Laboratoire IBISC

Biologie Intégrative et Systèmes Complexes

Pigraphs with Replicators: Finiteness and Boundedness Results

Frédéric Peschanski, Hanna Klaudel, and Raymond Devillers

IBISC University of Evry-Val d'Essonne, Genopole, France

RAPPORT DE RECHERCHE
IBISC-RR-2011-04
mai 2011

IBISC – Université d'Evry-Val d'Essonne
Tour Evry 2, 523 place des Terrasses de l'Agora
91000 Evry Cedex, France

Pigraphs with Replicators

Finiteness and Boundedness Results (Technical Report)

Frédéric Peschanski[1], Hanna Klaudel[2], and Raymond Devillers[3]

[1] UPMC – LIP6
[2] Université d'Évry–IBISC
[3] Université Libre de Bruxelles

Abstract. This paper describes the pi-graphs, a diagrammatic variant of the pi-calculus. The semantics is based on graph relabelling techniques, giving a much more concrete characterization compared to standard characterizations. This allows us to derive very precise results, for example the prediction of upper bounds for the dynamic resources produced and consumed by (thread-bounded) pi-graphs.

1 Introduction

This paper describes the *pi-graph calculus*, a variant of the pi-calculus [1, 2]. The most important feature of the formalism is that each construct possesses a simple, if not natural, diagrammatic interpretation. This emphasizes our first goal: the conception of (visual *and* textual) *modelling* artefacts for mobile systems.

Unlike previous pi-graphs variants [3–5], the calculus presented in this paper is quite a standard pi-calculus. Thus, it can be semantically characterized effortlessly by reusing one of the various known semantics (early, late, symbolic, etc.) and behavioral equivalences (barbed, open, etc.) for the pi-calculus. However, beyond modelling, our second goal is to develop efficient *implementation* and *verification* techniques for mobile systems. This lead us to develop more *concrete* semantics for the calculus as a *token-game* inspired by the Petri nets.

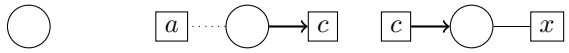
Technically we build on *graph relabelling* techniques which (unlike most graphical encodings of the pi-calculus, e.g. [6]) disallow node or edge creation or destruction in graph rewrites: the graph structure is invariant. This has practical advantages : the semantics is almost directly implementable, and structural invariance reflects a major trait of most popular modelling languages (e.g. finite state machines, statecharts, Petri nets, etc.). Most importantly, the essential notion of *thread* naturally emerges from the characterization. Although reasoning is less easy than in more abstract settings, we are able to derive quite precise results. For example, we exhibit upper bounds for the dynamic resources produced and consumed by systems in which the maximum number of simultaneously running threads is bounded, so-called *thread-bounded* pi-graphs. As a consequence, the "natural" behavioral equivalence on pi-graphs - namely *stateful bisimilarity* - is decidable for thread-bounded systems.

The outline of the paper is as follows. In Section 2 we introduce the diagram calculus and its informal semantics. The operational semantics, in terms of graph relabelling, is detailed is Section 4. The finiteness and boundedness results are presented in Section 5. The related and future works are discussed in Section 6.

2 The diagram calculus

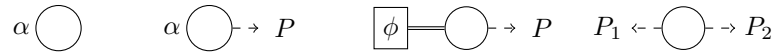
The pi-graph calculus is a relatively standard variant of the π-calculus, and *also* a language of *diagrams*. The syntax of the calculus, as well as the diagrammatic interpretation of each construct, is given below.

Action $\alpha ::=$ **Guard** $\phi ::= [a = b] \mid [a \neq b]$



τ (silent) $\quad \overline{c}\langle a \rangle$ (output) $\quad c(x)$ (input)

Process $P ::=$



α (suffix) $\quad \alpha.P$ (prefixing) $\quad \phi P$ (guard) $\quad P_1 + P_2$ (choice)

Replicator $R ::= r[k] : \widetilde{\nu a}\ !\alpha.P$ **Diagram** $\pi ::= \widetilde{\nu A}\ R_1 \ldots R_n$

In the graphical interpretation, each action corresponds to a node similar to a place in the Petri nets terminology, and indeed will contain *tokens*. The rectangular-shaped nodes are holders for *names* that can be used as channels. Of course, names can be transmitted dynamically as in the pi-calculus. The structure of the control-flow is encoded by dashed arrows. For instance, the choice operator involves two possible continuations. Unlike previous models of pi-graphs [3–5], we reuse the familiar pi-calculus concept of *replication* to model repetitive (and potentially infinite) behaviors. A difference with traditional replication is that in pi-graphs, the so-called *replicators* are not processes *per se* but can only be defined at the top-level. At the cost of some extra occurrences of tau transitions, nested replications can be encoded as top-level ones in the pi-calculus[4]. Relaxing this constraint would lead to semantic quirks (namely ϵ-divergence), and intuitively the reason is that replicated processes are spawned in a demand-driven way. Each replicator - identified as r - is associated to a bound k that limits the number of running *threads*. Control-infinite systems can be modelled with bound $k = \omega$. Note also that the replicated process must be prefixed, which also avoids semantic difficulties. If in the pi-calculus one can "play" with the scope of restrictions using *scope-extrusion* laws, the syntax is here constrained so that restrictions can only occur at the global level, or alternatively within a given replicator. The former are global restrictions, whereas the latter are thread-specific (although they might be shared). Note also the

[4] For a pi-calculus context C and a process P, $C[!P]$ can be encoded as e.g. $\nu c(c(x).!P \mid C[\overline{c}\langle c \rangle.0])$ provided c is fresh in $C[P]$ and x is not free in P.

absence of a 0 terminator (thus any action can be a suffix), which is simply not required (and indeed undesirable because tokens may accumulate indefinitely in the terminal nodes).

To illustrate the language, its diagrammatic interpretation and its informal semantics, we consider the following example.

Example 1. A (k-bounded) multi-session server (with initial redexes)

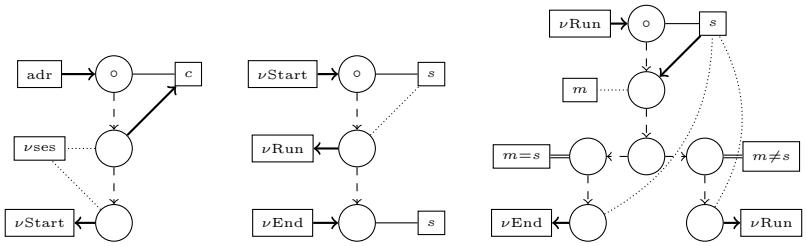
$\nu(\mathrm{Start}, \mathrm{Run}, \mathrm{End})$

$\mathrm{Service}[\omega] :! \boxed{\mathrm{Run}(s)}.s(m).([m = s]\overline{\mathrm{End}}\langle s\rangle + [m \neq s]\overline{\mathrm{Run}}\langle s\rangle)$

$\mathrm{Handler}[k] :! \boxed{\mathrm{Start}(s)}.\overline{\mathrm{Run}}\langle s\rangle.\mathrm{End}(s)$

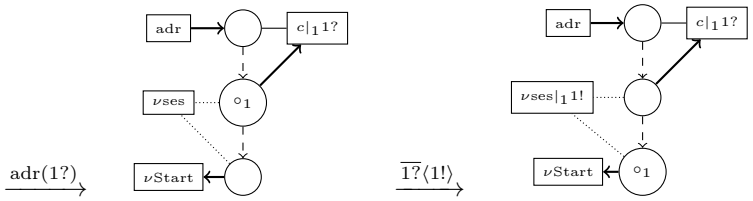
$\mathrm{Server}[1] : (\nu \mathrm{ses})\ ! \boxed{\mathrm{adr}(c)}.\overline{c}\langle \mathrm{ses}\rangle.\overline{\mathrm{Start}}\langle \mathrm{ses}\rangle$

The corresponding diagram (for $k = 3$) is depicted below.



Server$[1]$ **Handler**$[1, 2, 3]$ **Service**$[1, 2, \ldots]$

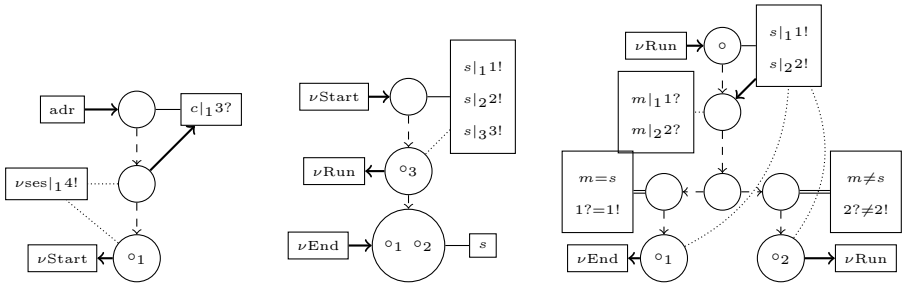
The tokens $\circ$ in the top-level place of each replicator represent the control state. In the textual representation, each token is represented as a frame surrounding an active part of process, a so-called *redex*. Each replicator label (Server, etc.) is associated to the set of available threads. For example, there is at most one thread (indexed 1) running in the server, three in the handler and an infinity in the service. In the first step, the server receives through the public channel adr a request for connection from a client.



The client request is a name bound to variable c, and it is given a *fresh identity* denoted 1?. The use of logical clock values as fresh names is a particularity of pi-graphs, which as explained in [3] gives a natural interpretation of freshness. Note also that the token within the server is labelled 1, which identifies the actual thread running the process, so the value of c is only 1? from the point of view of thread 1, hence the notation $c\,|_1\,1?$. Distinct threads may "see" different values.

In the second step, the locally private channel νses is sent back to the client. This channel will be used for further communications between the client and its specific service. Since the client is unknown (i.e. in the external environement), a *fresh output* name 1! is generated. This provides a concrete interpretation of a *bound output* in the pi-calculus terminology.

We depict below a reachable configuration of the system with two service threads running in parallel.



In the thread with identity 1 the session channel is 1! (bound to s) and a message 1? (bound to m) has been received. The match branch of the choice has been taken and thus 1? is asserted equal to 1! (i.e. $m = s$ for thread 1). The received message is thus the session channel itself, which triggers the termination of the service. In the second thread (identity 2), the message 2? is indeed distinct from the session channel 2! (i.e. $m \neq s$ for thread 2), and the service can be reentered at once through νRun (an example of self-recursion). In the Handler threads 1 and 2 are waiting for the end of their corresponding service. A further thread 3 is ready to start a service, which means a third client connection has been effected. We can see that any other potential client is now waiting for the handler to release at least one session (since $k = 3$).

3 Graphical representation

Process calculi are most often presented as term-rewrite systems whereas the pi-graphs follow the principle of *graph relabelling*. In this section we discuss the graphical model underlying the process algebra. Firstly, a precise definition of *names* manipulated by pi-graphs is required.

Definition 1. *The set of* ***names*** $\mathcal{N}$ *is the disjoint union of the following sets:*

$\mathcal{N}_f$ *the set of free names* $a, b, \ldots$
$\mathcal{N}_r$ *the set of global restrictions* $\nu A, \nu B, \ldots$
$\mathcal{N}_v$ *the set of input variables* $x, y, \ldots$
$\mathcal{N}_p$ *the set of local private names* $\nu a, \nu b, \ldots$
$\mathcal{N}_s$ *the set of shared names* $\nu a_{:\ell}, \nu b_{:\ell'} \ldots$ $(\ell, \ell' \in \mathbb{N})$
$\mathcal{N}_o \stackrel{\text{def}}{=} \{n! \mid n \in \mathbb{N}\}$ *the set of output names*
$\mathcal{N}_i \stackrel{\text{def}}{=} \{n? \mid n \in \mathbb{N}\}$ *the set of input names*

We define $\mathsf{Priv} \stackrel{\text{def}}{=} \mathcal{N}_r \cup \mathcal{N}_p \cup \mathcal{N}_s$ *(private names),* $\mathsf{Pub} \stackrel{\text{def}}{=} \mathcal{N} \setminus \mathsf{Priv}$ *(public names),* $\mathsf{Local} \stackrel{\text{def}}{=} \mathcal{N}_p \cup \mathcal{N}_v$ *and* $\mathsf{Global} \stackrel{\text{def}}{=} \mathcal{N}_r \cup \mathcal{N}_f \cup \mathcal{N}_s$.

This categorization is required because names are *globalized* in the semantics (i.e. all names have global scope). The sets $\mathcal{N}_i$ and $\mathcal{N}_o$ are names whose identity is generated fresh by construction. As illustrated before, they play a prominent role in the model. Together with the shared names in $\mathcal{N}_s$ - private names that have been communicated internally - they represent the *dynamic resources* produced and consumed by pi-graphs.

Definition 2. *a* ***pi-graph*** π *is a tuple* $\langle R, J, V, L, E, M\rangle$ *with*

- R *a set of replicator labels*
- $J \in R \to \mathbb{P}(\mathbb{N})$ *the available thread identifiers,*
- $V \stackrel{\text{def}}{=} \bigcup_{r\in R} V_r$ *a set of vertices (with all subsets* V_r *assumed disjoint),*
- $L \in V \times \{o, i, g_=, g_{\neq}\} \to \mathcal{N}^2$ *a vertex labelling,*
- $E \subset V \times V$ *a set of edges,*
- $M \in V \to \mathbb{P}(\mathbb{N})$ *a marking function,*

A pi-graph π is composed of a set of replicators, each one identified by a label r in set R^π. The vertices of the pi-graph define set $V^\pi \stackrel{\text{def}}{=} \bigcup_{r\in R^\pi} V_r^\pi$ where each set V_r^π identifies the vertices within replicator r (the vertex sets are assumed disjoint). The tokens in the diagrams - and equivalently the textual redexes - correspond in graphical terms to markings of vertices. We denote $M^\pi(v) \subseteq \mathbb{N}$ the set of tokens for vertex $v \in V^\pi$. In a diagram, if a node v (of replicator r) contains a token $\circ_i$, then $i \in M_r^\pi(v)$ represents the identity of the corresponding thread. We denote $J_r^\pi \subseteq \mathbb{N} \cup \{\omega\}$ the set of remaining thread identifiers in iterator r. If $J_r^\pi \neq \emptyset$ then a token $\circ$ is put in the initial place of the replicator. The maximum number of running threads of replicator r is denoted k_r^π.

Particularly important is the *context* in which the graphs are rewritten.

Definition 3. *A pi-graph* ***context*** Δ *is a triple* $\beta; \gamma; \kappa$ *with:*

- $\beta \in \mathcal{N} \to \mathcal{N}$ *the* ***name environment***
- $\kappa \in \mathcal{N}_o \to \mathbb{P}(\mathcal{N}_i)$ *a* ***causal clock****, and*
- $\gamma \stackrel{\text{def}}{=} (\gamma^=, \gamma^{\neq}) \in (\mathcal{N} \times \mathcal{N})^2$ *a dynamic match/mismatch* ***partition****.*

The name environment β relates the syntactic - and static - occurrences of names in the diagrams to their dynamic counterpart. To isolate a name x whose scope is bound to a given replicator r and thread j (e.g. a private name or an input variable), the manipulation functions of Table 1 reference such name as x_j^r. The reset function is used when a thread j finishes, and restores the identity for all its local names[5]. A special case is when a private name has been communicated to another thread and in such a situation it is replaced by a shared private name attached to a fresh identity $\ell \in \mathbb{N}$. The causal clock κ is used to

[5] For the sake of compactness, the identity is not explicitly encoded in the name environment β, but it is enforced by its manipulation functions.

Name environment β	
reference	$\beta^r_j(x) \stackrel{\text{def}}{=} \begin{cases} \beta(x) \text{ if } x \in \mathsf{dom}(\beta),\ \beta(x^r_j) \text{ if } x^r_j \in \mathsf{dom}(\beta), \\ x^r_j \text{ if } x \in \mathsf{Local},\ x \text{ otherwise} \end{cases}$
update[6]	$\beta^r_j[y/x] \stackrel{\text{def}}{=} \begin{cases} \beta \rhd \{x \mapsto y\} \cup \{z \mapsto y \mid \beta(z) = x\} \text{ if } x \in \mathsf{Global} \\ \beta \rhd \{x^r_j \mapsto y\} \cup \{z \mapsto y \mid \beta(z) = x^r_j\} \text{ otherwise} \end{cases}$
reset	$\mathsf{reset}^r_j(\beta) \stackrel{\text{def}}{=} \beta' \setminus \{x^r_j \mapsto y \mid x^r_j \in \mathsf{dom}(\beta)\}$
	with $\beta' \stackrel{\text{def}}{=} \beta \rhd \{z \mapsto x_{:\ell} \mid z \mapsto x^r_j \in \beta,\ \ell = \mathsf{min}(\mathbb{N}^+ \setminus \{\ell' \mid x_{:\ell'} \in \beta\})\}$
Causal clock κ	
fresh output	$\mathsf{out}(\kappa) \stackrel{\text{def}}{=} \kappa \cup \{\mathsf{next_o}(\kappa)! \mapsto \emptyset\}$
fresh input	$\mathsf{in}(\kappa) \stackrel{\text{def}}{=} \{o \mapsto (\kappa(o) \cup \{\mathsf{next_i}(\kappa)?\}) \mid o \in \mathsf{dom}(\kappa)\}$
freshness (in)	$\mathsf{next_i}(\kappa) \stackrel{\text{def}}{=} \mathsf{min}\left(\mathbb{N}^+ \setminus \{n \mid n? \in \bigcup \mathsf{cod}(\kappa)\}\right)$
freshness (out)	$\mathsf{next_o}(\kappa) \stackrel{\text{def}}{=} \mathsf{min}\left(\mathbb{N}^+ \setminus \{n \mid n! \in \mathsf{dom}(\kappa)\}\right)$
r-w causality	$x \prec_\kappa y \stackrel{\text{def}}{=} x \in \mathsf{dom}(\kappa) \wedge y \in \kappa(x)$
unify	$\kappa_{\lhd x=y} \stackrel{\text{def}}{=} \left\{ n! \mapsto \begin{cases} \kappa(n!) \setminus \{x\} \text{ if } y \notin \kappa(n!) \\ \kappa(n!) \setminus \{y\} \text{ if } x \notin \kappa(n!) \\ \kappa(n!) \text{ otherwise} \end{cases} \middle| n! \in \mathsf{dom}(\kappa) \right\}$
Partition $\gamma \stackrel{\text{def}}{=} (\gamma^=, \gamma^\neq)$	
match	$x =_\gamma y$ iff $\begin{cases} x = y \vee (x,y) \in \gamma^= \vee (y,x) \in \gamma^= \\ \vee \exists z,\ x =_\gamma z \wedge z =_\gamma y \end{cases}$
mismatch	$x \neq^\kappa_\gamma y$ iff $\begin{cases} (x,y) \in \gamma^\neq \vee (y,x) \in \gamma^\neq \vee y \neq^\kappa_\gamma x \\ \vee (\exists z_1\ z_2,\ x =_\gamma z_1 \wedge z_1 \neq^\kappa_\gamma z_2 \wedge z_2 =_\gamma y) \\ \vee (x,y \in \mathsf{Priv} \cup \mathcal{N}_o \wedge x \neq y) & \text{– private} \neq \text{private} \\ \vee (x \in \mathcal{N}_f \wedge y \in \mathsf{Priv} \cup \mathcal{N}_o) & \text{– public} \neq \text{private} \\ \vee (x \in \mathcal{N}_o \wedge y \in \mathcal{N}_i \wedge x \not\prec_\kappa y) & \text{– read-write order} \end{cases}$
compatibility	$x \overset{\gamma}{\sim}_\kappa y$ iff $\begin{cases} \neg(x \neq^\kappa_\gamma y) \wedge (x =_\gamma y \vee y \overset{\gamma}{\sim}_\kappa x \\ \vee (x,y \in \mathcal{N}_f \cup \mathcal{N}_i) & \text{– free and input compatibles} \\ \vee (x \in \mathcal{N}_o \wedge y \in \mathcal{N}_i \wedge x \prec_\kappa y)) & \text{– write-read order} \end{cases}$
refine	$\gamma_{\lhd x=y} \stackrel{\text{def}}{=} (\gamma^= \cup \{(x,y)\}, \gamma^\neq)$ if $\neg(x =_\gamma y)$, γ otherwise
refine set	$\gamma \lhd (\{(x,y)\} \cup E) \stackrel{\text{def}}{=} \gamma_{\lhd x=y} \lhd E$ and $\gamma \lhd \emptyset \stackrel{\text{def}}{=} \gamma$
discriminate	$\gamma_{\lhd x \neq^\kappa y} \stackrel{\text{def}}{=} (\gamma^=, \gamma^\neq \cup \{(x,y)\})$ if $\neg(x \neq^\kappa_\gamma y)$, γ otherwise

Table 1. The pi-graph context and associated operations.

generate fresh identities for names received from or sent to the environment, and it also implements *read-write causality* [7]. Finally, the γ component allows to record which names are *effectively* equal (in $\gamma^=$) (resp. inequal in $\gamma^\neq$) after a successful match (resp. mismatch). The functions manipulating the context, as defined in Table 1, will be discussed in the next section.

[6] For partial functions f_1 and f_2, $f_1 \rhd f_2 \stackrel{\text{def}}{=} f_1 \setminus \{x \mapsto f_1(x) \mid x \in \mathsf{dom}(f_2)\} \cup f_2$

The interpretation of a diagram $D \stackrel{\text{def}}{=} \nu(A_1,\ldots,A_p)\ R_1\ \ldots\ R_n$ is a graph π such that the statement $D \models \pi$ is provable by the following rule:

$$\frac{\begin{array}{l} P_1 \models_{A_1\mapsto\nu A_1,\ldots,A_p\mapsto\nu A_p,a_{1_1}\mapsto\nu a_{1_1},\ldots,a_{1_{m_1}}\mapsto\nu a_{1_{m_1}}} \langle V_1, i_{P_1}, L_1, E_1\rangle \\ \ldots \\ P_n \models_{A_1\mapsto\nu A_1,\ldots,A_p\mapsto\nu A_p,a_{n_1}\mapsto\nu a_{n_1},\ldots,a_{n_{m_n}}\mapsto\nu a_{n_{m_n}}} \langle V_n, i_{P_n}, L_n, E_n\rangle \\ V_1\cap\ldots\cap V_n = \emptyset \end{array}}{\begin{array}{l} \nu(A_1,\ldots,A_p)\ r_1[k_1]:\nu(a_{1_1},\ldots,a_{1_{m_1}})\ !P_1\ \ldots\ r_n[k_n]:\nu(a_{n_1},\ldots,a_{n_{m_n}})\ !P_n \\ \models \langle \{r_1,\ldots,r_n\}, \{r_1\mapsto[1,k_1],\ldots,r_n\mapsto[1,k_n]\}, V_1\cup\ldots\cup V_n, L_1\cup\ldots\cup L_n, \\ \quad E_1\cup\ldots\cup E_n, \{v\mapsto\emptyset \mid v\in V_1\cup\ldots V_n\}\rangle \end{array}}$$

In the resulting pi-graph π the (disjoint) sets of vertices $V_1,\ldots,V_n$, their labelling $L_1,\ldots,L_n$ and their edge sets $E_1,\ldots,E_n$ are obtained by proving, for each sub-process $P_1,\ldots,P_n$ an assertion of the form $P_j \models_\sigma \langle V_j, i_{P_j}, L_j, E_j\rangle$ where i_{P_j} is the node corresponding to the initial action of P_j. The substitution σ maps the bound names (the global restrictions and the private names) to their value (in $\mathcal{N}_s$ or $\mathcal{N}_p$). The rules for sub-processes are as follows :

$$\frac{\alpha \models_\sigma \langle v_\alpha, L_\alpha\rangle \qquad P \models_\sigma \langle V_P, i_P, L_P, E_P\rangle \quad v_\alpha\notin V_P}{\alpha.P \models_\sigma \langle \{v_\alpha\}\cup V_P, v_\alpha, L_\alpha\cup L_P, \{v_\alpha\mapsto i_P\}\cup E_P\rangle}$$

$$\frac{P \models_\sigma \langle V_P, i_P, L_P, E_P\rangle \quad L_\phi \stackrel{\text{def}}{=} \left\{\begin{array}{l}(v_\phi, g_=)\mapsto(\sigma(x),\sigma(y)) \text{ if } \phi=[x=y]\\ (v_\phi, g_\neq)\mapsto(\sigma(x),\sigma(y)) \text{ if } \phi=[x\neq y]\end{array}\right\} \quad v_\phi\notin V_P}{\phi P \models_\sigma \langle \{v_\phi\}\cup V_P, v_\phi, L_\phi\cup L_P, \{v_\phi\mapsto i_P\}\cup E_P\rangle}$$

$$\frac{\begin{array}{l} P_1 \models_\sigma \langle V_{P_1}, i_{P_1}, L_{P_1}, E_{P_1}\rangle \\ P_2 \models_\sigma \langle V_{P_2}, i_{P_2}, L_{P_2}, E_{P_2}\rangle \end{array} \quad v_+\notin V_{P_1}\cup V_{P_2} \quad V_{P_1}\cap V_{P_2}=\emptyset}{P_1+P_2 \models_\sigma \langle \{v_+\}\cup V_{P_1}\cup V_{P_2}, v_+, L_{P_1}\cup L_{P_2}, \{v_+\mapsto i_{P_1}, v_+\mapsto i_{P_2}\}\cup E_{P_1}\cup E_{P_2}\rangle}$$

Each action α generates a single vertex v_α yielding an assertion $\alpha \models_\sigma \langle v_\alpha, L_\alpha\rangle$ provable by one of the following rules:

$$\frac{}{\tau \models_\sigma \langle v_\tau, \emptyset\rangle}$$

$$\frac{}{\overline{c}\langle a\rangle \models_\sigma \langle v_o, \{(v_o,c)\mapsto(\sigma(c),\sigma(c)),(v_o,o)\mapsto(\sigma(a),\sigma(a))\}\rangle}$$

$$\frac{}{c(x) \models_\sigma \langle v_i, \{(v_i,c)\mapsto(\sigma(c),\sigma(c)),(v_i,i)\mapsto(x,x)\}\rangle}$$

The most important property of the graphical translation is that it generates graphs that have a size comparable to that of the initial (syntactic) diagrams.

Definition 4. *Let D be a diagram, we denote $|D|$ its size defined as follows:*

$$\left[\begin{array}{l} |\nu\widetilde{A}\ r_1[k_1]:\nu\widetilde{a_1}\ !P_1\ \ldots\ r_n[k_n]:\nu\widetilde{a_n}\ !P_n| \stackrel{\text{def}}{=} \sum_{i=1}^n |P_i| \\ |\alpha.P| \stackrel{\text{def}}{=} 1+|P| \\ |\phi P| \stackrel{\text{def}}{=} 1+|P| \\ |P_1+P_2| \stackrel{\text{def}}{=} 1+|P_1|+|P_2| \end{array}\right.$$

The size of a pi-graph π, denoted $|\pi|$, with vertices V is the cardinal of V.

In the diagrams we omit the restrictions and private names because they do not generate nodes in the translation. Moreover, in the translated graphs the name holders (square nodes) are not taken into account because at most two of these can be associated to each vertex.

Theorem 1. *Let D be a well-formed diagram. Then there exists a graph π such that $D \models \pi$. Moreover, $|\pi| = |D|$.*

Proof. By a simple structural induction on the translation rules □

4 Operational semantics

The operational semantics manipulates terms of the form $\Delta \vdash \pi$ where π is pi-graph (with explicit redexes) and $\Delta \stackrel{\text{def}}{=} \beta; \gamma; \kappa$ a context. The *initial context* has all empty components, it is denoted $\Delta_0 \stackrel{\text{def}}{=} \emptyset; (\emptyset, \emptyset); \emptyset$.

The rewrite rules are decomposed into layers. At the lower-level, the *commitment rules* (cf. Table 2) explain the atomic actions of pi-graphs. Above these, the *process rules* (cf. Table 3) describe the *local* behavior of processes. The *replicator rules* (cf. Table 4) describe what can happen for a prefix or suffix action of a replicator. Finally, there is a single *diagram rule* acting at the global-level.

[silent] $\beta; \gamma; \kappa \vdash r[j] : \tau \xrightarrow{\tau} \beta; \gamma; \kappa$

[out] $\beta; \gamma; \kappa \vdash r[j] : \overline{c}\langle a \rangle \xrightarrow{\overline{c'}\langle a' \rangle} \beta; \gamma; \kappa$ if $c' \stackrel{\text{def}}{=} \beta_j^r(c),\ a' \stackrel{\text{def}}{=} \beta_j^r(a) \in \mathsf{Pub}$

[o-fresh] $\beta; \gamma; \kappa \vdash r[j] : \overline{c}\langle a \rangle \xrightarrow{\overline{c'}\langle \mathsf{next_o}(\kappa)! \rangle} \beta_j^r[\mathsf{next_o}(\kappa)!/a]; \gamma; \mathsf{out}(\kappa)$
if $c' \stackrel{\text{def}}{=} \beta_j^r(c) \in \mathsf{Pub}$ and $\beta_j^r(a) \in \mathsf{Priv}$

[i-fresh] $\beta; \gamma; \kappa \vdash r[j] : c(x) \xrightarrow{c'(\mathsf{next_i}(\kappa)?)} \beta_j^r[\mathsf{next_i}(\kappa)?/x]; \gamma; \mathsf{in}(\kappa)$ if $c' \stackrel{\text{def}}{=} \beta_j^r(c) \in \mathsf{Pub}$

[com] $\beta; \gamma; \kappa \vdash r_1[j_1] : \overline{c}\langle a \rangle \parallel r_2[j_2] : d(x) \xrightarrow{\tau} \beta_{j2}^{r2}[a'/x]; \gamma_{\lhd c'=d'}; \kappa_{\lhd c'=d'}$
if $c' \overset{\gamma}{\sim}_{\kappa} d'$ with $c' \stackrel{\text{def}}{=} \beta_{j1}^{r1}(c),\ a' \stackrel{\text{def}}{=} \beta_{j1}^{r1}(a)$ and $d' \stackrel{\text{def}}{=} \beta_{j2}^{r2}(d)$

Table 2. Operational semantics: commitment rules.

Each atomic action performed by a pi-graph is handled by a specific commitment rule. The [silent] rule explains that a τ action performed in the context of a replicator r and thread j yields a commitment denoted $\xrightarrow{\tau}$. The context remains unmodified. The [out] rule generates a commitment $\xrightarrow{\overline{c'}\langle a' \rangle}$, where c' and a' are the values bound to "variables" c and a in the environment β. Both names must be public to trigger the rule. The rule [o-fresh] explains the emission

[prefix] $\beta;\gamma;\kappa \vdash r : \boxed{\alpha}_j.P \xrightarrow{\mu} \beta';\gamma';\kappa' \vdash r : \alpha.\boxed{P}_j$ if $\beta;\gamma;\kappa \vdash r[j] : \alpha \overset{\mu}{\dashrightarrow} \beta';\gamma';\kappa'$

[match] $\beta;\gamma;\kappa \vdash r : \boxed{[a=b]}_j P \xrightarrow{\varepsilon} \beta;\gamma_{\lhd a'=b'};\kappa_{\lhd a'=b'} \vdash r : [a=b]\boxed{P}_j$ if $a' \overset{\gamma}{\sim}_\kappa b'$
with $a' \overset{\text{def}}{=} \beta^r_j(a)$ and $b' \overset{\text{def}}{=} \beta^r_j(b)$

[miss] $\beta;\gamma;\kappa \vdash r : \boxed{[a\neq b]}_j P \xrightarrow{\varepsilon} \beta;\gamma_{\lhd a'\neq^\kappa b'};\kappa \vdash r : [a\neq b]\boxed{P}_j$ if $\neg(a' =_\gamma b')$
with $a' \overset{\text{def}}{=} \beta^r_j(a)$ and $b' \overset{\text{def}}{=} \beta^r_j(b)$

[sync] $\beta;\gamma;\kappa \vdash r_1 : \boxed{\alpha_1}_{j_1}.P \parallel r_2 : \boxed{\alpha_2}_{j_2}.Q \xrightarrow{\tau} \beta';\gamma';\kappa' \vdash r_1 : \alpha_1.\boxed{P}_{j_1} \parallel r_2 : \alpha_2.\boxed{Q}_{j_2}$
if $\beta;\gamma;\kappa \vdash r_1[j_1] : \alpha_1 \parallel r_2[j_2] : \alpha_2 \overset{\tau}{\dashrightarrow} \beta';\gamma';\kappa'$

[sum$_L$] $\beta;\gamma;\kappa \vdash r : P_1\boxed{+}_j P_2 \xrightarrow{\epsilon} \beta;\gamma;\kappa \vdash r : \boxed{P_1}_j + P_2$ – also [sum$_R$]

Table 3. Operational semantics: process rules.

of a private name (either a restriction νA, a private name νa or a shared name $\nu a_{:\ell}$) over a public channel. As illustrated previously, a fresh value - denoted $\mathsf{next}_o(\kappa)$! - is generated for the output name. The clock itself is modified so that the freshness guarantee is preserved. Part of the [o-fresh] commitment is the update of the environment to reflect the renaming by a fresh value. Note that if the name a is a global restriction, then the update is *permanent*. If it is a private name local to a thread, then the binding is tied to the life-cycle of that thread, except if it is shared with another thread. The [i-fresh] rule is almost the symmetric, it deals with names received from the environment, picked up freshly as $\mathsf{next}_i(\kappa)$?. Finally, the [com] rule explains the commitment of value-passing between two threads (here separated by a $\parallel$; they may arise from two different replicators and/or threads). For the sake of *compositionality*, a communication can take place between an output and an input on two *distinct* channels c' and d', as long as they can be equated explicitly, which we denote $c' \overset{\gamma}{\sim}_\kappa d'$ (cf. the [match] case below). The environment is updated to reflect this new equality.

Each process rule of Table 3 provides the description of a local rewrite of the form: $\beta;\gamma;\kappa \vdash \theta \xrightarrow{\mu} \beta';\gamma';\kappa' \vdash \theta'$ where $\beta;\gamma;\kappa$ is the context before the rewrite, and θ is a "subterm" of the pi-graph π with some redex(es) inside. In graphical terms, θ identifies the replicator r and a set of vertices $V_\theta \subseteq V^\pi_r$ of the graph π where the rewrite takes place. The [prefix] rule, for instance, describes the execution of an action in prefix position within a process, if the corresponding commitment can be proved. Here, the subterm θ is $\alpha.P$ and V_θ is defined as $\{v_\alpha\} \cup V_P$ where v_α is the single vertex corresponding to α, and V_P the vertices of (the graphical representation of) P. The notation $\boxed{\alpha}_j$ means that α is a *redex* for a thread identified as j. In the underlying graph, j is in $M^\pi(v_\alpha)$, i.e. the vertex v_α is marked with a token indexed by j. More generally, a redex can be any action $\boxed{\alpha}_j$, a guard $\boxed{\phi}_j P$ or a sum $P_1\boxed{+}_j P_2$. The notation extends to

arbitrary processes beyond action suffixes with $\boxed{\alpha.P}_j \stackrel{\text{def}}{=} \boxed{\alpha}_j.P$, $\boxed{\phi P}_j \stackrel{\text{def}}{=} \boxed{\phi}_j.P$ and $\boxed{P_1 + P_2}_j \stackrel{\text{def}}{=} P_1 \boxed{+}_j P_2$. A prefixed (resp. guarded) process is thus in its initial state for thread j if its initial prefix (resp. guard) is a redex. So in the right-hand side of the [prefix] rule the initial action of the continuation P (i.e. a place $v \in V_p$) received the token. The label μ of the transition as well as the update of the context are given by the commitment triggered for the action.

The [match] and [miss] rules are similar to a mix of commitment and prefixing, but cannot be decomposed because they are not "visible" commitment. Importantly, such *invisible steps*, labelled ϵ, cannot commit the choice of a given branch in a sum. Upon matching, the two names to equate - denoted a' and b' - must satisfy the *compatibility* relation $a' \overset{\gamma}{\sim}_\kappa b'$ (cf. Table 1). It is often the case that names are compatible, e.g. two free names are compatible as long as they are not explicitly inequated (by a mismatch). For dynamic names in $\mathcal{N}_i$ and $\mathcal{N}_o$ read-write causality is enforced by the causal clock κ. An output name $o!$ can only be equated to an input name $i?$ if the latter is *causally dependent* on the former, i.e. $o!$ "escaped" to the environment *before* $i?$ was received. This is the *write-read order* when $i? \in \kappa(o!)$. In the case of *read-write order* $?i \notin \kappa(o!)$, i.e. if $i?$ is received before $o!$ escapes, then the equality is forbidden. There are other subtilities, e.g. to reflect the equality in both γ and κ, but the formalization is routine. The mismatch is often omitted from pi-calculi (note that our introductory example motivates its use), mostly due to its non-trivial characterization. A natural treatment - similar to match - results from the concrete pi-graph semantics. The [miss] rule is very close to [match] except for the condition of name *incompatibility*. In Table 1 the relation $\neq_\gamma$ lists all the "natural" incompatibilities among names: e.g. restricted names are in essence distinguished, etc. Once again, read-write causality is enforced.

The synchronization rule [sync] involves two local contexts : an emission and a reception. These can be within the same replicator (albeit in distinct threads) or between two replicators. The communication must be committed (through [com]) and the tokens are "pushed" as in the [prefix] case.

Finally, the choice construct is interpreted by the [sum_L] rule (and its symmetric [sum_R]). The choice is made *arbitrarily* in the local context, but the chosen branch is *not* committed since an invisible ϵ-transition is taken. The notion of abstracted transition (cf. below) allows to recover the usual interpretation of the choice, which is to find a valid commitment within the branch.

Above the process rules are defined replicator rules involving at least a replicator-level context, i.e. its initial or one of its final actions. The [spawn] rule, for instance, is a variant of [prefix] that initiates a new thread with identity j, if available. The [term] rule explains the (normal) termination of a thread by its suffix action. The thread identity j is given back to the replicator, and the local names are reset. As explained previously, the private names of j that have been communicated to other threads become explicitly shared upon termination. The other rules are variants of [spawn] and [term] involving synchronization, which we interpret as "procedure" calls. The [call_L] (resp. [call_R]) rule synchronizes a

$$[\text{spawn}]\quad \beta;\gamma;\kappa \vdash r[J] :!\boxed{\alpha}.P \xrightarrow{\mu} \beta';\gamma';\kappa' \vdash r[J\setminus\{j\}] :!\alpha.\boxed{P}_j \qquad \text{with } j \stackrel{\text{def}}{=} \mathsf{min}(J)$$
$$\text{if } \beta;\gamma;\kappa \vdash r[j] : \alpha.P \overset{\mu}{\dashrightarrow} \beta';\gamma';\kappa'$$

$$[\text{term}]\quad \beta;\gamma;\kappa \vdash r[J] : \boxed{\alpha}_j \xrightarrow{\mu} \mathsf{reset}^r_j(\beta');\gamma';\kappa' \vdash r[J\cup\{j\}] : \alpha$$
$$\text{if } \beta;\gamma;\kappa \vdash r[j] : \alpha \overset{\mu}{\dashrightarrow} \beta';\gamma';\kappa'$$

$$[\text{call}_L]\quad \beta;\gamma;\kappa \vdash r_1[J_1] :!\boxed{\alpha_1}.P \parallel r_2 : \boxed{\alpha_2}_{j_2}.Q$$
$$\xrightarrow{\tau} \beta';\gamma';\kappa' \vdash r_1[J_1\setminus\{j_1\}] :!\alpha_1.\boxed{P}_{j_1} \parallel r_2 : \alpha_2.\boxed{Q}_{j_2} \qquad \text{with } j_1 \stackrel{\text{def}}{=} \mathsf{min}(J_1)$$
$$\text{if } \beta;\gamma;\kappa \vdash r_1[j_1] : \alpha_1 \parallel r_2[j_2] : \alpha_2 \overset{\tau}{\dashrightarrow} \beta';\gamma';\kappa' \quad \text{– also } [\text{call}_R] \text{ and } [\text{call}_{LR}]$$

$$[\text{call}_{Lt}]\quad \beta;\gamma;\kappa \vdash r_1[J_1] :!\boxed{\alpha_1}.P \parallel r_2[J_2] : \boxed{\alpha_2}_{j_2} \qquad \text{with } j_1 \stackrel{\text{def}}{=} \mathsf{min}(J_1)$$
$$\xrightarrow{\tau} \mathsf{reset}^{r_2}_{j_2}(\beta');\gamma';\kappa' \vdash r_1[J_1\setminus\{j_1\}] :!\alpha_1.\boxed{P}_{j_1} \parallel r_2[J_2\cup\{j_2\}] : \alpha_2$$
$$\text{if } \beta;\gamma;\kappa \vdash r_1[j_1] : \alpha_1 \parallel r_2[j_2] : \alpha_2 \overset{\tau}{\dashrightarrow} \beta';\gamma';\kappa' \qquad \text{– also } [\text{call}_{Rt}]$$

$$[\text{sync}_{Lt}]\ \beta;\gamma;\kappa \vdash r_1 : \boxed{\alpha_1}_{j_1}.P \parallel r_2[J_2] : \boxed{\alpha_2}_{j_2}$$
$$\xrightarrow{\tau} \mathsf{reset}^{r_2}_{j_2}(\beta');\gamma';\kappa' \vdash r_1 : \alpha_1.\boxed{P}_{j_1} \parallel r_2[J_2\cup\{j_2\}] : \alpha_2$$
$$\text{if } \beta;\gamma;\kappa \vdash r_1[j_1] : \alpha_1 \parallel r_2[j_2] : \alpha_2 \overset{\tau}{\dashrightarrow} \beta';\gamma';\kappa' \text{ – also } [\text{sync}_{Rt}] \text{ and } [\text{sync}_{tt}]$$

Table 4. Operational semantics: replicator rules.

replicator entry on the left (resp. on the right) with a local process on the right (resp. on the left), and intertwines the communication with the spawn of a new thread (i.e. it is a mix of [sync] and [spawn]). The rule $[\text{call}_{LR}]$ is when two replicator entries synchronize. The special case $[\text{call}_{Lt}]$ (resp. $[\text{call}_{Rt}]$) is when the right-hand (resp. left-hand) is the suffix of another thread, which means there is a simultaneous [sync], [spawn] and [term]. Finally, the $[\text{sync}_X]$ rules ($X \in \{Lt, Rt, tt\}$) involve a local process and a termination, or two simultaneous terminations. The redundancy here is the price to pay for *concreteness*, and each rule corresponds to a situation worth studying.

As explained previously, the left-hand side subterm θ in each non-global rule corresponds to a set V_θ of vertices where the local rewrite takes place in the pigraph π. We write $\pi[\theta]$ to identify explicitely this set of vertices in the context of the whole graph π. The unique global-level rule reflects the application of a process or replicator rule at the global level:

Definition 5. *A **global transition** $\beta;\gamma;\kappa \vdash \pi[\theta] \xrightarrow{\mu} \mathsf{gc}(\beta';\gamma';\kappa') \vdash \pi[\theta']$ is produced if $\beta;\gamma;\kappa \vdash \theta \xrightarrow{\mu} \beta';\gamma';\kappa' \vdash \theta'$ can be inferred by either a process or a replicator rule.*

The gc function "garbage collects" all the *inactive* names from the context. This clean-up allows to strictly control the usage of dynamic resources, and it is a key player of our main results (cf. Section 5).

Definition 6. *In a context $\beta;\gamma;\kappa$, the set of **inactive names** is:*
$$\mathsf{inact}(\beta;\gamma;\kappa) \stackrel{\text{def}}{=} \{n \mid (n \in \mathcal{N}_i \cup \mathcal{N}_s \wedge n \notin \mathsf{cod}(\beta)) \vee (n \in \mathcal{N}_o \wedge n \notin \mathsf{cod}(\beta) \wedge \neg(\exists m \in \mathsf{cod}(\beta),\ n =_\gamma m))\}$$

The input and shared names are considered inactive if they are not used in β. On the contrary, it is possible that an output name - not explicitely referenced in β - is indeed equated to another name in γ, which means it is still active. For example, if two names $o!$ and $i?$ are considered equal (i.e $o! =_\gamma i?$), even if $o!$ is not referenced (i.e. $o! \notin \mathsf{cod}(\beta)$), any other $o'!$ must be considered distinct from $i?$ as long as the latter remains active.

Now, the garbage collection function simply consists in removing all the occurrences of the inactive names in the context.

Definition 7. *In a pi-graph context $\beta;\gamma;\kappa$, let $E \stackrel{\text{def}}{=} \mathsf{inact}(\beta;\gamma;\kappa)$, then:*
$$\mathsf{gc}(\beta;\gamma;\kappa) \stackrel{\text{def}}{=} \beta;\gamma';\kappa' \text{ such that } \begin{cases} \gamma' \stackrel{\text{def}}{=} (\emptyset \lhd \{(x,y) \mid x \neq y \wedge x =_\gamma y \wedge x,y \notin E\}, \\ \qquad \{(x,y) \in \gamma^{\neq} \mid x,y \notin E\}) \\ \kappa' \stackrel{\text{def}}{=} \{(x, \kappa(x) \setminus E) \mid x \in \mathsf{dom}(\kappa) \setminus E\} \end{cases}$$

Abstracted transitions Despite their key role in describing the semantics of pi-graphs, the non-committing ϵ-transitions must be abstracted away when comparing behaviors at the global level. However, a "blind" abstraction of ϵ-transitions allows incorrect states to be reached, e.g. a state in which a branch in a choice has been taken (through rule $[\mathrm{sum}_L]$ yielding an ϵ) but with a first action that cannot be committed (i.e. no commitment rule applies). A solution is to enforce a causal dependence on abstracted ϵ-transitions, using the notion of *causal sequence*.

Definition 8. *Let $\Delta_1 \vdash \pi_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} \Delta_n \vdash \pi_n \xrightarrow{\alpha_n} \Delta_{n+1} \vdash \pi_{n+1}$ be a sequence of transitions. The sequence is **causal** iff for any i, $1 \leq i < n$, there exists j, $i < j \leq n$, and $\theta_i, \theta'_i, \theta_j, \theta'_j$, with $\Delta_i \vdash \pi_i[\theta_i] \xrightarrow{\alpha_i} \Delta_{i+1} \vdash \pi_i[\theta'_i]$ and $\Delta_j \vdash \pi_j[\theta_j] \xrightarrow{\alpha_j} \Delta_{j+1} \vdash \pi_j[\theta'_j]$ such that $M^{\pi_i}(V_{\theta'_i}) \cap M^{\pi_j}(V_{\theta_j}) \neq \emptyset$.*

In such a causal sequence, each transition but the last one produces tokens that are necessarily consumed by a further transition in the sequence. In particular the last transition is causally dependent on all the previous ones. The abstraction principle is then expressed as follows:

Definition 9. *An **abstracted transition** $\Delta_1 \vdash \pi_1 \stackrel{\alpha}{\Longrightarrow} \Delta_n \vdash \pi_n$ is inferred iff there is a causal sequence $\Delta_1 \vdash \pi_1 \xrightarrow{\epsilon} \ldots \xrightarrow{\epsilon} \Delta_n \vdash \pi_n \xrightarrow{\alpha} \Delta_{n+1} \vdash \pi_{n+1}$ $(\alpha \neq \epsilon)$.*

The definition says that an abstracted transition with label α corresponds to a path of length n beginning with $n-1$ ϵ-transitions, and ending with a single non-ϵ-transition labelled α. Moreover, all the intermediate invisible transitions are causally needed to produce the visible one. The following strengthens the definition.

Theorem 2. *Every causal ϵ-path starting from a state $\Delta \vdash \pi$ is finite.*

Proof. The syntax requires that the each replicator process is prefixed with an action, which cannot yield an ϵ-transition; hence an ϵ-path cannot spawn threads. Moreover, only finite sequences of ϵ can be triggered by a given thread (in fact, a bound can be easily computed from the syntax). Finally, causal paths involve at most the interleaving of two threads (in the case of [com] commitments), which allows to conclude □

Stateful bisimilarity The standard notion of bisimulation is not sufficient to compare pi-graph behaviours precisely enough. For example a process $[a = b]\,\overline{c}\langle a\rangle$ (put in an adequate replicator) will be distinguished with e.g. $[a = b]\,\overline{c}\langle b\rangle$ because the generated labels, respectively $\overline{c}\langle a\rangle$ and $\overline{c}\langle b\rangle$, are distinct! Bisimulation is indeed a *stateless* principle: only transition labels are compared and states are abstracted away. In a pi-graph, the only part of the state that must be inspected to decide whether labels are equal or not is the partition γ. The following definition makes this precise.

Definition 10. *In a context $\Delta = \beta;\gamma;\kappa$, the labels μ_1 and μ_2 are said equated, which we note $\mu_1 =_\Delta \mu_2$ iff* $\begin{cases} \mu_1 = \tau \wedge \mu_2 = \tau \\ \mu_1 = \overline{c_1}\langle a_1\rangle \wedge \mu_2 = \overline{c_2}\langle a_2\rangle \wedge c_1 =_\gamma c_2 \wedge a_1 =_\gamma a_2 \\ \mu_1 = c_1(x_1) \wedge \mu_2 = c_2(x_2) \wedge c_1 =_\gamma c_2 \wedge x_1 =_\gamma x_2 \end{cases}$

The bisimilarity relation we are looking for is thus a *stateful* variant of the former, defined as follows.

Definition 11. *A **(stateful) bisimulation** S is a symmetric relation on pi-graph states $\Delta \vdash \pi_1$ and $\Delta \vdash \pi_2$ such that $(\Delta \vdash \pi_1, \Delta \vdash \pi_2) \in S$ and $\Delta_1 \vdash \pi_1 \stackrel{\mu_1}{\Longrightarrow} \Delta' \vdash \pi_1'$ imply there is some μ_2 such that $\Delta \vdash \pi_2 \stackrel{\mu_2}{\Longrightarrow} \Delta' \vdash \pi_2'$, $[\mu_1]_\Delta = [\mu_2]_\Delta$ and $(\Delta' \vdash \pi_1', \Delta' \vdash \pi_2') \in S$*

*The **(stateful) bisimilarity** $\sim$ is the largest stateful bisimulation.*

5 Finiteness and boundedness results

Definition 12. *Let π be a pi-graph. We denote $\mathsf{lts}(\pi) \stackrel{\text{def}}{=} \langle Q, T\rangle$ its labelled transition system with Q the states reachable from $\Delta_0 \vdash \pi$, and T the set of triplets of the form $(\Delta' \vdash \pi', \alpha, \Delta'' \vdash \pi'')$ such that we can infer $\Delta' \vdash \pi' \stackrel{\alpha}{\Rightarrow} \Delta'' \vdash \pi''$.*

Replicators allow to characterize infinite systems. For example, the diagram $r[\omega] : a(x).\overline{b}\langle x\rangle$ (with a single replicator) yields an infinite transition system. In the intermediate state after the input and before the output, an unbounded number of threads can accumulate, so the number of reachable states is infinite.

An interesting subclass of the pi-graphs is when the number of threads is bounded, i.e. when in each iterator there is a finite number of tokens, which we denote $T_r^\pi \stackrel{\text{def}}{=} |\mathsf{cod}(M_r^\pi)|$.

Definition 13. *A pi-graph π is **thread-bounded** iff $\forall r \in R^\pi$, $\exists b_r < \omega$ such that $\forall(\Delta' \vdash \pi') \in \mathsf{lts}(\pi)$, $T_r^{\pi'} \leq b_r$. The **thread-bound** is then $\sum_{r\in R^\pi} b_r$.*

The server of Example 1 can be shown thread-bounded with bounds $(1, k, k+1)$, and the thread-bound is thus $2k+2$. These notions are semantic, and are obviously not decidable in the general case. A syntactic condition is *control-finiteness* when each replicator has an explicit finite bound k_r^π.

Definition 14. *A pi-graph π such that $\forall r \in R^\pi,\ k_r^\pi < \omega$ is said **control-finite**.*

Proposition 1. *Let π be a pi-graph. If π is control-finite then π is thread-bounded with thread-bound $\sum_{r \in R^\pi} k_r^\pi$.*

A decisive advantage of the concrete semantics developed for the pi-graphs is that very precise results can be derived. For instance, we can exhibit upper bounds for the amount of the dynamic resources - the number of output, input and shared names - manipulated by thread-bounded pi-graphs. For a state q, we denote $\mathsf{dyn}(q)$ the dynamic names (i.e. in set $\mathcal{N}_i \cup \mathcal{N}_o \cup \mathcal{N}_s$) that occur in q. We also identify an upper bound for the number of variables (input actions) simultaneously active in a given process:

Definition 15. *A process P has **variable-bound** $\mathcal{V}(P)$ such that:*

$$\begin{cases} \mathcal{V}(\alpha) \stackrel{\mathsf{def}}{=} 1 \text{ if } \alpha = c(x), 0 \text{ otherwise} \\ \mathcal{V}(\alpha.P) \stackrel{\mathsf{def}}{=} \mathcal{V}(\alpha) + \mathcal{V}(P),\ \mathcal{V}(\phi P) = \mathcal{V}(P), \mathcal{V}(P_1 + P_2) = \mathsf{max}(\mathcal{V}(P_1), \mathcal{V}(P_2)) \end{cases}$$

We shall denote $\mathcal{V}_r \stackrel{\mathsf{def}}{=} \mathcal{V}(\alpha.P)$ the maximum number of active variables of a replicator $r[k] : \widetilde{\nu a}\ !\alpha.P$. Now we may state an important boundedness result concerning the input and shared private names.

Lemma 1. *Let π be a pi-graph with thread-bounds b_r ($r \in R^\pi$) and $\mathsf{lts}(\pi) = \langle Q, T \rangle$. Then, for all $q \in Q$ we have $|\mathsf{dyn}(q) \cap (\mathcal{N}_i \cup \mathcal{N}_s)| \leq \sum_{r \in R^\pi} b_r \mathcal{V}_r$.*

Proof. The proof proceeds by contraposition and we first only consider the case for input names. Let $B \stackrel{\mathsf{def}}{=} \sum_{r \in R^\pi} b_r \mathcal{V}_r$ and assume there are two states in Q, $q_{n-1} \stackrel{\mathsf{def}}{=} \Delta_{n-1} \vdash \pi_{n-1}$ and $q_n \stackrel{\mathsf{def}}{=} \Delta_n \vdash \pi_n$, such that $|\mathsf{dyn}(q_{n-1}) \cap \mathcal{N}_i| = B$, $|\mathsf{dyn}(q_n) \cap \mathcal{N}_i| = 1 + B$ and we can infer $q_{n-1} \stackrel{c(n?)}{\Longrightarrow} q_n$ induced by the [i-fresh] commitment: $\beta_{n-1}; \gamma_{n-1}; \kappa_{n-1} \vdash \rho[j] : \alpha_n \stackrel{c(n?)}{\dashrightarrow} \beta_n; \gamma_n; \kappa_n$. By Definition 5, the gc function (cf. Def. 7) ensures that $\forall i? \in \mathsf{dyn}(q_{n-1}) \cap \mathcal{N}_i,\ i? \in \mathsf{cod}(\beta_{n-1})$. This implies that the number of active variables in q_{n-1} (i.e. with an image in β_{n-1}) is exactly B. Moreover, the [i-fresh] commitment requires the existence of a variable x of replicator ρ such that x is inactive in q_{n-1} and active in q_n for thread j. More precisely, $\beta_{n-1}(x_j^\rho) = x_j^\rho$ and $\beta_n(x_j^\rho) = n?$. Thus, the number of active variables in q_n would be $B+1$, which leads to a contradiction. In conclusion the state q_n is not reachable and thus $|\mathsf{dyn}(q_n) \cap \mathcal{N}_i| \leq B$. For the shared name the reasoning steps are similar, because the gc function requires the names to exist in $\mathsf{cod}(\beta) \cap \mathcal{N}_s$, and as seen above this codomain is bound by B; moreover, a variable name may not be associated to both an input name and a shared private name, hence the formula $\square$

For the output names the situation is more subtle. First, the restrictions and private names can be associated to output names in the environment β. However, we need to take into account only those names who are objects of output prefixes, hence the following definition.

Definition 16. *The **escape-bound** of:*

- *a pi-graph* $\pi \stackrel{\text{def}}{=} \nu\widetilde{A}\ r_1[k_1] : \nu\widetilde{a_1}!\alpha_1.P_1 \ldots r_n[k_n] : \nu\widetilde{a_n}!\alpha_n.P_n$ *is:*
  $\mathcal{E}_\pi \stackrel{\text{def}}{=} \sum_{A\in\widetilde{A}} \begin{bmatrix} 1 \text{ if } \exists i \text{ s.t } \mathcal{E}^A(\alpha_i.P_i) = 1 \\ 0 \text{ otherwise} \end{bmatrix}$
- *a replicator* $r[k] : \nu\widetilde{a}!\alpha.P$ *is* $\mathcal{E}_r \stackrel{\text{def}}{=} \sum_{a\in\widetilde{a}} \mathcal{E}^a(\alpha.P)$

$$\text{where} \begin{bmatrix} \mathcal{E}^\sigma(\alpha) \stackrel{\text{def}}{=} 1 \text{ if } \alpha = \overline{c}\langle\sigma\rangle,\ 0 \text{ otherwise} \\ \mathcal{E}^\sigma(\alpha.P) \stackrel{\text{def}}{=} 1 \text{ if } \mathcal{E}^\sigma(\alpha) = 1, \mathcal{E}^\sigma(P) \text{ otherwise} \\ \mathcal{E}^\sigma(\phi P) \stackrel{\text{def}}{=} \mathcal{E}^\sigma(P), \mathcal{E}^\sigma(P_1 + P_2) \stackrel{\text{def}}{=} \mathsf{max}(\mathcal{E}^\sigma(P_1), \mathcal{E}^\sigma(P_2)) \end{bmatrix}$$

Note that each restriction νA can escape *at most once* and thus the escape bound of the pi-graph cannot exceed the number of restrictions. Similarly, a (local) private name can escape at most once per spawned thread.

Lemma 2. *Let* π *be a pi-graph with thread-bounds* b_r *(*$r \in R^\pi$*) and* $\mathsf{lts}(\pi) = \langle Q, T\rangle$*. Then, for all* $q \in Q$ *we have* $|\mathsf{dyn}(q) \cap \mathcal{N}_o| \leq \mathcal{E}_\pi + \sum_{r\in R^\pi} b_r(\mathcal{E}_r + \mathcal{V}_r)$*.*

Proof. Beyond the (global and local) escape bounds, the situation appears as more subtle at first sight than for Lemma 1. By Definition 6 an output name $o!$ can exists in the partition γ even if it is not in the codomain of β. This is because the name can serve as a discriminant for an input name $i?$. However, there is at most one output name $o!$ equated to each input name $i?$ in γ, so that those extra cases are limited by the number of variables (not already used for output names), hence the formula derives from a similar proof scheme □

As an illustration our bounds predict that Example 1 generates at most $4k+3$ input names (exactly $k+1$) or shared names (exactly 0), and $4k+4$ output names (exactly $k+1$). The following is a direct consequence of the existence of bounds for the dynamic resources and the fact that we always choose the minimum available identity when generating a dynamic name (cf.[4]).

Lemma 3. *A thread-bounded system is finite-state.*

Finally, this obviously impacts bisimilarity as follows.

Theorem 3. *Bisimilarity is decidable for thread-bounded pi-graphs.*

Beyond decidability, we may remark that bisimilarity checking can exploit the following properties: (1) only (a subset of) the states and transition labels must be inspected (cf. Def. 11), and (2) for control-finite systems, the dynamic resources can be precomputed according to Lemmas 1 and 2.

6 Related and future works

In [3] we introduced a dynamic variant of the pi-graphs based on general graph rewriting techniques, similarly to [6]. The variants discussed in [4, 5] introduced *structural invariance* as in the present paper. Instead of replicators, we used previously an explicit parallel operator and a model of iterated processes. The introduction of replicators has several advantages. First they increase the expressivity of the language, previously restricted to control-finite systems. Moreover, they yield much more compact encodings since multiple threads can share common subgraphs. In [5] we show that pi-graphs represent a good intermediate language when translating pi-calculi into Petri-nets. The adaptation to the replicator-based variant should derive easily, although this is left as a future work. Finally the calculus presented in this paper adopts a standard (although constrained) syntax, which makes it a "real" pi-calculus unlike previous variants.

In [8] a verification framework is proposed for a pi-calculus with general recursion but no match nor mismatch. The translation produces Place/Transition Petri nets with sub-processes (so-called *fragments*) as places and either *reactions* between reachable fragments or communications through public channels as transitions. The translated nets produce the *reduction semantics* of the processes. In comparison we do not perform a semantic but a syntactic (graphical) translation of the pi-calculus, clearly much more compact. Most importantly, we characterize the *transition semantics* of the pi-graphs. Indeed, without the match (and mismatch) and considering the reduction semantics only, then most of the non-trivial components of the semantics disappear: the dynamics resources, hence the causal clock and the partition as well as all the commitment rules except [silent] and [com]. Interesting classes of finitely characterizable infinite systems are proposed in [8]. A further study of such classes in the setting of pi-graphs seems quite appealing. Another semantic translation of pi-calculi is proposed in the *history dependent automata* (HDA) framework [9]. The transitions of HDA provide injective correspondences between names, which ensures locally the freshness of the generated names. In the pi-graphs, the freshness property is enforced at the global level by the use of the causal clock. One advantage of the global approach is the possibility to implement non-trivial phenomena such as read-write causality [7]. We think the pi-graphs also enjoy a simpler notion of bisimilarity than HD-automata, in particular there is no renaming concern and the detection of inactive names is purely local (cf. Definition 6).

Our current work is the development of a verification framework for pi-graphs. Stateful bisimilarity as introduced in this paper seems amenable to the usual bisimulation checking techniques, using a mix of state-based and transition-based algorithms (cf.[10]). For control-finite systems Lemmas 1 and 2 can be exploited for precomputations (e.g."perfect" bitstate hashing), which may thus yield particularly efficient algorithms. Another line of work is the reinterpretation of the graph relabelling semantics such that the LTS states would be plain processes and not graphs. This would offer a simpler setting to demonstrate our conjecture that the pi-graph semantics presented in this paper is fully *compositional*. As a witness, consider the emblematic example of [11]. The pi-calculus

processes $[x = b]\overline{b}\langle b\rangle$ and 0 are both early and late bisimilar, although in a context with x equal to b they are discriminated. In the pi-graphs, the problem does not occur since $[x = b]\overline{b}\langle b\rangle$ produces a transition and is thus *not* (stateful) bisimilar to the deadlocked process.

References

1. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press (1999)
2. Sangiori, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
3. Peschanski, F., Bialkiewicz, J.A.: Modelling and verifying mobile systems using pi-graphs. In: SOFSEM. Volume 5404 of LNCS., Springer (2009) 437–448
4. Peschanski, F., Klaudel, H., Devillers, R.: A decidable characterization of a graphical pi-calculus with iterators. In: Infinity. Volume 39 of EPTCS. (2010) 47–61
5. Peschanski, F., Klaudel, H., Devillers, R.: A Petri net interpretation of open reconfigurable systems. In: Petri Nets. Volume to appear of LNCS. (2011)
6. Gadducci, F.: Graph rewriting for the π-calculus. Mathematical Structures in Computer Science **17** (2007) 407–437
7. Degano, P., Priami, C.: Causality for mobile processes. In: ICALP. Volume 944 of LNCS., Springer (1995) 660–671
8. Meyer, R., Gorrieri, R.: On the relationship between π-calculus and finite place/transition Petri nets. In: CONCUR. Volume 5710 of LNCS., Springer (2009) 463–480
9. Montanari, U., Pistore, M.: History-dependent automata: An introduction. In: SFM. Volume 3465 of LNCS., Springer (2005) 1–28
10. Baier, C., Katoen, J.P.: Chapter 7. Equivalence and Abstraction. In: Principles of Model Checking. MIT Press (2008)
11. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. Acta Inf. **33** (1996) 69–97