition must hold for the items "created" by corresponding production applications. Using the above notations, for each $x \in R - r(K)$, $\xi([h_i(in_R^s(x))]_\rho) = [h'_{i'}(in_R^{s'}(x))]_{\rho'}$.

Similarly, say that the five-tuple $\langle \rho, h_\sigma, f, h_\tau, \rho' \rangle$ is *consistent* if the "underlying" four-tuple $\langle \rho, h_\sigma, f, \rho' \rangle$ is consistent, f is a bijection, $h_\tau : \tau(\rho) \to \tau(\rho')$ is an isomorphism relating the target graphs, and the function ξ is an isomorphism that is consistent with h_τ as well (i.e., for each item $x \in \tau(\rho)$, $\xi([x]_\rho) = [h_\tau(x)]_{\rho'}$).

The next theorem provides a characterization of (c)tc-equivalence in terms of consistent four- (five-)tuples. We first recall some easy but useful composition properties of consistent five-tuples.

Proposition 3.5.2

1. *If $\langle \rho_1, h_\sigma, f, h, \rho'_1 \rangle$ and $\langle \rho_2, h, f', h'_\tau, \rho_2 \rangle$ are consistent five-tuples, such that $\rho_1 ; \rho_2$ and $\rho'_1 ; \rho'_2$ are defined, then $\langle \rho_1 ; \rho_2, h_\sigma, f | f', h'_\tau, \rho'_1 ; \rho'_2 \rangle$ is consistent as well.*

2. *If $\langle \rho, h_\sigma, f, h_\tau, \rho' \rangle$ and $\langle \rho', h'_\sigma, f', h'_\tau, \rho'' \rangle$ are consistent five-tuples then $\langle \rho, h'_\sigma \circ h_\sigma, f' \circ f, h'_\tau \circ h_\tau, \rho'' \rangle$ is consistent as well.* □

Theorem 3.5.3 (characterization of ctc- and tc-equivalence)
Let $\mathcal{G}$ be any graph grammar (also non-consuming) and let ψ and ψ' be decorated derivations of $\mathcal{G}$. Then

1. *ψ and ψ' are ctc-equivalent if and only if there is a permutation Π such that the five-tuple $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, M_{\psi'} \circ M_\psi^{-1}, \rho_{\psi'} \rangle$ is consistent;*

2. *similarly, ψ and ψ' are tc-equivalent if and only if there is a permutation Π such that the four-tuple $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, \rho_{\psi'} \rangle$ is consistent.*

Proof
Only if part of (1):
We show by induction that if two derivations are ctc-equivalent, i.e., $\psi \equiv^{c}_{\Pi} \psi'$ for some permutation Π, then the five-tuple $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, M_{\psi'} \circ M_\psi^{-1}, \rho_{\psi'} \rangle$ is consistent. By Definition 3.3.11, we have to consider the following cases:

($\mathsf{CTC-abs}$) If $\psi \equiv^{abs} \psi'$ then consistency of $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi_{id}^{\#\psi}, M_{\psi'} \circ M_\psi^{-1}, \rho_{\psi'} \rangle$ follows directly by the conditions of Definition 3.3.4;

($\mathsf{CTC-sh}$) Since $\psi \equiv^{sh}_{\Pi} \psi'$ we have $m_\psi = m_{\psi'}$, $M_\psi = M_{\psi'}$ and $\rho_\psi \equiv^{sh}_{\Pi} \rho_{\psi'}$. Therefore this case reduces to the proof that $\rho \equiv^{sh}_{\Pi} \rho'$ implies the consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi, id_{\tau(\rho)}, \rho' \rangle$. According to the rules of Definition 3.3.9, introducing shift equivalence on parallel derivations, we distinguish various cases:

- ($\mathsf{SH-id}$) The five-tuple $\langle \rho, id_{\sigma(\rho)}, \Pi_{id}^{\#\rho}, id_{\tau(\rho)}, \rho \rangle$ is trivially consistent;
- ($\mathsf{SH-\emptyset}$) The five-tuple $\langle G, id_G, \emptyset, id_G, \rho : G \Rightarrow_\emptyset G \rangle$ is consistent, since the isomorphism induced by ρ is id_G;
- ($\mathsf{SH-an}$) If $\rho \equiv^{an}_{\Pi} \rho'$, then the consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi, id_{\tau(\rho)}, \rho' \rangle$ can be grasped from the drawing of ρ' in Figure 3.9. Consider for example production q_{j1} in q'': any item x it consumes in graph X must be in the same $\approx_{\rho'}$-equivalence class of an item of G (by the existence of morphism s), which is exactly the item consumed by the j_1-th production of ρ. To prove this formally one would need an explicit analysis construction;
- ($\mathsf{SH-syn}$) If $\rho \equiv^{syn}_{\Pi} \rho'$, the consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi, id_{\tau(\rho)}, \rho' \rangle$ follows as in the previous case;
- ($\mathsf{SH-sym}$) The consistency of $\langle \rho', id_{\sigma(\rho')}, \Pi^{-1}, id_{\tau(\rho')}, \rho \rangle$ follows immediately from the consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi, id_{\tau(\rho)}, \rho' \rangle$, since all mappings relating ρ and ρ' are invertible;
- ($\mathsf{SH-comp}$) By the induction hypothesis $\langle \rho_1, id_{\sigma(\rho_1)}, \Pi_1, id_{\tau(\rho_1)}, \rho_1' \rangle$ and $\langle \rho_2, id_{\sigma(\rho_2)}, \Pi_2, id_{\tau(\rho_2)}, \rho_2' \rangle$ are consistent. Therefore,since $\tau(\rho_1) = \sigma(\rho_2)$, the consistency of $\langle \rho_1 ; \rho_2, id_{\sigma(\rho_1)}, \Pi_1 \mid \Pi_2, id_{\tau(\rho_2)}, \rho_1' ; \rho_2' \rangle$ follows from Proposition 3.5.2.(1);
- ($\mathsf{SH-trans}$) The consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi' \circ \Pi, id_{\tau(\rho)}, \rho'' \rangle$ follows by Proposition 3.5.2.(2), from the consistency of $\langle \rho, id_{\sigma(\rho)}, \Pi, id_{\tau(\rho)}, \rho' \rangle$ and of $\langle \rho', id_{\sigma(\rho')}, \Pi', id_{\tau(\rho')}, \rho'' \rangle$, which hold by induction hypothesis;

($\mathsf{CTC}-\mathsf{trans}$) By induction hypothesis $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, M_{\psi'} \circ M_\psi^{-1}, \rho_{\psi'} \rangle$ and $\langle \rho_{\psi'}, m_{\psi''} \circ m_{\psi'}^{-1}, \Pi', M_{\psi''} \circ M_{\psi'}^{-1}, \rho_{\psi''} \rangle$ are consistent. Thus, by Proposition 3.5.2.(2), $\langle \rho_\psi, m_{\psi''} \circ m_\psi^{-1}, \Pi' \circ \Pi, M_{\psi''} \circ M_\psi^{-1}, \rho_{\psi''} \rangle$ is consistent as well.

Only if part of (2):
Follows from the statement just proved and from the rules ($\mathsf{TC}-\mathsf{ctc}$) and ($\mathsf{TC}-\mathsf{iso}$) defining equivalence $\equiv$, since they do not affect the consistency of the underlying four-tuple.

If part of (1):
Suppose that the five-tuple $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, M_{\psi'} \circ M_\psi^{-1}, \rho_{\psi'} \rangle$ is consistent, and that $\Pi : \underline{\#\psi} \to \underline{\#\psi'}$ is a bijection (thus ψ and ψ' have the same order). By repeated applications of the analysis construction, ψ and ψ' can be transformed into equivalent, *sequential* derivations (i.e., such that at each direct derivation only one production is applied) ψ_1 and ψ'_1, such that $\psi \equiv^{sh}_{\Pi_1} \psi_1$ and $\psi' \equiv^{sh}_{\Pi_2} \psi'_1$, for suitable permutations Π_1 and Π_2. By the *only if* part, five-tuples $\langle \rho_{\psi_1}, id_{\sigma(\psi_1)}, \Pi_1^{-1}, id_{\tau(\psi_1)}, \rho_\psi \rangle$ and $\langle \rho_{\psi'}, id_{\sigma(\psi')}, \Pi_2, id_{\tau(\psi')}, \rho_{\psi'_1} \rangle$ are consistent, thus $\langle \rho_{\psi_1}, m_{\psi'_1} \circ m_{\psi_1}^{-1}, \Pi_2 \circ \Pi \circ \Pi_1^{-1}, M_{\psi'_1} \circ M_{\psi_1}^{-1}, \rho_{\psi'_1} \rangle$ is consistent as well. Now there are two cases.

1. Suppose that $\Pi_2 \circ \Pi \circ \Pi_1^{-1}$ is the identity permutation. Then it is possible to build a family of isomorphisms between the graphs of derivations ρ_{ψ_1} and $\rho_{\psi'_1}$, starting from $m_{\psi'_1} \circ m_{\psi_1}^{-1}$, and continuing inductively by exploiting the function $\xi : \mathit{Items}(\rho_{\psi_1}) \to \mathit{Items}(\rho_{\psi'_1})$ of Definition 3.5.1. This family of isomorphisms satisfies all the conditions of Definition 3.3.4: thus it provides a proof that $\psi_1 \equiv^{abs} \psi'_1$, and therefore we have $\psi \equiv^{sh} \psi_1 \equiv^{abs} \psi'_1 \equiv^{sh} \psi'$, showing that $\psi \equiv^{c} \psi'$.

2. If $\hat{\Pi} \stackrel{def}{=} \Pi_2 \circ \Pi \circ \Pi_1^{-1}$ is not the identity permutation, let $\hat{\imath} = min\{i \in \underline{\#\psi_1} \mid \hat{\Pi}(i) \neq i\}$. Then it can be shown that the $\hat{\Pi}(\hat{\imath})$-th direct derivation in ψ'_1 is sequential independent from the preceding one, essentially because it can not consume any item produced or explicitly preserved after the $\hat{\imath}$-th step. By applying the synthesis and the analysis constructions the two direct derivations can be exchanged, and this procedure can be iterated producing eventually a derivation ψ_2 such that $\psi'_1 \equiv^{sh}_{\Pi_3} \psi_2$ for some permutation Π_3, and such that $\Pi_3 \circ \hat{\Pi}$ is the identity permutation. Thus we are reduced to the case 1.

If part of (2):
Let $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, \rho_{\psi'} \rangle$ be a consistent four-tuple, where Π is a permutation.

By exploiting isomorphisms $m_{\psi'} \circ m_{\psi}^{-1}$, Π and $\xi : Items(\rho_{\psi}) \to Items(\rho_{\psi'})$, it can be proved that the target graphs of ρ_{ψ} and $\rho_{\psi'}$ are isomorphic, and that there exists a unique isomorphism $h_{\tau} : \tau(\psi) \to \tau(\psi')$ such that $\langle \rho_{\psi}, m_{\psi'} \circ m_{\psi}^{-1}, \Pi, h_{\tau}, \rho_{\psi'} \rangle$ is a consistent five-tuple. Now, let α be the discrete decorated derivation $\langle M_{\psi'}, \tau(\psi'), h_{\tau} \circ M_{\psi} \rangle$. Then $\psi'; \alpha = \langle m_{\psi'}, \rho_{\psi'}, h_{\tau} \circ M_{\psi} \rangle$. Clearly, the five-tuple $\langle \rho_{\psi}, m_{\psi'} \circ m_{\psi}^{-1}, \Pi, h_{\tau} \circ M_{\psi} \circ M_{\psi}^{-1}, \rho_{\psi'} \rangle$ is consistent, and by *if part of (1)* we get $\psi \equiv^{c} \psi'; \alpha$. Then $\psi \equiv \psi'$ follows by rule $(\mathsf{CTC} - \mathsf{iso})$. □

3.5.2 From processes to traces and backwards

We are now ready to prove the isomorphism between the categories $\mathbf{Tr}[\mathcal{G}]$ and $\mathbf{CP}[\mathcal{G}]$ for a given grammar $\mathcal{G}$. First we introduce two functions which map each trace into a process and viceversa. Then we show that such functions extend to functors from $\mathbf{Tr}[\mathcal{G}]$ to $\mathbf{CP}[\mathcal{G}]$ and backward, which are inverse to each other.
Given an abstract c-process $[cp]$ we can obtain a derivation by "executing" the productions of cp in any order compatible with the causal order and then considering the corresponding concatenable derivation trace.

Definition 3.5.4 (linearization)
Let $cp = \langle m, p, M \rangle$ be a c-process. A *linearization* of the set P_p of productions of cp is any bijection $e : \underline{|P_p|} \to P_p$, such that the corresponding linear order, defined by $q_0 \sqsubseteq q_1$ iff $e^{-1}(q_0) \leq e^{-1}(q_1)$, is compatible with the causal order of cp, i.e., $q_0 \leq_p q_1$ implies $q_0 \sqsubseteq q_1$.

Definition 3.5.5 (from processes to traces)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a typed graph grammar and let $[cp]$ be an abstract c-process of $\mathcal{G}$, where $cp = \langle m, p, M \rangle$. Consider any linearization e of P_p and define the decorated derivation $\psi(cp, e)$ as follows:

$$\psi(cp, e) = \langle m, \rho, M \rangle, \quad \text{where } \rho = \{G_{j-1} \Rightarrow_{q_j, g_j} G_j\}_{j \in \underline{|P_p|}}$$

such that $G_0 = Min(cp)$, $G_{|P_p|} = Max(cp)$, and for each $j \in \underline{|P_p|}$

- $q_j = mp_p(e(j))$;
- $G_j = Gr(S_{\{e(1),\ldots,e(j)\}})$, i.e., the subgraph of the type graph TG_p of the process determined by the reachable set $S_{\{e(1),\ldots,e(j)\}}$, typed by mg_p;
- each derivation step $G_{j-1} \Rightarrow_{q_j, g_j} G_j$ is as in Figure 3.14.(a), where unlabelled arrows represent inclusions.

Finally, we associate to the abstract process $[cp]$ the concatenable derivation trace $\mathcal{L}_A([cp]) = [\psi(cp, e)]_c$.

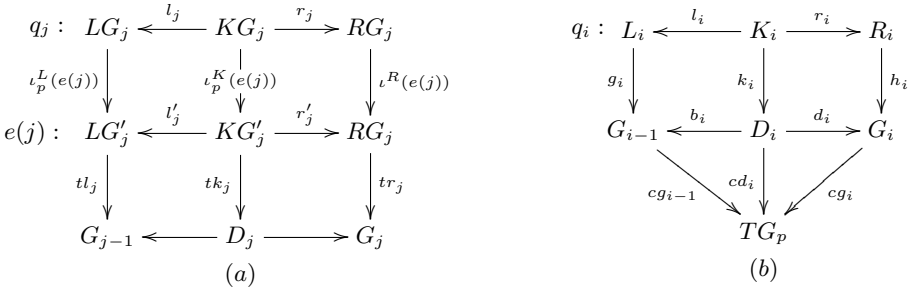


Figure 3.14: From abstract c-processes to concatenable derivation traces and backward.

By using Theorem 3.4.8, it is not difficult to prove that $\psi(cp, e)$ is a legal decorated derivation in $\mathcal{G}$. The proof can be carried out by adapting the argument in [16], Theorem 29. The only differences, here, are the fact that the source graph of the derivation is not, in general, the start graph of the grammar and the presence of the decorations.
The following lemma shows that the mapping $\mathcal{L}_A$ can be lifted to a well-defined functor $\mathcal{L}_A : \mathbf{CP}[\mathcal{G}] \to \mathbf{Tr}[\mathcal{G}]$ from the category of abstract c-processes to the category of derivation traces, which acts as identity on objects.

Lemma 3.5.6 (functoriality of $\mathcal{L}_A$)
Let $\mathcal{G}$ be a typed graph grammar, and let cp_1 and cp_2 be c-processes of $\mathcal{G}$. Then

1. ($\mathcal{L}_A$ is a well-defined mapping from abstract c-processes to traces)
   if $cp_1 \cong cp_2$ then $\psi(cp_1, e_1) \equiv^c \psi(cp_2, e_2)$, for any choice of linearizations e_1 and e_2;
2. ($\mathcal{L}_A$ preserves sequential composition)
   if defined, $\psi(cp_1 ; cp_2, e) \equiv^c \psi(cp_1, e_1) ; \psi(cp_2, e_2)$, for any choice of linearizations e_1, e_2 and e;
3. ($\mathcal{L}_A$ maps discrete processes to discrete derivation traces, and, in particular, it preserves identities)
   $\psi(Sym_{\mathcal{G}}(m, G, M), e) \equiv^c \langle m, G, M\rangle$, for any choice of linearization e.

Proof
1. Let $cp_1 \cong cp_2$ be two isomorphic c-processes of $\mathcal{G}$, with $cp_i = \langle m_i, p_i, M_i\rangle$

for $i \in \{1,2\}$. Let $f = \langle fg, fp\rangle : cp_1 \to cp_2$ be an isomorphism, and let e_1 and e_2 be linearizations of P_{p_1} and P_{p_2}, respectively. We show that the two decorated derivations $\psi_1 = \psi(cp_1, e_1)$ and $\psi_2 = \psi(cp_2, e_2)$ are ctc-equivalent by exhibiting a consistent five-tuple $\langle \rho_{\psi_1}, m_{\psi_2} \circ m_{\psi_1}^{-1}, \Pi, M_{\psi_2} \circ M_{\psi_1}^{-1}, \rho_{\psi_2}\rangle$.
First, it is quite easy to recognize that $Items(\psi_i)$ is (isomorphic to) $Items(TG_{p_i})$, for $i \in \{1,2\}$ (see Definition 3.5.1) and thus $fg : TG_{p_1} \to TG_{p_2}$ induces, in an obvious way, an isomorphism $\xi : Items(\psi_1) \to Items(\psi_2)$.
By Corollary 3.4.12, the restrictions of fg to the source and target graphs of the processes are isomorphisms, therefore we can take

$$h_\sigma = fg_{|Min(cp_1)} : Min(cp_1) \to Min(cp_2) \text{ and}$$
$$h_\tau = fg_{|Max(cp_1)} : Max(cp_1) \to Max(cp_2),$$

which are compatible with $m_{\psi_2} \circ m_{\psi_1}^{-1}$ and $M_{\psi_2} \circ M_{\psi_1}^{-1}$, by definition of isomorphism of c-processes (and obviously compatible with ξ). Finally we can define the permutation $\Pi : \underline{\#\psi_1} \to \underline{\#\psi_2}$ as $\Pi = e_2^{-1} \circ fp \circ e_1$.
Now, it is not difficult to check that $\langle \rho_{\psi_1}, m_{\psi_2} \circ m_{\psi_1}^{-1}, \Pi, M_{\psi_2} \circ M_{\psi_1}^{-1}, \rho_{\psi_2}\rangle$ is a consistent five-tuple. Thus, by Theorem 3.5.3.(1), we conclude that $\psi_1 \equiv^c \psi_2$.

2. Let $cp_1 = \langle m_1, p_1, M_1\rangle$ and $cp_2 = \langle m_2, p_2, M_2\rangle$ be two c-processes such that their sequential composition is defined, i.e., $Max(cp_1) = Min(cp_2)$, $M_1 = m_2$ and all other items in cp_1 and cp_2 are distinct. Let $cp = \langle m_1, p, M_2\rangle$ be their sequential composition. Let us fix linearizations e_1, e_2 and e of P_{p_1}, P_{p_2} and P_p, respectively, and let $\psi_1 = \psi(cp_1, e_1)$, $\psi_2 = \psi(cp_2, e_2)$ and $\psi = \psi(cp, e)$.
Observe that $\sigma(\psi) = \sigma(\psi_1) = \sigma(\psi_1 ; \psi_2)$ and similarly $\tau(\psi) = \tau(\psi_2) = \tau(\psi_1 ; \psi_2)$. Furthermore, we have that $Items(\psi)$ and $Items(\psi_1;\psi_2)$ are (isomorphic to) $Items(TG_p) = Items(TG_{p_1}) \cup Items(TG_{p_2})$. Therefore one can verify that the five-tuple $\langle \rho_\psi, id_{\sigma(\psi)}, \Pi, id_{\tau(\psi)}, \rho_{\psi_1 ; \psi_2}\rangle$, where $\Pi = (e_1 \cup e_2)^{-1} \circ e$ is consistent (the function ξ being identity on classes). This fact, together with the observation that $m_\psi = m_{\psi_1} = m_{\psi_1;\psi_2}$ and $M_\psi = M_{\psi_2} = M_{\psi_1;\psi_2}$, allows us to conclude, by Theorem 3.5.3.(1), that $\psi \equiv^c \psi_1;\psi_2$.

3. Obvious. □

The backward step, from concatenable derivation traces to processes, is performed via a colimit construction that, applied to a decorated derivation ψ, essentially constructs the type graph as a copy of the source graph plus the items created during the rewriting process. Productions are instances of production applications.

Definition 3.5.7 (from derivations to processes)
Let $\mathcal{G} = \langle TG, G_s, P, \pi\rangle$ be a typed graph grammar and let $\psi = \langle m, \rho, M\rangle$ be a decorated derivation, with $\#\psi = n$. We associate to ψ a c-process $cp(\psi) = \langle m', p, M'\rangle$, defined as follows:

- $\langle TG_p, mg_p \rangle$ is a colimit object (in category **TG-Graph**) of the diagram representing derivation ψ, as depicted (for a single derivation step and without typing morphisms) in Figure 3.14.(b);

- $P_p = \{\langle prod(\psi)(j), j\rangle \mid j \in \underline{n}\}$. For all $j \in \underline{n}$, if $prod(\psi)(j) = q_{is}$ (notation of Definition 3.2.9, Figure 3.6), then $\pi_p(\langle prod(\psi)(j), j\rangle)$ is essentially production q_{is}, but typed over TG_p (see Figure 3.14.(b)). Formally $\pi_p(\langle prod(\psi)(j), j\rangle)$ is the production

  $$\langle LG_{q_{is}}, cg_{i-1} \circ g_i \circ in^L_{is} \rangle \overset{l_{q_{is}}}{\leftarrow} \langle KG_{q_{is}}, cd_i \circ k_i \circ in^K_{q_{is}} \rangle \overset{r_{q_{is}}}{\rightarrow} \langle RG_{q_{is}}, cg_i \circ h_i \circ in^R_{q_{is}} \rangle$$

  and $mp_p(\langle prod(\psi)(j), j\rangle) = prod(\psi)(j)$. Finally $\iota_p(\langle prod(\psi)(j), j\rangle) = \langle id_{LG_{q_{is}}}, id_{KG_{q_{is}}}, id_{RG_{q_{is}}} \rangle$.

  Note that for $j_1 \neq j_2$ we may have $prod(\psi)(j_1) = prod(\psi)(j_2)$; instead, the productions in P_p are all distinct, as they can be considered as production occurrences of $\mathcal{G}$;

- notice that $Min(cp) = cg_0(G_0)$ and $Max(cp) = cg_n(G_n)$ (and the cg_i's are injective, because so are the horizontal arrows) so that we can define $m' = cg_0 \circ m$ and $M' = cg_n \circ M$;

Finally we define the image of the trace $[\psi]_c$ as $\mathcal{P}_A([\psi]_c) = [cp(\psi)]$.

As an example, the process of Figure 3.12 can be obtained (up to isomorphism) by applying the construction just described to either the derivation of Figure 3.5 or that of Figure 3.10.
Notice that it is quite easy to have a concrete characterization of the colimit graph TG_p. Since all the squares in the diagram representing derivation $\psi : G_0 \Rightarrow^* G_n$ commute (they are pushouts), TG_p can be regarded equivalently as the colimit of the bottom line of the derivation, $G_0 \overset{b_1}{\leftarrow} D_1 \overset{d_1}{\rightarrow} G_1 \overset{b_2}{\leftarrow} \cdots D_n \overset{d_n}{\rightarrow} G_n$. Thus TG_p can be characterized explicitly as the graph having as items $Items(\rho_\psi)$ (see Definition 3.5.1), and where source and target functions are determined in the obvious way. The injections cx_i ($x \in \{g, d\}$) simply map every item into its equivalence class.

Lemma 3.5.8 ($\mathcal{P}_A$ is well-defined)
Let $\mathcal{G} = \langle TG, G_s, P, \pi\rangle$ be a typed graph grammar and let $\psi_1 \equiv^c \psi_2$ be two decorated derivations of $\mathcal{G}$. Then $cp(\psi_1) \cong cp(\psi_2)$.

Proof
Let ψ_1 and ψ_2 be ctc-equivalent decorated derivations of $\mathcal{G}$. Then, by Theorem 3.5.3 there exists a five-tuple $\langle \rho_{\psi_1}, m_{\psi_2} \circ m_{\psi_1}^{-1}, \Pi, M_{\psi_2} \circ M_{\psi_1}^{-1}, \rho_{\psi_2} \rangle$ with a function $\xi : \mathit{Items}(\psi_1) \to \mathit{Items}(\psi_2)$ witnessing its consistence.
Let $cp(\psi_1) = cp_1 = \langle m_1, p_1, M_1 \rangle$ and $cp(\psi_2) = cp_2 = \langle m_2, p_2, M_2 \rangle$. First notice that each $\mathit{Items}(\psi_i)$ is isomorphic to the set of items of the corresponding type graph TG_{p_i} ($i \in \{1, 2\}$), and thus ξ induces readily a function $fg : TG_{p_1} \to TG_{p_2}$, which, by definition of consistent five-tuple, is consistent with the decorations of the processes. Moreover the permutation Π induces a bijection $fp : P_{p_1} \to P_{p_2}$, defined as $fp(\langle \mathit{prod}(\psi_1)(j), j \rangle) = \langle \mathit{prod}(\psi_2)(\Pi(j)), \Pi(j) \rangle$, for $j \in \underline{|P_{p_1}|}$.
From the definition of consistent five-tuple it follows easily that $\langle fg, fp \rangle : cp_1 \to cp_2$ is an isomorphism of c-processes. □
The next lemma shows that the constructions performed by $\mathcal{P}_A$ and $\mathcal{L}_A$ are, at abstract level, "inverse" each other.

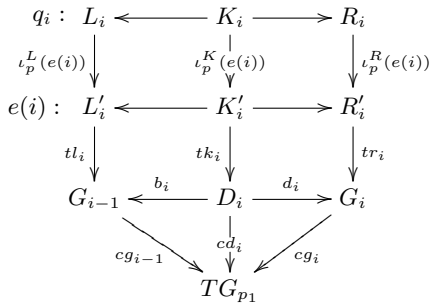
Lemma 3.5.9 (relating $\mathcal{P}_A$ and $\mathcal{L}_A$)
Let $\mathcal{G}$ be a graph grammar. Then:

1. *$\mathcal{P}_A(\mathcal{L}_A([cp])) = [cp]$, for any c-process cp and*
2. *$\mathcal{L}_A(\mathcal{P}_A([\psi]_c)) = [\psi]_c$, for any decorated derivation ψ.*

Proof
1. Let $cp = \langle m, p, M \rangle$ be a c-process of $\mathcal{G}$ and let $e : \underline{|P_p|} \to P_p$ be a linearization of P_p. Consider the decorated derivation:

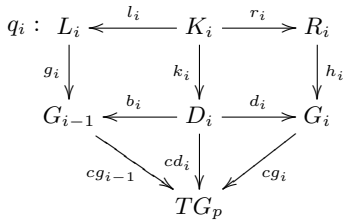
$$\psi(cp, e) = \langle m, \{G_{i-1} \Rightarrow_{e(i)} G_i\}_{i \in \underline{|P_p|}}, M \rangle$$

as in Definition 3.5.5.
Let us now construct the process $cp_1 = cp(\psi) = \langle m_1, p_1, M_1 \rangle$ as in Definition 3.5.7, with the type graph TG_{p_1} obtained as colimit of the diagram:

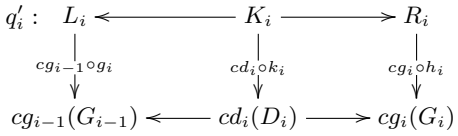


Observe that a possible concrete choice for TG_{p_1} is TG_p, with all cg_i's defined as inclusions.
If we define $fp : P_{p_1} \to P_p$ as $fp(\langle prod(\psi)(i), i\rangle) = e(i)$, for $i \in \underline{\#\psi}$, then it is easy to prove that $f = \langle id_{TG_p}, fp\rangle : cp_1 \to cp$ is a c-process isomorphism.

2. First of all, if ψ is a discrete decorated derivation, the result is obvious. Otherwise we can suppose $\psi = \langle m, \{G_{i-1} \Rightarrow_{q_i} G_i\}_{i\in\underline{n}}, M\rangle$ to be a derivation using a single production at each step (recall that, by Lemma 3.5.8, we can apply the concrete construction to any derivation in the trace and that in $[\psi]_c$ a derivation of this shape can always be found).
Let $cp = cp(\psi) = \langle m, p, M\rangle$ the c-process built as in Definition 3.5.7. In particular, the type graph TG_p is defined as the colimit of the diagram:



The set of productions is $P_p = \{q'_i = \langle q_i, i\rangle \mid i \in \underline{n}\}$, with $\pi_p(q'_i) = \langle LG_i, cg_{i-1} \circ g_i\rangle \overset{l_i}{\leftarrow} \langle KG_i, cd_i \circ k_i\rangle \overset{r_i}{\rightarrow} \langle RG_i, cg_i \circ r_i\rangle$. Moreover $m = cg_0 \circ m_\psi$ and $M = cg_n \circ M_\psi$. Remember that all cg_i's are injections and $cg_0 : G_0 \to Min(cp)$, $cg_n : G_n \to Max(cp)$ are isomorphisms.
Now it is not difficult to verify that $prod(\psi)$ is a linearization of P_p and $\psi' = \psi(cp, prod(\psi))$ is the derivation whose i^{th} step is depicted below.



The family of isomorphism $\{\theta_{X_i} : X_i \to X'_i \mid X \in \{G, D\},\ i \in \underline{n}\} \cup \{\theta_{G_0}\}$ between corresponding graphs in the two (linear) derivations defined as:

$$\theta_{D_i} = cd_i \qquad \text{and} \qquad \theta_{G_i} = cg_i$$

satisfies the conditions of Definition 3.3.4 and thus we have that $\psi' \equiv^{abs} \psi$. Therefore $[\psi]_c = [\psi']_c = \mathcal{L}_A([cp]) = \mathcal{P}_A(\mathcal{L}_A([\psi]_c)$. □
The previous lemma, together with functoriality of $\mathcal{L}_A$, allows us to conclude that $\mathcal{P}_A : \mathbf{Tr}[\mathcal{G}] \to \mathbf{CP}[\mathcal{G}]$, defined as identity on objects and as $\mathcal{P}_A([\psi]_c) =$

$[cp(\psi)]$ on morphisms, is a well-defined functor. In fact, given two decorated derivations ψ_1 and ψ_2, we have

$$\begin{aligned}
&\mathcal{P}_A([\psi_1]_c\,;\,[\psi_2]_c) = && \text{by Lemma 3.5.9.(2)}\\
&\quad = \mathcal{P}_A(\mathcal{L}_A(\mathcal{P}_A([\psi_1]_c))\,;\,\mathcal{L}_A(\mathcal{P}_A([\psi_2]_c))) = && \text{by functoriality of } \mathcal{L}_A\\
&\quad = \mathcal{P}_A(\mathcal{L}_A(\mathcal{P}_A([\psi_1]_c)\,;\,\mathcal{P}_A([\psi_2]_c))) = && \text{by Lemma 3.5.9.(1)}\\
&\quad = \mathcal{P}_A([\psi_1]_c)\,;\,\mathcal{P}_A([\psi_2]_c)
\end{aligned}$$

Similarly one proves also that $\mathcal{P}_A$ preserves identities. Moreover, again by Lemma 3.5.9, functors $\mathcal{L}_A$ and $\mathcal{P}_A$ are inverse to each other, thus implying the main result of this section.

Theorem 3.5.10
Let $\mathcal{G}$ be a graph grammar. Then $\mathcal{L}_A : \mathbf{CP}[\mathcal{G}] \to \mathbf{Tr}[\mathcal{G}]$ and $\mathcal{P}_A : \mathbf{Tr}[\mathcal{G}] \to \mathbf{CP}[\mathcal{G}]$ are inverse to each other, establishing an isomorphism of categories. □

3.6 Event Structure Semantics

The aim of this section is to define a semantics for graph grammars based on Winskel's event structures [24], a simple event based semantic model where events are considered as atomic and instantaneous steps, which can appear only once in a computation. An event can occur only after some other events (its causes) have taken place and the execution of an event can inhibit the execution of other events. We concentrate here on *prime event structures* (with binary conflict), shortly *PES*'s, where the above situation is formalized via two binary relations: *causality*, modelled by a partial order relation, and *conflict*, modelled by a symmetric and irreflexive relation, hereditary with respect to causality. It is worth noticing that such semantics will differ from the previously described trace and process semantics not only because it represents in a unique structure both the concurrent and non-deterministic aspects of grammar computations, but also because it abstracts out completely from the nature of the states of the system.
Starting from the assumption that (concatenable) derivation traces are an adequate abstract representation of deterministic truly concurrent computations of a grammar, we construct the category of objects of $\mathbf{Tr}[\mathcal{G}]$ under the start graph G_s, namely $(G_s \downarrow \mathbf{Tr}[\mathcal{G}])$. Such category provides a synthetic representation of all the possible computations of the grammar beginning from the start graph. For consuming grammars, the obvious partial order $\mathbf{Dom}[\mathcal{G}]$ induced by the preorder associated to category $(G_s \downarrow \mathbf{Tr}[\mathcal{G}])$ is shown to have the desired algebraic structure, namely to be a prime algebraic, finitely coherent

and finitary partial order, or briefly a *domain*. Thus, by well known results of [24], $\mathbf{Dom}[\mathcal{G}]$ indirectly determines a PES. To prove the algebraic properties of the domain $\mathbf{Dom}[\mathcal{G}]$, we present first an explicit construction of a prime event structure $\mathbf{ES}[\mathcal{G}]$ associated to a graph grammar. Roughly speaking, in the event structure $\mathbf{ES}[\mathcal{G}]$ an event is determined by a specific direct derivation α belonging to a derivation ψ that begins from the start graph of $\mathcal{G}$, together with all its causes (i.e., all the preceding direct derivations of ψ that caused α, either directly or indirectly). Then we show that, in the consuming case, the finite configurations of $\mathbf{ES}[\mathcal{G}]$ are one-to-one with the elements of the domain $\mathbf{Dom}[\mathcal{G}]$. Thus these two structures are conceptually equivalent, in the sense that one can be recovered from the other. This presentation has been preferred to an explicit proof of the algebraic properties of the domain, since we think it has the advantage of giving a more explicit understanding of the event structure, and of simplifying slightly the proofs.

Nicely, the events and the configurations of the event structure $\mathbf{ES}[\mathcal{G}]$ associated to a consuming grammar $\mathcal{G}$ can futhermore be characterized in terms of processes. The corresponding result strongly relies on the the close relationship between traces and processes proved in the previous section.

It is worth remarking that the PES and domain semantics are shown to give a meaningful representation of the behaviour of the original grammar only in the consuming case. Concretely, only part of the results are carried out for general grammars and, in particular, what fails for consuming grammars is the correspondence between the domain and the event structure associated to a grammar. This limitation is not surprising. Similarly to what happens for Petri nets, given a production with empty pre-set, an unbounded number of indistinguishable copies of such production can be applied in parallel in a derivation. This phenomenon, called *autoconcurrency*, cannot be modelled within Winskel's event structures, since for a fixed set of events they can describe only computations where each event can fire just once.

Another limitation of traditional event structures is their inability faithfully modelling *asymmetric conflicts*, a kind of dependency among events arising in formalisms, like graph grammars, where part of the state can be preserved by a step of a computation. The adequateness of Winskel's event structures for representing the behaviour of grammars and the possibility of adopting alternative formalisms will be discussed at the end of the section.

3.6.1 Prime event structures and domains

Let us start by introducing prime event structures and prime algebraic domains, and recalling the relationships existing between such structures.

Definition 3.6.1 (prime event structures)
A *prime event structure* (*PES*) is a tuple $\mathcal{E} = \langle E, \leq, \# \rangle$, where E is a set of *events* and $\leq$, $\#$ are binary relations on E called *causal dependency relation* and *conflict relation*, respectively, such that:

1. the relation $\leq$ is a partial order, satisfying the *axiom of finite causes*, i.e., $\lfloor e \rfloor = \{e' \in E \mid e' \leq e\}$ is finite for all $e \in E$;

2. the relation $\#$ is irreflexive, symmetric and hereditary with respect to $\leq$, i.e., $e \# e'$ and $e' \leq e''$ implies $e \# e''$ for all $e, e', e'' \in E$;

A configuration of a PES is a set of events representing a possible computation of the system modelled by the event structure.

Definition 3.6.2 (configuration)
A *configuration* C of a prime event structure $\mathcal{E}$ is a subset of events $C \subseteq E$ such that

1. C is *conflict-free*: for all $e, e' \in C$, $\neg(e \# e')$.

2. C is *left-closed*: for all $e, e' \in E$, $e' \leq e$ and $e \in C$ implies $e' \in C$.

Given two configurations C_1 and C_2 such that $C_1 \subseteq C_2$, configuration C_1 can be extended to C_2 by performing the events in $C_2 - C_1$ in any order compatible with the causal order. Therefore set-theoretical inclusion on configurations can be interpreted safely as a computational ordering.

Definition 3.6.3 (domain of configurations)
The *domain of configurations* of a prime event structure $\mathcal{E}$, denoted $\mathcal{L}(\mathcal{E})$, is the partial order $\mathcal{L}(\mathcal{E}) = \langle \mathcal{C}, \subseteq \rangle$, where $\mathcal{C}$ is the set of all configurations of $\mathcal{E}$, and $\subseteq$ is the subset inclusion relation. The *domain of finite configurations* of $\mathcal{E}$, denoted $\mathcal{FL}(\mathcal{E})$, is the partial order of all *finite* configurations of $\mathcal{E}$ ordered by inclusion.

As anticipated in the introduction, the domain of finite configurations $\mathcal{FL}(\mathcal{E})$ of a prime event structure $\mathcal{E}$ has a very nice algebraic structure, namely it is a *prime algebraic, finitely coherent, finitary partial order*,[i] briefly referred to

[i] Recall that a partial order $\langle D, \sqsubseteq \rangle$ is *finitely coherent* if each pairwise compatible finite subset X (namely each set where every pair of elements have a common upper bound) has a least upper bound. A *complete prime* of D is an element $p \in D$ such that if $p \sqsubseteq \bigsqcup X$, with X compatible subset of D, then $p \sqsubseteq x$ for some $x \in X$. Then D is called *prime algebraic* if each element in $x \in D$ is the least upper bound of the complete primes below x itself. Finally D is finitary if for each $x \in D$ the set of elements below x $\{y \in D \mid y \sqsubseteq x\}$ is finite.

in the following as *domain* [39,24].[j] Conversely, each domain $\langle D, \sqsubseteq \rangle$ can be viewed as the domain of finite configurations of a uniquely determined (up to isomorphism) prime event structure $\mathcal{E}(D) = \langle Pr(D), \leq, \# \rangle$ where

1. $Pr(D)$ is the set of complete primes of D,
2. $p \leq p'$ iff $p \sqsubseteq p'$, and
3. $p \# p'$ iff p and p'' are incompatible, namely $\neg \exists x \in D.\ (p \sqsubseteq x \ \wedge \ p' \sqsubseteq x)$.

As shown by Winskel [24], the relation between PES's and domains lifts, at categorical level, to an equivalence of categories.

3.6.2 Event structure semantics for a grammar

Let $\mathcal{G}$ be a graph grammar and let $\mathbf{Tr}[\mathcal{G}]$ be its category of concatenable derivation traces (see Definition 3.3.12). The construction of the domain associated to a grammar strongly relies on the category of objects under $[G_s]$ in $\mathbf{Tr}[\mathcal{G}]$ (see Definition 3.2.2), namely $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$.
We first observe that, for a consuming grammars such a category is a preorder, i.e., for any pair of objects there is at most one arrow from the first one to the second one. The result is an easy consequence of the following technical lemma.

Lemma 3.6.4
Let $\mathcal{G}$ be a consuming graph grammar and let $\psi_1, \psi_2, \psi_1', \psi_2'$ be decorated derivations. If $\psi_1 ; \psi_2 \equiv^c_{\Pi} \psi_1' ; \psi_2'$ and $\psi_1 \equiv^c_{\Pi_1} \psi_1'$ then $\Pi_{|\underline{\#\psi_1}} = \Pi_1$ and $\psi_2 \equiv^c \psi_2'$.

Proof
We first observe that if $\langle \rho_1 ; \rho_2, h_\sigma, f, h_\tau, \rho_1' ; \rho_2' \rangle$ and $\langle \rho_1, h_\sigma, f_1, h_\tau', \rho_1' \rangle$ are consistent five-tuples then surely $f_1 = f_{|\underline{\#\rho_1}}$. Otherwise, let $\hat{\imath} = \min\{i \in \underline{\#\rho_1} \mid f(i) \neq f_1(i)\}$. Since the two five-tuples have the same h_σ and permutations f and f_1 coincide for $i < \hat{\imath}$, it is easy to see that the productions of $\rho_1' ; \rho_2'$ corresponding to the indexes $f(\hat{\imath})$ and $f_1(\hat{\imath})$ should consume the same items. But this is impossible since $f(\hat{\imath}) \neq f_1(\hat{\imath})$ and productions really consume something, being $\mathcal{G}$ consuming. Therefore the set $\{i \in \underline{\#\rho_1} \mid f(i) \neq f_1(i)\}$ must be empty, namely $f_1 = f_{|\underline{\#\rho_1}}$. Notice moreover that, in the situation

[j] Usually in the literature a domain is a complete partial order satisfying further suitable algebraic properties. Here we call "domain" the set of finite elements of such a structure, which uniquely determines the whole domain through an ideal completion. For example, $\mathcal{L}(\mathcal{E})$ is the ideal completion of $\mathcal{FL}(\mathcal{E})$.

above, $\langle \rho_2, h'_\tau, f_2, h_\tau, \rho'_2 \rangle$ is a consistent five-tuple as well, where f_2 is defined as $f_2(j) = f(j + \#\rho_1) - \#\rho_1$ for $j \in \underline{\#\rho_2}$.
Now, let $\psi_1 ; \psi_2 \equiv^c \psi'_1 ; \psi'_2$ and $\psi_1 \equiv^c \psi'_1$. By Theorem 3.5.3 we can find two consistent five-tuples $\langle \rho_{\psi_1 ; \psi_2}, m_{\psi'_1} \circ {m_{\psi_1}}^{-1}, \Pi, M_{\psi'_2} \circ M_{\psi_2}^{-1}, \rho_{\psi'_1 ; \psi'_2} \rangle$ and $\langle \rho_{\psi_1}, m_{\psi'_1} \circ {m_{\psi_1}}^{-1}, \Pi_1, M_{\psi'_1} \circ M_{\psi_1}^{-1}, \rho_{\psi'_1} \rangle$. Therefore, as an instance of the observation above $\Pi_{|\underline{\#\psi_1}} = \Pi_1$. Moreover there is a consistent five-tuple

$$\langle \rho_{\psi_2}, M_{\psi'_1} \circ M_{\psi_1}^{-1}, \Pi_2, M_{\psi'_2} \circ M_{\psi_2}^{-1}, \rho_{\psi'_2} \rangle. \tag{3.1}$$

Since sequential composition is defined it must be $M_{\psi_1} = m_{\psi_2}$ and $M_{\psi'_1} = m_{\psi'_2}$, and therefore the existence of the five-tuple (3.1), by Theorem 3.5.3, implies $\psi_2 \equiv^c \psi'_2$. □
In the following, we will use $\delta, \delta', \delta_1, \ldots$ to range over concatenable derivation traces, and $\eta, \eta', \eta_1, \ldots$ to range over derivation traces.

Lemma 3.6.5 (structure of $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$)
Let $\mathcal{G}$ be a consuming graph grammar. Then category $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$ is a pre-order.

Proof
By Definition 3.2.2, the objects of $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$ have the form $\langle \delta, [H] \rangle$, with $\delta : [G_s] \to [H]$ concatenable derivation trace, while an arrow $\delta_1 : \langle \delta, [H] \rangle \to \langle \delta', [H'] \rangle$ is a concatenable derivation trace such that $\delta ; \delta_1 = \delta'$. Now suppose that there are two such arrows with the same source and target, δ_1 and δ_2. Then, since $\delta ; \delta_1 = \delta' = \delta ; \delta_2$, by Definition 3.3.12 we have that if $\psi \in \delta$ is a derivation, then $\psi ; \psi_1 \equiv^c \psi ; \psi_2$ for suitable $\psi_1 \in \delta_1$ and $\psi_2 \in \delta_2$. Since obviously $\psi \equiv^c \psi$, by Lemma 3.6.4, $\psi_1 \equiv^c \psi_2$. Therefore $\delta_1 = \delta_2$. □
Although, in general, for a possibly non-consuming grammar $\mathcal{G}$, $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$ is not a preorder, we can associate to such category a preorder, called the *pre-domain* of the grammar, in a canonical way. Then the *domain* of the grammar is defined as the partial order induced by the pre-domain.

Definition 3.6.6 (pre-domain and domain of a graph grammar)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a graph grammar. The *pre-domain* of $\mathcal{G}$, denoted $\mathbf{PreDom}[\mathcal{G}]$, is the preorder having objects of $([G_s] \downarrow \mathbf{Tr}[\mathcal{G}])$ as elements and where $x \sqsubseteq y$ iff there exists an arrow $f : x \to y$. The *domain* of $\mathcal{G}$, denoted $\mathbf{Dom}[\mathcal{G}]$, is defined as the partial order induced by $\mathbf{PreDom}[\mathcal{G}]$.

Recall that if $\sim$ is the equivalence relation on objects of $\mathbf{PreDom}[\mathcal{G}]$ defined as $x \sim y$ if $x \sqsubseteq y \sqsubseteq x$, namely if there are two arrows $f : x \to y$ and $g : y \to x$, then $\mathbf{Dom}[\mathcal{G}]$ is the set of $\sim$-equivalence classes of objects of $\mathbf{PreDom}[\mathcal{G}]$,

with the ordering given by $[x]_\sim \leq [y]_\sim$ iff there is an arrow $f : x \to y$ in **PreDom**$[\mathcal{G}]$.
The next lemma characterizes the structure of the domain **Dom**$[\mathcal{G}]$. The elements of the domain can be identified with the derivation traces of $\mathcal{G}$ with source in the start graph. The order is a kind of prefix ordering on such traces.

Lemma 3.6.7 (structure of Dom$[\mathcal{G}]$)
Let $\mathcal{G}$ be a graph grammar. Then the elements of the partial order **Dom**$[\mathcal{G}]$ *are one-to-one with derivation traces starting from $[G_s]$, and the partial ordering is given by $\eta \leq \eta'$ iff there are concatenable derivation traces $\delta \subseteq \eta$ and δ' such that $\delta\,;\,\delta' \subseteq \eta'$.*

Proof
Let $\langle \delta, [H] \rangle$, $\langle \delta', [H'] \rangle$ be two elements of **PreDom**$[\mathcal{G}]$ and suppose $\langle \delta, [H] \rangle \sim \langle \delta', [H'] \rangle$, i.e., there exist δ_1 and δ_2 such that $\delta\,;\,\delta_1 = \delta'$ and $\delta'\,;\,\delta_2 = \delta$. Thus $\delta\,;\,\delta_1\,;\,\delta_2 = \delta$, which is possible only if the order of both δ_1 and δ_2 is 0, i.e., they contain derivations using only empty or discrete derivations. By Definition 3.3.11 it follows immediately that the elements of **Dom**$[\mathcal{G}]$ are exactly the derivation traces. The characterization of the ordering in **Dom**$[\mathcal{G}]$ follows from the definitions. □
We give now an explicit definition of the event structure of a graph grammar. For consuming grammars such event structure will be shown to be closely related with the domain just introduced. More precisely we will show that **Dom**$[\mathcal{G}]$ is isomorphic to the domain of finite configuration of the event structure.

Definition 3.6.8 ((prime) event structure of a graph grammar)
Let $\mathcal{G} = \langle TG, G_s, P, \pi \rangle$ be a graph grammar. A *pre-event* for $\mathcal{G}$ is a pair $\langle \psi, \alpha \rangle$, where ψ is a decorated derivation starting from a graph isomorphic to G_s, and α is a direct derivation using a single production in P (namely $\#\alpha = 1$) such that

1. $\psi\,;\,\alpha$ is defined (i.e., $\tau(\psi) = \sigma(\alpha)$, and $M_\psi = m_\alpha$) and
2. $\psi\,;\,\alpha \equiv^c_\Pi \psi'$ implies $\Pi(\#\psi + 1) = \#\psi + 1$.

If $\langle \psi, \alpha \rangle$ is a pre-event, let ε^ψ_α be the corresponding derivation trace, namely $\varepsilon^\psi_\alpha = [\psi\,;\,\alpha]$.
An *event* ε for $\mathcal{G}$ is then defined as a derivation trace $\varepsilon = \varepsilon^\psi_\alpha$ for some pre-event $\langle \psi, \alpha \rangle$. For each event ε let $Der(\varepsilon)$ denote the set of derivations containing such event, formally defined as follows:

$$Der(\varepsilon) = \bigcup \{\delta \mid \exists \delta_\varepsilon \subseteq \varepsilon.\ \exists \delta'.\ \delta_\varepsilon ; \delta' = \delta\}.$$

Notice that in particular, $\varepsilon \subseteq Der(\varepsilon)$, since for all $\delta_\varepsilon \subseteq \varepsilon$ we can concatenate it with the (concatenable trace corresponding to) a discrete derivation. Then the *(prime) event structure of grammar* $\mathcal{G}$, denoted $\mathbf{ES}[\mathcal{G}]$, is the triple $\mathbf{ES}[\mathcal{G}] = \langle E, \leq, \# \rangle$, where E is the set of all events for $\mathcal{G}$, $\varepsilon \leq \varepsilon'$ if $Der(\varepsilon') \subseteq Der(\varepsilon)$, and $\varepsilon \# \varepsilon'$ if $Der(\varepsilon) \cap Der(\varepsilon') = \emptyset$.

It is worth explaining informally the last definition. Conceptually, an event $\varepsilon^{\psi}_{\alpha}$ of a grammar $\mathcal{G}$ is determined by the application of a production to a graph reachable from the start graph of $\mathcal{G}$ (i.e., by the direct derivation α), together with the history that generated the graph items needed by that production application (i.e., the derivation ψ). The fact that in the pre-event $\psi\,;\,\alpha$ the last step α cannot be shifted backward (requirement (2) for pre-events) guarantees that α (causally) depends from all the previous steps in ψ. It is worth stressing that the same requirement implies that if $\psi\,;\,\alpha \equiv^{c}_{\Pi} \psi'$ then $\psi' = \psi''\,;\,\alpha'$ with $\psi \equiv^{c}_{\Pi_{\lfloor \underline{\#\psi}}} \psi''$ and $\alpha \equiv^{abs} \alpha'$. Clearly, isomorphic production applications or different linearizations of the same history should determine the same event. Therefore an event is defined as a set of derivations of $\mathcal{G}$, namely the set of all derivations having (a copy of) α as the last step and containing (a copy of) all its causes in any order consistent with the causality relation.
Given this definition of event, the causality and conflict relations are easily defined. In fact, considering for each event ε the set $Der(\varepsilon)$ of all the derivations that performed ε at some point, we have that two events are in conflict if there is no derivation that can perform both of them, and they are causally related if each derivation that performs one also performs the other.

Theorem 3.6.9 ($\mathbf{ES}[\mathcal{G}]$ is well-defined)
$\mathbf{ES}[\mathcal{G}]$ *is a well-defined prime event structure.*

Proof
By Definition 3.6.1, we have to show that $\leq$ is a partial order satisfying the axiom of finite causes, and that $\#$ is symmetric, irreflexive and hereditary.
Let us start by noticing that given two events ε_1 and ε_2, if $Der(\varepsilon_2) \subseteq Der(\varepsilon_1)$, then ε_1 is a "prefix" of ε_2, namely

$$\forall \delta_2 \subseteq \varepsilon_2.\ \exists \delta_1 \subseteq \varepsilon_1.\ \delta_2 = \delta_1\,;\,\delta, \quad \text{for some } \delta.$$

In fact, let $Der(\varepsilon_2) \subseteq Der(\varepsilon_1)$. Given a concatenable trace $\delta_2 \subseteq \varepsilon_2$, recalling that $\varepsilon_2 \subseteq Der(\varepsilon_2)$, we have $\delta_2 \subseteq Der(\varepsilon_1)$ and thus by Definition 3.6.8, there exists $\delta_1 \subseteq \varepsilon_1$ and δ such that $\delta_1\,;\,\delta = \delta_2$.
In particular, if $Der(\varepsilon_1) = Der(\varepsilon_2)$ we deduce that $\varepsilon_1 = \varepsilon_2$. In fact, for any $\delta_2 \subseteq \varepsilon_2$, we have

$$\delta_2 = \delta_1\,;\,\delta, \tag{3.2}$$

for some $\delta_1 \subseteq \varepsilon_1$. Symmetrically $\delta_1 = \delta_2' \,;\, \delta'$, for some $\delta_2' \subseteq \varepsilon_2$. By definition of event, $\delta_2' \equiv \delta_2 \,;\, \delta_\emptyset$, with $\delta_\emptyset$ (class of an) empty derivation and thus, summing up $\delta_2 = \delta_2 \,;\, \delta_\emptyset \,;\, \delta' \,;\, \delta$. We conclude that $\delta_\emptyset \,;\, \delta' \,;\, \delta$ consists of empty and discrete derivations. Thus, by (3.2), we have that $\delta_2 \subseteq \varepsilon_1$, which suffices to conclude $\varepsilon_2 \subseteq \varepsilon_1$. The converse inclusion follows by symmetry.
Now the proof that $\leq$ is a partial order is trivial: if $\varepsilon_1 \leq \varepsilon_2$ and $\varepsilon_2 \leq \varepsilon_1$ then, by definition of $\leq$, $Der(\varepsilon_1) \subseteq Der(\varepsilon_2)$ and $Der(\varepsilon_2) \subseteq Der(\varepsilon_1)$ and therefore, by the previous observation, $\varepsilon_1 = \varepsilon_2$. The fact that $\leq$ satisfies the axiom of finite causes follows easily from the observation that the "prefixes" of a derivations (modulo empty direct derivations) are finite.
For the properties of relation $\#$, symmetry and irreflexivity are obvious; for hereditarity, $(\varepsilon \# \varepsilon_1 \wedge \varepsilon_1 \leq \varepsilon_2) \Rightarrow (Der(\varepsilon) \cap Der(\varepsilon_1) = \emptyset \wedge Der(\varepsilon_2) \subseteq Der(\varepsilon_1)) \Rightarrow (Der(\varepsilon) \cap Der(\varepsilon_2) = \emptyset) \Rightarrow (\varepsilon \# \varepsilon_2)$. □

Example 3.6 (Event structure of grammar $\mathcal{C}$-$\mathcal{S}$)
Figure 3.15 depicts part of the event structure of the graph grammar $\mathcal{C}$-$\mathcal{S}$ of Example 3.1. Continuous arrows form the Hasse diagram of the causality relation, while dotted lines connect events in direct conflict (inherited conflicts are not drawn explicitly).
Recalling that, intuitively, an event of a grammar corresponds to a specific application of a production together with its "history", a careful analysis of grammar $\mathcal{C}$-$\mathcal{S}$ allows to conclude that its event structure contains the following events:

$$E \;=\; \{req(n) \mid n \in \mathbb{N}\} \;\cup\; \{ser(w), rel(w) \mid w \in \mathbb{N}^{\otimes}\},$$

where $\mathbb{N}^{\otimes}$ denotes the set of non-empty sequences of *distinct* natural numbers. In fact, an application of production *REQ* only depends on previous applications of the same production (because it consumes a *job* edge, and only *REQ* can produce one), and therefore a natural number is sufficient to represent its history: conceptually, $req(n)$ is the event corresponding to the n-th application of *REQ*. An application of production *SER*, instead, depends both on a specific application of *REQ* (because it consumes a *req* edge), and, because of the S node it consumes and produces, either on the start graph or on a previous application of *SER* itself followed by *REL* (*SER* cannot be applied in presence of a *busy* edge connected to node S because of the dangling condition). It is not difficult to check that such an event can be determined uniquely by a non empty sequence of distinct natural numbers: $ser(n_1 n_2 \cdots n_k)$ is the event corresponding to the application of *SER* which serves the n_k-th *REQ*uest, after requests $n_1, \ldots, n_{k-1}$ have been served in this order. In turn, an application of production *REL* only depends on a previous application of *SER* (because of the *busy* edge), and we denote by $rel(w)$ the event caused directly by $ser(w)$.

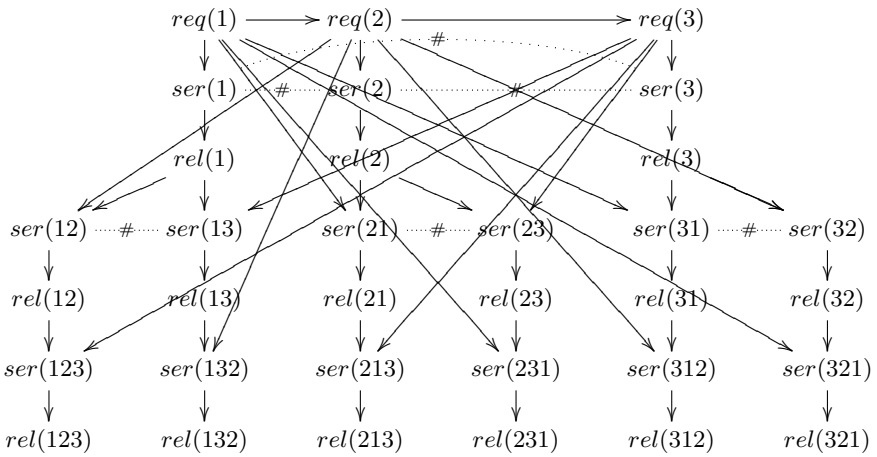


Figure 3.15: Event structure of the grammar $\mathcal{C}$-$\mathcal{S}$.

This informal description should be sufficient to understand the part of the event structure shown in Figure 3.15, including only the events which are caused by the first three requests, and the relationships among them. The causality and conflict relations of event structure $\mathbf{ES}[\mathcal{C}\text{-}\mathcal{S}]$ are defined as follows:

- $req(n) \leq req(m)$ iff $n \leq m$;
- $req(n) \leq ser(w)$ iff $n \in w$, that is, an application of *SER* only depends on the request it serves and on those served in its history;
- $ser(w) \leq ser(w')$ iff $w \sqsubseteq w'$, where $\sqsubseteq$ is the prefix ordering (the application of *SER* depends only on applications of itself in its history);
- $ser(w) \leq rel(w')$ iff $w \sqsubseteq w'$;
- $rel(w) \leq ser(w')$ iff $w \sqsubset w'$.
- for $x, y \in \{rel, ser\}$, $x(w)\#y(w')$ iff w and w' are incomparable with respect to the prefix ordering. □

The result relating the domain and the event structure of a grammar relies on a property of tc-equivalence, which only holds for consuming grammars.

It essentially states that any two tc-equivalent derivations are related by a unique "consistent" permutation among the applied productions. Therefore each production application can be seen as a uniquely determined event. Notice that this is the key point where the hypothesis of having a consuming grammar plays a rôle.

Lemma 3.6.10 (unique function between productions)
Let ρ and ρ' be two derivations of a consuming grammar $\mathcal{G}$, and suppose that $\langle \rho, h, f, \rho' \rangle$ and $\langle \rho, h, f', \rho' \rangle$ are consistent four-tuples. Then $f = f'$. The corresponding statement holds for consistent five-tuples as well.

Proof
First note that for each equivalence class of items $\gamma \in \mathit{Items}(\rho)$ and for each graph G_i in ρ, there is at most one item $x \in G_i$ in γ: this is due to the injectivity of all morphisms d_i, b_i in ρ (that follows from the injectivity of the productions).
Now suppose by absurd that $f \neq f'$, let $\hat{\imath} = \mathit{min}\{i \in \underline{\#\rho} \mid f(i) \neq f'(i)\}$, let $\mathit{prod}(\rho)(\hat{\imath}) : \left(L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R\right)$, and let $x \in L - l(K)$: such an x certainly exists, because all productions are consuming. Suppose moreover that $f(\hat{\imath}) = \left(\sum_{j=1}^{r} k'_j\right) + s$ and that $f'(\hat{\imath}) = \left(\sum_{j=1}^{r'} k'_j\right) + s'$. Since both four-tuples are consistent, by the conditions of Definition 3.5.1 we deduce that $\mathit{prod}(\rho')(f(\hat{\imath})) = \mathit{prod}(\rho')(f'(\hat{\imath}))$ and that $[g_r(in_L^s(x))]_{\rho'} = [g_{r'}(in_L^{s'}(x))]_{\rho'} \stackrel{def}{=} \gamma$ (see Figure 3.6). Now, since the squares in a derivation are pushouts and $x \in L - l(K)$, it follows that $g_r(in_L^s(x)) \notin d_r(D_r)$; this implies that G_{r-1} is the last graph of ρ such that $\mathit{Items}(G_{r-1}) \cap \gamma \neq \emptyset$, because by the above observation G_{r-1} contains at most one item of γ. Applying the same reasoning to $g_{r'}(in_L^{s'}(x))$, we deduce that $r = r'$ (thus necessarily $s \neq s'$) and $g_r(in_L^s(x)) = g_{r'}(in_L^{s'}(x))$. In words, we have found two distinct applications of the same production in the r-th direct derivation of ρ', and both applications consume the same item of G_{r-1}. But this clearly violates the identification condition, implying that the left square of the r-th direct derivation of ρ' cannot be a pushout, thus yielding a contradiction. More formally, this fact can be deduced by observing that the images of x in the left-hand side L_r of the r-th parallel production of ρ' must be distinct ($in_L^s(x) \neq in_L^{s'}(x)$) because L_r is a coproduct object; and that neither $in_L^s(x)$ nor $in_L^{s'}(x)$ is in $l_r(K_r)$.
The same proof applies to consistent five-tuples as well, by considering their underlying consistent four-tuples. □
We stress that this result strongly relies on the assumption that productions are consuming, and it implies that no "autoconcurrency" is possible for consuming

productions. If instead grammar $\mathcal{G}$ contains a non-consuming production q (i.e., $q : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$ and l is an isomorphism), then it is easy to find a counterexample to Lemma 3.6.10. Consider indeed the parallel production $q + q$, and the match $g \stackrel{def}{=} [id_L, id_L] : L + L \to L$: since l is an isomorphism, the gluing conditions are satisfied, and thus there is a graph H such that $\rho : L \Rightarrow_{\langle q+q,g\rangle} H$.[k] Then it is easy to check that both $\langle \rho, id_L, \Pi^2_{id}, \rho\rangle$ and $\langle \rho, id_L, \Pi^2_{\times}, \rho\rangle$ are consistent four-tuples, where $\Pi^2_{\times}(1) = 2$ and $\Pi^2_{\times}(2) = 1$.

Corollary 3.6.11 (unique permutation for tc-equivalent derivations)
Let ψ and ψ' be two decorated derivations of a consuming grammar. If $\psi \equiv \psi'$, then there is a unique permutation Π on $\underline{\#\psi}$ such that $\psi \equiv_{\Pi} \psi'$.
A fortiori, *the same holds for equivalence relations $\equiv^c$, $\equiv^{sh}$, and $\equiv^{abs}$.*

Proof
The statement follows from Theorem 3.5.3 and from Lemma 3.6.10. □
As one would expect, the number of events "appearing" in a derivation trace is bounded by the order of the trace. Moreover, as a consequence of the Lemma 3.6.10, for consuming grammars the bound is reached.

Lemma 3.6.12 (counting events in a trace)
Let $\mathcal{G}$ be a graph grammar and let η be a derivation trace in $\mathcal{G}$, with source in the start graph. Then

$$|\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}| \leq \#\eta,$$

where $\#\eta$ denote the order of any derivation in η. Moreover, if $\mathcal{G}$ is consuming then the equality holds, namely $|\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}| = \#\eta$.

Proof
For the first part, consider two different events ε_1 and ε_2 in $\mathbf{ES}[\mathcal{G}]$ and let ψ be a decorated derivation such that $\psi \in Der(\varepsilon_1) \cap Der(\varepsilon_2)$. We show that $\#\psi \geq 2$. By definition of $Der(\cdot)$, we have that

$$\psi \equiv^c \psi_{\varepsilon_1} ; \psi_1 \quad \text{and} \quad \psi \equiv^c \psi_{\varepsilon_2} ; \psi_2,$$

for some decorated derivations $\psi_{\varepsilon_i} \in \varepsilon_i$, ψ_i ($i \in \{1, 2\}$). Therefore $\psi_{\varepsilon_1} ; \psi_1 \equiv^c_{\Pi} \psi_{\varepsilon_2} ; \psi_2$, and surely $\Pi(\#\psi_{\varepsilon_1}) \neq \#\psi_{\varepsilon_2}$. Otherwise, by definition of event we could conclude $\varepsilon_1 = \varepsilon_2$. Thus necessarily $\#\psi \geq 2$.
The above observation can be easily generalized to prove that, for a derivation trace η, $|\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}| \leq \#\eta$.

[k] Clearly, if l is not surjective, then the match g would not satisfy the identification condition. Thus this counterexample does not apply if p is consuming.

Suppose now that $\mathcal{G}$ is a consuming grammar, let η be a derivation trace and let $\psi \in \eta$ be a decorated derivation. To gain more intuition, without loss of generality, we can restrict ourselves to "sequential" derivations which apply just one production at each step.
For each $i \in \underline{\#\psi}$ choose a derivation ψ_i such that $\psi \equiv^c_{\Pi_i} \psi_i$, which minimizes the value of $\overline{\Pi_i(i)}$ and let k_i be such a value. Let $\psi_i = \psi'_i ; \psi''_i$, with $\#\psi'_i = k_i$ (working with linear derivations surely ψ_i is decomposable in this way). It is easy to verify that $\varepsilon_i = [\psi'_i]$ is an event in $\mathbf{ES}[\mathcal{G}]$. Moreover, all such events are distinct. In fact suppose that for $i_1, i_2 \in \underline{\#\psi}$ the events ε_{i_1} and ε_{i_2} coincide. Then $\psi'_{i_1} \equiv^c_{\Pi} \psi'_{i_2} ; \psi_\emptyset$, where $\psi_\emptyset$ is a suitable decorated discrete derivation. Notice that necessarily, by definition of event, Π maps $k_{i_1} = k_{i_2}$ into itself.
Now, let Π' be the permutation relating ψ_{i_1} and ψ_{i_2}, namely $\psi_{i_1} \equiv^c_{\Pi'} \psi_{i_2}$. Since $\psi_{i_1} = \psi'_{i_1} ; \psi''_{i_1}$ and $\psi_{i_2} = \psi'_{i_2} ; \psi_\emptyset ; \psi_\emptyset^{-1} ; \psi''_{i_2}$, by Lemma 3.6.4, $\Pi'_{\underline{|\#\psi'_1}} = \Pi$ and thus also Π' maps k_{i_1} into itself. Summing up

$$\psi \equiv^c_{\Pi_{i_1}} \psi_{i_1} \equiv^c_{\Pi'} \psi_{i_2} \equiv^c_{\Pi^{-1}_{i_2}} \psi,$$

and therefore, by two applications of $(\mathsf{CTC} - \mathsf{trans})$ we obtain $\psi \equiv^c \psi$ via a permutation $\Pi^{-1}_{i_2} \circ \Pi' \circ \Pi_{i_1}$, mapping i_1 into i_2. Since $\mathcal{G}$ is consuming, by Lemma 3.6.10, such permutation must be the identity and thus $i_1 = i_2$.
This means that all events ε_i are distinct, showing that also the converse inequality $|\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}| \geq \#\eta$ holds. □
We are now ready to show the main result of this section, namely, that the domain of a consuming grammar is exactly the domain of finite configurations of the associated event structure.

Theorem 3.6.13 (domain and event structure)
The domain $\mathbf{Dom}[\mathcal{G}]$ *of a consuming graph grammar* $\mathcal{G}$ *is isomorphic to the domain of finite configurations of its event structure* $\mathbf{ES}[\mathcal{G}]$.

Proof
If Δ is a set of decorated derivations, let the *order* of Δ, denoted $\#\Delta$, be the smallest among the orders of derivations in Δ, i.e., $\#\Delta \stackrel{def}{=} min\{\#\psi \mid \psi \in \Delta\}$, and let $\mu(\Delta) \subseteq \Delta$ be the set of all the derivations of Δ which have minimal order: $\mu(\Delta) = \{\psi \in \Delta \mid \#\psi = \#\Delta\}$.
For each configuration C of $\mathbf{ES}[\mathcal{G}]$, let $\underline{\mu}(C) = \mu\left(\bigcap_{\varepsilon \in C} Der(\varepsilon)\right)$. We prove that $\underline{\mu} : \mathcal{FL}(\mathbf{ES}[\mathcal{G}]) \to \mathbf{Dom}[\mathcal{G}]$ is a well-defined function from finite configurations of $\mathbf{ES}[\mathcal{G}]$ to elements of $\mathbf{Dom}[\mathcal{G}]$, i.e., that $\underline{\mu}(C)$ is a derivation trace, by showing (1) that $\psi \in \underline{\mu}(C) \ \wedge \ \psi \equiv \psi' \Rightarrow \ \psi' \in \underline{\mu}(C)$, and (2) that $\psi, \psi' \in \underline{\mu}(C) \Rightarrow \psi \equiv \psi'$.

1. Suppose that $\psi \in \underline{\mu}(C)$ and $\psi \equiv \psi'$, or equivalently $\psi \equiv^c \psi' ; \psi''$ where

ψ'' is a discrete decorated derivation. Therefore for all $\varepsilon \in C$, there exists δ_ε and δ such that $[\psi]_c = \delta_\varepsilon ; \delta$. Recalling that $\psi \equiv^c \psi' ; \psi''$, we conclude that $\psi' ; \psi''$ is in $Der(\varepsilon)$ and therefore that $\psi' \in Der(\varepsilon)$ for all $\varepsilon \in C$. Therefore, observing that ψ' has the same order of ψ, we conclude $\psi' \in \underline{\mu}(C)$.

2. Suppose now that $\psi, \psi' \in \underline{\mu}(C)$. They clearly have the same order, and a permutation Π on $\overline{\#\psi}$ can be defined as follows: $\Pi(x) = y$ iff there exists an event $\varepsilon \in C$ and $\overline{\psi}_1, \psi_2 \in \varepsilon$ such that $\psi_1 ; \psi_1' \equiv^c_{\Pi_1} \psi$, $\psi_2 ; \psi_2' \equiv^c_{\Pi_2} \psi'$ and $\Pi_1(x) = \Pi_2(y) = \#\psi_1$. It can be shown that Π is well-defined (by using the defining property of events) and that the resulting four-tuple $\langle \rho_\psi, m_{\psi'} \circ m_\psi^{-1}, \Pi, \rho_{\psi'} \rangle$ is consistent; thus $\psi \equiv \psi'$ by Theorem 3.5.3.

Now, given a derivation trace η starting from $[G_s]$, let $\chi(\eta) = \{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}$. It is easy to verify that $\chi(\eta)$ is a configuration, i.e., it is conflict-free (because $\bigcap\{Der(\varepsilon) \mid \varepsilon \in \mathbf{ES}[\mathcal{G}] \ \wedge \ \varepsilon \in \chi(\eta)\} \supseteq \eta \neq \emptyset$), and left-closed, since $\varepsilon \leq \varepsilon'$ implies $Der(\varepsilon') \subseteq Der(\varepsilon)$. Finally, functions $\underline{\mu}$ and χ are inverse of each other.
In fact, for any derivation trace η

$$\begin{aligned}
&\underline{\mu}(\chi(\eta)) = \\
&\quad = \underline{\mu}(\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}) \\
&\quad = \mu\left(\bigcap\{Der(\varepsilon) \mid \varepsilon \in \mathbf{ES}[\mathcal{G}] \ \wedge \ \eta \subseteq Der(\varepsilon)\}\right)
\end{aligned}$$

Let $A = \bigcap\{Der(\varepsilon) \mid \varepsilon \in \mathbf{ES}[\mathcal{G}] \ \wedge \ \eta \subseteq Der(\varepsilon)\}$. Then, as observed above, $A \supseteq \eta$. Moreover, by Lemma 3.6.12, we have $|\{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \eta \subseteq Der(\varepsilon)\}| = \#\eta$. Thus for any $\psi \in A$, by applying again the same lemma, we conclude that $\#\psi \geq \#\eta$. Therefore the derivations in η has minimum length and thus $\underline{\mu}(\chi(\eta)) = \mu(A) = \eta$.
Viceversa, for every configuration C,

$$\chi(\underline{\mu}(C)) = \{\varepsilon \in \mathbf{ES}[\mathcal{G}] \mid \underline{\mu}(C) \subseteq Der(\varepsilon)\}.$$

Consider an event $\varepsilon \in \mathbf{ES}[\mathcal{G}]$ such that $\underline{\mu}(C) \subseteq Der(\varepsilon)$ and let $\psi \in \underline{\mu}(C)$. Since $\psi \in Der(\varepsilon)$ there are $\psi_\varepsilon \in \varepsilon$ and ψ' suitable decorated derivations such that $\psi \equiv^c_\Pi \psi_\varepsilon ; \psi'$. Then it is not difficult to prove that ε coincides with the event of C, corresponding to the $\Pi^{-1}(\#\psi_\varepsilon)$ production application in ψ. Therefore $\chi(\underline{\mu}(C)) \subseteq C$. The converse inclusion follows trivially from the definitions.

□

From this result it follows as an immediate consequence the main result of the section.

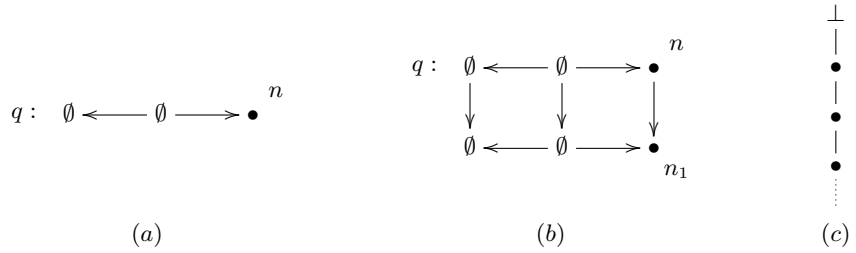


Figure 3.16: Domain and event structure for a consuming grammar.

Corollary 3.6.14 (algebraic structure of the domain of a grammar) *The domain of a consuming grammar $\mathcal{G}$, $\mathbf{Dom}[\mathcal{G}]$, is a a prime algebraic, finitely coherent and finitary partial order.*

Proof
The statement follows from Theorem 3.6.13 because such algebraic properties uniquely characterize the domain of finite configurations of a PES [25]. □
It is worth remarking that the correspondence between $\mathbf{Dom}[\mathcal{G}]$ and $\mathbf{ES}[\mathcal{G}]$ fails to hold when $\mathcal{G}$ is a non-consuming grammar. Consider, for instance, a grammar $\mathcal{G}_0$ with a unique production q, depicted in Figure 3.16.(a). It is not difficult to see that $\mathbf{ES}[\mathcal{G}_0]$ contains a unique event e, namely the derivation trace containing the derivation of Figure 3.16.(b), decorated in the only possible way. This fact is extremely intuitive: the unique event e represents the application of the production q, with the unique possible history, namely the empty history.
Computations in $\mathcal{G}_0$ would be hypothetically represented by sets of copies (multisets) of such event. We already noticed that in this situation the configurations of traditional event structures are not sufficiently expressive. Consequently, the domain $\mathbf{Dom}[\mathcal{G}_0]$ associated to the grammar, which is depicted in Figure 3.16.(c), is not isomorphic to the domain of configurations of $\mathbf{ES}[\mathcal{G}_0]$. In fact, the event structure associated to $\mathbf{Dom}[\mathcal{G}_0]$ has a countably infinite set of events $\{e_i \mid i \in \mathbb{N}\}$, with $e_i \leq e_{i+1}$ for each $i \in \mathbb{N}$. Intuitively, we can think that e_i represents the i^{th} firing of the event e and this justifies the causal dependencies among events: the $i + 1^{th}$ occurrence of e must follow the i^{th} occurrence. However this interpretation is meaningful only if we limit ourselves to sequential (non parallel) computations, which is unacceptable for a formalism which is intended as a model of concurrency.

3.6.3 Processes and events

The domain $\mathbf{Dom}[\mathcal{G}]$ and the event structure $\mathbf{ES}[\mathcal{G}]$ of a consuming grammar $\mathcal{G}$ have been constructed by using the category of derivation traces $\mathbf{Tr}[\mathcal{G}]$ of the grammar. Thanks to the very close relation existing between concatenable processes and concatenable derivation traces, we are able to provide a nice characterization of the finite configurations (elements of the domain $\mathbf{Dom}[\mathcal{G}]$) and of the events of $\mathbf{ES}[\mathcal{G}]$ in terms of processes. The result resembles the analogous correspondence existing for P/T nets [13] and is based on a similar notion of left concatenable process.

Definition 3.6.15 (abstract left c-process)
Two c-processes cp_1 and cp_2 are *left isomorphic*, denoted by $cp_1 \cong_l cp_2$, if there exists a pair of functions $f = \langle fg, fp \rangle$ satisfying all the requirements of Definition 3.4.14, but, possibly, the commutativity of the right triangle of the correspondent figure. An *abstract left c-process* is a class of left isomorphic c-processes $[cp]_l$. It is *initial* if $Min(cp) \simeq G_s$. It is *prime* if the causal order $\leq$ of cp, restricted to the set P of its productions, has a maximum element.

It is worth noticing that abstract left c-processes are the process-theoretical counterpart of (non concatenable) derivation traces. Indeed, it can be shown that abstract left c-processes are one-to-one with derivation traces, the correspondence being induced by the (arrow component of) functors $\mathcal{L}_A$ and $\mathcal{P}_A$. The next lemma states a property of concatenable derivation traces which is used in the characterization of the event structure of a grammar in terms of processes. However the result is also of some interest in itself. In fact, it formalizes the claim, reported several times in the chapter as an informal statement, according to which the dependency between two events in a trace is represented implicitly by the fact that the two events appears in the same order in all the derivations contained in the trace.

Lemma 3.6.16
Let ψ be a decorated derivation of a grammar $\mathcal{G}$ and let $i, j \in \#\psi$. Consider the process $cp = cp(\psi) = \langle m, p, M \rangle$ and let $q_i = \langle prod(\psi)(i), i \rangle$ and $q_j = \langle prod(\psi)(j), j \rangle$ be the productions of cp corresponding to the i^{th} and j^{th} productions applied in the derivation. Then

$$\Pi(i) \leq \Pi(j), \text{ for all } \psi' \text{ such that } \psi \equiv^c_\Pi \psi' \qquad \Leftrightarrow \qquad q_i \leq q_j.$$

Proof
($\Rightarrow$) The proof is done by contraposition. Suppose $q_i \not\leq q_j$; then we can find a linearization e of P_p such that $e^{-1}(q_i) > e^{-1}(q_j)$. By the proof of Lemma 3.5.9.(2) we have that $\psi \equiv^c_{\Pi^{\#\psi}_{id}} \psi(cp, prod(\psi))$ and, by the proof of

Lemma 3.5.6.(1), $\psi(cp, prod(\psi)) \equiv^c_\Pi \psi(cp, e)$, with $\Pi = e^{-1} \circ id_{P_p} \circ prod(\psi) = e^{-1} \circ prod(\psi)$. Therefore $\psi \equiv^c_\Pi \psi(cp, e)$ and notice that $\Pi(i) = e^{-1}(q_i) > \Pi(j) = e^{-1}(q_j)$, by the choice of e. This concludes the proof of the right implication.

($\Leftarrow$) Suppose $q_i \leq q_j$. Let ψ' be a derivation such that $\psi \equiv^c_\Pi \psi'$ and consider the process $cp' = cp(\psi')$. Since $\psi \equiv^c_\Pi \psi'$, by Lemma 3.5.8, $cp \cong cp'$ via an isomorphism $f = \langle fg, fp \rangle$, with $fp(\langle prod(\psi)(k), k \rangle = \langle prod(\psi')(\Pi(k)), \Pi(k) \rangle$. Since by Corollary 3.4.11 the function fp is monotonic w.r.t. causal order, we have $fp(q_i) \leq_{cp'} fp(q_j)$, namely $\langle prod(\psi')(\Pi(i)), \Pi(i) \rangle \leq_{cp'} prod(\psi')(\Pi(j)), \Pi(j) \rangle$. Recalling the way the function $cp(\cdot)$ is defined one concludes that necessarily $\Pi(i) \leq \Pi(j)$. □

We finally obtain the announced characterization, which has a clear intuitive meaning if one thinks of the productions of (the occurrence grammar of) a process as instances of production applications in the original grammar $\mathcal{G}$, and therefore as possible events in $\mathcal{G}$.

Theorem 3.6.17
Let $\mathcal{G}$ be a consuming grammar. Then there is a one to one correspondence between:

1. *initial left c-processes and elements of* $\mathbf{Dom}[\mathcal{G}]$*;*

2. *prime initial left c-processes and elements of* $\mathbf{ES}[\mathcal{G}]$.

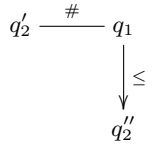
Proof
1. Just use the correspondence between abstract left c-processes and derivation traces stressed before, and then Lemma 3.6.7.

2. By Lemma 3.6.16 it is immediate to conclude that prime initial left c-processes are one-to-one with derivation traces $[\psi]$ such that for all $\psi' \equiv^c_\Pi \psi'$, $\Pi(\#\psi) = \#\psi$. Then, to conclude, simply notice that such derivation traces are exactly the events of $\mathbf{ES}[\mathcal{G}]$. □

This result confirms the appropriateness of the chosen notion of (concatenable) process and makes us confident on the possibility of carrying on the program of extending notions and results from the theory of Petri nets to graph transformation systems.
Notice that, although developed under the assumption that grammar $\mathcal{G}$ is consuming, most of the results of this subsection hold for non consuming grammars as well. More precisely, the only result which uses the assumption of having consuming grammars in an essential way is point (2) of Theorem 3.6.17.

3.6.4 Adequateness of PES: asymmetric conflict in graph grammars

The result presented in this section, as well as many other contributions in the literature, consider (prime) event structures as an adequate semantic domain for graph transformation systems. However, not all researchers agree on this. The point is that graph transformation systems manifest a form of asymmetric conflict that cannot be faithfully captured by prime event structures.
In fact, suppose that there is an item that is only read by a production q_1 (i.e., it is in the interface graph K_1) and consumed by another production q_2. Then one would say that these production have some conflicting behavior on the common item. Indeed after having applied q_1 we can safely apply q_2, but the application of q_2 inhibits q_1. This situation cannot be modelled in a direct way within a traditional prime event structure: q_1 and q_2 are neither in conflict nor concurrent nor causal dependent. Simply, as for a traditional conflict, the application of q_2 prevents q_1 to be executed, so that q_1 can never follow q_2 in a derivation. But the converse is not true, since q_1 *can* be applied before q_2. Therefore, this situation can be naturally interpreted as an *asymmetric conflict* between the two productions. Equivalently, since q_1 precedes q_2 in any derivation where both are applied, in such derivations, q_1 acts as a cause of q_2. However, differently from a true cause, q_1 is not necessary for q_2 to be applied. Therefore we can also think of the relation between the two transition as a *weak* form of *causal dependency*.
This problem has been overcome in our approach via a reasonable encoding of this situation in a PES, namely representing the application of q_2 with two distinct mutually exclusive events: q_2', representing the execution of q_2 that prevents q_1, thus mutually exclusive with q_1, and q_2'', representing the application of q_2 after q_1 (caused by q_1).



This encoding can be unsatisfactory since it determines an "explosion" of the number of events (notice that not only q_2, but also all its consequences are duplicated) and it does not represent faithfully the dependencies between events. Solutions which explicitly take into account the presence of asymmetric conflicts have been proposed in [40,41,42], where generalized notions of event structure are defined, enriched with an explicit relation modeling a non-symmetric version of conflict. In particular, in [40,43] Pinna and Poigné, starting from the "operational" notion of event automaton, suggest an enrich-

ment of prime event structures and flow event structures with *possible causes*. The basic idea is that if e is a possible cause of e', then e can precede e' or it can be ignored, but the execution of e never follows e'. This is formalized by introducing an explicit subset of possible events in prime event structures or adding a "possible flow relation" (weak causality) in flow event structures. As one could expect the asymmetric conflict relation allows to represent the usual symmetric conflict [41,42]. Moreover, in [42] the relation with traditional PES (and with domains) is formalized by showing that there exists a coreflection between the category of PES and a category of *asymmetric event structures*. This allows for an elegant translation from the asymmetric event structure semantics to the PES semantics via an adjoint functor.
Some authors have also explored the possibility of allowing the parallel execution of weakly dependent events. Under this assumption also a generalized (in the sense described above) event structure semantics is no more suited (in the sense that it does not express the fact that two events can happen concurrently without allowing for all possible serializations of them): new semantic domains have to be looked for, possibly based on the work by Janicki and Koutny [44,45], who address similar problems in the setting of Petri nets with inhibitor arcs. It is interesting to recall that, although the classical solution for the DPO approach consists in forbidding the parallel application of weakly dependent productions, some proposals exist in the literature (see e.g., [46]) exploring the possibility of allowing for the synchronized application of weakly dependent productions.

Besides discussing the problems and the possible solutions arising when one tries to provide graph grammars with an event structure semantics, one could also wonder how far event structures are adequate as a concurrent semantics for graph grammars. An event structure semantics abstracts completely from the structure of states, as it only shows the causal and conflict relations among the transitions of a system. Thus, since both nets and grammars have a *set* of transitions, it comes of no surprise that, under a certain degree of abstraction, they have a similar semantics. However, being the state of a graph grammar far more complex than the state of a net, the doubt arises that in the case of grammars, forgetting completely the structure of the state is no more a reasonable choice and the obtained semantics turns out to be scarcely significant with respect to the real behaviour of the original system.
Obviously the answer is not unique and depends on what aspects of the computation of a grammar one is interested to observe. On the one hand a more concrete semantics is readily given by graph processes, where the type graph provides full information on the state. On the other hand one could also consider the event structure semantics too concrete, since it does not validate, for

instance, the natural equality $P = P + P$, for a process P. To gain such level of abstraction one could resolve to "observe" the behaviour of grammars via suitable bisimulation relations. Notice that based on the process and event structure semantics for grammars, classical notions of bisimulation (e.g., history preserving bisimulation) can be used. However also in this case trying to define new notions of bisimulation, explicitly tailored on graph transformation systems, seems an interesting task.

3.7 Related work

Various authors have proposed concurrent semantics for graph transformation systems that are related to the process and event structure semantics presented in the previous sections. Here we give a short overview of the contributions to the literature we are aware of.

Given the strong relationship between Petri nets and graph grammars sketched in the Introduction and elaborated in Section 3.2.1, and considering the importance of the notion of process in Petri net theory, it is not surprising that various kinds of notions related to processes have been considered for the concurrent semantics of graph grammars as well. Kreowski and Wilharm [28] define *derivation processes* as partial orders of direct derivations. An equivalence is defined on processes, that essentially relates all processes containing the same derivation steps, disregarding the order and the multiplicity in which they appear. *Complete processes*, i.e., processes that for each pair of independent derivation steps contain the whole Church-Rosser diamond, are shown to be standard representatives of equivalence classes. In particular, complete *conflict-free* processes are representatives of shift-equivalence classes of sequential derivations, and thus they are one-to-one with the graph processes of Section 3.4. The main difference is that derivation processes of [28] are not themselves grammars. The same authors study various kinds of transformations of derivation processes in [47].

A process semantics has been proposed by Janssens for Extended Structure Morphisms systems, which are an evolution of Actor Grammars [48], based on the Node Label Controlled approach to graph transformation. In [49] the notion of *computation structure* is introduced, which formalizes a rewriting process of an ESM system. The notion may be seen as a generalized version of the computation graphs of [50]. Moreover, it is shown in [51] that abstract computation structures and concrete configuration graphs form a category that is a natural generalization of the well-known notion of a transition system. The set (or category) of computation structures may be taken as the basic process semantics of an ESM system.

In [52] Ribeiro proposes a notion of non-deterministic unfolding of graph grammars in the SPO approach. Such unfolding is itself a grammar and can be regarded as a non-deterministic process. Interestingly, the construction is carried out at categorical level, by defining an unfolding functor from a category of graph grammars to a category of (abstract) *occurrence grammars*, and then showing that it is a right adjoint to a suitable *folding functor*. This result is then exploited to prove that the unfolding semantics is compositional with respect to certain operations on grammars defined in terms of limits.

Maggiolo and Winkowski introduce in [35] *dynamic graphs*, which are partially ordered structures closely related to (a nondeterministic variant of) the occurrence grammars underlying the concatenable processes of Section 3.4. Such dynamic graphs are studied as mathematical structures of their own, emphasizing their composition/decomposition properties. Even if they are quite similar to concatenable processes, they miss a formal result of correspondence with graph derivations, and they are not defined as grammars.

For what concerns the event structure semantics of graph transformation systems, the first proposal on the topic, to our knowledge, is the paper by Schied [38], where a PES is associated to a grammar by using a *deterministic* variation of the DPO approach. The key idea in his approach is that at each direct derivation the derived graph is uniquely determined by the host graph, the applied production and the occurrence morphism. This is achieved by giving to newly created items of the derived graph an identity that is uniquely determined by their generation history. To obtain a PES from the collection of all derivations of a grammar starting from the start graph, as intermediate step a *trace language* (defined using the shift-equivalence) is constructed, and then general results from [53] are applied to extract the event structure from the trace language.

The equivalence of the event structure semantics by Schied and that of Section 3.6 has been proved recently in [54], where a further equivalent characterization of the PES of a grammar is proposed. The idea there is to build first a nondeterministic unfolding of the graph grammar, and then to extract from it the prime algebraic domain of its configurations, which uniquely determines a PES. This approach is the closest (among those we are aware of) to the classical way of extracting a prime event structure from a Petri net, as described in [10].

For the algebraic Single-Pushout approach to graph transformation, an event structure semantics has been proposed by Korff in [55], together with an application to Actor Systems. In a more abstract framework, Richard Banach proposes in [56] a very general categorical construction based on "op-fibrations", intended to be applicable to various graph rewriting formalisms. The proposed

construction is able to associate an event structure semantics with a variety of graph transformation approaches. However, in the case of the algebraic, DPO approach, the resulting event structure does not coincide with ours: The precise reasons of this mismatch deserve further investigation.

Let us conclude by recalling the main sources of the technical material presented in this chapter, and by pointing out some little changes that we adopted in our presentation.

The derivation trace semantics presented in Section 3.3 is essentially taken from [17]. It is obtained by rephrasing in the setting of *typed* graph grammars the construction proposed in [18,5] for classical DPO grammars. The main difference with respect to those papers is the technical solution adopted to make traces concatenable. The solution used in [15,18,5] consists of choosing for each pair of isomorphic graphs a distinguished isomorphism relating them, named *standard isomorphism* (see Definition 3.8.3). Then two concrete derivations are called *abstraction equivalent* if they are isomorphic and, in addition, the two isomorphisms relating the source and the target graphs, respectively, are standard. Like the decoration of the source and target graphs of a derivation adopted in this chapter, this technique ensures that the sequential composition of derivations, defined at the concrete level, can be lifted naturally to the composition of abstract derivations. Interestingly, the resulting category of derivation traces, say $\mathbf{Tr}^s[\mathcal{G}]$, turns out to be isomorphic to category $\mathbf{Tr}[\mathcal{G}]$ of Definition 3.3.12. In fact two functors can be defined, $\mathbf{UnDec} : \mathbf{Tr}[\mathcal{G}] \to \mathbf{Tr}^s[\mathcal{G}]$ and $\mathbf{Dec} : \mathbf{Tr}^s[\mathcal{G}] \to \mathbf{Tr}[\mathcal{G}]$ that are inverse each other. Intuitively, **Dec** is defined on the basis of a transformation that equips a given derivation with a "canonical decoration", consisting of the standard isomorphisms from the canonical graphs to the source and target graphs. Viceversa, **UnDec** essentially transforms every decorated derivation into a derivation between canonical graphs, by pre- and post-composing it with suitable empty derivations.

Both technical solutions have their advantages. On the one hand, standard isomorphisms allow to define the composition of derivations as they are defined in the classical way, as sequences of double pushout diagrams (without additional decorations). On the other hand, a drawback of this approach is that the notion of equivalence between derivations does depend on representation details, i.e., on the real identity of the items of the graphs involved in the derivation. For instance, given a derivation, if we change uniformly the name of a node in all the involved graphs and relations, in general we obtain a new derivation that can be non-equivalent to the original one. On the contrary, it has been already emphasized in the chapter that the solution based on decorations does not suffer of this problem.

The definitions of abstraction equivalence here and in [5,18] differ for another

little point. According to [5,18] derivations which differ for the order in which productions are composed inside a single direct parallel steps are abstraction equivalent. Instead, in our approach also this kind of "switching" is taken into account only by the shift equivalence.

With respect to [17], a minor, technical difference of our presentation of the trace semantics resides in the way the sequential composition of decorated derivations is defined. Here two such derivations ψ and ψ' are composable only if the target graph of ψ and the source graph of ψ', as well as the corresponding decorations M_ψ and $m_{\psi'}$, coincide, while according to [17] it was sufficient that $\tau(\psi) \simeq \sigma(\psi')$, and the composed derivation was defined by inserting in the middle a suitable empty derivation. Despite that, we already emphasized that the notion of concatenation induced at a more abstract level on traces is exactly the same.

The contents of Section 3.4 is elaborated from [16] for the part concerning graph processes, and from [17] for concatenable processes. With respect to the original definition in [16], our graph processes exhibit a subtle but relevant difference. According to our definition the isomorphisms between productions in the process and corresponding productions in the grammar (the isomorphisms "ι" of Definition 3.4.9) are part of the process, and the notion of process isomorphism is required to be consistent with such isomorphisms. In [16], instead, such isomorphisms are not explicitly provided, but they are only required to exist: This leads to a looser notion of process isomorphism. Furthermore, unlike [16], we allow processes to start from any graph and not only from the *start graph* of the grammar. This is clearly needed to define a reasonable notion of concatenable process. With respect to the concatenable processes originally proposed in [17] the only difference is, again, that sequential composition is defined in a more restrictive way at the concrete level, but the notions of composition at the level of abstract processes coincide.

Finally, the construction of the event structure $\mathbf{ES}[\mathcal{G}]$ for a grammar $\mathcal{G}$ in Section 3.6 is essentially taken from [18], by adapting the definitions and some of the results to the "typed" framework and to non-consuming grammars.

Acknowledgements

Most of the research results presented in this chapter have been developed with the support of TMR Network GETGRATS, Esprit WG APPLIGRAPH and MURST project Tecniche Formali per Sistemi Software.

3.8 Appendix: Construction of canonical graphs

We present here an explicit construction of the canonical graph associated to a given finite typed graph, as introduced in Definition 3.3.1. This construction is a straightforward adaptation to the typed framework of the analogous construction of [35] for labelled graphs. We also discuss how a class of *standard isomorphism* in the sense of [34,15] can be defined using canonical graphs, under a mild assumption on the category of graphs.
In the following we suppose that $TG = \langle N', E', s', t' \rangle$ is a fixed type graph. Furthermore, we assume that the set of nodes N' and the set of arcs E' of TG are equipped with fixed total orders.[l]

Definition 3.8.1 (arranged graph)
Let $G = \langle \langle N, E, s, t \rangle, f \rangle$ be a finite graph typed over TG. An *arranged graph* associated with G is a typed graph G_1, isomorphic to G, obtained as follows:

- for each $\lambda \in N'$, the nodes of G labelled with λ (i.e., in $f_N^{-1}(\lambda)$) are arranged in a sequence

$$n_0, \ldots, n_i, \ldots, n_p \in f^{-1}(\lambda)$$

  and then renamed as

$$\langle \lambda, 0 \rangle, \ldots, \langle \lambda, i \rangle, \ldots, \langle \lambda, p \rangle;$$

- for each $\nu : \lambda \to \mu \in E'$ and for each pair of (renamed) nodes $\langle \lambda, i \rangle$, $\langle \mu, j \rangle$ the arcs of G labelled with ν (i.e., in $f_E^{-1}(\nu)$) with source $\langle \lambda, i \rangle$ and target $\langle \mu, j \rangle$ are arranged in a sequence

$$e_0, \ldots, e_k, \ldots, e_q \in f^{-1}(\nu)$$

  and then renamed as

$$\langle \nu, \langle \lambda, i \rangle, \langle \mu, j \rangle, 0 \rangle, \ldots, \langle \nu, \langle \lambda, i \rangle, \langle \mu, j \rangle, k \rangle, \ldots, \langle \nu, \langle \lambda, i \rangle, \langle \mu, j \rangle, q \rangle.$$

We assign to G_1 a code

$$code(G_1) = uv,$$

[l] Recall that a partial order $\langle A, \leq \rangle$ is a *total order* if for all $a, b \in A$, $a \leq b$ or $b \leq a$. The partial order $\langle A, \leq \rangle$ is a *well-order* if each non-empty subset of A has a least element. Clearly each finite total order is well-ordered.

where u is the sequence of ordered nodes and v is the sequence of ordered arcs.[m]

Notice that the code of an arranged graph uniquely determines the graph and its typing. Hence if G_1 and G_2 are two arranged graphs with $code(G_1) = code(G_2)$ then $G_1 = G_2$.
Obviously if two arranged graphs G_1 and G_2 are isomorphic then $code(G_1) = uv_1$ and $code(G_2) = uv_2$, i.e., they have the same node code and arc codes of the same length.
Moreover since N' and E' are well-ordered, also the set of "arranged edges", $AE = E' \times (N' \times \mathbb{N})^2 \times \mathbb{N}$, with the lexicographical order, is well-ordered. Therefore the set of sequences of arranged edges of fixed length AE^n (ordered lexicographically) is well-ordered, for each choice of n. Thus we can associate to each finite graph G typed over TG an (isomorphic) arranged graph with minimum code. We denote such a graph by $Can(G)$, and we call it the *canonical graph* for G.
The following proposition ensures that the second condition of Definition 3.3.1 is satisfied as well, i.e., that we can consider Can as an operation on abstract graphs.

Proposition 3.8.2
Let G and G' be graphs typed over TG. If $G \simeq G'$, then $Can(G) = Can(G')$.

Proof
Let $f : G \to G'$ be an isomorphism and let $i : Can(G) \to G$, $i' : Can(G') \to G'$ be the obvious isomorphisms induced by the construction of the arranged graph.
Since $f \circ i : Can(G) \to G'$ is an isomorphism, $Can(G)$ is an arranged graph for graph G'. Hence $code(Can(G')) \leq code(Can(G))$ and in the same way $code(Can(G)) \leq code(Can(G'))$. Therefore $code(Can(G)) = code(Can(G'))$ and thus $Can(G) = Can(G')$. □
We show now how a class of standard isomorphisms can be defined by using the construction of canonical graphs just presented. As summarized in the previous section, standard isomorphisms were introduced in [34,15] to define the composition of abstract derivations, therefore as an alternative technique with respect to the use of decorations. Let us fist recall the definition.

[m] We consider on arranged nodes and arcs the usual lexicographical order.

Definition 3.8.3 (standard isomorphisms)
A family s of *standard isomorphisms* in category **TG-Graph** is a family of isomorphisms indexed by pairs of isomorphic graphs (i.e., $s = \{s(G, G') \mid G \simeq G'\}$), satisfying the following conditions for each G, G' and $G'' \in |\mathbf{TG\text{-}Graph}|$:

- $s(G, G') : G \to G'$;
- $s(G, G) = id_G$;
- $s(G'', G') \circ s(G, G'') = s(G, G')$.

Now let G be a finite graph typed over TG. An isomorphism $f : Can(G) \to G$ can be represented as a sequence of pairs:

$$\langle n_0, f(n_0)\rangle, \ldots, \langle n_p, f(n_p)\rangle,\ \langle e_0, f(e_0)\rangle, \ldots, \langle e_q, f(e_q)\rangle,$$

with $n_i \in N_{Can(G)}$ and $e_k \in E_{Can(G)}$. Therefore, if we suppose that the sets of (possible) nodes and arcs are well-ordered, we can choose a minimum (w.r.t. lexicographical order) isomorphism, denoted by

$$i(G) : Can(G) \to G.$$

Now, given two TG-typed graphs G and G' we can define the standard isomorphism $s(G, G')$ as

$$s(G, G') = i(G') \circ i(G)^{-1}.$$

It is immediate to check that this definition satisfies the conditions Definition 3.8.3. Notice also that if we consider only finite graphs, we can safely assume that the sets of nodes and arcs of the type graph and of all the other graphs we deal with are well-ordered without assuming the axiom of choice: for example, we can simply assume that nodes and arcs are natural numbers. Under this assumption the construction of both $Can(G)$ and of the standard isomorphisms is effective.

References

1. C.A. Petri. *Kommunikation mit Automaten.* Schriften des Institutes für Instrumentelle Matematik, Bonn. 1962.
2. W. Reisig. *Petri Nets: An Introduction.* EACTS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
3. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations.* World Scientific, 1997.
4. H. Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 3–14. Springer Verlag, 1987.

5. 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. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations.* World Scientific, 1997.
6. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoret. Comput. Sci.*, 109:181–224, 1993.
7. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations.* World Scientific, 1997.
8. G. Rozenberg. Behaviour of Elementary Net Systems. In *Petri Nets: Central Models and Their Properties*, volume 254 of *LNCS*, pages 60–94. Springer Verlag, 1987.
9. U. Golz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
10. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoret. Comput. Sci.*, 13:85–108, 1981.
11. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 4: Semantic Modelling.* Oxford University Press, 1995.
12. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
13. J. Meseguer, U. Montanari, and V. Sassone. On the semantics of Petri nets. In *Proceedings CONCUR '92*, volume 630 of *LNCS*, pages 286–301. Springer Verlag, 1992.
14. H.-J. Kreowski. *Manipulation von Graphmanipulationen.* PhD thesis, Technische Universität Berlin, 1977.
15. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Abstract Graph Derivations in the Double-Pushout Approach. In H.-J. Schneider and H. Ehrig, editors, *Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science*, volume 776 of *LNCS*, pages 86–103. Springer Verlag, 1994.
16. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
17. P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 283–295. Springer Verlag, 1998.

18. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An Event Structure Semantics for Graph Grammars with Parallel Productions. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*. Springer Verlag, 1996.
19. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. An event structure semantics for safe graph grammars. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, IFIP Transactions A-56, pages 423–444. North-Holland, 1994.
20. H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer Verlag, 1979.
21. H. Ehrig. Aspects of Concurrency in Graph Grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *LNCS*, pages 58–81. Springer Verlag, 1983.
22. P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. *Acta Informatica*, 33:641–647, 1996.
23. V. Sassone. An axiomatization of the algebra of Petri net concatenable processes. *Theoret. Comput. Sci.*, 170:277–296, 1996.
24. G. Winskel. Event Structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 325–392. Springer Verlag, 1987.
25. G. Winskel. An Introduction to Event Structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 325–392. Springer Verlag, 1989.
26. S. Mac Lane. *Categories for the working mathematician.* Springer Verlag, 1971.
27. H.-J. Kreowski. A comparison between Petri nets and graph grammars. In H. Noltemeier, editor, *Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science*, volume 100 of *LNCS*, pages 306–317. Springer Verlag, 1981.
28. H.-J. Kreowski and A. Wilharm. Net processes correspond to derivation processes in graph grammars. *Theoret. Comput. Sci.*, 44:275–305, 1986.
29. H.-J. Schneider. On categorical graph grammars integrating structural transformations and operations on labels. *Theoret. Comput. Sci.*, 109:257–274, 1993.

30. A. Corradini. Concurrent Graph and Term Graph Rewriting. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR'96*, volume 1119 of *LNCS*, pages 438–464. Springer Verlag, 1996.
31. R. Janicki and M. Koutny. Semantics of inhibitor nets. *Information and Computation*, 123:1–16, 1995.
32. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32, 1995.
33. W. Vogler. Efficiency of asynchronous systems and read arcs in Petri nets. Technical Report 352, Institüt für Mathematik, Augsburg University, 1996.
34. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Note on standard representation of graphs and graph derivations. In H.-J. Schneider and H. Ehrig, editors, *Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science*, volume 776 of *LNCS*, pages 104–118. Springer Verlag, 1994.
35. A. Maggiolo-Schettini and J. Winkowski. Dynamic Graphs. In *Proceedings of MFCS'96*, volume 1113 of *LNCS*, pages 431–442, 1996.
36. H.-J. Kreowski. Is parallelism already concurrency? Part 1: Derivations in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 343–360. Springer Verlag, 1987.
37. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and Concurrency in High-Level Replacement Systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.
38. G. Schied. On relating Rewriting Systems and Graph Grammars to Event Structures. In H.-J. Schneider and H. Ehrig, editors, *Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science*, volume 776 of *LNCS*, pages 326–340. Springer Verlag, 1994.
39. G. Berry. Stable models of typed lambda-calculi. In *Proceeding of the 5^{th} ICALP*, volume 62 of *LNCS*, pages 72–89. Springer-Verlag, 1978.
40. G. M. Pinna and A. Poigné. On the nature of events. In *Mathematical Foundations of Computer Science*, volume 629 of *LNCS*, pages 430–441. Springer Verlag, 1992.
41. R. Langerak. Bundle Event Structures: A Non-Interleaving Semantics for Lotos. In *5^{th} Intl. Conf. on Formal Description Techniques (FORTE'92)*, pages 331–346. North-Holland, 1992.
42. P. Baldan, A. Corradini, and U. Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. In M. Nivat, editor, *Proceedings of FoSSaCS '98*, volume 1378, pages 63–80. Springer

Verlag, 1998.

43. G. M. Pinna and A. Poigné. On the nature of events: another perspective in concurrency. *Theoretical Computer Science*, 138:425–454, 1995.
44. R. Janicki and M. Koutny. Invariant semantics of nets with inhibitor arcs. In *Proceedings CONCUR '91*, volume 527 of *LNCS*. Springer Verlag, 1991.
45. R. Janicki and M. Koutny. Structure of concurrency. *Theoret. Comput. Sci.*, 112:5–52, 1993.
46. A. Corradini and F. Rossi. Synchronized Composition of Graph Grammar Productions. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *LNCS*. Springer Verlag, 1996.
47. H.-J. Kreowski and A. Wilharm. Is parallelism already concurrency? Part 2: Non-sequential processes in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of the 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 361–377. Springer Verlag, 1987.
48. D. Janssens and G. Rozenberg. Actor Grammars. *Mathematical Systems Theory*, 22:75–107, 1989.
49. D. Janssens. ESM systems and the composition of their computations. In *Graph Transformations in Computer Science*, volume 776 of *LNCS*, pages 203–217. Springer Verlag, 1994.
50. D. Janssens, M. Lens, and G. Rozenberg. Computation graphs for actor grammars. *Journal of Computer and System Science*, 46:60–90, 1993.
51. D. Janssens and T. Mens. Abstract semantics for ESM systems. *Fundamenta Informaticae*, 26:315–339, 1996.
52. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
53. M.A. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, University of Sussex, 1988. Report no. 1/88.
54. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578, pages 73–89. Springer Verlag, 1999.
55. M. Korff. True Concurrency Semantics for Single Pushout Graph Transformations with Applications to Actor Systems. In *Proceedings International Workshop on Information Systems – Corretness and Reusability, IS-CORE'94*, pages 244–258. Vrije Universiteit Press, 1994. Tech. Report IR-357.

56. R. Banach. DPO Rewriting and Abstract Semantics via Opfibrations. In A. Corradini and U. Montanari, editors, *Proceedings SEGRAGRA'95*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences, 1995. `http://www.elsevier.nl/locate/entcs/volume2.html`.