Proceedings of the
Workshop on Petri Nets and Graph Transformation
(PNGT 2006)

Event Structure Semantics for Dynamic Graph Grammars

Roberto Bruni, Hernán Melgratti, and Ugo Montanari

19 pages

Event Structure Semantics for Dynamic Graph Grammars

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.

Abstract: Dynamic graph grammars (DGGs) are a reflexive extension of Graph Grammars that have been introduced to represent mobile reflexive systems and calculi at a convenient level of abstraction. Persistent graph grammars (PGGs) are a class of graph grammars that admits a fully satisfactory concurrent semantics thanks to the fact that all so-called asymmetric conflicts (between items that are read by some productions and consumed by other) are avoided. In this paper we introduce a slight variant of DGGs, called *persistent dynamic graph grammars* (PDGGs), that can be encoded in PGGs preserving concurrency. Finally, PDGGs are exploited to define a concurrent semantics for the Join calculus enriched with persistent messaging (if preferred, the latter can be naively seen as dynamic nets with read arcs).

Keywords: Dynamic graph grammars, Event structures, Join calculus

1 Introduction

Petri nets and Graph Transformation Systems (GTSs) are two well-known models for concurrent systems. Petri nets, conceptually simpler, had became a reference model for experimenting with and developing new semantic approaches to concurrency, notably nonsequential processes, unfolding constructions, event structures and algebraic models. GTSs provide a more sophisticated and expressive framework for handling resource types, distribution, access control.

The close analogy between the two models neatly emerges when viewing Petri nets as particular GTSs over discrete typed graphs: places form the nodes of the type graph, markings are typed graphs (whose nodes model tokens) and transitions are productions. This correspondence has facilitated the mutual transfer of concepts and techniques, like the development of process and unfolding semantics for GTSs obtained in [Bal00] by extending the chain of coreflections defined for Petri nets [NPW81] to graph grammars. However, general Double Pushout (DPO) grammars require a much more sophisticated notion of events, called *inhibitor event structures* and the adjunction between unfolding and event structures breaks down to a functorial construction $\mathcal{E}_g$ in just one direction. A recent result [BCMR07] re-established the missing link for the Single Pushout (SPO) approach, by defining an adjunction $\mathcal{N}_s \dashv \mathcal{E}_s$ between the category of semi-weighted SPO occurrence grammars and the category of the so-called *asymmetric event structures*. The category of prime algebraic domains is equivalent to the category of prime event structures (PES), thus all constructions can ultimately lead to PES, that offers a reference denotational model of true concurrency. This is summarized in the three upper rows of Figure 1.

From the point of view of concurrency, there are two interesting extensions of Petri nets that have been studied separately: (1) read arcs and (2) mobility. The first allow for multiple, concurrent read-accesses to resources, which is a very common situation in e.g. databases transactions,
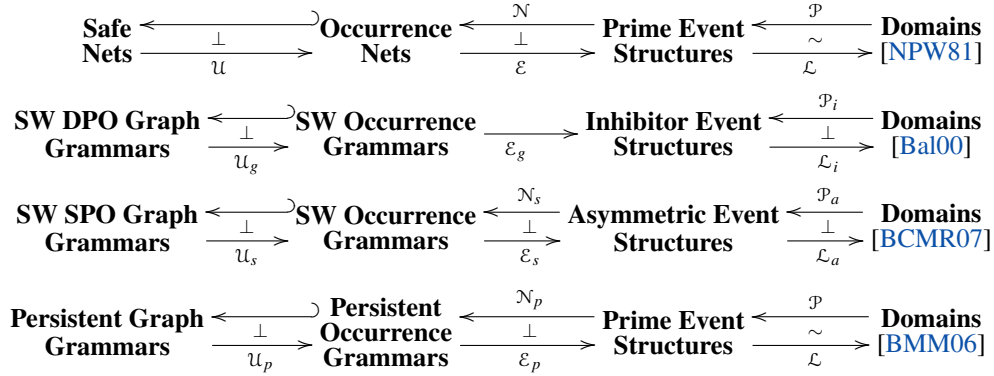
Figure 1: A recollection of event structure semantics.

concurrent constraint programming, and asynchronous systems. The second accounts for dynamic changes in the topology of the net, like deploying new places and transitions, that can occur in highly dynamic environments like global computing. In the literature, Petri net semantics have been extended to cope with read arcs, while the mobile extension has not received yet a fully satisfactory assessment (some preliminary investigation is in [BM06b]).

Exploiting the analogy between nets and GTSs, here we first introduce *persistent dynamic graph grammars* (PDGGs) as a suitable generalization of mobile nets with read arcs by combining *dynamic graph grammars* (DGGs) [BM06a] and *persistent productions* [BMM06]. Then we are able to equip PDGGs with a nice event structure semantics by exploiting: (i) the encoding of DGG in ordinary graph grammars given in [BM06a] and (ii) the coreflective event structure semantics for *persistent graph grammars* (PGGs) given in [BMM06]). In DGGs, the application of a production can release fresh types and fresh productions. The encoding in [BM06a] enriches configurations with additional run-time typing information for the dynamically generated structure. We slightly refine it so to map PDGGs to PGGs and then show that it preserves concurrency. The main theorem of [BMM06] has shown that the construction of the prime event structure associated to a PGG is expressed by a chain of coreflections (see the bottom row in Figure 1), and since PGGs are node preserving, the constructions under DPO and SPO approaches coincide. By combining all results, we obtain a concurrent semantics for PDGGs.

As a case study we present a version of the Join calculus with persistent messages, which emerges as a rather elegant way of combining nets, read arcs, and mobility in a process calculus. In fact, it is well-known that dynamic nets are in one-to-one correspondence with processes of the Join calculus [BS01], and we propose persistent messages as a disciplined mean to introduce reading primitives for multiple concurrent accesses to resources.

Structure of the paper: Section 2 recalls some important preliminaries about persistent graph grammars, and dynamic graph grammars. Section 3 introduces PDGGs and their (concurrency-preserving) encoding in PGGs. Section 4 describes PJoin and equips it with a concurrent semantics via encoding PJoin processes in PDGGs. Some concluding remarks are in Section 5.

$$p: \quad L \xleftarrow{\quad l \quad} K \xrightarrow{\quad r \quad} R$$

$$\left. \begin{array}{ccc} m \downarrow & (1) & k \downarrow & (2) & \downarrow h \\ G \xleftarrow{\quad b \quad} & D & \xrightarrow{\quad d \quad} H \end{array} \right.$$

Figure 2: A DPO direct derivation.

2 Background

2.1 Persistent Graph Grammars

A *(directed) graph* is a tuple $G = \langle N_G, E_G, s_G, t_G \rangle$, where N_G is a set of *nodes*, E_G is a set of *edges* (or *arcs*), and $s_G, t_G : E_G \rightarrow N_G$ are the *source* and *target* functions. We shall omit subscripts when obvious from the context. A *graph morphism* $f : G \rightarrow G'$ is a couple $f = \langle f_N : N \rightarrow N', f_E : E \rightarrow E' \rangle$ such that: $s' \circ f_E = f_N \circ s$ and $t' \circ f_E = f_N \circ t$.

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 $\tau_G : |G| \rightarrow T$ is a total morphism. A *morphism* between T-typed graphs $f : G_1 \rightarrow G_2$ is a graph morphisms $f : |G_1| \rightarrow |G_2|$ such that $\tau_{G_1} = \tau_{G_2} \circ f$. The category of T-typed graphs and their morphisms is denoted by T-**Graph**. Since we work only with typed notions, we will usually omit the qualification "typed", and we will not indicate explicitly the typing morphisms.

In GGs the graph $|G|$ defines the configuration of the system and its items (nodes and edges) model resources, while τ_G defines the *typing* of the resources. Hence, the underlying graph $|G|$ evolves dynamically, while the type graph T is statically fixed and cannot change at run-time.

The key notion to *glue* graphs together is that of a categorical pushout. Roughly, a pushout pastes two graphs 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 $b : A \rightarrow B$ and $c : A \rightarrow C$. A *pushout* of the span (b, c) is then an object D together with two (co-final) morphisms $f : B \rightarrow D$ and $g : C \rightarrow D$ such that: (i) $f \circ b = g \circ c$ and (ii) for any other choice of $f' : B \rightarrow D'$ and $g' : C \rightarrow D'$ s.t. $f' \circ b = g' \circ c$ there is a unique $d : D \rightarrow D'$ s.t. $f' = d \circ f$ and $g' = d \circ g$. If the pushout is defined, then c and g form the *pushout complement* of $\langle b, f \rangle$.

Definition 1 (DPO graph grammar) A *(T-typed graph) DPO production* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is a span of injective graph morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$. 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 $l : K \rightarrow L$ is not surjective.

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*, P is a set of DPO *productions*.

Given a graph G, a production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, and a *match* $m : L \rightarrow G$, a *direct derivation* δ *from G to H using p (based on m)* exists, written $\delta : G \Rightarrow_p H$, if and only if the diagram in Figure 2 can be constructed, where both squares are pushouts in T-**Graph**: (1) the rewriting step removes from the graph G the items $m(L - l(K))$, yielding the graph D (with k, b as a pushout complement of $\langle m, l \rangle$); (2) then, fresh copies of the items in $R - r(K)$ 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 $\langle m, l \rangle$ is subject to the satisfaction of the following *gluing conditions* [CMR$^+$97]:

- *identification condition:* $\forall x, y \in L$ if $x \neq y$ and $m(x) = m(y)$ then $x, y \in l(K)$;

- *dangling condition:* no arc in $G \setminus m(L)$ should be incident to a node in $m(L \setminus l(K))$.

The identification condition is satisfied by *valid matches*: a match is not valid if it requires a single item to be consumed twice, or to be both consumed and preserved.

A *derivation* is a sequence $\gamma = \{ \delta_i : G_{i-1} \Rightarrow_{p_{i-1}} G_i \}_{i \in \{1, \ldots, n\}}$ of direct derivations.

In what follows we will focus on semi-weighted GGs that enforce disambiguation in the semantics by preventing the generation of "equivalent" resources carrying the same history. A graph grammar $\mathcal{G} = \langle T, G_{in}, P \rangle$ is *semi-weighted* if G_{in} is injectively typed and the target of every production $p \in P$ is injective.

In [BMM06], we have introduced a particular class of Graph Grammars, called *Persistent Graph Grammars* (PGGs), which allow us to obtain a PES via a chain of coreflections. The remaining of this section is devoted to summarise PGGs and their event structure 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. In the following we assume a persistent type graph T is given. Given 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.

Definition 2 (Persistent graph grammar)   A production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ is *persistent* if:

- *node persistence:* $N_L = l(N_K)$ (i.e., nodes are never deleted);

- *removal of removable arcs:* $E_K^- = \emptyset$ (i.e., all matched removable arcs are deleted);

- *preservation of persistent arcs:* $E_L^+ = l(E_K^+)$ (i.e., all persistent arcs are preserved).

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

Theorem 1   (cfr. [BMM06]) *The construction of the prime event structure associated to a persistent graph grammar is expressed by the chain of coreflections in the bottom row of Figure 1.*

Remark 1   *The Single Pushout approach (SPO) offers an alternative to DPO. A (T-typed) SPO production is an injective partial graph morphism $q : L \rightarrowtail R$ (without loss of generality, assume that q is just the partial inclusion $L \cap R \subseteq R$). The key difference w.r.t. the DPO approach is that in SPO there is no dangling condition preventing a rule to be applied: when a node is deleted by the application of a rule, then all dangling edges are also deleted as some kind of side-effect. However, in the special case of node preserving rules, the effect of SPO and DPO is very close: there is an isomorphism between SPO and DPO node preserving grammars that maps each production $q : L \rightarrowtail R$ to $p_q : (L \hookleftarrow dom(q) \hookrightarrow R)$. The isomorphism makes the PES construction independent from the approach [BMM06].*

Remark 2   *For PGGs, the identification condition reduces to require the matching to be injective over removable arcs.*

2.2 Dynamic Graph Grammars

In this section we describe *Dynamic Graph Grammars* (DGGs) and their encoding into graph grammars as presented in [BM06a]. DGGs are graph grammars that dynamically modifies their graph of types and their set of productions. A T-typed dynamic production takes the form: $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T'})$ where $\mathcal{G}_{T'}$ is a suitable (T'-typed) dynamic graph grammar. A dynamic graph grammar can contain any number of such productions. In order to provide the formal definition we need the following notion of graph retyping.

Definition 3 (Graph Retyping)   Given a T-typed graph $G = \langle |G|, \tau_G \rangle$ and a morphism $\sigma : T \to T'$ we denote by $\sigma \cdot G$ the T'-typed graph $\sigma \cdot G = \langle |G|, \sigma \circ \tau_G \rangle$.

The set of DGGs is formally defined as follows:

Definition 4 (Dynamic Graph Grammars)   The domain of dynamic graph grammars can be expressed as the least set DGG satisfying the equation:

$$\text{DGG} = \{ (T, G_{in}, P) \mid G_{in} \in \mathbf{Graph}_T \land$$
$$\forall p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p}) \in P. (L_T, K_T \in \mathbf{Graph}_T \land T \subset T_p \land \mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p) \in \text{DGG}) \}$$

where $\mathbf{Graph}_T$ is the set of T-typed graphs and $r : \iota \cdot K_T \to G_{T_p}$ is a morphism between T_p-typed graphs, where $\iota : T \hookrightarrow T_p$ denotes the obvious sub-graph injection.

Any element $\mathcal{G} = (T, G_{in}, P) \in \text{DGG}$ is called a *dynamic graph grammar*.

When $P = \emptyset$, then the grammar is roughly a T-typed graph, which is statically fixed and cannot change. Such grammars are called *static*. A production p is *static* if its right-hand side $\mathcal{G}_{T_p}$ is a static grammar and $T_p = T$. If all productions are static, then the grammar is called *shallow* and is essentially an ordinary GGs: the application of any production can neither change the type graph nor spawn new rules.

Example 1   *Figures 3 and 4 introduce a small dynamic graph grammar. Let T_a be the singleton type graph with just one node a. Let $T_g \supset T_a$ consisting of nodes a and b and two edges $f : a \to b$, and $g : b \to b$.*

Take the T_a-typed dynamic grammar $\mathcal{G}_a$ with the dynamic production p in Figure 3. For simplicity, we take the inclusions as legs of the span and draw the typing (dotted lines) only once for each item. The left-hand side of p consists of a T_a-typed graph with just one node n_1, which is preserved by the context, and the right-hand side spawns a shallow T_g-typed grammar $\mathcal{G}_p$ whose initial graph has, beside n_1, one additional node n_2 and one arc h. The grammar $\mathcal{G}_p$ itself has just one static production $q \in P_p$, illustrated in Figure 4: the left-hand side is a graph with three nodes m_0 and m_1, m_2, which are all preserved, and two arcs h_1, h_2, which are deleted, and the right-hand side spawns the static T_g-typed grammar $\mathcal{G}_q$ (i.e., a graph) with one additional arc l from m_1 to m_2 and type g.
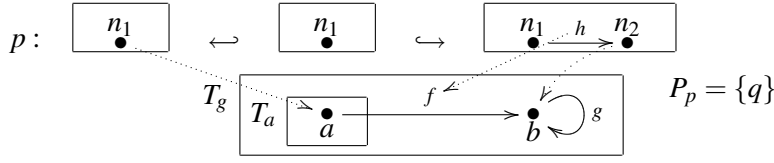
Figure 3: A dynamic production p.
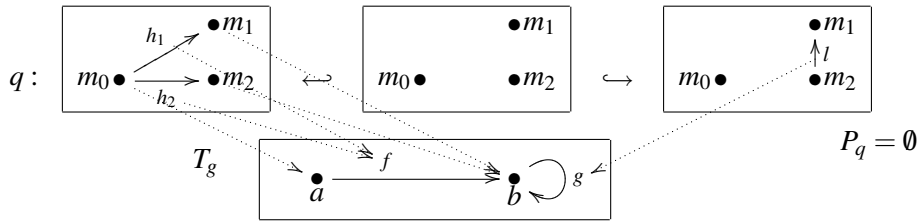
Figure 4: A static production q.

The dynamics of DGGs relies on a notion of retyping, which can be used to generate fresh items in the type graph. In the following, when considering type graph constructions, we assume that a standard choice of pushout objects satisfying the following requirements is available: Let $T \subset T''$ and $\sigma : T \to T'$ injective, then we denote by $T_{\sigma,T''}$ the pushout object of the inclusion $\iota : T \hookrightarrow T''$ and σ such that T' embeds in $T_{\sigma,T''}$ via set-theoretical inclusion $\iota_{\sigma,T''}$, while T'' embeds via an injection $\rho_{\sigma,T''}$ (see Figure 5(a)) that renames items in $T'' \setminus T$ with fresh names. When σ is the inclusion $T \subseteq T'$ we replace the subscripting $(-)_{\sigma,T''}$ with $(-)_{T',T''}$.

Definition 5 (Fresh Graph Retyping)   Let $T \subset T''$. Given a T''-typed graph $G = \langle |G|, \tau_G \rangle$ and an injection $\sigma : T \to T'$ we let $\sigma \cdot G = \langle |G|, \rho_{\sigma,T''} \circ \tau_G \rangle$.

Definition 6 (Dynamic Retyping)   Given a T-typed DGG $\mathcal{G} = (T, G_{in}, P)$ and an injective morphism $\sigma : T \to T'$ we denote by $\sigma \cdot \mathcal{G}$ the T'-typed grammar defined recursively by letting $\sigma \cdot \mathcal{G} = (T', \sigma \cdot G_{in}, \sigma \cdot P)$, with $\sigma \cdot P = \{\sigma \cdot p \mid p \in P\}$ where $\sigma \cdot p : (\sigma \cdot L_T \xleftarrow{l} \sigma \cdot K_T \xrightarrow{r} \rho_{\sigma,T_p} \cdot \mathcal{G}_{T_p})$ for any $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p})$. (Note that $\rho_{\sigma,T_p} \cdot \mathcal{G}_{T_p}$ is a T_{σ,T_p}-typed grammar and $\sigma \cdot p$ is a T'-typed production.)

In DGGs, productions are nested inside (the right-hand sides of) other productions, but only top-level productions can be applied, by finding a matching of their left-hand sides into the initial graph. When such a production p is applied, then fresh instances of the productions P_p, nested one level below p, become available at the top-level, and can be unwound themselves in successive steps. Given a DGG $\mathcal{G} = (T, G_{in}, P)$, a production $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p}) \in P$ with $\mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p)$, and a matching $m : L_T \to G_{in}$ that satisfies the gluing conditions, we proceed as follows (see Figure 5(b)):

$$\mathcal{G}'_p = \rho_{T,T_p} \cdot \mathcal{G}_{T_p} = (T_{T,T_p}, G'_p, P')$$
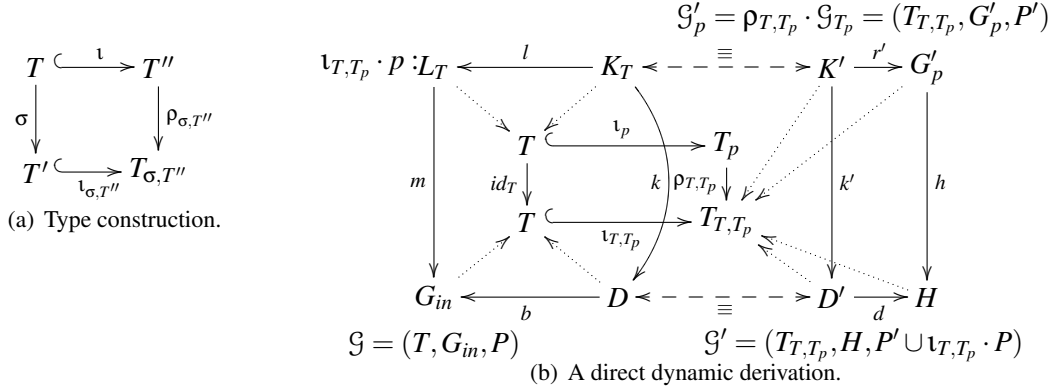

(a) Type construction.

(b) A direct dynamic derivation.

Figure 5: The dynamics of DGGs

- We build the pushout complement of $\langle m, l \rangle$, obtaining a T-typed graph D with morphisms $k : K_T \to D$ and $b : D \to G_{in}$.

- We build the standard type graph T_{T,T_p} associated with $\sigma = id_T : T \to T$ and $\iota_p : T \hookrightarrow T_p$. Note that $\iota_{T,T_p} = \rho_{T,T_p} \circ \iota_p$. Fresh items of the underlying graph produced by the application of p must be typed over T_{T,T_p}.

- We build the retyped graphs $D' = \iota_{T,T_p} \cdot D$ and $K' = \iota_{T,T_p} \cdot K_T$ and take the morphism $k' : K' \to D'$ induced by k.

- We build the retyped DGG $\mathcal{G}'_p = \rho_{T,T_p} \cdot \mathcal{G}_{T_p} = (T_{T,T_p}, G'_p, P')$ with $G'_p = \rho_{T,T_p} \cdot G_{T_p}$ and $P' = \rho_{T,T_p} \cdot P_p$ and take the morphism $r' : K' \to G'_p$ induced by $r : \iota_p \cdot K_T \to G_{T_p}$.

- The pushout of k' and r' gives us a T_{T,T_p}-typed graph H with $d : D' \to H$ and $h : G'_p \to H$.

- Finally, we build the DGG $\mathcal{G}' = (T_{T,T_p}, H, P' \cup \iota_{T,T_p} \cdot P)$.

When all the above is applicable, we say that there is a *direct dynamic derivation* α from $\mathcal{G}$ to $\mathcal{G}'$ using p (based on m) and write $\alpha : \mathcal{G} \Rightarrow_p \mathcal{G}'$. A *dynamic derivation* is a sequence of direct dynamic derivations starting from the initial graph.

Example 2   *For the DGG introduced in Example 1, assume the initial graph of $\mathcal{G}_a$ is a discrete graph G_0 with one node k typed over a. The application of the production p (with the obvious matching from n_1 to k) spawns a fresh instance $\mathcal{G}_p^1$ of $\mathcal{G}_p$: the type graph becomes T_{g_1} and the underlying graph becomes $G_1 \supset G_0$ with nodes k (typed a) and k_1 (typed b_1) connected by an arc h_1 (typed f_1). Moreover, a production q_1 is now available beside p. A second application of p spawns another fresh instance $\mathcal{G}_p^2$ of $\mathcal{G}_p$: the type graph becomes $T_{g_1} \cup T_{g_2}$ and the underlying graph becomes $G_2 \supset G_1$ with a new node k_2 and an arc $h_2 : k \to k_2$. Again, a production q_2 is now available. Similarly, p can be applied again and again (see Figure 6). However, no suitable matching can ever be found for the application of q_1, q_2, etc. In fact, it is not possible to find two arcs with the same type, say f_i, and the identification condition prevents a non-injective matching of the two arcs in the left-hand side of q_i.*
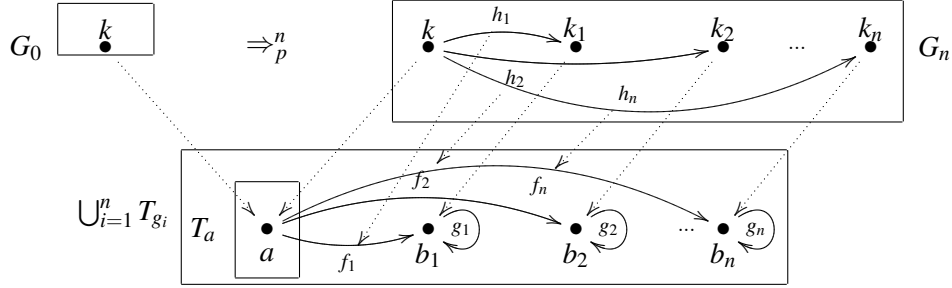
Figure 6: A derivation where p is applied n times.

Note that the type graphs syntactically appearing in $\mathcal{G} = (T, G_{in}, P) \in$ DGG form a finite tree $\mathbb{T}(\mathcal{G})$ rooted in T, with parent relation given by immediate subsetting (i.e., T_i is parent of T_j iff $T_i \subset T_j$ and no T_k appears in $\mathcal{G}$ such that $T_i \subset T_k \subset T_j$) and where leaves are associated with static grammars. We remark that each type graph $T_p \supset T$ extends T with local declarations $T_p \setminus T$, whose scope is bounded by the specific production p. For simplicity, but without loss of generality, we assume that all additional items introduced by different type graphs inside $\mathcal{G}$ are named differently.

Definition 7 (Flat Type Graph)   We let $\mathbf{T}(\mathcal{G}) = \bigcup_{T_i \in \mathbb{T}(\mathcal{G})} T_i$ denote the *overall flat type graph* of $\mathcal{G}$, and let $\iota_{T_i} : T_i \hookrightarrow \mathbf{T}(\mathcal{G})$ denote the obvious sub-graph inclusion. Note that, by the structuring of $\mathbb{T}(\mathcal{G})$, the type graph $\mathbf{T}(\mathcal{G})$ is just the union of all the leaves of $\mathbb{T}(\mathcal{G})$.

Similarly, all nested productions in $\mathcal{G}$ form the tree $\mathbb{P}(\mathcal{G})$ rooted in P with parent relation given by immediate inclusion (i.e., the set of productions P_i is the parent of P_p iff $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} (T_p, G_{T_p}, P_p)) \in P_i)$.

Definition 8 (Flattening)   Given a T-typed dynamic production $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p})$ with $\mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p)$ we say that the ordinary $\mathbf{T}(\mathcal{G})$-typed production $\mathsf{flat}(p) : (\iota_T \cdot L_T \xleftarrow{l} \iota_T \cdot K_T \xrightarrow{r} \iota_{T_p} \cdot G_{T_p})$ is the *flattening of p*. We let $\mathbf{P}(\mathcal{G}) = \bigcup_{P_i \in \mathbb{P}(\mathcal{G})} \{\mathsf{flat}(p) \mid p \in P_i\}$ denote the *overall set of flat productions* of $\mathcal{G}$. The $\mathbf{T}(\mathcal{G})$-typed shallow grammar $\mathbf{F}(\mathcal{G}) = (\mathbf{T}(\mathcal{G}), \iota_T \cdot G_{in}, \mathbf{P}(\mathcal{G}))$ is called the *flattening of $\mathcal{G}$*.

2.2.1 Encoding DGG into graph grammars

In what follows we provide an overview of the encoding of DGG into GGs. A detailed presentation can be found in [BM06a].

The encoding of a DGG $\mathcal{G}$ relies on the definition of a unique type graph expressive enough for distinguishing all the types generated dynamically by $\mathcal{G}$. Such structure is called the *refined type graph* and it is intuitively described as follows. Consider a chain of types $\mathcal{T}$ describing the evolution of the type graph of $\mathcal{G}$, i.e., a list of graph ordered by inclusions, e.g., $T_1 \subset T_2 \subset \ldots \subset T_n$, then the evolution of the type graph is given by the refined version $\{[T_n]\}_{\mathcal{T}}$ of the last graph in the

(a) A T_g-typed Graph.   (b) Refined T_g-Typed Graph.
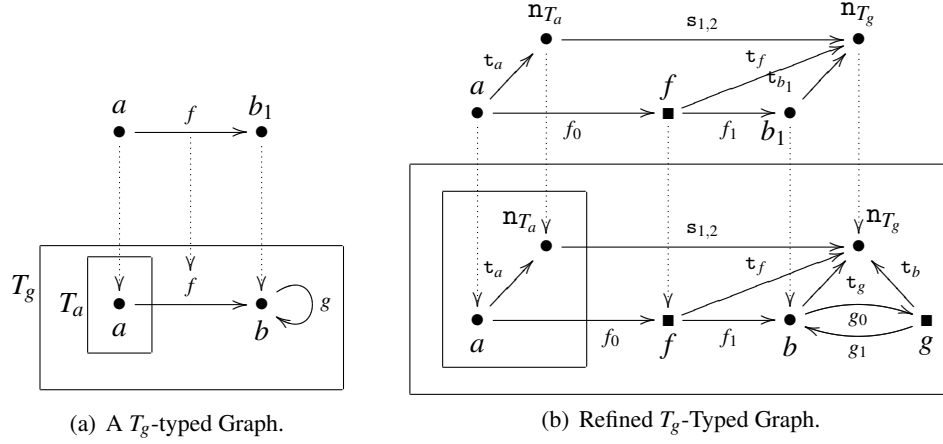
Figure 7: A refined T-Typed Graph.

list. Informally, any item (i.e., node or arc) of T_n is mapped to a node in $\{[T_n]\}_{\mathcal{T}}$. Every graph T_i in the chain $\mathcal{T}$ is also represented in $\{[T_n]\}_{\mathcal{T}}$ by a node $\mathbf{n}_{T_i}$. Moreover, any node w corresponding to an item k of T_n has an arc $\mathbf{t}_w$ to the node $\mathbf{n}_{T_i}$ if T_i is the minimal type in $\mathcal{T}$ that includes k. We call T_i the *type of k in $\mathcal{T}$*. Formally, for any $k \in T_n$, the type of k is $\mathcal{T}(k) = T_i$ if $k \in T_i \backslash T_{i-1}$.

Definition 9 (Refined type graph)   Given a type graph T and a chain of types $\mathcal{T} = T_1 \subset \ldots \subset T_n$, with $T_n = T$, the *refined type graph* is $\{[T]\}_{\mathcal{T}} = \langle N_R, E_R, s_R, t_R \rangle$, where:

- $N_R = N_T \cup E_T \cup \{\mathbf{n}_{T_i} | T_i \in \mathcal{T}\}$ where $\mathbf{n}_{T_i}$ are fresh names, i.e., the nodes of $\{[T]\}_{\mathcal{T}}$ are the nodes and arcs of T plus one extra node for any type in $\mathcal{T}$;

- $E_R = \{e_0, e_1 | e \in E_T\} \cup \{\mathbf{t}_w | w \in N_T \cup E_T\} \cup \{\mathbf{s}_{i,i+1} | 0 < i \leq n-1\}$ where all edge names are fresh. Source and target functions are defined s.t. the following holds:

  - $e_0 : s_T(e) \to e$ and $e_1 : e \to t_T(e)$, i.e., e_0 connects the node $e \in N_R$ to its original source in T while e_1 connects e to its target;

  - $\mathbf{t}_w : w \to \mathbf{n}_{\mathcal{T}(w)}$, i.e., $\mathbf{t}_w$ connects w to the node representing its type;

  - $\mathbf{s}_{i,i+1} : \mathbf{n}_{T_i} \to \mathbf{n}_{T_{i+1}}$ denotes the inclusion of types $T_i \subset T_{i+1}$.

Example 3   *Consider the type graph T_g depicted in the bottom part of the Figure 7(a). The refined type graph for the chain $T_a = \{\mathbf{a}\} \subset T_g = T_a \cup \{f, b, g\}$ is shown at the bottom of Figure 7(b). The original arc f (resp. g) of T_g is represented by the homonymous "squared" node f (resp. g) and the pair of fresh arcs f_0 and f_1 (resp. g_0 and g_1). The types T_a and T_g are represented by the fresh nodes $\mathbf{n}_{T_a}$ and $\mathbf{n}_{T_g}$, while the inclusion relation $T_a \subset T_g$ is denoted by the arc $\mathbf{s}_{1,2}$. Finally, for any item w, $\mathbf{t}_w$ connects w to its type node.*

Then, a T-typed graph is also mapped to a graph typed over the refined version of its type T.

Definition 10 (Refined T-Typed Graph)   Given a T-typed graph $G = \langle |G|, \tau_G \rangle$ and a chain $\mathcal{T} = T_1 \subset \ldots \subset T_n = T$, the $\{[T]\}_{\mathcal{T}}$-typed graph $\{[G]\}_{\mathcal{T}} = \langle |H|, \tau_H \rangle$ is defined as:

- $N_H = N_G \cup E_G \cup \{\mathtt{n}_{T_i} | T_i \in \mathcal{T}\}$, i.e., N_H has all items of G plus nodes denoting types;

- $E_H = \{e_0, e_1 | e \in E_G\} \cup \{\mathtt{t}_w | w \in N_G \cup E_G\} \cup \{\mathtt{s}_{i,i+1} | 0 < i < n-1\}$, where:

  - $e_0 : s_T(e) \to e$ and $e_1 : e \to t_T(e)$;
  - $\mathtt{t}_w : w \to \mathtt{n}_{\mathcal{T}(\tau_G(w))}$, i.e., $\mathtt{t}_w$ connects w to the node representing its type in $\mathcal{T}$, which is obtained by using the typing morphism $\tau_G(w)$;
  - $\mathtt{s}_{i,i+1} : \mathtt{n}_{T_i} \to \mathtt{n}_{T_{i+1}}$, for the inclusion of types.

- The typing morphism τ_H is defined as follows

$$
\begin{array}{llllllll}
\tau_H(k) & = & \tau_G(k) & \text{if } k \in G & \tau_H(\mathtt{n}_{T_i}) & = & \mathtt{n}_{T_i} & \\
\tau_H(e_i) & = & \tau_G(e)_i & i = 0,1 & \tau_H(\mathtt{t}_w) & = & \mathtt{t}_{\tau_G(w)} & \tau_H(\mathtt{s}_{i,i+1}) & = & \mathtt{s}_{i,i+1}
\end{array}
$$

the nodes $\mathtt{n}_{T_i}$ and the arcs $\mathtt{t}_w$ and $\mathtt{s}_{i,i+1}$ of a refined type graph (resp., a refined T-typed graph) are referred to as the *location of the type graph* (respectively, *location of the graph*).

Example 4 Consider the T_g-typed graph G shown in Figure 7(a). The refined version of G for the chain $T_a = \{\mathtt{a}\} \subset T_g = T_a \cup \{f, b, g\}$ is shown in Figure 7(b) (we omit the representation of the obvious typing of arcs). The type graph corresponds to the refined version of T_g explained in Example 3.

Definition 11 (Refined T-Typed DGG) Let $\mathcal{G} = (T, G_{in}, P)$ be a DGG, and $\mathcal{T} = T_1 \subset \ldots \subset T_n$, with $T_n = T$ be a chain of types. Then, the refined version of $\mathcal{G}$ is $\{[\mathcal{G}]\}_{\mathcal{T}} = (\{[T]\}_{\mathcal{T}}, \{[G_{in}]\}_{\mathcal{T}}, \{[P]\}_{\mathcal{T}})$, where $\{[P]\}_{\mathcal{T}} = \{\{[p]\}_{\mathcal{T}} | p \in P\}$ is obtained by encoding any production $p : (L \xleftarrow{l} K \xrightarrow{r} (T', G'_{in}, P'))$ in P as follows:

$$
\{[p]\}_{\mathcal{T}} : (\{[L]\}_{\mathcal{T}} \xleftarrow{l'} \{[K]\}_{\mathcal{T}} \xrightarrow{r'} \{[(T', G'_{in}, P')]\}_{\mathcal{T} \subset T'}) \tag{1}
$$

where morphisms l' and r' are the obvious extensions of l and r with the identity over the location of the graphs.

Example 5 Consider the production p in Figure 3. Its refined version is shown in Figure 8. Note that the type graphs correspond to the refined versions of the original type graphs, while the left-hand-side, the interface, and the right-hand-side correspond to the refined version of the original ones. In particular, note that the left-hand-side is typed over the refined version of T_a, while the right-hand-side grammar is typed over the refined version of T_g. Moreover, the new production $\{[q]\}_{T_a \subset T_g}$ created by the reduction correspond to the refined versions of the original one q (for clarity we do not draw the typing morphism, which is the obvious one).

The refined version of a grammar $\mathcal{G}$ recreates the overall flat type graph $\mathbf{T}(\mathcal{G})$ as static tree of types (see Definition 7). Any production $p : (L_T \leftarrow K_T \to \mathcal{G}_{T'})$ is encoded by considering the path $\mathcal{T}$ of $\mathbf{T}(\mathcal{G})$ leading from the root of $\mathbf{T}(\mathcal{G})$ to T. Moreover, since previous definitions can be straightforwardly extended to consider the whole tree instead of a path, we will use $\{[\_]\}_{\mathcal{T}}$ to denote also the refined versions obtained by considering the tree of types $\mathcal{T}$.

Definition 12 (Encoding) Let $\mathcal{G} = \langle T, G_{in}, P \rangle$ be a DGG. Then, the equivalent graph grammar $[\![\mathcal{G}]\!]$ is defined as $[\![\mathcal{G}]\!] = \mathbf{F}(\{[\mathcal{G}]\}_T)$, where $\mathbf{F}(\_)$ stands for flattening (see Definition 8).
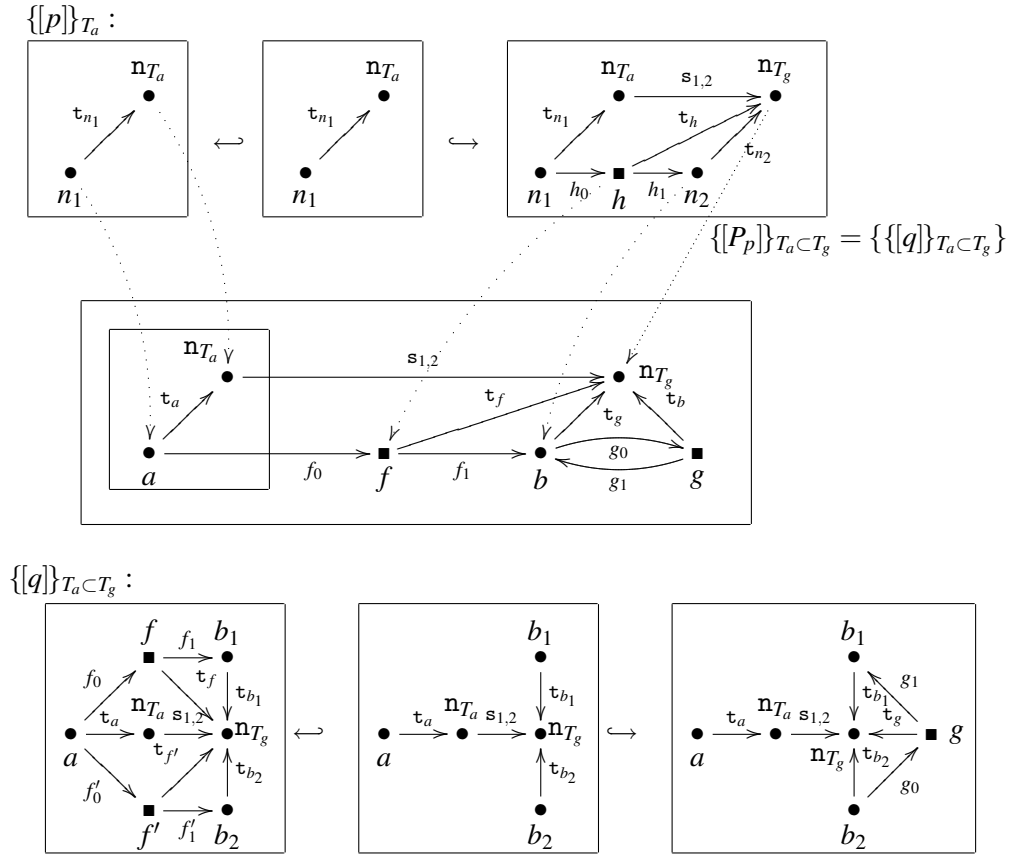
Figure 8: A refined production p.

3 Persistent Dynamic Graph Grammars

As done for GGs, we define the class of persistent DGGs by restricting the way in which resources are used. In particular, we assume that any type graph T is *persistent* if its edges are partitioned in two subsets: E_T^+ of persistent edges and E_T^- of removable edges. In the following assume any type graph T being persistent. Given 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.

Definition 13 (Persistent DGG)   A DGG $\mathcal{G}$ is *persistent* if all its productions are consuming, semi-weighted and persistent, where a production $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p})$ is *persistent* if all the following conditions hold:

- *node persistence:* $N_L = l(N_K)$;

- *removal of removable arcs:* $E_K^- = \emptyset$;

- *preservation of persistent arcs:* $E_L^+ = l(E_K^+)$;

- *dynamic persistency:* $\mathcal{G}_{T_p}$ is persistent.

Note that the three first conditions are the same as those in Definition 2. The fourth one requires persistency to hold recursively in the right-hand side of each production.

3.1 Encoding a PDGG into a PGG

In this section we show that by starting from a PDGG we can obtain a PGG, and hence, we can indirectly have the event structure semantics of PDGGs. The encoding is defined analogously to the original encoding of DGGs into GGs overviewed in Section 2.2.1. The only difference here is when encoding a production. Originally, a production $p : L \xleftarrow{l} K \xrightarrow{r} \mathcal{G}_{T_p}$ is encoded as in Equation (1). Note that for any edge e in L, the encoding generates one node e in $\{[L]\}_{\mathcal{T}}$, and three edges e_0, e_1 and $\mathtt{t}_e$. Nevertheless, when e is not in the image of l (that is, e is a removable arc), the node e in $\{[L]\}_{\mathcal{T}}$ is not preserved by $\{[p]\}_{\mathcal{T}}$. Therefore, we slightly modify the encoding of productions by requiring that for any removable arc e in L, the production $\{[p]\}_{\mathcal{T}}$ preserves the node encoding e to guarantee node persistency. Formally,

$$\{[p]\}_{\mathcal{T}} : (\{[L]\}_{\mathcal{T}} \xleftarrow{l'} \{[K]\}_{\mathcal{T}} \uplus G \xrightarrow{r'} \{[(T', G'_{in}, P')]\}_{\mathcal{T} \subset T'} \uplus G) \tag{2}$$

where G is the subgraph of $\{[L]\}_{\mathcal{T}}$ containing the nodes encoding the removable arcs of L (i.e., $e \in E_L^-$) and the nodes encoding the type of removable arcs (i.e., $\mathtt{n}_{\mathcal{T}(\tau_G(e))}$ for any $e \in E_L^-$). Moreover, we consider l' and r' as the identity over elements in G.

Lemma 1 $[\![\_]\!]$ *preserves behaviour, persistency, consumption and semi-weightedness.*

Proof. (Sketch)
Behaviour: The proof follows as for the original encoding of DGGs in GGs in [BM06a], by showing that any reduction step that applies a production p in the original grammar corresponds to a reduction step on the encoded grammar obtained by applying the encoded version of the original production p.
Persistency: Moreover, $\{[p]\}_{\mathcal{T}}$ is persistent when p is such because nodes are now persistent, and for edges we have that:

- Edges e_0, e_1, and $\mathtt{t}_e$ are removable if e is removable. Otherwise they are preserved.

- Type inclusion edges $s_{i,i+1}$ are preserved.

Semi-weighted: (i) Since the initial graph is injectively typed and the encoding does not add multiple elements with the same type, then the initial graph of the encoded grammar is injectively typed. (ii) Similarly, since productions are injective, the encoding produces injective productions.
Consumption: The proof follows by noting that the encoding preserves the name of edges, but the arcs connecting the name of an edge to its attachment points are actually consumed. Hence, if p is a rule of a PDGG, then it is consuming, and hence there is at least one deleted edge. Therefore the encoded form of p consumes at least two arcs. $\square$

Hence, this slightly variation of the encoding gives an equivalent PGG for any PDGG.

Lemma 2 *The encoding $[\![\_]\!]$ preserves concurrency.*

Proof. (Sketch) We prove that the encoding $[\![\_]\!]$ preserves the shift equivalence. This follows from the fact that for any two productions p and q that are sequentially independent (i.e., they are not in conflict) in a PDGG $\mathcal{G}$, then the corresponding rules in $[\![\mathcal{G}]\!]$ are also sequentially independent. This is proved by showing that the information added by the encoding of p and q corresponds to the location of the types and the names of edges, which will be always preserved by both generated rules. $\qquad\square$

4 Persistent Join

This section illustrates the use of PDGG for modelling a version of the Join calculus enriched with persistent messages, called *Persistent Join* (PJoin). Persistent messages are guaranteed not to be consumed after being produced [PSVV06], and hence are naturally useful for modelling situations like persistent, monotonic global stores such as in concurrent constraint programming. The syntax of PJoin relies on an infinite set $\mathcal{C}$ of names for consumable messages ranged over by $c, d, \ldots$, and a set $\mathcal{P}$ of names for persistent messages ranged over by $p, q, \ldots$. We write $x, y, z, \ldots$ to range over $\mathcal{C} \cup \mathcal{P}$. The syntax of PJoin is given by the grammar

$$P ::= 0 \mid x\langle y\rangle \mid \textbf{def } D \textbf{ in } P \mid P|Q \qquad D ::= J \triangleright P \mid D \wedge D \qquad J ::= x\langle y\rangle \mid J|J$$

A basic form of processes is an asynchronous emission of a message y on channel x, which is written $x\langle y\rangle$. The input is performed by local definitions $x_1\langle y_1\rangle \mid \ldots \mid x_n\langle y_n\rangle \triangleright P$ that is enabled when every channel x_i contains a message (in this case the names y_i denote the formal parameters of the input to be substituted in P with the actual content of the messages).

Example 6 *The following process illustrates the definition of a constant that can be read concurrently, and two concurrent accesses to it. We indicate persistent names by underlining their definitions.*

$$P = \textbf{def } read\langle k\rangle \mid \underline{val}\langle v\rangle \triangleright k\langle v\rangle \ \textbf{ in } \ val\langle c\rangle \mid read\langle k_1\rangle \mid read\langle k_2\rangle$$

The definition $read\langle k\rangle|\underline{val}\langle v\rangle \triangleright k\langle v\rangle$ introduces two private channels: read and val. The message $val\langle c\rangle$ to the persistent channel val sets the actual value c of the constant, while the messages $read\langle k_1\rangle$ and $read\langle k_2\rangle$ request the value of the constant.

The occurrences of x and y in the message $x\langle y\rangle$ are *free*. Differently, x and y occur bound in $P = \textbf{def } x\langle z\rangle|y\langle z'\rangle \triangleright P_1 \textbf{ in } P_2$ by the local definition $D = x\langle z\rangle|y\langle z'\rangle \triangleright P_1$. Of course, the parameters z and z' occur bound in D, but not in P_2. The sets of free and bound names of P are written respectively $fn(P)$ and $bn(P)$. Moreover, x and y are the defined names of D (written $dn(D)$). We denote with $cbn(P)$ and $pbn(P)$ the sets of consumable and persistent bound names of P. Similarly, $cdn(D)$ and $pdn(D)$ are the consumable and persistent defined names of D. We write $J_1|\underline{J_2}$ meaning that all defined names in J_1 are consumable while all defined names in J_2 are persistent.

The semantics of the PJoin is given with the *reflexive chemical abstract machine* model [FG96]. In this model a solution is roughly a multiset of active definitions $J \triangleright P$ and messages $x\langle u \rangle$ (separated by comma). Moves are distinguished between *structural* $\rightleftharpoons$, which heat or cool processes, and reductions $\rightarrow$, which are the basic computational steps. The multiset rewriting rules for PJoin are as follows:

$$
\begin{aligned}
0 &\rightleftharpoons \\
P \mid Q &\rightleftharpoons P, Q \\
D \wedge E &\rightleftharpoons D, E \\
\textbf{def } D \textbf{ in } P &\rightleftharpoons D\sigma_{dn(D)},\ P\sigma_{dn(D)} \qquad (range(\sigma_{dn(D)}) \text{ globally fresh}) \\
J_1 \mid \underline{J_2} \triangleright P,\ (J_1 \mid J_2)\sigma &\rightarrow J_1 \mid \underline{J_2} \triangleright P,\ J_2\sigma,\ P\sigma
\end{aligned}
$$

Structural moves are identical to the Join calculus, and allow for the rearrangement of terms inside a solution. In particular, a term denoting a process with local definitions can be represented by two terms (one for the definitions and other for the process) only when the locally defined ports are renamed by fresh names (this rule stands for the dynamic generation of new names). The only difference between Join and PJoin is in the unique reduction rule that can be applied when the solution contains a reaction $J_1 \mid \underline{J_2} \triangleright P$ and an instance $(J_1 \mid J_2)\sigma$ of the Join pattern $J_1 \mid \underline{J_2}$: when such a match is found, the consumable messages $J_1\sigma$ are removed from the solution, while persistent messages $J_2\sigma$ are kept. Moreover, a suitable instance of the guarded process $P\sigma$ is activated. We require σ to substitute persistent (resp. consumable) names by persistent (resp. consumable) names. We will write $P \mapsto P'$ for $P \rightleftharpoons^* Q \rightarrow Q' \rightleftharpoons^* P'$.

Example 7   *The process introduced in Example 6 behaves as follows*

$$
\begin{aligned}
P &= \textbf{def } read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle \textbf{ in } val\langle c \rangle \mid read\langle k_1 \rangle \mid read\langle k_2 \rangle \\
&\rightleftharpoons read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle ,\ val\langle c \rangle \mid read\langle k_1 \rangle \mid read\langle k_2 \rangle \\
&\rightleftharpoons^* read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle ,\ val\langle c \rangle ,\ read\langle k_1 \rangle ,\ read\langle k_2 \rangle
\end{aligned}
$$

At this point the system may reduce in two different ways. One alternative is the firing of the unique reaction $read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle$ *with the messages* $val\langle c \rangle$ *and* $read\langle k_1 \rangle$*, and by producing the message* $k_1\langle c \rangle$*, i.e.,*

$$P \mapsto read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle ,\ val\langle c \rangle ,\ k_1\langle c \rangle ,\ read\langle k_2 \rangle .$$

Note that $val\langle c \rangle$ *is preserved while* $read\langle k_1 \rangle$ *is consumed. Similarly, the system may evolve by firing the same reaction with the messages* $val\langle c \rangle$ *and* $read\langle k_2 \rangle$*. In this way,*

$$P \mapsto read\langle k \rangle \mid \underline{val}\langle v \rangle \triangleright k\langle v \rangle ,\ val\langle c \rangle ,\ k_2\langle c \rangle ,\ read\langle k_1 \rangle .$$

We remark that such reductions can be computed concurrently since the message $val\langle c \rangle$ *involved in both reductions is always preserved, i.e.,*
Differently, consider a variant of P in which $val\langle v \rangle$ *is a consumable channel.*

$$P' = \textbf{def } read\langle k \rangle \mid val\langle v \rangle \triangleright k\langle v \rangle \mid val\langle v \rangle \textbf{ in } val\langle c \rangle \mid read\langle k_1 \rangle \mid read\langle k_2 \rangle$$

In this case, the firing of the reaction $read\langle k \rangle \mid val\langle v \rangle \triangleright k\langle v \rangle \mid val\langle v \rangle$ *does not preserve the original message on the port val. Instead, it consumes the message and produces a new one. Hence, the accesses* $read\langle k_1 \rangle$ *and* $read\langle k_2 \rangle$ *cannot be served concurrently, but only in a serialised way.*

Example 8   *This example illustrates the dynamic features of* PJoin *which are exploited for creating constants. In particular, the following definition is used for creating fresh constants that can be read concurrently.*

$$D = new\langle w \rangle \vartriangleright \textbf{def } read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle \textbf{ in } val\langle c \rangle \,|\, w\langle read \rangle$$

The idea is that any message to the port new *will create a new constant with value* c*, which will be read by accessing a fresh port* read*. (For simplicity, we create all constants with the same value* c*, but it would be possible to write a slightly more complex process that allows to set an arbitrary initial value). Then, the following process*

$$Q = \textbf{def } D \textbf{ in } new\langle k_1 \rangle \,|\, new\langle k_2 \rangle$$

may be rewritten as follows

$$Q \quad \rightleftharpoons^* \quad D, new\langle k_1 \rangle, new\langle k_2 \rangle$$

By firing D *with* $new\langle k_1 \rangle$*, we obtain*

$$\begin{aligned} Q \quad &\mapsto \quad D, \textbf{def } read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle \textbf{ in } val\langle c \rangle \,|\, k_1\langle read \rangle, new\langle k_2 \rangle \\ &\rightleftharpoons^* \quad D, read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle, \; val\langle c \rangle, \, k_1\langle read \rangle, new\langle k_2 \rangle \end{aligned}$$

By firing D *again with* $new\langle k_2 \rangle$*, we obtain*

$$\begin{aligned} &\quad D, read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle, \; val\langle c \rangle, \, k_1\langle read \rangle, new\langle k_2 \rangle \\ \rightarrow \quad &\quad D, read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle, \; val\langle c \rangle, \, k_1\langle read \rangle, \textbf{def } read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle \textbf{ in } val\langle c \rangle \,|\, k_1\langle read \rangle \\ \rightleftharpoons^* \quad &\quad D, read\langle k \rangle \,|\, \underline{val}\langle v \rangle \vartriangleright k\langle v \rangle, \; val\langle c \rangle, \, k_1\langle read \rangle, read'\langle k \rangle, \underline{val'}\langle v \rangle \vartriangleright k\langle v \rangle, val'\langle c \rangle, k_1\langle read' \rangle \end{aligned}$$

Note that the heating of the solution renames the ports read *and* val *by the fresh names* read' *and* val'*, which assures any firing of* D *to produce fresh instances of local names.*

PJoin processes as DGGs.   The encoding for PJoin is analogous to the encoding of Join presented in [BM06a]. For simplicity we assume definitions with different scopes not to share defined names. Any process P is encoded as a DGG $\mathcal{G}_P = \langle T_P, G_{in}, Q \rangle$. Generally speaking, a channel x is encoded as a node n_x but the fact that the channel is named x is denoted by an arc $x : n_x \rightarrow n_x$. If node n_x corresponds to x and n_y to y, then a message $x\langle y \rangle$ is represented with an arc $n_x \rightarrow n_y$. Any firing rule $J \vartriangleright P$ will be encoded as a production. More formally, the initial type graph T_P is shown in Figure 9(a), where $fn(P) \cup bn(P) = \{c_1, \ldots, c_k, p_1, \ldots, p_l\}$. T_P has two nodes c and p standing respectively for consumable and persistent channels, four arcs m_{cc}, m_{cp}, m_{pc}, m_{pp} for denoting different kinds of messages, where the subindexes indicate the type of involved channels, one arc c_i for each consumable name in $fn(P) \cup bn(P)$, and one arc p_j for any persistent name in $fn(P) \cup bn(P)$. We call the *context* of P the T_P-typed graph C_P with one node c_{c_i} and one arc $c_i : c_{c_i} \rightarrow c_{c_i}$ for each consumable name $c_i \in fn(P) \cup bn(P)$, and one node p_{p_i} and one arc $p_i : p_{p_i} \rightarrow p_{p_i}$ for each persistent name $p_i \in fn(P) \cup bn(P)$. Then, the initial graph G_{in} and the set of productions Q of $\mathcal{G}_P = \langle T_P, G_{in}, Q \rangle$ are defined by structural induction as follows:

- $[P = 0]$. $G_{in} = C_P$ is the empty graph and $Q = \emptyset$.

(a) Initial Type Graph.

(b) Encoding of $x\langle y\rangle$.

(c) Encoding of $x\langle x\rangle$.

(d) Encoding of $c_1\langle d_1\rangle|\dots|c_k\langle q_k\rangle|p_1\langle d_1'\rangle|\dots|p_l\langle q_l'\rangle \triangleright P_i$.
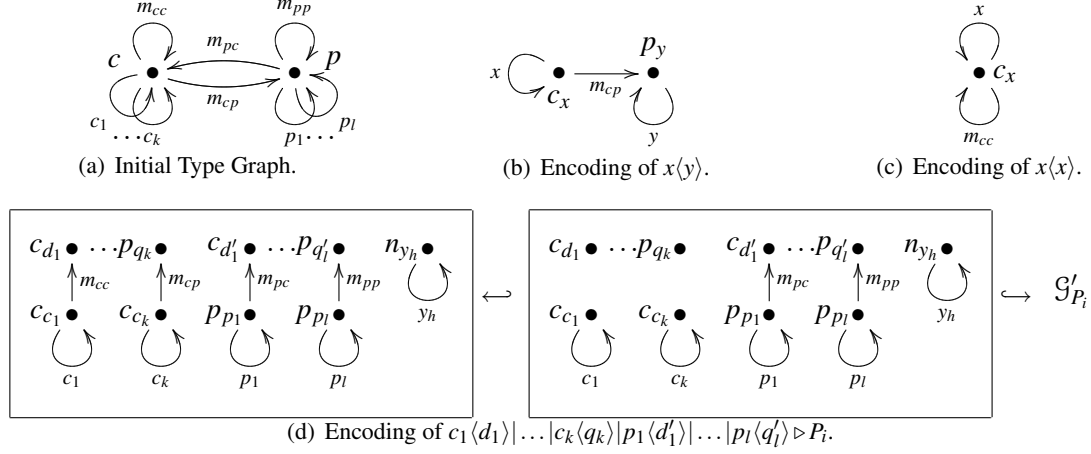
Figure 9: Join processes as Dynamic Graph Grammars.

- $[P = x\langle y\rangle]$. The encoding of a message depends on the type of involved names. There are four possibilities. If x is consumable and y persistent, then $G_{in} = C_P \cup \{m_{cp} : c_x \to p_y\}$ is the graph shown in Figure 9(b), with the typing morphism mapping c_x into c, p_y into p, and being the identity on the remaining arcs. Remaining cases are analogous. When $x = y$, $G_{in} = C_P \cup \{m_{cc} : n_x \to n_x\}$ is as in Figure 9(c). In all cases $Q = \emptyset$.

- $[P = \mathbf{def}\ J_1 \triangleright P_1 \wedge \dots \wedge J_n \triangleright P_n\ \mathbf{in}\ P']$. Let $\mathcal{G}_{P'} = \langle T_{P'}, G'_{in}, Q'\rangle$ be the encoding of P', then $\mathcal{G}_P = \langle T_P, C_P \cup G'_{in}, Q' \cup \bigcup_{1\leq i\leq n}\{r_i\}\rangle$, where $T_P \supseteq T_{P'}$ (because $J_i \triangleright P_i$ may contribute free names not present in P') and each r_i encodes $J_i \triangleright P_i$. Assuming $\mathcal{G}_{P_i} = \langle T_i, G_{in_i}, Q_i\rangle$ and the general shape for patterns $J_i = c_1\langle d_1\rangle|\dots|c_k\langle q_k\rangle|p_1\langle d_1'\rangle|\dots|p_l\langle q_l'\rangle$ (to expose all combinations of consumable and persistent names), then r_i is shown in Figure 9(d), where $\mathcal{G}'_{P_i} = \langle T_i \cup T_P, G'_{in_i}, Q'_i\rangle$ extends $\mathcal{G}_{P_i}$ over the type graph $T_i \cup T_P$ with G'_{in_i} being the union of G_{in_i} and the items preserved by the production, and with Q'_i being the suitable retyping of Q_i. The self-loop arcs naming the nodes d_j, d_j', q_j, and q_j' are not included in rule r_i because the actual name of formal parameters is not known a priori and it will be provided by valid matchings. Moreover, the left-hand-side and the interface contain a node n_{y_h} and an arc y_h for any free name y_h of P_i not bound in J_i. In this way the context of the initial graph of $\mathcal{G}_{P_i}$ is bound to the names of the left-hand-side of the production. Note that the production preserves all persistent messages, since the edges of type m_{pc} and m_{pp} are kept, while consumable messages are removed, because edges of type m_{cc} and m_{cp} are deleted.

- $[P = P_1|P_2]$. Let $\mathcal{G}_{P_1} = \langle T_1, G_{in_1}, Q_1\rangle$ and $\mathcal{G}_{P_2} = \langle T_2, G_{in_2}, Q_2\rangle$ be the encoding of P_1 and P_2, then the initial graph is the pushout object of the span $C_P \cup G_{in_1} \hookleftarrow C_P \hookrightarrow C_P \cup G_{in_2}$, and Q is the union of Q_1 and Q_2 (upon production retyping over T_P).

Given a PJoin process P, it might be the case that $\mathcal{G}_P$ is not persistent. This may happen when P yields non-consuming (e.g., containing a definition $p\langle x\rangle \triangleright P$, with p persistent) or non semi-weighted (e.g., $J \triangleright x\langle y\rangle | x\langle y\rangle$) productions. However, such situations can be checked by a static analysis of processes.

In particular, processes considered in our examples yield persistent grammars.

Example 9 *Consider the process Q introduced in Example 8, the PDGG encoding the behaviour of Q is shown in Figure 10. The initial type graph is depicted in 10(a): (i) nodes p and c are the types of persistent and consumable channels respectively, (ii) edges m_{cc}, m_{cp}, m_{pc} and m_{pp} are the types of edges encoding messages, (iii) edges c, k_1 and k_2 are the free names of Q, and (iv) edge new is the type of the unique defined name of Q.*

Figure 10(b) shows the initial graph, which contains the nodes and edges corresponding to the three free names of Q, i.e., c, k_1 and k_2, and the unique defined name new. Moreover, the initial graph contains the edges m_{cc}^1 and m_{cc}^2 representing the two initial messages $new\langle k_1\rangle$ and $new\langle k_2\rangle$.

The encoding of D is in Figure 10(c). The graph on the left-hand side requires a message on port new (arrow m_{cc}). Moreover, it contains the node c_c and the edge c that matches the name c occurring free in the guarded process of D. The names c and new act as binders in the right-hand side of the production $\mathcal{G}_{P_i}$, which is detailed in Figures 10(d), 10(e), and 10(f). The type graph of $\mathcal{G}_{P_i}$ adds to the initial type graph (in Figure 10(a)) the edges describing the names read, w, and val of the guarded process of D. The initial graph of $\mathcal{G}_{P_i}$ is shown in Figure 10(e), which encodes the messages $val\langle c\rangle$ and $w\langle read\rangle$ of the guarded process of D. It is worth noting that the initial graph contains also a new edge named w attached to the node c_w. In this way, an existing channel is associated with a new name, a kind of local alias for that channel. Finally, the unique production of $\mathcal{G}_{P_i}$ is shown in Figure 10(f). For simplicity, the right-hand of the production shows only the initial graph as we omit the type graph, which adds only the edges $k : c \rightarrow c$ and $v : c \rightarrow c$. In this case, the right-hand side is static, i.e., adds no productions.
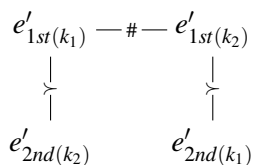
The following result hold up to aliasing of names, i.e., by removing aliasing from grammars.

Proposition 1 *For any PJoin process P we have:*

- *If $P \mapsto P'$ using $J_i \triangleright P_i$, then $\exists Q$ s.t. $\mathcal{G}_P \Rightarrow_{p_i} \mathcal{G}_Q$ and $Q \leftrightharpoons^* P'$;*

- *If $\mathcal{G}_P \Rightarrow_{p_i} \mathcal{G}'$, then $\exists P'$ s.t. $P \mapsto P'$ using $J_i \triangleright P_i$ and $\mathcal{G}' = \mathcal{G}_{P'}$.*

When $\mathcal{G}_P$ is persistent, we can combine the results discussed in Sections 2 and 3 to flatten the PDGG $\mathcal{G}_P$ into the PGG $[\![\mathcal{G}_P]\!]$, and then take the PES semantics $\mathcal{E}_p(\mathcal{U}_p([\![\mathcal{G}_P]\!]))$ (see Figure 1).

Example 10 *The PDGG $\mathcal{G}_P$ associated with the PJoin process P of Example 6 yields a PES with just two concurrent events (say $e_{read(k_1)}$ **co** $e_{read(k_2)}$). Differently, the PDGG $\mathcal{G}_{P'}$ associated with P' introduced in Example 7 would yield four events: $e'_{1st(k_1)}$ and $e'_{1st(k_2)}$ denoting respectively the fact that the first served request is $read\langle k_1\rangle$ or $read\langle k_2\rangle$, and similarly $e'_{2nd(k_1)}$ and $e'_{2nd(k_2)}$ for the second served request; where the causal and (immediate) conflict relations are shown below:*
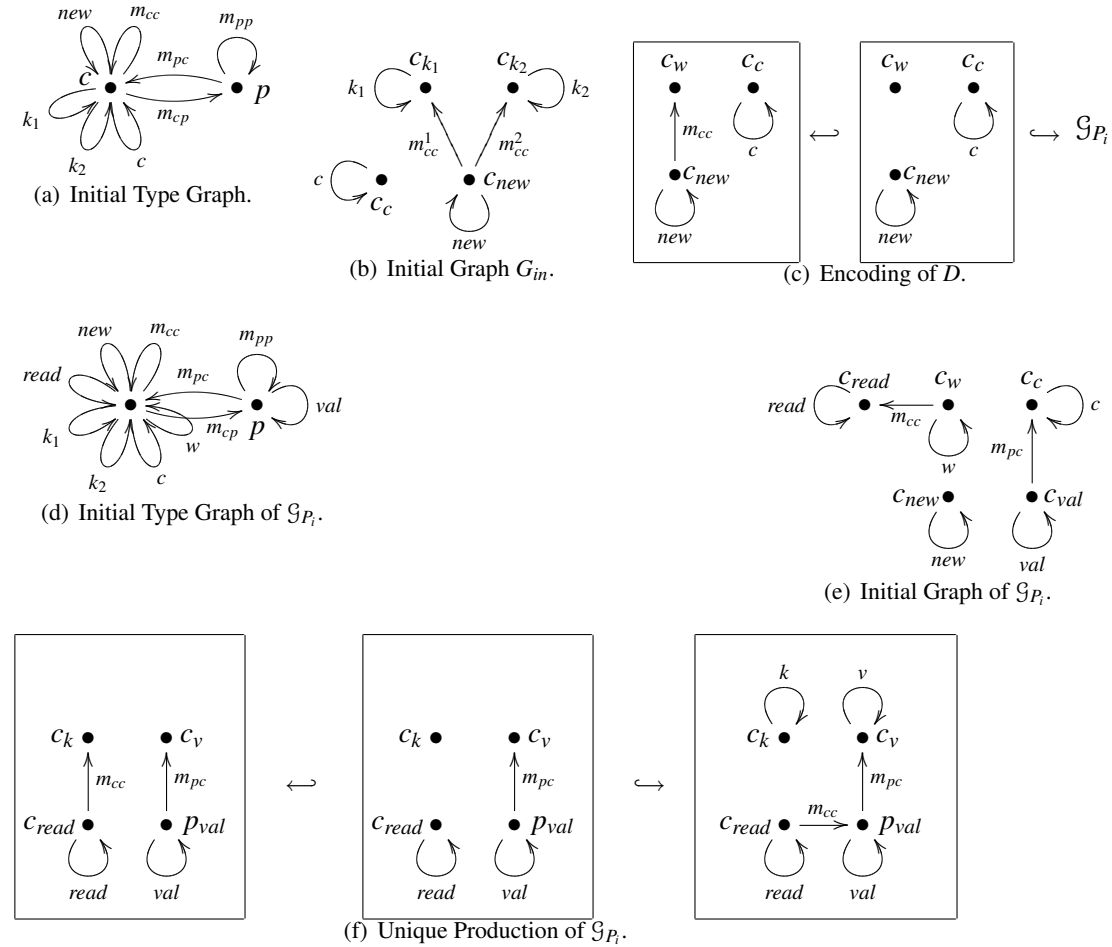
$$
\begin{array}{cc}
e'_{1st(k_1)} \text{---}\#\text{---} e'_{1st(k_2)} \\
\uparrow \qquad\qquad \uparrow \\
e'_{2nd(k_2)} \qquad e'_{2nd(k_1)}
\end{array}
$$

(a) Initial Type Graph.

(b) Initial Graph G_{in}.

(c) Encoding of D.

(d) Initial Type Graph of $\mathcal{G}_{P_i}$.

(e) Initial Graph of $\mathcal{G}_{P_i}$.

(f) Unique Production of $\mathcal{G}_{P_i}$.

Figure 10: Join processes as Dynamic Graph Grammars.

5 Conclusions

In this paper we propose PDGGs as a convenient model for concurrent reflexive systems. In fact PDGGs can be encoded in ordinary PGGs preserving concurrency, and PGGs admit a nice prime event structures semantics defined via a chain of adjunctions. Moreover, the case study of PJoin shows that PDGGs can allow for a more convenient representation of mobile calculi w.r.t. PGGs as evident by comparing the encoding in Section 4 with the one of ordinary Join in PGGs given in [BMM06]. In particular, here there is a more tight and neat connection between concepts: rules as productions (and not hyperarcs as in [BMM06]), channels as types (partitioned in persistent and consumable, with the behaviour defined by the corresponding productions) and messages as resources (typed accordingly the subject and object names).

References

[Bal00] P. Baldan. *Modelling concurrent computations: From contextual Petri nets to graph grammars*. PhD thesis, Computer Science Department, University of Pisa, 2000.

[BCMR07] P. Baldan, A. Corradini, U. Montanari, L. Ribeiro. Unfolding Semantics of Graph Transformation. *Information and Computation*, 2007. To appear.

[BM06a] R. Bruni, H. Melgratti. Dynamic Graph Transformation Systems. In Corradini et al. (eds.), *Proceedings of ICGT 2006, 3rd International Conference on Graph Transformation*. Lect. Notes in Comput. Sci. 4178, pp. 230–244. Springer Verlag, 2006.

[BM06b] R. Bruni, H. Melgratti. Non-sequential behaviour of dynamic nets. In Donatelli and Thiagarajan (eds.), *Proceedings of ATPN 2006, 27th International Conference on Application and Theory of Petri Nets and Other Models Of Councurrency*. Lect. Notes in Comput. Sci. 4024, pp. 105–124. Springer Verlag, 2006.

[BMM06] R. Bruni, H. Melgratti, U. Montanari. Event Structure Semantics for Nominal Calculi. In Baier and Hermanns (eds.), *Proceedings of CONCUR 2006, 17th International Conference on Concurrency Theory*. Lect. Notes in Comput. Sci. Springer Verlag, 2006. To Appear.

[BS01]   M. Buscemi, V. Sassone. High-level Petri nets as type theories in the Join calculus. In Honsell and Miculan (eds.), *Proceedings of FoSSaCS 2001, Foundations of Software Science and Computation Structures*. Lect. Notes in Comput. Sci. 2030, pp. 104–120. Springer Verlag, 2001.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Chapter Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. World Scientific, 1997.

[FG96]   C. Fournet, G. Gonthier. The reflexive chemical abstract machine and the Join calculus. In *Proceedings of POPL'96, 23rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*. Pp. 372–385. ACM Press, 1996.

[NPW81] M. Nielsen, G. Plotkin, G. Winskel. Petri Nets, Event Structures and Domains, Part 1. *Theoret. Comput. Sci.* 13:85–108, 1981.

[PSVV06] C. Palamidessi, V. Saraswat, F. Valencia, B. Victor. On the Expressiveness of Linearity vs Persistence in the Asychronous Pi-Calculus. In *LICS*. Pp. 59–68. 2006.