Event Structure Semantics for Nominal Calculi*

Roberto Bruni[1], Hernán Melgratti[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italia.
[2] IMT Lucca Institute for Advance Studies, Italia.
bruni@di.unipi.it, hernan.melgratti@imtlucca.it, ugo@di.unipi.it

Abstract. Event structures have been used for giving true concurrent semantics to languages and models of concurrency such as CCS, Petri nets and graph grammars. Although certain nominal calculi have been modeled with graph grammars, and hence their event structure semantics could be obtained as instances of the general case, the main limitation is that in the case of graph grammars the construction is more complex than strictly necessary for dealing with usual nominal calculi and, speaking in categorical terms, it is not as elegant as in the case of Petri nets. The main contribution of this work is the definition of a particular class of graph grammars, called *persistent*, that are expressive enough to model name passing calculi while simplifying the denotational domain construction, which can be expressed as an adjunction. Finally, we apply our technique to derive event structure semantics for pi-calculus and join-calculus processes.

1 Introduction

The paper by Varacca and Yoshida [23] advocates the definition of true concurrent semantics for π-calculus, renewing the interest in the use of event structures in connection with process calculi, a long-standing thread initiated by Winskel's semantics of Milner's CCS [24]. Their main contribution is an original typing system on the event structure for controlling the behavior of linear processes. Actually, they also suggest that their formalization is the first event structure semantics for the π-calculus, which (as also discussed in their concluding section) is true just in part.

We argue that the techniques were already available for deriving an event structure semantics (but not the results in [23]), even if the pieces were not put together yet. To explain this, we have to go back to the joint work of the third author with Pistore [19] on the encoding of π-calculus in Graph Transformation Systems (GTS), under the so-called Double Pushout approach (DPO) [9,10]. While Petri nets can account for CCS-like languages, it seems that nominal calculi fit better in the GTS approach, where name creation, dynamic network topology, and causality due to name passing can be more easily accounted for. However, some of the latest results about concurrent semantics for GTS were not available at that time, and the existing techniques were not as much sophisticated as available today, so that no explicit definition of the associated event structure semantics was given. More recently, [1] made some substantial advancements on the true concurrent semantics of DPO, by explaining in terms of the so-called *inhibitor event structures* the semantics of a large class of GTS. This result was achieved

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA

$$\begin{tikzcd}
\textbf{Safe Nets} \arrow[r, shift right, "\mathcal{U}"'] \arrow[r, phantom, "\perp"] & \textbf{Occurrence Nets} \arrow[l, hook', shift right] \arrow[r, shift right, "\mathcal{E}"'] \arrow[r, phantom, "\perp"] & \textbf{Prime Event Structures} \arrow[l, shift right, "\mathcal{N}"'] \arrow[r, shift right, "\mathcal{L}"'] \arrow[r, phantom, "\sim"] & \textbf{Domains} \arrow[l, shift right, "\mathcal{P}"'] \\
\textbf{Semi-Weighted Contextual Nets} \arrow[r, shift right, "\mathcal{U}_a"'] \arrow[r, phantom, "\perp"] & \textbf{SW Occurrence Nets} \arrow[l, hook', shift right] \arrow[r, shift right, "\mathcal{E}_a"'] \arrow[r, phantom, "\perp"] & \textbf{Asymmetric Event Structures} \arrow[l, shift right, "\mathcal{N}_a"'] \arrow[r, shift right, "\mathcal{L}_a"'] \arrow[r, phantom, "\perp"] & \textbf{Domains} \arrow[l, shift right, "\mathcal{P}_a"'] \\
\textbf{SW DPO Graph Grammars} \arrow[r, shift right, "\mathcal{U}_g"'] \arrow[r, phantom, "\perp"] & \textbf{SW Occurrence Grammars} \arrow[l, hook', shift right] \arrow[r, "\mathcal{E}_g"'] & \textbf{Inhibitor Event Structures} \arrow[r, shift right, "\mathcal{L}_i"'] \arrow[r, phantom, "\perp"] & \textbf{Domains} \arrow[l, shift right, "\mathcal{P}_i"'] \\
\textbf{SW SPO Graph Grammars} \arrow[r, shift right, "\mathcal{U}_s"'] \arrow[r, phantom, "\perp"] & \textbf{SW Occurrence Grammars} \arrow[l, hook', shift right] \arrow[r, shift right, "\mathcal{E}_s"'] \arrow[r, phantom, "\perp"] & \textbf{Asymmetric Event Structures} \arrow[l, shift right, "\mathcal{N}_s"'] \arrow[r, shift right, "\mathcal{L}_a"'] \arrow[r, phantom, "\perp"] & \textbf{Domains} \arrow[l, shift right, "\mathcal{P}_a"']
\end{tikzcd}$$

Fig. 1. A recollection of event structure semantics.

as part of a larger research programme aimed at extending the chain of coreflections defined for Petri nets [20,24,18] first to contextual nets [3] (using *asymmetric event structures*) and then to graph grammars (exploiting some analogy between these two models). The price to pay was the introduction of much more sophisticated event structures. For DPO grammars the adjunction between unfolding and event structures breaks down to a functorial construction in just one direction. A recent result [4] has shown that the missing link can be re-established when considering the Single Pushout (SPO) approach [17,12]. This is summarized in Figure 1. The category of prime algebraic domains is equivalent to the category of prime event structures (PES), thus all constructions can ultimately lead to PES. It is worth noting that, under a mild assumption on graph grammars, namely *node preservation*, the elegant SPO constructions can be transferred also to the DPO approach. To some extent, the above informal discussion paves the way to the definition of an event structure semantics of π-calculus, (almost) obtained by applying the PES construction available for GTS to the encoding in [19].

We observe that the event structure semantics in this case are unnecessarily complicated (by the need of dealing with features pertaining to graph grammars but not needed in the encoding). Hence, we devise a simpler class of grammars, large enough to allow the encoding, but restricted as much as needed to obtain a PES via a chain of coreflections. Incidentally, the class we take is node preserving, and thus we can carry the construction under both the DPO and the SPO approach, still getting the same result. Our contribution aims to promote GTS as a suitable modeling framework for nominal calculi. The technique is demonstrated by addressing the case studies of π-calculus and join-calculus. We remark that this is the first event structure semantics for the latter, whose synchronization pattern challenges the reuse of other techniques in the literature.

Structure of the paper. Section 2 summarizes the basics of typed graph grammars under the DPO (§ 2.1) and SPO (§ 2.2) approaches, and their event structure semantics (§ 2.3). Section 3 defines the class of persistent grammars. Sections 4 and 5 illustrate, respectively, how to associate PES to π-calculus processes and join calculus processes. Related works and final remarks are in Section 6.

2 Typed Graph Grammars and True Concurrency

Given a partial function $f : A \rightarrowtail B$ its *domain* is $dom(f) = \{a \in A \mid f(a) \text{ is defined}\}$. For $f, g : A \rightarrowtail B$ partial functions, we write $f \subseteq g$ when $dom(f) \subseteq dom(g)$ and $f(x) = g(x)$ for all $x \in dom(f)$. When $dom(f) = A$ we say that f is *total* and write $f : A \to B$.

A *(directed, unlabeled) graph* is a tuple $G = \langle N_G, E_G, s_G, t_G \rangle$, where N_G is a set of *nodes* (or *vertices*), E_G is a set of *edges* (or *arcs*), and $s_G, t_G : E_G \to N_G$ are the *source* and *target* functions. We shall omit subscripts when obvious from the context.

A *partial graph morphism* $f : G \rightarrowtail G'$ is a couple $f = \langle f_N : N \rightarrowtail N', f_E : E \rightarrowtail E' \rangle$ such that: $s' \circ f_E \subseteq f_N \circ s$ and $t' \circ f_E \subseteq f_N \circ t$. It is *total* if both components are total. The inclusions ensure that *any* subgraph of a graph G can be the domain of a partial morphism $f : G \rightarrowtail H$. Instead, the stronger constraint $s' \circ f_E = f_N \circ s$ and $t' \circ f_E = f_N \circ t$ would require f to be defined over an edge if it is defined on its source or target nodes.

In *typed graph grammars* [9], graphs are typed over a structure that is itself a graph, i.e., the typing is a graph homomorphism. In this setting, category theory serves as a tool to characterize constructions in a succinct, elegant way, favoring flexibility and generality. Since category theory is mainly a *theory of morphisms*, structure / behavior preserving mappings play a key role. Given a *graph of types* T, a T*-typed graph* is a pair $\langle |G|, \tau_G \rangle$, where $|G|$ is the *underlying* graph and $\tau_G : |G| \to T$ is a total morphism.

In GTS the graph $|G|$ defines the (dynamically evolving) configuration of the system and its elements (nodes and edges) model resources, while τ_G defines the (static) *typing* of the resources. For example, when encoding Petri nets in GTS the places of the net form the (discrete) graph of types, while tokens form the configuration of the system.

A *partial (resp. total) morphism* between T-typed graphs $f : G_1 \rightarrowtail G_2$ is a partial (resp. total) graph morphism $f : |G_1| \rightarrowtail |G_2|$ consistent with the typing, i.e., such that $\tau_{G_1} \supseteq \tau_{G_2} \circ f$. We denote by T-**PGraph** the category of T-typed graphs and partial morphisms and by T-**Graph** its subcategory of total morphisms. Focusing on total morphisms, the DPO approach is based T-**Graph**, whereas the SPO approach exploits T-**PGraph**. Since in this paper we work only with typed notions, we will usually omit the qualification "typed", and we will not indicate explicitly the typing morphisms.

In GTS the key notion to *glue* graphs together is that of a categorical pushout. Roughly, a pushout pastes two graphs together by injecting them in a larger graph that is (isomorphic to) their disjoint union modulo the collapsing of some common part. We recall that a *span* is a pair (b, c) of morphisms $b : A \to B$ and $c : A \to C$. A *pushout* of the span (b, c) is then an object D together with two (co-final) morphisms $f : B \to D$ and $g : C \to D$ such that: (i) $f \circ b = g \circ c$ and (ii) for any other choice of $f' : B \to D'$ and $g' : C \to D'$ s.t. $f' \circ b = g' \circ c$ there is a unique $d : D \to D'$ s.t. $f' = d \circ f$ and $g' = d \circ g$. If the pushout is defined, then c and g is called the *pushout complement* of $\langle b, f \rangle$.

2.1 DPO Direct Derivations

A *(T-typed) DPO production* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is a span of injective typed graph morphisms $l : K \to L$ and $r : K \to R$. The T-typed graphs L, K, and R are called the *left-hand side*, the *interface*, and the *right-hand side* of the production, respectively. The production is called *consuming* if the morphism $l : K \to L$ is not surjective.

$$
\begin{tikzcd}
p: L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "h"] \\
G & D \arrow[l, "b"] \arrow[r, "d"'] & H
\end{tikzcd}
$$

(a) A DPO direct derivation.

$$
\begin{tikzcd}
L \arrow[r, tail, "q"] \arrow[d, "g"'] & R \arrow[d, tail, "h"] \\
G \arrow[r, tail, "d"'] & H
\end{tikzcd}
$$

(b) An SPO direct derivation.

Fig. 2. Graph grammar derivations.

Definition 2.1 (DPO graph grammar). *A* (T-typed) DPO graph grammar $\mathcal{G}$ *is a tuple* $\langle T, G_{in}, P\rangle$*, where* G_{in} *is the* initial (T-typed) graph *and* P *is a set of DPO* productions.

Given a graph G, a production $p : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, and a *match* (i.e., a total graph morphism) $g : L \to G$, a *direct derivation* δ *from* G *to* H *using* p *(based on* g*)* exists, written $\delta : G \Rightarrow_p H$, if and only if the diagram in Figure 2(a) can be constructed, where both squares are pushouts in T-**Graph**: (1) the rewriting step removes from the graph G the items $g(L - l(K))$ (images of the left-hand side but not of the interface), yielding the graph D (with k, b as a pushout complement of $\langle g, l\rangle$); (2) then, fresh copies of the items in the right-hand side R that are not in the image of the interface, namely $R - r(K)$, are added to D yielding H (as a pushout of (k, r)). The interface K specifies both what is preserved and how fresh items must be glued to the existing part.

The existence of the pushout complement of $\langle g, l\rangle$ is subject to the satisfaction of the following *gluing conditions* [10]:

- *identification condition:* $\forall x, y \in L$ if $x \neq y$ and $g(x) = g(y)$ then $x, y \in l(K)$ (note however that the match can be non-injective on preserved items: the same resource can be used with multiplicity greater than one if preserved by the derivation);
- *dangling condition:* no arc in $G - g(L)$ is attached to a node in $g(L - l(K))$ (otherwise the derivation would leave such arc dangling after the removal of the node).

The identification condition is satisfied by the so-called *valid matches*: a match is not valid if it requires an item to be consumed twice, or to be both deleted and preserved.

2.2 SPO Direct Derivations

A *(T-typed) SPO production* is an injective partial graph morphism $q : L \rightarrowtail R$. It is called *consuming* if the morphism is not total. Without loss of generality, we will assume that q is just the partial inclusion $L \cap R \subseteq R$. The typed graphs L and R are called the *left-hand side* and the *right-hand side* of the production, respectively.

Definition 2.2 (SPO graph grammar). *A* (T-typed) SPO graph grammar $\mathcal{G}$ *is a tuple* $\langle T, G_{in}, Q\rangle$*, where* G_{in} *is the* initial (T-typed) graph *and* Q *is a set of* productions.

Given a graph G and a *match* $g : L \rightarrow G$, there is a *direct derivation* δ *from* G *to* H *using* q *(based on* g*)*, written $\delta : G \Rightarrow_q H$, if the diagram in Figure 2(b) forms a pushout square in T-**PGraph**. Roughly, the rewriting step removes from the graph G the image of the items of the left-hand side that are not in the domain of q, namely $g(L-R)$, and it adds the items of the right-hand side that are not in the image of q, namely $R-L$. The items in the image of $dom(q) = L \cap R$ are preserved by the rewriting step.

The key difference w.r.t. the DPO approach is that in SPO there is no *dangling condition* preventing a rule to be applied. In fact, as T-**PGraph** is the base category, when a node is deleted by the application of a rule, then all the edges having such node as source or target are deleted by the rewriting step, as a kind of *side-effect*.

On the contrary, the *identification condition* and the notion of *valid match* are still required to hold for the correct application of a production.

In the special case of *node preserving* grammars, the effect of SPO and DPO is very close. An SPO grammar is node preserving if each production $q : L \rightarrowtail R$ defines a total map on nodes. Similarly, a DPO grammar is node preserving if in each production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ the functions l and r are surjective on nodes. Then there is an isomorphism between SPO and DPO node preserving grammars that maps each production $q : L \rightarrowtail R$ to $D(q) : (L \xleftarrow{l} dom(q) \xrightarrow{r} R)$, with l and r the obvious inclusions.

2.3 Unfolding Constructions and Event Structure Semantics

A DPO/SPO *derivation* $\rho = \{G_{i-1} \Rightarrow_{q_{i-1}} G_i\}_{i \in \{1,\ldots,n\}}$ in $\mathcal{G}$ is a sequence of direct derivations, with $G_0 = G_{in}$. A derivation is *valid* if it involves only valid matches.

We will consider only *consuming* graph grammars and *valid* derivations. The restriction to consuming grammars is essential to obtain a meaningful semantics combining concurrency and nondeterminism. In fact, the presence of non-consuming productions, which can be applied without deleting any item, would lead to an unbounded number of concurrent events with the same causal history. This would not fit with the approach to concurrency (see, e.g., [16,24]) where events in computations are identified with their causal history (formally, the unfolding construction would not work properly). This corresponds, in the theory of Petri nets, to the common requirement that transitions must have non-empty preconditions. The requirement about valid derivations is needed to have a computational interpretation that is resource-conscious, i.e., where a resource can be consumed only once.

To equip graph grammars with event structure semantics, by analogy with Petri nets, the idea is to first unfold all graph grammar derivations into the same "space of computations", collecting all items that can ever be produced and relating them to the applicable direct derivations. Then, we can project such unfolding so to keep just the events and the causality $\prec$, concurrency **co** and conflict # relationship between them.

In the case of Petri nets, the unfolding can be represented as a special kind of acyclic net, called occurrence net, whose places model all the tokens that can ever be produced and whose transitions model all the possible firings (events). For example, two events requesting the same token are in conflict, while an event is causally dependent on those events that generated the tokens it fetches and two events can be concurrently executed if they are neither causally dependent nor in conflict. The event structure is then obtained by keeping the events and forgetting the tokens. The appropriateness of the construction is supported by categorical arguments: (1) the maps from Petri nets to their unfoldings and from unfoldings to event structures are functors, i.e. they preserve net morphisms; (2) it is possible to go backward, in the sense of deriving a standard occurrence net from each event structure, and a standard net from each occurrence net (actually itself); (3) the backward maps are again functors; (4) forward and backward maps form a particularly nice kind of adjunctions, called co-reflections, which are the categorical means to relate different domains in the best possible way (formally, the unit of the adjunction is a natural isomorphism establishing an equivalence between a full subcategory of the domain of computational models and the denotational domain).

The case of DPO grammars is complicated by the fact that derivations can introduce subtle dependencies between events. Here the "tokens" are both the nodes and the arcs of the graph, hence it is possible: (1) to access the same resource concurrently, in read-only modality so to speak; (2) to have *asymmetric conflicts* between a direct derivation that attempts to read a resource and one that wants to fetch it; (3) to have events that by attempting to remove a node are inhibited by the presence of edges connected to that node. The consequences are that: (1) it is still possible to unfold DPO grammars in special acyclic DPO occurrence grammars accounting for all the above features; (2) a more complicated notion of event structure is needed, called *inhibitor event structures*; (3) the constructions are still functorial, but there is no fully satisfactory way back from inhibitor event structures to occurrence graph grammars.

The case of SPO is still more sophisticated than Petri nets, but more satisfactory than DPO. In fact: (1) it is possible to unfold SPO graph grammars in special acyclic SPO occurrence grammars; (2) a more sophisticated notion of event structure is needed, called *asymmetric event structures*, which can account for multiple concurrent readings and asymmetric conflicts; (3) all the constructions are coreflections.

Notably, for the special case of node preserving grammars the DPO construction can be carried on in close analogy with SPO, yielding the same asymmetric event structures. We do not have enough space here to formalize the above discussion, but details are not needed to follow the rest of this paper. Interested readers can check [4] for technicalities.

An important point to mention is that all the above constructions work only for a special kind of grammars, called *semi-weighted* and inspired by a similar requirements on Petri nets. Roughly, semi-weighted grammars enforce disambiguation in the semantics by preventing the generation of "equivalent" resources carrying the same history. We recall that a typed graph G is called *injective* if the typing morphism τ_G is injective.

Definition 2.3 (Semi-Weighted Grammar). *A graph grammar* $\mathcal{G} = \langle T, G_{in}, P\rangle$ *is* semi-weighted *if* G_{in} *is injective and the target of every production* $p \in P$ *is injective.*

3 Persistent Graph Grammars

In this section we revisit the general theory developed for SPO and DPO approaches when considering a special kind of graph grammars, called *persistent*. Sections 4 and 5 show that such restriction is a reasonable enough compromise between the applicability of the approach to nominal calculi and the categorical adequacy of the semantics.

A type graph T is *persistent* if its edges are partitioned in two subsets: E_T^+ of persistent edges and E_T^- of removable edges. Given a persistent T, and a T-typed graph G, we denote by E_G^+ and E_G^- the set of edges mapped respectively to persistent edges and to removable edges of T. In the following assume a persistent type graph T is given.

Definition 3.1 (Persistent Productions). *A DPO production* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ *is* persistent *if all of the following hold:*

- node persistence: $N_L = l(N_K)$ *(i.e., all nodes in L are images of nodes in K);*
- removal of removable arcs: $E_K^- = \emptyset$ *(i.e., no removable arc is in K);*
- preservation of persistent arcs: $E_L^+ = l(E_K^+)$ *(i.e., all persistent arcs in L are images of persistent arcs in K).*

Similarly, an SPO production $q : L \rightarrowtail R$ *is* persistent *if:*

- node persistence: $N_L \subseteq N_R$ *(i.e., all nodes in L are also in R);*
- removal of removable arcs: $E_L^- \cap E_R = \emptyset$ *(i.e., no removable arc in L is preserved);*
- preservation of persistent arcs: $E_L^+ \subseteq E_R^+$ *(i.e., all persistent arcs in L are in R).*

Definition 3.2 (Persistent graph grammar). *A* (T-typed, DPO/SPO) graph grammar $\mathcal{G}$ *is* persistent *if all its productions are consuming, semi-weighted and persistent.*

We have already analyzed and discussed the requirements about the grammar being consuming and semi-weighted. A first motivation for the persistence requirement is the fact that it characterizes a whole class of grammars for which there is no need of checking the dangling arc condition when applying any direct derivation.

Lemma 3.1. *Given any (T-typed) graph G, any persistent DPO production* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$*, and any valid match* $g : L \to G$*, the dangling arc condition is trivially satisfied.*

Lemma 3.2. *Given any (T-typed) graph G, any persistent SPO production* $q : L \rightarrowtail R$*, and any valid match* $g : L \to G$*, no side-effect is produced on G.*

The proofs of the above lemmas exploit just node persistence. A remarkable consequence of the above properties is that in the unfolding construction we can completely disregard the precedences between productions induced by the dangling arc condition.

A second motivation is that there is no resource that can be both read and consumed during a derivation: nodes and persistent arcs can be just produced once and then read; removable arcs can be produced (once) and removed (once) but never read. A remarkable consequence of this property is that the event structure associated to the unfolding does not impose inhibitor conditions between events.

Theorem 3.1. *The construction of the prime event structure associated to a persistent graph grammar is expressed by the chain of coreflections in Figure 3.*

The isomorphism between node preserving SPO grammars and node preserving DPO grammars (see end of § 2.2) makes the result independent from the approach, in the sense that the PES $\mathcal{E}_p(\mathcal{U}_p(\mathcal{G}))$ associated to a persistent SPO grammar $\mathcal{G}$ is isomorphic to the one associated with the corresponding persistent DPO grammar $D(\mathcal{G})$.

$$\begin{tikzcd} \textbf{Persistent Graph Grammars} \arrow[r, shift left, "\mathcal{U}_p"'] \arrow[r, phantom, "\perp"] & \textbf{Persistent Occurrence Grammars} \arrow[l, hook', shift left] \arrow[r, shift right, "\mathcal{E}_p"'] \arrow[r, phantom, "\perp"] & \textbf{Prime Event Structures} \arrow[l, shift right, "\mathcal{N}_p"'] \arrow[r, shift right, "\mathcal{L}"'] \arrow[r, phantom, "\sim"] & \textbf{Domains} \arrow[l, shift right, "\mathcal{P}"'] \end{tikzcd}$$

Fig. 3. True concurrent semantics of persistent graph grammars.

$$P ::= 0 \mid \overline{x}\langle y\rangle \mid x(y).P \mid !x(y).P \mid (\nu x)P \mid P|P$$

(a) Syntax

$$\begin{array}{lll} P \mid 0 \equiv P & P \mid Q \equiv P \mid Q & (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\ P \equiv Q \ \mathit{if}\ P \equiv_\alpha Q & (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & (\nu x)P \mid Q \equiv (\nu x)(P|Q) \ \mathit{if}\ x \notin fn(Q) \end{array}$$

(b) Structural equivalence

$$\begin{array}{llll} (\text{SYNC}) & \overline{x}\langle y\rangle \mid x(z).P \to P\{^y/_z\} & (!\text{SYNC}) & \overline{x}\langle y\rangle \mid !x(z).P \to P\{^y/_z\} \mid !x(z).P \\ (\text{PAR}) & P \to P' \Rightarrow P|Q \to P'|Q & (\text{RES}) & P \to P' \Rightarrow (\nu x)P \to (\nu x)P' \end{array}$$

(c) Reduction Semantics

Fig. 4. Syntax and reduction Semantics of the asynchronous *pi*-calculus

The proof of the main result (for SPO) is carried on along the lines of [4], but it is omitted because of space limitation. We are confident that the case studies in Sections 4 and 5 can be understood without looking at the details of our constructions. Proofs will be included in the full version of this work. We just remark that the unfolding functor ($\mathcal{U}_p$) and the event structure ($\mathcal{E}_p$) are just the restrictions to the domain of persistent graph grammars of the functors already designed for (node preserving) graph grammars ($\mathcal{U}_s$ and $\mathcal{E}_s$). Moreover, asymmetric conflicts are due to ordinary mutual exclusion arguments (and causality) and thus the event structure associated to the unfolding is morally a PES (in disguise). Hence, the only adjunct that must be redefined is the one associating a (persistent) occurrence graph grammar to a PES, as otherwise the adjoint functor $\mathcal{N}_s$ would generate a node preserving, but non persistent occurrence graph grammar.

4 Event Structure Semantics for the π-calculus

In this section we show the encoding of asynchronous π processes as persistent graph grammars and the construction of their event structure.

Given an infinite set of names $\mathcal{N}$ ranged over by $a, b, x, y, z, \ldots$, the *asynchronous* π processes over $\mathcal{N}$ are defined by the grammar in Figure 4(a). The reduction semantics is the least relation satisfying the rules in Figure 4(c) (modulo the structural congruence rules in Figure 4(b)). Free and bound names (written $fn(P)$ and $bn(P)$) are defined as usual. A process P is a *sequential agent* if it is either $\overline{x}\langle y\rangle$, $x(y).P$ or $!x(y).P'$.

For simplicity, we represent π processes as *hypergraphs* instead of graphs, like in [19]. A hyperarc can be connected to several nodes. Hence, any hyperarc has an ordered set of attachment points, which is represented by a sequence. As usual, $|s|$ stands for the length of the sequence s, and $s[i]$ for $0 < i \leq |s|$ refers to the ith element of s.

Definition 4.1 (Hypergraph). *A* (hyper)graph *is a triple $H = (N_H, E_H, \phi_H)$, where N_H is the set of nodes, E_H is the set of edges, and $\phi_H : E_H \to N_H^*$ describes the connections of the graph. We call $|\phi_H(e)|$ the* rank *of e and assume that $|\phi_H(e)| > 0$ for any e.*

Note that every hypergraph H can be straightforwardly encoded as a graph G, whose nodes are the nodes and arcs of H, and whose arcs connect the nodes corresponding to edges with the original nodes of H. Formally, $G = (N_H \cup E_H, E, s, t)$, where $E = \bigcup_{e\in E_H}\{e_1,\ldots,e_{|e|}\}$, $s(e_i) = e$ and $t(e_i) = \phi_H(e)[i]$ for all $e_i \in E$.

A process P corresponds to a hypergraph $H = (N_H, E_H, \phi_H)$, where nodes stand for the names used by P, and hyperarcs denote sequential agents of P. Given $e \in E_H$ denoting a sequential agent S, the definition of ϕ_H attaches e to the nodes corresponding to the free names $fn(S)$ of S. In particular, $|\phi(e)|$ is equal to the number of occurrences of free variables in S, and $\phi(e)[i] = n_x$ if n_x is the node associated to the variable x and the ith occurrence of a free name in S corresponds to x (we assume free names in S to be ordered in some fixed form, e.g., from left to right).

For simplicity, and w.l.o.g., we will consider a canonical form for processes in which all bound variables are different from each other. The canonical form of P is $can(P) = P'$ where $\{[P]\}_1 = P', n$ and $\{[\_]\}_n : P \to P \times \mathbb{N}$ is defined s.t. $\{[P]\}_n = P', n'$ iff P' is obtained by renaming (from the left to the right) all bound variables of P with consecutive natural numbers in the range $[n, n'-1]$. Moreover, we assume $fn(P) \cap \mathbb{N} = \emptyset$.

$$
\begin{array}{rll}
\{[0]\}_n = 0, n & & \\
\{[\overline{x}\langle y\rangle]\}_n = \overline{x}\langle y\rangle, n & & \\
\{[x(y).P]\}_n = x(n).P', n' & where \{[P\{^n/_y\}]\}_{n+1} = P', n' \\
\{[(\nu x)P]\}_n = (\nu n)P', n' & where \{[P\{^n/_x\}]\}_{n+1} = P', n' \\
\{[P_1|P_2]\}_n = P_1' \mid P_2', n' & where \{[P_1]\}_n = P_1', n'' \; and \; \{[P_2]\}_{n''} = P_2', n' \\
\{[!P]\}_n = !P', n' & where \{[P]\}_n = P', n'
\end{array}
$$

We associate a type $\lfloor S \rfloor$ to any sequential agent S, defined as follows:

$$
\begin{array}{rl}
\lfloor \overline{x}\langle y\rangle \rfloor = \overline{x}\langle \bullet \rangle & \\
\lfloor x(n).P \rfloor = x(n).(P\{^\bullet/_{x_1}, \ldots, ^\bullet/_{x_n}\}) & where \; \{x_1,\ldots,x_n\} = fn(P)\backslash\{n\} \\
\lfloor !x(n).P \rfloor = !x(n).(P\{^\bullet/_{x_1}, \ldots, ^\bullet/_{x_n}\}) & where \; \{x_1,\ldots,x_n\} = fn(P)\backslash\{n\}
\end{array}
$$

The special mark $\bullet$ denotes an occurrence of a free variable. We say $n+1$ the rank of $\lfloor S \rfloor$ with n the number of occurrences of $\bullet$ in $\lfloor S \rfloor$.

Example 4.1. Consider the following process $P = x(z).(\overline{z}\langle y\rangle \mid \overline{z}\langle y\rangle) \mid \overline{x}\langle y\rangle \mid \overline{x}\langle x\rangle$. Then $can(P) = x(1).(\overline{1}\langle y\rangle \mid \overline{1}\langle y\rangle) \mid \overline{x}\langle y\rangle \mid \overline{x}\langle x\rangle$. Moreover, the types of all sequential agents in P are $x(1).(\overline{1}\langle \bullet\rangle \mid \overline{1}\langle \bullet\rangle)$ and $\overline{x}\langle \bullet\rangle$.

Differently from the encoding of [19], sequential agents differing on their first free name have different types. For instance, here $P_1 = \overline{x}\langle z\rangle$, $P_2 = \overline{x}\langle x\rangle$ and $P_3 = \overline{z}\langle z\rangle$ are typed $\lfloor P_1\rfloor = \lfloor P_2\rfloor = \overline{x}\langle\bullet\rangle \neq \lfloor P_3\rfloor = \overline{z}\langle\bullet\rangle$ (contrastingly to the original proposal that assigns the same type $\overline{\bullet}\langle\bullet\rangle$ to all of them). Our definition generates more productions, but it produces semi-weighted grammars in many more cases, as explained below.

Given a set A of agent types, the *type graph* associated with A is $T_A = \langle\{x\}, A, \phi_{T_A}\rangle$ s.t. for all t, $\phi_{T_A}(t) = s$, and $s[i] = x$ for $0 < i \leq |s| = rank(t)$. The set of T_A-typed hypergraphs we consider is the least set built using the following constants and operations.

- 0 is the empty graph $(\emptyset, \emptyset, \emptyset)$.
- x denotes the discrete graph $(\{x\}, \emptyset, \emptyset)$ containing the node x.
- $H_1 \oplus H_2 = (N_{H_1} \cup N_{H_2}, E_{H_1} + E_{H_2}, \phi)$ is the composition of H_1 and H_2 , where $+$ stands for the disjoint union of sets and ϕ is defined as follows

$$\phi(e) = \phi_{H_1}(e) \quad \textit{if} \;\; e \in E_{H_1}(e) \qquad\qquad \phi(e) = \phi_{H_2}(e) \quad \textit{if} \;\; e \in E_{H_2}(e)$$

- $H_1\{^y/x\}$ is the graph obtained by renaming the node x of H_1 by y, i.e., $H_1\{^y/x\} = ((N_{H_1} \setminus \{x\}) \cup \{y\}, E_{H_1}, \phi_H)$, where $\phi_H(e) = \phi_{H_1}(e)\{^y/_x\}$ for all $e \in E_{H_1}$.
- S s.t $\lfloor S\rfloor \in A$ is the graph whose nodes are the free names of S and its unique arc has type $\lfloor S\rfloor$, i.e., $H = (fn(S), \{\lfloor S\rfloor\}, \{\lfloor S\rfloor \mapsto s\})$, where $|s|$ is equal to the rank of $\lfloor S\rfloor$ and $s[i] = x$ if the ith occurrence of a free name in S is x.

In all cases, the typing morphisms map nodes to x and arcs to their type.

Remark 4.1. The graphs $H_1 = \overline{x}\langle y\rangle$ and $H_2 = \overline{k}\langle y\rangle\{^x/_k\}$ are different since they contain the same nodes $\{x, y\}$, but H_1 has a unique arc with type $\overline{x}\langle\bullet\rangle$, while the arc of H_2 has type $\overline{k}\langle\bullet\rangle$. Differently, $H_3 = \overline{x}\langle y\rangle$ and $H_4 = \overline{x}\langle k\rangle\{^x/_k\}$ are identical. In this case, we will use the first notation as an abbreviation for the second one.

The next definition provides the mapping from π processes to (hyper)graphs.

Definition 4.2. *Given a canonical π process P, its corresponding agent hypergraph is $H_P = unw(P)$, where unw is inductively defined as follows:*

$$\begin{array}{ccc} unw(0) = 0 & unw(\overline{x}\langle y\rangle) = \overline{x}\langle y\rangle & unw(x(y).P) = x(y).P \\ unw(!x(y).P) = !x(y).P & unw((\nu x)P) = unw(P) & unw(P_1|P_2) = unw(P_1) \oplus unw(P_2) \end{array}$$

Example 4.2. The agent hypergraph corresponding to the process P in Example 4.1 is $H_P = x(1).(\overline{1}\langle y\rangle \mid \overline{1}\langle y\rangle) \,\oplus\, \overline{x}\langle y\rangle \,\oplus\, \overline{x}\langle x\rangle$.

Then, the graph grammar corresponding to a particular process is defined as follows. We use $I \xrightarrow[C]{} O$ to denote rule patterns that can be instantiated by providing agent types. Any instance is a production $q : L \xleftarrow{l} K \xrightarrow{r} R$ with $K = C \oplus n(I)$, $L = I$ and $R = O \oplus K$, with $n(I)$ being the nodes of I, and where the morphisms are the obvious inclusions.

Definition 4.3 **(π process as a graph grammar).** *The graph grammar corresponding to a π process P is $\mathcal{G}_P = \langle T, G_{in}, Q\rangle$, where T contains the types of all possible subagents of P, $G_{in} = H_P = unw(P)$ and productions $q \in Q$ are obtained by instantiating the two* *patterns below (where $\overline{k_1}\langle\bullet\rangle$,$k_2(y).P$ and $k_2(y).P$ are types of T and z is a fresh name) with the types of the subagents of P.*

$$\text{GRAPH-SYNC}: \quad \overline{k_1}\langle z\rangle\{^x/_{k_1}\} \oplus k_2(y).P\{^x/_{k_2}\} \longrightarrow unw(P)\{^x/_{k_2}\}\{^z/_y\}$$

$$\text{GRAPH-!SYNC}: \quad \overline{k_1}\langle z\rangle\{^x/_{k_1}\} \xrightarrow[!k_2(y).P\{^x/_{k_2}\}]{} unw(P)\{^x/_{k_2}\}\{^z/_y\}$$

The rewriting rules do not specify the actual name over which the communication takes place, but just that the output and the input action take place over the same node.

Example 4.3. Consider P defined in Example 4.1. The corresponding graph grammar $\mathcal{G}_P = \langle T, G_{in}, Q\rangle$ is defined as follows:

- T contains a unique node, and its arcs correspond to the different types of all the sequential agents occurring in P, i.e., $E_T = \{\overline{1}\langle\bullet\rangle,\ \overline{x}\langle\bullet\rangle,\ x(1).(\overline{1}\langle\bullet\rangle \mid \overline{1}\langle\bullet\rangle)\}$.
- $G_{in} = H_P = x(1).(\overline{1}\langle y\rangle \mid \overline{1}\langle y\rangle) \oplus \overline{x}\langle y\rangle \oplus \overline{x}\langle x\rangle$;
- $Q = \{p_1, p_2\}$, where:

$$p_1:\ \overline{x}\langle z\rangle \oplus x(1).(\overline{1}\langle y\rangle \mid \overline{1}\langle y\rangle) \xleftarrow{l_1} x \oplus y \oplus z \xrightarrow{r_1} \overline{1}\langle y\rangle\{^z/_1\} \oplus \overline{1}\langle y\rangle\{^z/_1\} \oplus x$$

$$p_2:\ \overline{1}\langle z\rangle\{^x/_1\} \oplus x(1).(\overline{1}\langle y\rangle \mid \overline{1}\langle y\rangle) \xleftarrow{l_2} x \oplus y \oplus z \xrightarrow{r_2} \overline{1}\langle y\rangle\{^z/_1\} \oplus \overline{1}\langle y\rangle\{^z/_1\} \oplus x$$

with l_1, l_2, r_1 and r_2 being inclusion morphisms.

By applying rule p_1, we can derive $G_{in} \Rightarrow_{p_1} \overline{1}\langle y\rangle\{^y/_1\} \oplus \overline{1}\langle y\rangle\{^y/_1\} \oplus \overline{x}\langle x\rangle$.

The evolution of any process P is described by a finite rewriting system (since the set of sequential agents contained in P is finite). Moreover, graph productions are persistent since all nodes are persistent and removable arcs do not appear in contexts. Nevertheless, the grammar may not be semi-weighted. In fact, the initial graph G_{in} in Example 4.3 is not injective (it contains two arcs with the same type $\overline{x}\langle\bullet\rangle$). Similarly, the targets of both productions are not injective (they have two arcs with type $\overline{1}\langle\bullet\rangle$). In what follows, we restrict our analysis to semi-weighted processes, i.e., processes that produce semi-weighted grammars. Semi-weighted processes disambiguate the production of identical elements having the same history. Hence, P can be written as

$$P' = x(1).(\overline{1}\langle y\rangle \mid (\nu 2)(2(3).\overline{1}\langle y\rangle|\overline{2}\langle 2\rangle)) \mid \overline{x}\langle y\rangle \mid (\nu 4)(4(5).\overline{x}\langle y\rangle|\overline{4}\langle 4\rangle)$$

where the production of identical elements with the same history is avoided by introducing an internal reduction. Note that the initial graph

$$G'_{in} = x(1).(\overline{1}\langle y\rangle \mid (\nu 2)(2(3).\overline{1}\langle y\rangle|\overline{2}\langle 2\rangle)) \oplus \overline{x}\langle y\rangle \oplus 4(5).\overline{x}\langle y\rangle \oplus \overline{4}\langle 4\rangle$$

is injective (arcs of graphs $\overline{x}\langle y\rangle$ and $\overline{4}\langle 4\rangle$ have different types). Similarly, the target of associated transitions are injective, and hence, the associated grammar is semi-weighted.

Remark 4.2. PGGs do not imply a severe limitation for encoding the π-calculus since (i) node and arc persistency have no influence, (ii) consuming rules have no effect when following a reduction approach, (iii) although semi-weighted rules prevent us from encoding processes having tokens with identical causal history, it is possible to encode any process as a PGGs by disambiguating identical tokens (for instance, by introducing internal reductions).

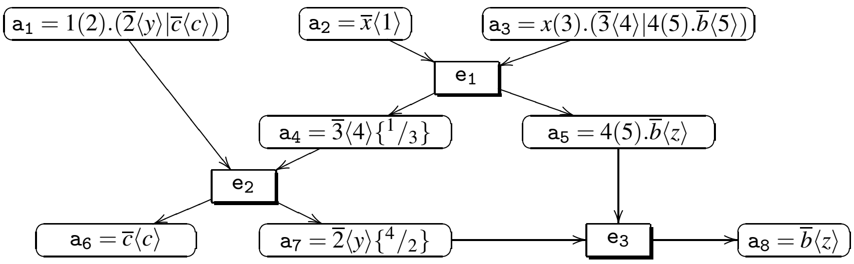

Fig. 5. Unfolding of the grammar corresponding to the π process P

Example 4.4 (Event Structure). Consider the following asynchronous π process corresponding to the encoding of the synchronous process $\overline{x}\langle y\rangle.\overline{c}\langle c\rangle.0|x(z).\overline{b}\langle z\rangle.0$.

$$P = (\nu k)(\overline{x}\langle k\rangle|k(a).(\overline{a}\langle y\rangle|\overline{c}\langle c\rangle)) \mid x(k).(\nu a)(\overline{k}\langle a\rangle|a(z).\overline{b}\langle z\rangle)$$

After obtaining the canonical form of P by renaming all bound names by natural numbers, the initial graph of the corresponding grammar is

$$G_{in} = \overline{x}\langle 1\rangle \oplus 1(2).(\overline{2}\langle y\rangle|\overline{c}\langle c\rangle)) \oplus x(3).(\overline{3}\langle 4\rangle|4(5).\overline{b}\langle 5\rangle)$$

The set of productions is obtained by instantiating the pattern rule for GRAPH-SYNC with all possible agent types: $\overline{x}\langle\bullet\rangle$, $1(2).(\overline{2}\langle\bullet\rangle|\overline{\bullet}\langle\bullet\rangle)$, $\overline{2}\langle\bullet\rangle$, $\overline{c}\langle\bullet\rangle$, $x(3).(\overline{3}\langle\bullet\rangle|\bullet(5).\overline{\bullet}\langle 5\rangle)$, $\overline{3}\langle\bullet\rangle$, $4(5).\overline{\bullet}\langle 5\rangle$, $\overline{b}\langle\bullet\rangle\}$. For instance, one possible instantiation is:

$$p_1 : \overline{x}\langle y\rangle \oplus (1(2).(\overline{2}\langle v\rangle|\overline{w}\langle z\rangle))\{^x/_1\} \xleftarrow{l_1} x \oplus y \oplus v \oplus w \oplus z \xrightarrow{r_1} \overline{2}\langle v\rangle\{^y/_2\} \oplus \overline{w}\langle z\rangle \oplus x$$

The corresponding unfolding is in Figure 5. We use a net-like pictorial representation, where productions are shadow-shaped boxes connected to the consumed and produced resources by incoming and outcoming arrows respectively. For the sake of clarity, we omit the representation of graph nodes (i.e., names x and y) and edge attachments, since nodes are preserved by productions and they do not introduce additional dependencies to those shown in Figure 5. The minimal elements of the unfolding, i.e., a_1, a_2 and a_3, are the elements of G_{in}. Any event stands for the application of a production on a set of concurrent events. Note that a_6 and a_8 causally depend on $\mathsf{a}_1, \mathsf{a}_2$ and a_3. In particular, the output in a_6 causally depends on the input action in a_3, even though a_1 and a_3 share no names.

5 Event Structure Semantics for join-calculus

As done for the π calculus, we provide the event structure semantics of join processes through its mapping to persistent graph grammars. For simplicity, we focus on *core (recursive) join-calculus* [13], but our approach smoothly extends to full join.

$$P = x\langle u\rangle \mid P|P \mid \mathbf{def}\, x\langle u\rangle|y\langle v\rangle \triangleright P \,\mathbf{in}\, P$$

(a) Syntax

$$P \mid Q \equiv P\,,\,Q$$
$$\mathbf{def}\, x\langle u\rangle|y\langle v\rangle \triangleright P_1 \,\mathbf{in}\, P_2 \equiv (x\langle u\rangle|y\langle v\rangle \triangleright P_1)\sigma\,,\,P_2\sigma$$
$$(\sigma \textit{ renames } x \textit{ and } y \textit{ with globally fresh names})$$
$$x\langle u\rangle|y\langle v\rangle \triangleright P,\; x\langle z_1\rangle,\; y\langle z_2\rangle \rightarrow x\langle u\rangle|y\langle v\rangle \triangleright P,\; P\{^{z_1}/_x, ^{z_2}/_y\}$$

(b) Semantics

Fig. 6. Core Join

The syntax of core Join is in Figure 6(a). The occurrences of x and u in $x\langle u\rangle$ are *free*, while x and y occur bound in $P = \mathbf{def}\, x\langle u\rangle|y\langle v\rangle \triangleright P_1 \,\mathbf{in}\, P_2$, and u and v occur bound in $x\langle u\rangle|y\langle v\rangle \triangleright P_1$. Free and bound names of P are written respectively $fn(P)$ and $bn(P)$.

The semantics of the join calculus relies on the *reflexive chemical abstract machine* model [13]. In this model a solution is a multiset of active definitions and processes (separated by comma). New definitions may become active dynamically. Moves are distinguished between *structural* $\equiv$, which heat or cool processes, and reductions $\rightarrow$, which are the basic computational steps (disjoint reductions can be executed in parallel). The rewriting rules are shown in Figure 6(b).

As done for π processes, we only consider canonical processes. The canonical form of P is $can(P) = P'$ for $\{[P]\}_1 = P', n$ and

$$\begin{aligned}
\{[x\langle u\rangle]\}_n &= \overline{x}\langle u\rangle, n\\
\{[P_1|P_1]\}_n &= P_1' \mid P_2', n' \; \textit{where} \{[P_1]\}_n = P_1', n'' \textit{ and } \{[P_2]\}_{n''} = P_2', n'\\
\{[\mathbf{def}\, x\langle u\rangle|y\langle v\rangle \triangleright P_1 \,\mathbf{in}\, P_2]\}_n &= \mathbf{def}\, n\langle n+1\rangle|n+2\langle n+3\rangle \triangleright P_1' \,\mathbf{in}\, P_2', n'\\
&\quad \textit{where } \{[P_1\{^n/_x, ^{n+1}/_u, ^{n+2}/_y, ^{n+3}/_v\}]\}_{n+4} = P_1', n''\\
&\quad \textit{and } \{[P_2\{^n/_x, ^{n+2}/_y\}]\}_{n''} = P_2', n'
\end{aligned}$$

By analogy with π, we consider subterms $x\langle u\rangle$ and $x\langle u\rangle|y\langle v\rangle \triangleright P$ as sequential agents. Then, the type $\lfloor S\rfloor$ of a sequential agent S is defined as follows:

$$\begin{aligned}
\lfloor \overline{x}\langle y\rangle\rfloor &= \overline{x}\langle \bullet\rangle\\
\lfloor x\langle u\rangle|y\langle v\rangle \triangleright P\rfloor &= x\langle u\rangle|y\langle v\rangle \triangleright (P\{^\bullet/_{x_1},\ldots,^\bullet/_{x_n}\}) \textit{ where } \{x_1,\ldots,x_n\} = fn(P)\setminus\{u,v\}
\end{aligned}$$

The mapping of processes to graph grammar is defined below.

Definition 5.1. *Given a canonical join process P, its corresponding hypergraph is $H_P = unw(P)$, where unw is inductively defined as follows:*

$$\begin{aligned}
unw(x\langle u\rangle) &= x\langle u\rangle\\
unw(\mathbf{def}\, x\langle u\rangle|y\langle v\rangle \triangleright P_1 \,\mathbf{in}\, P_2) &= x\langle u\rangle|y\langle v\rangle \triangleright P_1 \oplus unw(P_2)\\
unw(P_1|P_2) &= unw(P_1) \oplus unw(P_2)
\end{aligned}$$

Definition 5.2 (Join process as a graph grammar). *The graph grammar $\mathcal{G}_P$ corresponding to the join process P is $\mathcal{G}_P = \langle T, G_{in}, Q\rangle$, where T contains the types of all the subagents of P, $G_{in} = H_P = unw(P)$ and productions $q \in Q$ are obtained by instantiating the following pattern with the types in T.*

$$\overline{k_1}\langle u\rangle\{^x/_{k_1}\}\oplus\overline{k_2}\langle v\rangle\{^y/_{k_2}\}\xrightarrow[(k_3\langle u_1\rangle|k_4\langle v_1\rangle\triangleright P)\{^x/_{k_3},^y/_{k_4}\}]{} unw(P)\{^x/_{k_3},^y/_{k_4}\}\{^u/_{u_1},^v/_{v_1}\}$$

Example 5.1. Consider the canonical join process $P = \textbf{def}\ D\ \textbf{in}\ 1\langle 3\rangle \mid 3\langle 1\rangle$, with $D = 1\langle 2\rangle|3\langle 4\rangle\triangleright 4\langle 3\rangle|3\langle 1\rangle$. Then, the initial graph of the grammar is

$$G_{in} = H_P = D\oplus 1\langle 3\rangle\oplus 3\langle 1\rangle$$

and the types $\{1\langle 2\rangle|3\langle 4\rangle\triangleright 4\langle\bullet\rangle|\bullet\langle\bullet\rangle,\ 1\langle\bullet\rangle,\ 3\langle\bullet\rangle,\ 4\langle\bullet\rangle\}$. Hence, we have nine possible rules p_{k_1,k_2}, one for any possible combination of $k_1,k_2\in 1,3,4$, defined as follows

$$D\{^x/_1,^y/_3\}\oplus\overline{k_1}\langle u\rangle\{^x/_{k_1}\}\oplus\overline{k_2}\langle v\rangle\{^y/_{k_2}\}\xleftarrow{l_1}D\{^x/_1,^y/_3\}\oplus u\oplus v$$
$$\xrightarrow{r_1}D\{^x/_1,^y/_3\}\oplus\overline{4}\langle y\rangle\{^v/_4\}\oplus\overline{3}\langle 1\rangle\{^x/_1,^y/_3\}\oplus u$$

Then, we have the following computation

$$G_{in}\ \Rightarrow_{p_{1,3}}\ D\oplus 4\langle 3\rangle\{^1/_4\}\oplus 3\langle 1\rangle\ \Rightarrow_{p_{4,3}}\ D\oplus 4\langle 3\rangle\{^1/_4\}\oplus 3\langle 1\rangle$$

The unfolding of $\mathcal{G}_P$ can be obtained as for π processes. In this case, the causal relation of the event structure is the total order $\mathtt{e}_1\prec\mathtt{e}_2,\ldots$, while the $\# = \emptyset$ and $\mathbf{co} = \emptyset$.

6 Related works and concluding remarks

We have introduced Persistent Graph Grammars (PGGs) as a convenient model for equipping nominal calculi with truly concurrent semantics. Our results collect, so to say, the best of two worlds: the event structure semantics is defined by a chain of coreflections as in [4] and the encoding of nominal calculi is rather direct as in [19]

We improve on [4] by restricting the format of productions so as to guarantee that there is no information loss when viewing the asymmetric event structure associated to the grammar as a PES. The restricted format considerably simplifies the construction w.r.t. fully general grammars. We improve on [19] by refining the type system so as to apply the unfolding construction to a much broader class of π-processes that comprises all those processes whose associated productions are semi-weighted. The generality of our technique is also supported by the original case study of join calculus.

Being a special case of semi-weighted grammars, PGGs enjoys the nice property of reconciling the SPO with the DPO approach. Note that two other event structure semantics proposed in the literature for DPO [8,22] coincide with the one obtained from the unfolding, which thus can be called *the* event structure semantics of GTS.

We exploit a “hierarchical” encoding of sequential processes, as opposed to the “flat” (DPO) encodings of [14,15], where finitely many productions encode all (finite) agents and where node fusion is requested in the right-hand side of some productions. The latter feature prevented the straightforward reuse of some techniques, although [5] develops a non-sequential semantics also in the presence of node fusion.

Due to space limitation we leave the comparison with previous non-interleaving semantics of the π-calculus [11,6] to the full version of this paper. The linearity constraint in [23] shares some similarities with the semi-weightedness criterion and it would be interesting to see if their type systems can be transferred to graph grammar productions.

Acknoweldgement. We warmfully thank Paolo Baldan for his guidance and support during the writing of the paper. We also thank the referees for their careful revisions.

References

1. P. Baldan. *Modelling concurrent computations: from contextual Petri nets to graph grammars.* PhD thesis, University of Pisa, 2000.
2. P. Baldan, A. Corradini, and B. König. Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. *Proc. CONCUR'04*, *LNCS* 3170, pp. 83–98. Springer, 2004.
3. P. Baldan, A. Corradini, and U. Montanari. Contextual Petri nets, asymmetric event structures and processes. *Inform. and Comput.*, 171(1):1–49, 2001.
4. P. Baldan, A. Corradini, U. Montanari, and L. Ribeiro. Concurrency and Nondeterminism in Graph Rewriting: From Graph Grammars to Asymmetric Event Structures and Backwards. *Technical Report CS-2005-2*, University Ca' Foscari of Venice, 2005.
5. P. Baldan, F. Gadducci, and U. Montanari. Concurrent Semantics for Graph Rewriting with Fusions. *Proc. CONCUR'06*. This volume.
6. M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the pi-calculus. *Acta Informatica*, 35(3):353–400, 1998.
7. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and its adjunctions with categories of derivations. *Proc. TAGT'94*, *LNCS* 1073, pp. 56–74. Springer, 1996.
8. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, F. Rossi. An event structure semantics for graph grammars with parallel productions. *Proc. TAGT'94*, *LNCS* 1073, pp.240–256, 1996.
9. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fund. Inf.*, 26:241–265, 1996.
10. 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 [21].
11. P. Degano and C. Priami. Non-interleaving semantics for mobile processes. *Theoret. Comput. Sci.*, 216(1-2):237–270, 1999.
12. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. Algebraic approaches to graph transformation II: SPO approach and comparison with DPO. In [21].
13. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. In *Proc. POPL'96*, pp. 372–385. ACM Press, 1996.
14. F. Gadducci. Term Graph Rewriting for the phi-Calculus. In *Proc. APLAS'03*, *LNCS* 2895, pp. 37–54. Springer, 2003.
15. F. Gadducci and U. Montanari. A Concurrent Graph Semantics for Mobile Ambients. In *Proc. MFPS'01*, *ENTCS* 45. Elsevier, 2001.
16. U. Golz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
17. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoret. Comput. Sci.*, 109:181–224, 1993.
18. J. Meseguer, U. Montanari, and V. Sassone. On the semantics of Place/Transition Petri nets. *Mathematical Structures in Computer Science*, 7:359–397, 1997.
19. U. Montanari and M. Pistore. Concurent semantics for the π-calculus. *ENTCS* 1. 1995.
20. M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoret. Comput. Sci.*, 13:85–108, 1981.
21. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.
22. G. Schied. On relating Rewriting Systems and Graph Grammars to Event Structures. *Proc. Graph Transformations in Computer Science*, *LNCS* 776, pp. 326–340. Springer, 1994.
23. D. Varacca and N. Yoshida. Typed event Structures and the π-calculus. In *Proc. MFPS'06*.
24. G. Winskel. Event Structures. *Advances in Petri Nets'86*, *LNCS* 255, pp. 325–392. 1987.