



A Petri Net Interpretation of Open Reconfigurable Systems

Frédéric Peschanski¹, Hanna Klaudel², and Raymond Devillers³

¹ UPMC – LIP6

² Université d'Évry–IBISC

³ Université Libre de Bruxelles

Abstract. We present a Petri net interpretation of the pi-graphs - a graphical variant of the pi-calculus. Characterizing labelled transition systems, the translation can be used to reason in Petri net terms about open reconfigurable systems. We demonstrate that the pi-graphs and their translated Petri nets agree at the semantic level. In consequence, existing results on pi-graphs naturally extend to the translated Petri nets, most notably a guarantee of finiteness by construction.

Keywords: Reconfigurable systems, Pi-calculus, Petri nets

1 Introduction

Systems that reconfigure themselves dynamically are ubiquitous: from Internet servers that create and destroy dynamic connections with clients, to mobile systems that scrutinize their surrounding environment to adapt their behaviour dynamically. The pi-calculus [1] is acknowledged as a particularly concise and expressive formalism to specify the dynamic structure of reconfigurable systems.

There is now a relatively rich tradition of translations of pi-calculus variants into Petri nets. One motivation of such studies is the possibility to apply efficient verification techniques on pi-calculus models. Another interest relates to expressivity: which features of the pi-calculus can be translated? What is the abstraction level required from the target Petri net formalism? Existing related research studies roughly decompose in either semantic [2–4] or syntax-driven [5] translations. In the first case, a reachability analysis of the semantics of a given term is performed, and a Petri net is synthesized from this. In the second case, a Petri net is built directly from the syntax. Most of the time, the semantic analysis allows to produce lower level nets. On the other side of the coin, semantic encodings generally cover less features (especially no support for match or mismatch operators; *i.e.*, without comparison of names) and often only apply on *reduction*, or *chemical* semantics for *closed systems*. With the increased expressivity of high-level nets, it is possible to support more constructs. Most importantly, it is possible to capture the semantics of *open systems* by translating the richer *labelled transition semantics* which is most customary in studies about the pi-calculus. To quote Robin Milner: “*you can’t do behavioural analysis*”

with the chemical semantics [...] I think the strength of labels is that you get the chance of congruential behaviours” [6]. More prosaically, labelled transition semantics enables compositional reasoning about partial specifications.

In this paper, we continue our work about translating rich pi-calculus variants in labelled transition semantics [5]. By introducing a “translation-friendly” variant of the pi-calculus, namely the pi-graphs [7, 8], we are able to provide a much simpler syntax-driven translation to one-safe coloured Petri nets. The translation supports most of the constructs of the original pi-calculus, including the match and mismatch operators with their non-trivial semantics. For non-terminating behaviours we use a notion of *iterator* along the line of [9]. The semantics for the intermediate calculus rely on graph relabelling techniques (hence the name pi-graphs) but in the present paper, to avoid confusion with the Petri net semantics, we provide a more abstract presentation. An important result of the paper is that these semantics are in complete *agreement*. In consequence, existing results on pi-graphs (cf. [8]) naturally extend to the translated Petri nets, most notably a guarantee that only finite-state systems can be constructed.

The outline of the paper is as follows. In Section 2 we describe the syntax and informal semantics of the pi-graph calculus. In Section 3 we present the operational semantics more formally. Section 4 presents the translation to Petri nets, and discusses its main properties. The related and future works are discussed in Section 5.

2 The process calculus

The pi-graph process calculus is a variant of the pi-calculus that enjoys a natural graphical interpretation⁴ hence its name. The syntax is based on prefixes, processes, iterators and graphs. The prefixes are as follows :

$$p ::= \tau \mid \bar{c}\langle a \rangle \mid c(x) \mid \sum [P_1 + \dots + P_n] \mid \prod [P_1 \parallel \dots \parallel P_n] \quad (n > 1)$$

The first three elements correspond to the standard action prefixes of the pi-calculus: silent action τ , output $\bar{c}\langle a \rangle$ of name a on channel c , and input $c(x)$ through channel c of a name bound to the variable x . The remaining two elements are an n -ary non-deterministic choice operator \sum between a set of possible processes P_1, \dots, P_n and an n -ary parallel operator \prod . Unlike most variants of the pi-calculus, these are not considered as top-level binary operators. The reason is that the sub-processes of choice and parallel must be *terminated* (cf. below). The syntax for processes is as follows:

$$P, Q, P_i ::= p.P \mid [a = b]P \mid [a \neq b] P \mid p.0$$

A process can be constructed inductively by prefixing another process P by a prefix p , denoted $p.P$. The match $[a = b] P$ (resp. mismatch $[a \neq b] P$) is such

⁴ The graphical interpretation of pi-graphs is not detailed in the present paper to avoid any confusion with the translated Petri nets. This interpretation is discussed at length in previous publications [7, 8].

that the process P is only enabled if the names a and b are proved equal (resp. inequal). Finally, the construction $p.0$ corresponds to the suffixing of a prefix p by a *terminator* 0 . It is the only way to terminate a prefixed sequence. Moreover, no match or mismatch is allowed in front of 0 , for semantic reasons (cf. [8]).

The top-level components of pi-graphs are iterators, which allow to encode control-finite recursive behaviours without relying on explicit recursive calls. The syntax of iterators is as follows:

$$\mathcal{I} ::= I : (\nu a_1, \dots, \nu a_n) * P \text{ with } a_1, \dots, a_n \text{ names, } P \text{ a process and } I \text{ a label}$$

The behaviour of an iterator is the repeated execution of its main process P in which the names a_1, \dots, a_n are considered *locally private*.

Iterators can be composed to form a pi-graph with the following syntax:

$$\pi ::= (\nu A_1, \dots, \nu A_m) \mathcal{I}_1 \parallel \dots \parallel \mathcal{I}_p$$

with A_1, \dots, A_m names and $\mathcal{I}_1, \dots, \mathcal{I}_p$ iterators

The names A_1, \dots, A_m are considered *globally private* in all the iterators $\mathcal{I}_1, \dots, \mathcal{I}_p$. These (roughly) play the role of static restrictions as in CCS, whereas locally private names (i.e. $\nu a_1, \dots, \nu a_n$) correspond to the (dynamic) restrictions of the pi-calculus. Iterators are implicitly executed in parallel.

To illustrate the syntax constructors and their informal meaning, we consider the example of a simple server mode.

Example 1. a sequential server (empty parentheses are omitted)

$$\begin{aligned} & (\nu \text{Start}, \nu \text{Run}, \nu \text{End}) \\ \text{Service} & : * \text{Run}(s).s(m). \sum \left[\begin{array}{l} [m = s] \overline{\text{End}}(s).0 \\ + [m \neq s] \tau.\overline{\text{Start}}(s).0 \end{array} \right].0 & \parallel \\ \text{Handler} & : * \text{Start}(s).\overline{\text{Run}}(s).0 & \parallel \\ \text{Server} & : (\nu \text{session}) * \text{adr}(c).\overline{c}(\text{session}).\overline{\text{Start}}(\text{session}).\text{End}(q).0 \end{aligned}$$

The depicted pi-graph has three iterators: the main server process, a session handler and a service delivered to clients. The server initially expects a connection request from a client on the public channel adr . A (locally) private session channel is then sent back to the connected client and an instance of the service is started through the session handler. The (globally) private channels - Start , Run and End - are used to confine the server model, disallowing external interferences. When a request is emitted by a client (through channel s in the Service iterator) two cases are discriminated. If the received message m is the session channel s itself, then this indicates the end of the session. The control is passed back to the Server iterator using the (globally private) End channel. Otherwise, an internal activity is recorded (as a silent step τ) and the service is reactivated by the Handler component.

Note that in this example we only specify the server infrastructure. We see the model as an open system independent from any particular client model.

3 Operational semantics

The operational semantics of pi-graphs is based on *graph relabelling*, a simplistic form of *graph rewriting* where nodes and edges can have their content updated but neither created nor deleted. To avoid any confusion with the Petri net semantics - also a form of graph relabelling - we adopt in this paper a process calculus presentation. As a preliminary, we require a precise definition of what is a name in a pi-graph.

Definition 1. *The set of **names** is $\mathcal{N} \stackrel{\text{def}}{=} \mathcal{N}_f \uplus \mathcal{N}_v \uplus \mathcal{N}_r \uplus \mathcal{N}_p \uplus \mathcal{N}_o \uplus \mathcal{N}_i$ with:*

$$\left[\begin{array}{l} \mathcal{N}_f \text{ the set of free names } a, b, \dots \\ \mathcal{N}_v \text{ the set of variables } x^I, y^J, \dots \text{ with } I, J \text{ iterator labels} \\ \mathcal{N}_r \text{ the set of restrictions } \nu A, \nu B, \dots \\ \mathcal{N}_p \text{ the set of private names } \nu^I a, \nu^J b, \dots \text{ with } I, J \text{ iterator labels} \\ \mathcal{N}_o \stackrel{\text{def}}{=} \{n! \mid n \in \mathbb{N}\} \text{ the set of output names} \\ \mathcal{N}_i \stackrel{\text{def}}{=} \{n? \mid n \in \mathbb{N}\} \text{ the set of input names} \end{array} \right.$$

We define $\text{Priv} \stackrel{\text{def}}{=} \mathcal{N}_r \cup \mathcal{N}_p$ (private names) and $\text{Pub} \stackrel{\text{def}}{=} \mathcal{N} \setminus \text{Priv}$ (public names)

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. They play a prominent role in the model.

3.1 Terms, context and initial state

The semantics manipulates terms of the form $\Gamma \vdash \pi$ where π is a pi-graph with names globalized (cf. below) and $\Gamma = \beta; \gamma; \kappa$ is a global context with:

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

The operations available on the context are formalized in Table 1. We will discuss these in the remainder of the section.

The initial state and context is denoted $\beta_0; \gamma_0; \kappa_0 \vdash \pi_0$ where π_0 is a syntactic term with globalized names. If the syntactic term is $(\nu A_1, \dots, \nu A_m) \mathcal{I}_1 \parallel \dots \parallel \mathcal{I}_p$ then we transform it as $(\mathcal{I}_1 \parallel \dots \parallel \mathcal{I}_p) \{ \nu A_1 / A_1, \dots, \nu A_m / A_m \}$, i.e. each restricted bound name A_i is replaced by an explicit restricted occurrence νA_i in the set \mathcal{N}_r . Moreover, we rewrite each iterator of the form $I_j : (\nu a_1, \dots, \nu a_n) * P_j$ as $I_j : *P_j \{ \nu^{I_j} a_1 / a_1, \dots, \nu^{I_j} a_n / a_n \}$ where each name $\nu^{I_j} a_k$ is in \mathcal{N}_p . As such, we encode at the global level the local nature of these private names. Finally we repeat this process for each input variable x of iterator I_j , renamed (as well as its occurrences) as x^{I_j} in set \mathcal{N}_v . We thus end up with a term containing only names with global scope but ranging from disjoint subsets of \mathcal{N} (all iterator labels are assumed different).

In the initial state, the instantiation function associates each name of π_0 with itself, i.e. $\beta_0 \stackrel{\text{def}}{=} \{n \mapsto n \mid n \text{ a name of } \pi_0\}$. By default, there are no explicit name

Instantiation	β
update	$\beta_{\triangleleft x \mapsto y} \stackrel{\text{def}}{=} (\beta \setminus \{x \mapsto \beta(x)\}) \cup \{x \mapsto y\}$
Causal clock	κ
fresh output	$\text{out}(\kappa) \stackrel{\text{def}}{=} \kappa \cup \{\text{next}_o(\kappa)! \mapsto \emptyset\}$
fresh input	$\text{in}(\kappa) \stackrel{\text{def}}{=} \{o \mapsto (\kappa(o) \cup \{\text{next}_i(\kappa)?\}) \mid o \in \text{dom}(\kappa)\}$
freshness (input)	$\text{next}_i(\kappa) \stackrel{\text{def}}{=} \min(\mathbb{N}^+ \setminus \{n \mid n? \in \bigcup \text{cod}(\kappa)\})$
freshness (output)	$\text{next}_o(\kappa) \stackrel{\text{def}}{=} \min(\mathbb{N}^+ \setminus \{n \mid n! \in \text{dom}(\kappa)\})$
r-w causality	$x \prec_{\kappa} y \stackrel{\text{def}}{=} x \in \text{dom}(\kappa) \wedge y \in \kappa(x)$
Partitions	$\gamma \stackrel{\text{def}}{=} (\gamma^{\bar{\cdot}}, \gamma^{\neq})$
match	$x =_{\gamma} y \text{ iff } \begin{cases} x = y \vee (x, y) \in \gamma^{\bar{\cdot}} \vee (y, x) \in \gamma^{\bar{\cdot}} \\ \vee \exists z, x =_{\gamma} z \wedge z =_{\gamma} y \end{cases}$
mismatch	$x \neq_{\gamma}^{\kappa} y \text{ iff } \begin{cases} (x, y) \in \gamma^{\neq} \vee (y, x) \in \gamma^{\neq} \vee (x, y \in \text{Priv} \cup \mathcal{N}_o \wedge x \neq y) \\ \vee (x \in \mathcal{N}_f \wedge y \in \text{Priv} \cup \mathcal{N}_o) \vee (y \in \mathcal{N}_f \wedge x \in \text{Priv} \cup \mathcal{N}_o) \\ \vee (x \in \mathcal{N}_o \wedge y \in \mathcal{N}_i \wedge x \not\prec_{\kappa} y) \vee (y \in \mathcal{N}_o \wedge x \in \mathcal{N}_i \wedge y \not\prec_{\kappa} x) \\ \vee (\exists z_1 z_2, x =_{\gamma} z_1 \wedge z_1 \neq_{\gamma}^{\kappa} z_2 \wedge z_2 =_{\gamma} y) \end{cases}$
compatibility	$x \overset{\sim}{\prec}_{\kappa} y \text{ iff } \begin{cases} \neg(x \neq_{\gamma}^{\kappa} y) \wedge (x, y \in \mathcal{N}_f \cup \mathcal{N}_i \\ \vee (x \in \mathcal{N}_o \wedge y \in \mathcal{N}_i \wedge x \prec_{\kappa} y) \vee x =_{\gamma} y \vee y \overset{\sim}{\prec}_{\kappa} x) \end{cases}$
refine	$\gamma_{\triangleleft x=y} \stackrel{\text{def}}{=} (\gamma^{\bar{\cdot}} \cup \{(x, y)\}, \gamma^{\neq}) \text{ if } \neg(x =_{\gamma} y), \gamma \text{ otherwise}$
discriminate	$\gamma_{\triangleleft x \neq y} \stackrel{\text{def}}{=} (\gamma^{\bar{\cdot}}, \gamma^{\neq} \cup \{(x, y)\}) \text{ if } \neg(x \neq_{\gamma}^{\kappa} y), \gamma \text{ otherwise}$

Table 1. The global context and associated operations.

(in)equality so the partitions are empty, i.e. $\gamma_0 \stackrel{\text{def}}{=} (\emptyset, \emptyset)$. Finally, the initial clock is also empty, i.e. $\kappa_0 \stackrel{\text{def}}{=} \emptyset$. The initial state of the server example (cf. Example 1) is depicted below.

Example 2. the sequential server with global (and shortened) names

$$\begin{aligned}
 & \beta_0; \gamma_0; \kappa_0 \vdash \\
 V : & \boxed{*} \nu \text{Run}(x^V).x^V(y^V). \sum \left[\begin{array}{l} [y^V = x^V] \bar{\nu} \text{End}\langle x^V \rangle.0 \\ + [y^V \neq x^V] \tau. \bar{\nu} \text{Start}\langle x^V \rangle.0 \end{array} \right].0 \quad \parallel \\
 H : & \boxed{*} \nu \text{Start}(x^H). \bar{\nu} \text{Run}\langle x^H \rangle.0 \quad \parallel \\
 S : & \boxed{*} a(x^S).x^S \langle \nu^S \text{ses} \rangle. \bar{\nu} \text{Start}\langle \nu^S \text{ses} \rangle. \nu \text{End}(q^S).0
 \end{aligned}$$

To model control-flow we use boxes to surround the active parts of pi-graphs. A prefix denoted \boxed{p} is considered active, and it is said to be a *redex*. Anticipating on the Petri net semantics, the redex prefixes will be associated to places with an active control token inside. The left-part of a match $[a = b] P$ (resp. mismatch $[a \neq b] P$) can also be a redex, which is denoted $\boxed{[a = b]} P$ (resp. $\boxed{[a \neq b]} P$). The 0 suffix of a process can also be a redex $\boxed{0}$, which means the process has terminated. Finally, an iterator $I : *P$ can be a redex, denoted $I : \boxed{*} P$. This means the iterator is ready to execute a (first or) new iteration of P .

A process in its initial state is denoted \boxed{P} such that:

$$\boxed{p.P} \stackrel{\text{def}}{=} \boxed{p} P, \quad \boxed{p.0} \stackrel{\text{def}}{=} \boxed{p} 0 \quad \text{and} \quad \boxed{[a = b] P} \stackrel{\text{def}}{=} \boxed{[a = b]} P \quad (\text{resp. } \neq).$$

A process is thus in its initial state if its initial prefix is a redex. We also introduce a complementary notation for a process in its terminal state, which we denote \boxed{P} , when the terminator of a process is a redex, i.e:

$$\boxed{pP} \stackrel{\text{def}}{=} p\boxed{P}, \quad \boxed{p0} \stackrel{\text{def}}{=} p\boxed{0} \quad \text{and} \quad \boxed{[a = b]P} \stackrel{\text{def}}{=} [a = b]\boxed{P} \quad (\text{resp. } \neq).$$

In the initial state, the term π_0 has all and only its iterators marked as redex, i.e. π_0 is of the form $I_1 : \boxed{[*]P_1} \parallel \dots \parallel I_p : \boxed{[*]P_p}$, cf. Example 2 above.

3.2 The semantic rules

The operational semantics rules are listed in Table 2. Each rule is of the form: $\beta; \gamma; \kappa \vdash \theta \xrightarrow{\alpha} \beta'; \gamma'; \kappa' \vdash \theta'$ where θ is a subterm of a pi-graph π with some redex(es) inside⁵. We write $\pi[\theta]$ to denote π with the designated subterm θ . The local application of a rule to the subterm θ is reflected in the global pi-graph π as follows:

Definition 2. A *global transition* $\beta; \gamma; \kappa \vdash \pi[\theta] \xrightarrow{\alpha} \text{gc}(\beta'; \gamma'; \kappa') \vdash \pi[\theta']$ is inferred iff $\beta; \gamma; \kappa \vdash \theta \xrightarrow{\alpha} \beta'; \gamma'; \kappa' \vdash \theta'$ is provable

In a global transition, the label α can be either a low-level rewrite ϵ , a silent step τ , an output $\bar{c}\langle a \rangle$ or an input $c(x)$. In the subterm θ' after the rewrite, the only possibility is a move of redex(es): i.e. an evolution of the control graph. Moreover, the rules are *local*, no other subterm of π is changed. The global context, however, can be updated through a rewrite. First, the gc function “garbage collects” all the *inactive* names from the context. This clean-up is required to ensure the finiteness of the model (cf. [8]).

Definition 3. In a global context $\beta; \gamma; \kappa$, the set of *inactive names* is:

$$\text{inact}(\beta; \gamma; \kappa) \stackrel{\text{def}}{=} \{n \mid (n \in \mathcal{N}_i \wedge n \notin \text{cod}(\beta)) \vee (n \in \mathcal{N}_o \wedge n \notin \text{cod}(\beta) \wedge \neg(\exists m \in \text{cod}(\beta), n =_{\gamma} m))\}$$

All the static names, i.e. the names in $\mathcal{N} \setminus (\mathcal{N}_i \cup \mathcal{N}_o)$ are considered active. An input name n is considered inactive if and only if it is not instantiated in β . For an output name the situation is slightly more complicated because it is possible that the name has been equated to another name in γ , which means that even if it is not instantiated its existence may be required. For example, if two names $o!$ and $i?$ are considered equal (i.e. $o! =_{\gamma} i?$), even if $o!$ is not instantiated (i.e. $o! \notin \text{cod}(\beta)$), any other $o'!$ must be considered distinct from $i?$ as long as the latter remains itself instantiated.

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

⁵ The shape of a subterm θ in a pi-graph $\pi[\theta]$ follows the left-hand side of the semantic rules (cf. Table 2). The only non-trivial case is for the [sync] rule because the subterm involves two separate sub-processes, potentially in distinct iterators. In [8] such a subterm indeed corresponds to a subgraph.

[silent]	$\beta; \gamma; \kappa \vdash \boxed{\tau}P \xrightarrow{\tau} \beta; \gamma; \kappa \vdash \tau \boxed{P}$
[out]	$\beta; \gamma; \kappa \vdash \boxed{\bar{\Phi}\langle\Delta\rangle}P \xrightarrow{\beta(\bar{\Phi})\langle\beta(\Delta)\rangle} \beta; \gamma; \kappa \vdash \bar{\Phi}\langle\Delta\rangle \boxed{P} \quad \text{if } \beta(\bar{\Phi}), \beta(\Delta) \in \text{Pub}$
[o-fresh]	$\beta; \gamma; \kappa \vdash \boxed{\bar{\Phi}\langle\Delta\rangle}P \xrightarrow{\beta(\bar{\Phi})\langle\text{next}_0(\kappa)!\rangle} \beta_{\triangleleft\Delta \mapsto \text{next}_0(\kappa)}; \gamma; \text{out}(\kappa) \vdash \bar{\Phi}\langle\Delta\rangle \boxed{P}$ if $\beta(\bar{\Phi}) \in \text{Pub}, \beta(\Delta) \in \text{Priv}$
[i-fresh]	$\beta; \gamma; \kappa \vdash \boxed{\Phi(x)}P \xrightarrow{\beta(\bar{\Phi})\langle\text{next}_i(\kappa)!\rangle} \beta_{\triangleleft x \mapsto \text{next}_i(\kappa)}; \gamma; \text{in}(\kappa) \vdash \Phi(x) \boxed{P} \quad \text{if } \beta(\bar{\Phi}) \in \text{Pub}$
[match]	$\beta; \gamma; \kappa \vdash \boxed{[\Phi = \Delta]}P \xrightarrow{\varepsilon} \beta; \gamma_{\triangleleft \beta(\Phi) = \beta(\Delta)}; \kappa \vdash [\Phi = \Delta] \boxed{P} \quad \text{if } \beta(\bar{\Phi}) \sim_{\kappa} \beta(\Delta)$
[miss]	$\beta; \gamma; \kappa \vdash \boxed{[\Phi \neq \Delta]}P \xrightarrow{\varepsilon} \beta; \gamma_{\triangleleft \beta(\Phi) \neq \beta(\Delta)}; \kappa \vdash [\Phi \neq \Delta] \boxed{P} \quad \text{if } \neg(\beta(\bar{\Phi}) =_{\gamma} \beta(\Delta))$
[sync]	$\beta; \gamma; \kappa \vdash \boxed{\bar{\Phi}\langle\Delta\rangle}P \parallel \boxed{\Phi'(x)}Q$ $\xrightarrow{\tau} \beta_{\triangleleft x \mapsto \beta(\Delta)}; \gamma_{\triangleleft \beta(\Phi) = \beta(\Phi')}; \kappa \vdash \bar{\Phi}\langle x \rangle \boxed{P} \parallel \Phi'(x) \boxed{Q} \quad \text{if } \beta(\bar{\Phi}) \sim_{\kappa} \beta(\Phi')$
[sum]	$\beta; \gamma; \kappa \vdash \sum [P_1 + \dots + P_i + \dots + P_n]Q \xrightarrow{\varepsilon} \beta; \gamma; \kappa \vdash \sum [P_1 + \dots + \boxed{P_i} + \dots + P_n]Q$
[sum ₀]	$\beta; \gamma; \kappa \vdash \sum [P_1 + \dots + \boxed{P_i} + \dots + P_n]Q \xrightarrow{\varepsilon} \beta; \gamma; \kappa \vdash \sum [P_1 + \dots + P_i + \dots + P_n] \boxed{Q}$
[par]	$\beta; \gamma; \kappa \vdash \prod [P_1 \parallel \dots \parallel P_i \parallel \dots \parallel P_k]Q \xrightarrow{\varepsilon} \beta; \gamma; \kappa \vdash \prod [\boxed{P_1} \parallel \dots \parallel \boxed{P_i} \parallel \dots \parallel \boxed{P_k}]Q$
[par ₀]	$\beta; \gamma; \kappa \vdash \prod [\boxed{P_1} \parallel \dots \parallel \boxed{P_i} \parallel \dots \parallel \boxed{P_k}]Q \xrightarrow{\varepsilon} \beta; \gamma; \kappa \vdash \prod [P_1 \parallel \dots \parallel P_i \parallel \dots \parallel P_k] \boxed{Q}$
[iter]	$\beta; \gamma; \kappa \vdash I : \boxed{*}P \xrightarrow{\varepsilon} \beta; \gamma; \kappa \vdash I : * \boxed{P}$
[iter ₀]	$\beta; \gamma; \kappa \vdash I : * \boxed{P} \xrightarrow{\varepsilon} \beta_{\triangleleft (\cup_{\Phi I \in \text{dom}(\beta)} \Phi I \mapsto \Phi I)}; \gamma; \kappa \vdash I : * \boxed{P}$

Table 2. The operational semantics rules.

Definition 4. In a pi-graph context $\beta; \gamma; \kappa$, let $E \stackrel{\text{def}}{=} \text{inact}(\beta; \gamma; \kappa)$.

Then $\text{gc}(\beta; \gamma; \kappa) \stackrel{\text{def}}{=} \beta; \gamma'; \kappa'$ such that
$$\begin{cases} \gamma' \stackrel{\text{def}}{=} (\{(x, y) \mid x =_{\gamma} y \wedge x, y \notin E\}, \\ \quad \{(x, y) \in \gamma \neq \mid x, y \notin E\}) \\ \kappa' \stackrel{\text{def}}{=} \{(x, \kappa(x) \setminus E) \mid x \in \text{dom}(\kappa) \setminus E\} \end{cases}$$

We now proceed to the individual description of the semantics rules.

The [silent] rule describes a subterm of π with a prefix τ as a redex. In such a situation, a transition through τ may be fired, leading to a state π' identical to π except that the continuation process P is activated.

The [par] rule is similar but the control-flow is now duplicated, simulating the fork of parallel processes. The latter works in conjunction with the [par₀] rule, which waits for all the forked processes to terminate before passing the control to the continuation Q . The non-deterministic choice among a set of possible sub-processes is implemented by the [sum] and [sum₀] rules. Unlike parallel, only one branch is non-deterministically activated, and only the termination of this particular branch is required to activate the continuation of the choice.

The [iter] rule explains the start of a new iteration of an iterator named I . At the end of the iteration the rule [iter₀] reactivates the iterator so that a

next iteration can start. The important step is the reinitialization of the names instantiated locally during the last iteration (cf. below).

As an illustration, we consider the example of an infinite generator of fresh names:

$$\{\nu^I a \mapsto \nu^I a\}; \emptyset; \emptyset \vdash I : \boxed{*} \bar{c}\langle \nu^I a \rangle.0$$

The only applicable rule is [iter] to start a new iteration, which yields:

$$\xrightarrow{\epsilon} \{\nu^I a \mapsto \nu^I a\}; \emptyset; \emptyset \vdash I : * \boxed{\bar{c}\langle \nu^I a \rangle}.0$$

We are now in a state where a private name must be sent over a public channel, a situation handled by the [o-fresh] rule. An important remark is that after the emission, the name $\nu^I a$ cannot be considered private anymore but *shared*. We implement the sharing of a private name by the instantiation of a fresh output name in set \mathcal{N}_o . To ensure freshness, we use the clock component κ of the context. The identity of the name is denoted $\text{next}_o(\kappa)!$. In Table 1 we see that the principle is to take the least strictly positive integer that is not already in the domain of κ . Moreover, the clock is updated so that the newly generated identity is recorded, which is denoted $\text{out}(\kappa)$ and simply consists in adding the new output name in the domain of the clock. In our running example, the clock is initially empty so the generated identity is 1!. The whole transition is depicted below:

$$\xrightarrow{\bar{c}\langle 1! \rangle} \{\nu^I a \mapsto 1!\}; \emptyset; \{1! \mapsto \emptyset\} \vdash I : * \bar{c}\langle \nu^I a \rangle. \boxed{0}$$

In this state the iteration terminates by [iter₀] and we must not forget to reinitialize the local private name $\nu^I a$ and apply garbage collection, which gives:

$$\xrightarrow{\epsilon} \{\nu^I a \mapsto \nu^I a\}; \emptyset; \emptyset \vdash I : \boxed{*} \bar{c}\langle \nu^I a \rangle.0$$

We are back in the initial state and we remark that the infinite fresh name generator is characterized finitely.

The rule [out] is a simpler variant triggered when a process emits a public name on a public channel. The rule [i-fresh] (fresh input) is quite similar to [o-fresh]. When a name is received from the environment, a fresh identity $\text{next}_i(\kappa)?$ is generated for it. The clock is updated using $\text{in}(\kappa)$ which adds the fresh input in the co-domains of all the existing output names (i.e. the domain of κ). This records *read-write causality* [10] that we illustrate together with the [match] rule below.

$$\begin{aligned} & \beta, \nu^I a \mapsto \nu^I a, x^I \mapsto x^I; \emptyset; \emptyset \vdash I : * \boxed{\bar{c}\langle \nu^I a \rangle}.d(x^I).[\nu^I a = x^I] P \\ & \xrightarrow{\bar{c}\langle 1! \rangle} \beta, \nu^I a \mapsto 1!, x^I \mapsto x^I; \emptyset; \{1! \mapsto \emptyset\} \vdash I : * \bar{c}\langle \nu^I a \rangle. \boxed{d(x^I)}. [\nu^I a = x^I] P \\ & \xrightarrow{\bar{d}\langle 1? \rangle} \beta, \nu^I a \mapsto 1!, x^I \mapsto 1?; \emptyset; \{1! \mapsto \{1?\}\} \vdash I : * \bar{c}\langle \nu^I a \rangle. d(x^I). \boxed{[\nu^I a = x^I]} P \end{aligned}$$

In the first step the [o-fresh] rule is triggered because we send a private name on a public channel. In consequence the private name $\nu^I a$ is instantiated by a fresh output 1!. The clock is also updated to record this fact. In the second step the

[i-fresh] rule instantiates for x^I the fresh input $1?$. The read-write causal link between the output $1!$ and the subsequent input $1?$ is recorded in the clock. The rationale is that it is indeed possible to receive an instance of $1!$ as $1?$ because the former is shared. Actually, the two names may be equated, which in term of Table 1 would be written $1! \overset{\gamma}{\sim}_{\kappa} 1?$ because $1! \prec_{\kappa} 1?$ (i.e. $1? \in \kappa(1!)$) where κ is the current clock and γ the partition. In the next step, the [match] rule is enabled and thus we can infer the following transition:

$$\xrightarrow{\epsilon} \beta'; \{1! = 1?\}; \{1! \mapsto \{1?\}\} \vdash I : *c\langle \nu^I a \rangle . d(x^I) . [\nu^I a = x^I] \boxed{P}$$

(where the instantiations β' remain unchanged)

The match has been effected and in the continuation P (left undetailed), the names $1!$ and $1?$ (and thus the occurrences of $\nu^I a$ and x^I) are considered equal. If we inverse the input and the output, then the input $1?$ will appear before the output $1!$, and thus in the clock the two names will not be related. In consequence the final match would not be enabled and P could not be reached. We refer to [8] for a more thorough discussion about these non-trivial aspects.

The proposed dynamic interpretation of match suggests a similar treatment for mismatch, which is implemented by the [miss] rule. First, a mismatch between two names x, y is only possible if they are not provably equal (i.e. $x =_{\gamma} y$). There are then more possibilities for x, y to be inequal (denoted $x \neq_{\gamma}^{\kappa} y$). First they are inequal if either the name are explicitly distinguished in $\gamma \neq$ or they are distinct private names. Another case is if one name is a (public) free name (in \mathcal{N}_f) and the other one is a private or an output name. The most non-trivial case is if one name is an input and the other one is an output, which makes them provably distinct if they are not causally related in κ (hence the causal clock is required). Finally, we must be careful not to forget about the interaction between equality and inequality. In all the other cases the mismatch leads to the explicit addition of a discriminating pair in $\gamma \neq$.

Finally, the rule [sync] is for a communication taking place internally in a pi-graph. The subterm triggering the rule is composed of a pair of redexes: an output in one process and an input in another process (potentially from two distinct iterators). The effect of the rule is a communication from the output to the input process, the received name being instantiated in β . Because we need to characterize open systems, a non-trivial aspect here is that the communication can be triggered on distinct channel names, under the constraint that the match between the two names can be performed. This makes name matching and synchronization intimately related in the proposed semantics.

3.3 Abstracted transitions

An important aspect of the pi-graphs semantics is the possibility to abstract away from low-level ϵ -transitions. In [8] we propose an inductive rule that allows to omit the ϵ -transitions altogether, blindly. The main problem with this inductive principle is that it does not translate easily to the semantics of Petri nets. In this paper we use an alternative approach, which is to allow to branch directly on

ϵ -transitions in non-deterministic choices. At first sight, this may lead to an incorrect situation where a deadlock may result from an unobservable ϵ -transition. To overcome this problem we introduce a *causal* principle of abstraction.

Definition 5. Let $\Gamma_1 \vdash \pi_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} \Gamma_n \vdash \pi_n \xrightarrow{\alpha_n} \Gamma_{n+1} \vdash \pi_{n+1}$ be a sequence of transitions, with θ_i, θ'_i the subterms inducing the i -th transition ($1 \leq i \leq n$), following Definition 2. This sequence will be said **causal** iff $\forall i, 1 \leq i < n, \exists j, i < j \leq n$ such that $\theta'_i \cap \theta_j \neq \emptyset$, i.e., the i -th transition produces a redex (maybe many) needed by the j -th one.

In such a causal sequence, the first transitions are all causally needed to produce the next ones, and in particular the last transition causally uses all the previous ones. The abstraction principle is then expressed as follows:

Definition 6. An **abstracted transition** $\Gamma_1 \vdash \pi_1 \xrightarrow{\alpha} \Gamma_n \vdash \pi_n$ is inferred iff there is a causal sequence $\Gamma_1 \vdash \pi_1 \xrightarrow{\epsilon} \dots \xrightarrow{\epsilon} \Gamma_n \vdash \pi_n \xrightarrow{\alpha} \Gamma_{n+1} \vdash \pi_{n+1}$ ($\alpha \neq \epsilon$).

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. This abstracts from invisible transitions and gets rid of the invisible transitions not causally needed to perform a visible one, which excludes to choose a branch in a choice leading to a deadlock through ϵ transitions. The definition is sound because we show in [8] that every ϵ -path is of finite (bounded, in fact) length.

4 Translation to Petri nets

The key component of our proposition is the translation of the pi-graphs and their semantics into (concise) Petri nets. As a matter of fact, the pi-graph semantics has been designed to implicitly address the most complex issues of such a translation (control-flow, freshness, read-write causality, etc.). In this section we thus mostly assemble the pieces of the puzzle.

4.1 Petri net class and transition rule

Our translation requires a relatively simple class of coloured Petri nets. As usual in coloured models, places have types, arcs have annotations with variables and constants, and transitions have labels and guards. A particularity of these nets is that they accept only 1-safe markings. More precisely each place always has a unique token (possibly the “empty” token). Moreover, all the arcs are bidirectional and labelled by pairs R/W intended to *read* from the adjacent place a token $\rho(R)$ and *write* an updated token $\rho(W)$ (where ρ is a *binding* function for the variables in adjacent arcs and in the guard of the transition).

Definition 7. A (coloured) Petri net is a tuple $N = (S, T, U, G; M)$, where

- S and T are the sets of places and transitions with $S \cap T = \emptyset$;
- $S \times T$ is the set of bidirectional arcs;
- U is a labeling mapping for each element of $S \cup T \cup (S \times T)$, such that
 - for each place $s \in S$, $U(s)$ gives its type (the set of admissible tokens);
 - for each transition $t \in T$, $U(t)$ is its (possibly empty) label;
 - for each arc in $S \times T$, $U((s, t))$ is a pair R/W , where R and W are constants or variables (or tuples thereon) compatible with $U(s)$.
- G is the mapping associating a guard (Boolean formula) to each $t \in T$;
- M is the marking associating to each place $s \in S$ a unique token in $U(s)$.

As usual in high-level Petri nets, a transition $t \in T$ is *enabled* at a marking M iff there exists a *binding* ρ for the variables in its guard and in the arc inscriptions adjacent to t , such that ρ is compatible with the type of each place s adjacent to t , matches each token of $M(s)$ and makes the guard $G(t)$ true. The *occurrence* of t under ρ produces a new marking M' by consuming the tokens in all places adjacent to t and producing the new ones according to the arc annotations and conditions expressed in the guard. Such an occurrence of t is denoted $M[t : \rho > M']$.

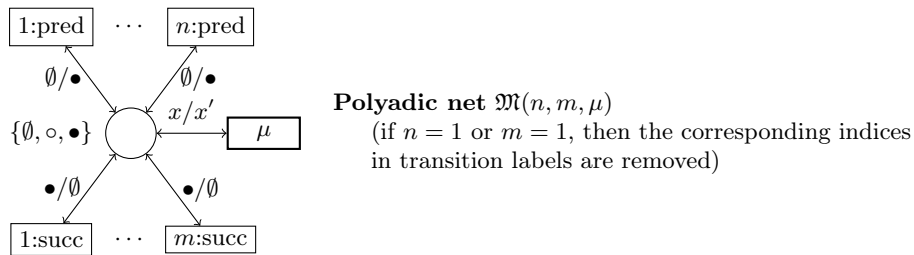
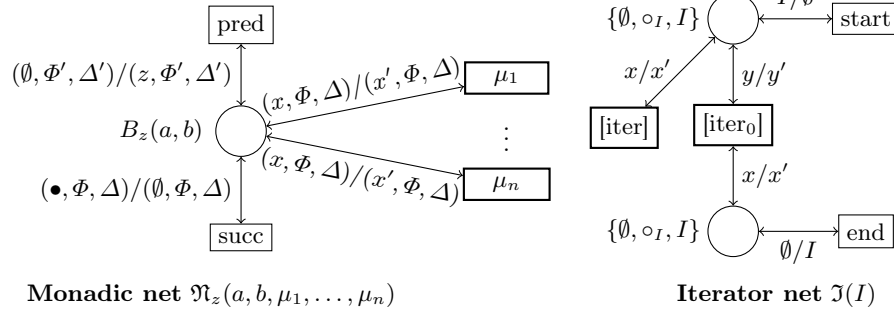
4.2 Translation

Let $\mathcal{E} \stackrel{\text{def}}{=} \beta_0; \gamma_0; \kappa_0 \vdash \pi_0$ be a pi-graph in its initial state (cf. Section 3.1). The translation is performed in two steps:

1. first, a *term net* $\mathfrak{T}(\pi_0)$ is obtained from a syntax-driven translation of π_0 ;
2. then, the translation $\text{Pnet}(\mathcal{E})$ is obtained by composing $\mathfrak{T}(\pi_0)$ with a generic *context net* \mathfrak{C} , and by associating to the resulting net structure an initial marking.

Step 1. The syntax-driven translation of pi-graph terms is based on a reduced set of basic *control-flow nets* that can be composed together using only three operators: (trivial) relabelling, disjoint union and merge. Table 3 gives the complete definition of the translation.

The *monadic control-flow nets*, denoted $\mathfrak{N}_z(a, b, \mu_1, \dots, \mu_n)$, have only a single predecessor transition, labelled *pred*, and a single successor labelled *succ*. The control-flow is obtained from the predecessor and forwarded to the successor. The unique place of the net is connected to a set of *context transitions* μ_1, \dots, μ_n that correspond to rule names in the operational semantics of Table 2 (cf. the context net in Step 2 below). In the translation, the monadic nets are used to directly encode the constructs that reference names: input/output and match/mismatch. The set of context transition labels correspond to all the semantic rules that are potentially enabled when these constructs are in redex position, e.g. [i-fresh] and [sync] for the input prefix (cf. Table 2 and Figure 1). The place type of a monadic net is $B_z(a, b) \stackrel{\text{def}}{=} \{(z, a, b), (\bullet, a, b), (\emptyset, a, b)\}$, with



$$\begin{aligned}
& \mathfrak{T}([I_1 : *P_1 \parallel \dots \parallel I_k : *P_k]) \stackrel{\text{def}}{=} \bigoplus_{j=1}^k \mathfrak{T}(I_j : *P_j) \\
& \mathfrak{T}(I : *P) \stackrel{\text{def}}{=} \mathfrak{J}(I) \sqcap_{\{\text{start} \bowtie \text{pred}, \text{end} \bowtie \text{succ}\}} \mathfrak{T}(P) \\
& \mathfrak{T}(p\emptyset) \stackrel{\text{def}}{=} \mathfrak{T}(p) \\
& \mathfrak{T}(pP) \stackrel{\text{def}}{=} \mathfrak{T}(p) \sqcap_{\{\text{succ} \bowtie \text{pred}\}} \mathfrak{T}(P) \\
& \mathfrak{T}(\tau) \stackrel{\text{def}}{=} \mathfrak{M}(1, 1, [\text{silent}]) \\
& \mathfrak{T}(\bar{a}(b)) \stackrel{\text{def}}{=} \mathfrak{N}_{\circ_o}(a, b, [\text{out}], [\text{o-fresh}], [\text{sync}]) \\
& \mathfrak{T}(a(b)) \stackrel{\text{def}}{=} \mathfrak{N}_{\circ_i}(a, b, [\text{i-fresh}], [\text{sync}]) \\
& \mathfrak{T}([a = b]P) \stackrel{\text{def}}{=} \mathfrak{N}_{\circ}(a, b, [\text{match}]) \sqcap_{\{\text{succ} \bowtie \text{pred}\}} \mathfrak{T}(P) \\
& \mathfrak{T}([a \neq b]P) \stackrel{\text{def}}{=} \mathfrak{N}_{\circ}(a, b, [\text{miss}]) \sqcap_{\{\text{succ} \bowtie \text{pred}\}} \mathfrak{T}(P) \\
& \mathfrak{T}(\sum[P_1 + \dots + P_k]) \stackrel{\text{def}}{=} \mathfrak{M}(1, k, [\text{sum}]) \sqcap_A (\bigoplus_{j=1}^k j : \mathfrak{T}(P_j)) \sqcap_A \mathfrak{M}(k, 1, [\text{sum}_0]) \\
& \quad \text{where } A \stackrel{\text{def}}{=} \{j : \text{succ} \bowtie j : \text{pred} \mid j \in [1, k]\} \\
& \mathfrak{T}(\prod[P_1 \parallel \dots \parallel P_k]) \stackrel{\text{def}}{=} \mathfrak{M}(1, 1, [\text{par}]) \sqcap_B (\bigoplus_{j=1}^k j : \mathfrak{T}(P_j)) \sqcap_{B'} \mathfrak{M}(1, 1, [\text{par}_0]) \\
& \quad \text{where } B \stackrel{\text{def}}{=} \{\text{succ} \bowtie j : \text{pred} \mid j \in [1, k]\} \text{ and } B' \stackrel{\text{def}}{=} \{j : \text{succ} \bowtie \text{pred} \mid j \in [1, k]\}
\end{aligned}$$

where the net operations are defined as follows:

- $j:N$ renames in N transition labels pred and succ to $j:\text{pred}$ and $j:\text{succ}$;
- $\bigoplus_{j=1}^k N_j$ is a disjoint union of N_j 's;
- $N \sqcap_A N'$, with $A \subseteq U(T) \times U'(T')$, is the disjoint union $N \oplus N'$ where for each pair $(a, a') \in A$, denoted $a \bowtie a'$, the transitions labeled a in N have been subsequently merged with the transitions labeled a' in N' ; the merged transitions are anonymous; i.e., have empty labels and true guards.

Table 3. Term nets – Step 1 of the translation.

a, b names in \mathcal{N} , and $z = \circ_i$ for an input prefix, $z = \circ_o$ for an output, and $z = \circ$ for match and mismatch. The control markers are: \emptyset for inactive, $\circ_\lambda \in \{\circ, \circ_i, \circ_o\}$ for the activation of the place (redex position for the corresponding term), and \bullet to trigger the successor transition (we illustrate the dynamics of the translated Petri net below).

The *polyadic control-flow nets*, denoted $\mathfrak{M}(n, m, \mu)$ are used to encode the remaining constructs but iterators. They have potentially multiple predecessors and successors but are connected to a single transition of the context net (e.g. [silent] for the τ prefix). The place type of polyadic nets is $\{\emptyset, \circ, \bullet\}$ with the same meaning as the monadic case for control markers. Finally, the *iterator nets* denoted $\mathfrak{I}(I)$ directly encode the iterator constructs. The type of the initial and final places of a translated iterator is $\{\emptyset, \circ_I, I\}$ where I is the iterator label and \circ_I the control marker.

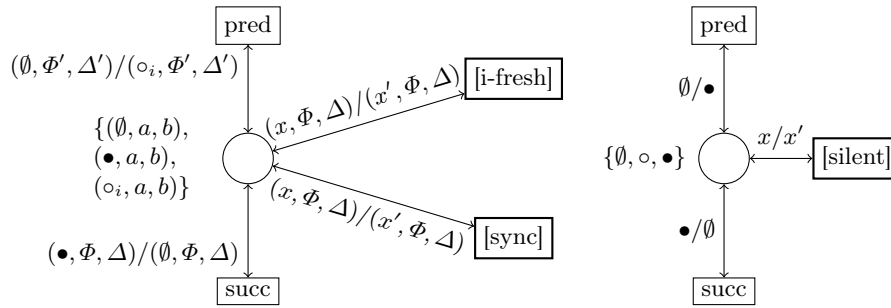


Fig. 1. Term net for Input $\mathfrak{T}(a(b))$ (left) and silent prefix $\mathfrak{T}(\tau)$ (right).

The translations for input and silent prefixes are illustrated in Figure 1. For the input prefix, the symbols Φ, Δ and Δ' are variables always bound to the translated channel name a and datum name b . The variables x and x' are used to modify the control-state of the place. This is managed by the context net (cf. Step 2 below). The other symbols are constants. The connection to the continuation of a prefix consists in merging the succ transition of the prefix with the pred transition of the next prefix (if it is a termination the prefix remains unmodified). The translation for output is similar to the input case. The match and mismatch are special prefixes (that cannot be suffixed by 0) but otherwise have quite similar translations. For parallel and sum, the entry and the exit places of the constructs have separate translations. A (polyadic) sum entry - connected to [sum] - has k successors (because a choice must be made) whereas the successor is unique for parallel entry (activation of all successor places at once through [par]). Symmetrically, the sum exit - connected to [sum₀] has k predecessors (only one must terminate) whereas it is unique for parallel exit (termination of all sub-processes at once through [par₀]). An iterator term net $\mathfrak{I}(I : * P)$ is obtained by the disjoint union of the translation of the iterator

body $\mathfrak{T}(P)$ and the term net $\mathfrak{J}(I)$, with the explicit merge of the start/pred, and end/succ transitions. The translation of Example 2 is proposed in Figure 2.

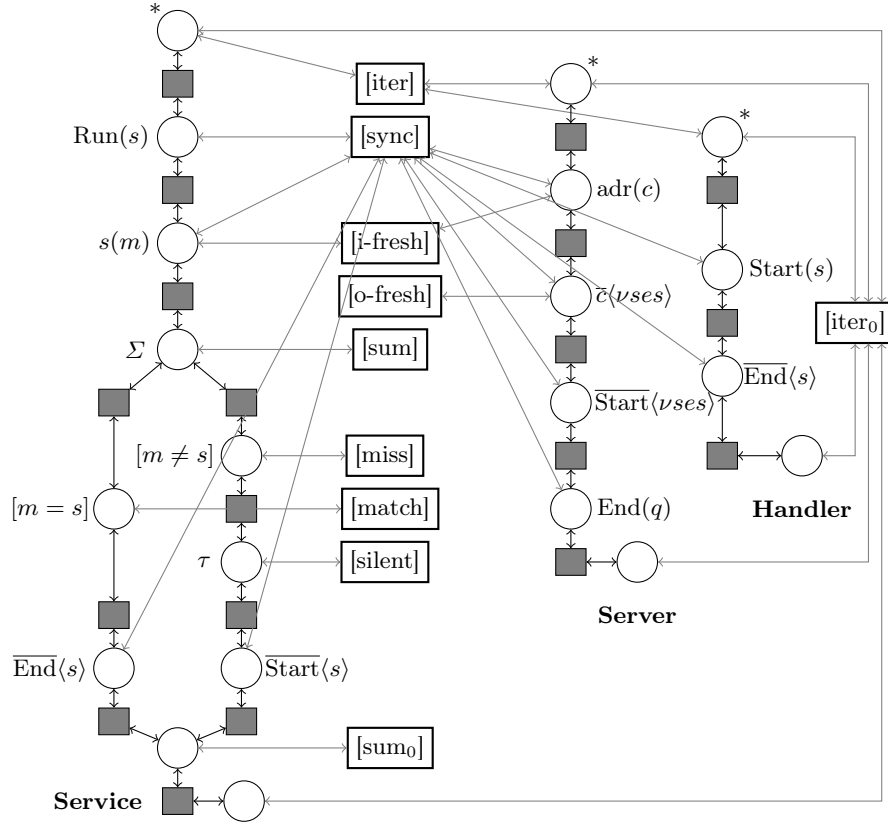


Fig. 2. Translation of the server specification (without place and arc labels).

Step 2. The term nets produced by the first step of the translation encode the basic control-structure of the pi-graphs. However, an extra layer is required to *orchestrate* faithfully the semantics of Table 2. Technically speaking, we must connect the context transitions ([sync], [par], etc.) to a Petri net layer responsible of the orchestration: namely the *context net*. Given the relatively high-level and interleaving nature of the pi-graphs (as well as most pi-calculus variants), the role of the context net is to centralize the control so that each observation (i.e. a labelled transition in process calculus terms) can be generated *atomically*.

The generic context net \mathfrak{C} - represented in Figure 3 - has only two places and fourteen transitions. The *global context place* Γ contains a single token (β, γ, κ)

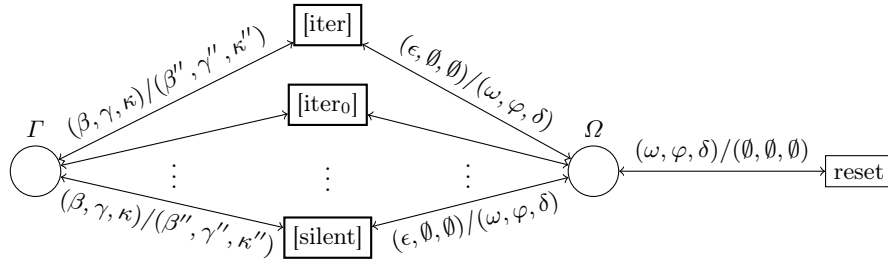


Fig. 3. Generic context net \mathfrak{C} – all context transitions are present.

describing the current instantiations, partitions and causal clock (cf. Section 3). The *observation place* Ω contains informations about the current *observation* or \emptyset for “unobservable” states. This place allows to abstract away from the internal states of the Petri nets and it also ensures the *mutual exclusion* among potential observations for the connected term net.

$$\begin{aligned}
 g([\text{silent}]) &\stackrel{\text{def}}{=} \omega = \tau \wedge \exists u : x_u = \circ \wedge x'_u = \bullet \\
 g([\text{out}]) &\stackrel{\text{def}}{=} \omega = o \wedge \exists u (x_u = \circ_o \wedge \varphi = \beta(\Phi_u) \in \text{Pub} \wedge \delta = \beta(\Delta_u) \in \text{Pub} \wedge x'_u = \bullet) \\
 g([\text{o-fresh}]) &\stackrel{\text{def}}{=} \omega = o \wedge \delta = \text{next}_o(\kappa)! \wedge \exists u (x_u = \circ_o \wedge \varphi = \beta(\Phi_u) \in \text{Pub} \wedge \\
 &\quad \beta(\Delta_u) \in \text{Priv} \wedge x'_u = \bullet \wedge \beta'(\Delta_u) = \delta) \wedge \kappa' = \text{out}(\kappa) \\
 g([\text{i-fresh}]) &\stackrel{\text{def}}{=} \omega = i \wedge \delta = \text{next}_i(\kappa)? \wedge \exists u (x_u = \circ_i \wedge \varphi = \beta(\Phi_u) \in \text{Pub} \wedge \\
 &\quad x'_u = \bullet \wedge \beta'(\Delta_u) = \delta) \wedge \kappa' = \text{in}(\kappa) \\
 g([\text{match}]) &\stackrel{\text{def}}{=} \omega = \epsilon \wedge \exists u (x_u = \circ \wedge \varphi = \beta(\Phi_u) \wedge \delta = \beta(\Delta_u) \wedge \varphi \stackrel{\gamma}{\sim} \kappa \delta \wedge x'_u = \bullet) \wedge \\
 &\quad \gamma' = \gamma_{\triangleleft \varphi = \delta} \\
 g([\text{miss}]) &\stackrel{\text{def}}{=} \omega = \epsilon \wedge \exists u (x_u = \circ \wedge \varphi = \beta(\Phi_u) \wedge \delta = \beta(\Delta_u) \wedge \neg(\varphi = \gamma \delta) \wedge x'_u = \bullet) \wedge \\
 &\quad \gamma' = \gamma_{\triangleleft \varphi \neq \kappa \delta} \\
 g([\text{sync}]) &\stackrel{\text{def}}{=} \omega = \tau \wedge \exists u, v (x_u = \circ_o \wedge x_v = \circ_i \wedge \beta(\Phi_u) \stackrel{\gamma}{\sim} \kappa \beta(\Phi_v) \wedge x'_u = x'_v = \bullet \wedge \\
 &\quad \beta' = \beta_{\triangleleft \Delta'_v \mapsto \beta(\Delta_u)} \wedge \gamma' = \gamma_{\triangleleft \beta(\Phi_u) = \beta(\Phi_v)}) \\
 g([\text{sum}]) &= g([\text{sum}_0]) = g([\text{par}]) = g([\text{par}_0]) \stackrel{\text{def}}{=} \omega = \epsilon \wedge \exists u : x_u = \circ \wedge x'_u = \bullet \\
 g([\text{iter}]) &\stackrel{\text{def}}{=} \omega = \epsilon \wedge \exists u : x_u = \circ_I \wedge x'_u = I \\
 g([\text{iter}_0]) &\stackrel{\text{def}}{=} \omega = \epsilon \wedge \exists u : x_u \neq \emptyset \wedge \{\beta'(a^{x_u}) = a^{x_u} \mid a^{x_u} \in \text{dom}(\beta)\} \wedge x'_u = \emptyset \wedge \\
 &\quad \exists v : y_v = \emptyset \wedge y'_v = x_u \\
 g([\text{reset}]) &\stackrel{\text{def}}{=} (\omega, \varphi, \delta) \neq (\emptyset, \emptyset, \emptyset)
 \end{aligned}$$

Table 4. Formulas for Petri net transition guards (if not explicitly indicated $x'_r = x_r$, $\beta'(r) = \beta(r)$, $\gamma' = \gamma$, $\kappa' = \kappa$, $\varphi = \emptyset$ and $\delta = \emptyset$) and $(\beta'', \gamma'', \kappa'') = \text{gc}(\beta', \gamma', \kappa')$.

The reset transition is used to *discharge* the previous observation (by putting $(\emptyset, \emptyset, \emptyset)$ in Ω). The thirteen other transitions correspond to the thirteen rule names of Table 2. All the preconditions and side-effects of the semantic rules

are encoded as guards for the corresponding context transitions. The guards are listed in Table 4 and the correspondence with the semantic rules of Table 2 is immediate. A context transition is enabled if the marking on Γ gives an appropriate global context, and if the marking on Ω is $(\emptyset, \emptyset, \emptyset)$ indicating that an observation is possible (mutual-exclusion). If an observation is enabled, the markings on Ω and Γ allow to retrieve it. For example, if the token on Ω is of the form (o, c, d) , it means that the corresponding observation is the output $\bar{c}\langle d \rangle$, and analogously for the other actions. Firing the reset transition discharges the observation, allowing further observations.

The final net structure is a disjoint union of the nets $\mathfrak{T}(\pi)$ and \mathfrak{C} , in which:

- we remove from \mathfrak{C} all the context transitions which are not present in $\mathfrak{T}(\pi)$;
- we merge all context transitions with the same label, and add to each resulting transition the corresponding guard.

Formally, this is defined as follows, using net operation $\mathfrak{T}(\pi) \diamond_D \mathfrak{C}$, indexed by a set $D \stackrel{\text{def}}{=} \{(\mu, g(\mu)) \mid \mu \in \mathcal{M}\}$ of pairs (transition label, transition guard), where \mathcal{M} is the set of labels containing reset and all interface transition labels present in $\mathfrak{T}(\pi)$, and the guards $g(\mu)$ are as specified in Table 4. For each pair $(\mu, g(\mu)) \in D$, let t_1, \dots, t_n be transitions with label μ in $\mathfrak{T}(\pi)$; for each arc (s, t_u) , the variables in arc annotations $U(s, t_u)$ are indexed by u ; all these transitions are merged together and with the transition with the same label in \mathfrak{C} ; the resulting transition has label μ and guard $g(\mu)$.

The Petri net $\text{Pnet}(\mathcal{E})$ is then obtained by associating to the structure resulting from Step 2 an initial marking M_0 , which is $(\beta_0, \gamma_0, \kappa_0)$ on Γ , $(\emptyset, \emptyset, \emptyset)$ on Ω , (\emptyset, a, b) on input $a(b)$, output $\bar{a}\langle b \rangle$, match $[a = b]$ and mismatch $[a \neq b]$ places, I_j on the initial place of iterator I_j , and \emptyset on any other place.

4.3 Main properties

The first important property of the proposed translation scheme is its *concision*.

Theorem 1. *The size of $\text{Pnet}(\mathcal{E})$ is linear in the size of \mathcal{E}*

Proof. The proof is by trivial (inductive) case analysis, with exactly one place and one transition for each prefix except sum and parallel (two places and one transition each), two places and one transition for iterators, and the context net has only two places and fourteen transitions. More precisely the size is (tightly) bounded by $2i + p + 2s + 2$ places and $i + p + s + 14$ transitions (with i the number of iterators, p the number of simple prefixes and s the number of parallels and sums) \square

The most important property we expect from the translation is that it agrees with the operational semantics of pi-graphs described in Section 3. Our objective is to show that each abstracted rewrite of a pi-graph is matched, in the translated Petri net, by an *abstracted occurrence* (defined below).

The Petri nets have lower-level semantics than the pi-graphs; in particular we may abstract away from any occurrence of an *anonymous transition* (i.e. a transition t such that $U(t) = \emptyset$). First, we may observe:

Lemma 1. *There is no infinite sequence of anonymous occurrences, and successive anonymous occurrences are causally independent.*

Proof. In the translation, we can easily see that between two anonymous transitions there is always a place connected to a context transition. As such, the context transition must be fired between the two anonymous occurrences. Of course, causally independent anonymous occurrences (in parallel processes or iterators) can still be performed in arbitrary order, but there can only be a finite number of these. \square

Hence, we may consider sequences of occurrences $M_0[t_1 : \rho_1 > M_1 \dots M_{n-1}[t_n : \rho_n > M_n$ such that $\forall i, 0 \leq i \leq n, U(t_i) = \emptyset$ and such that there is no anonymous transition t_{n+1} enabled from M_n . The markings M_0, \dots, M_n are considered as equivalent and the witness of the equivalence class is M_n , i.e. the (unique) state from which no further anonymous transition is enabled.

With this first level of abstraction, the Petri net markings are almost in one-to-one correspondence with pi-graph states.

Definition 8. *Let $\beta_1, \gamma_1, \kappa_1 \vdash \pi[\theta]$ be a pi-graph state with a subterm θ . We define $M^{\pi[\theta]}$ a marking of the corresponding Petri net such that $M^{\pi[\theta]}(\Gamma) = (\beta_1, \gamma_1, \kappa_1)$, $M^{\pi[\theta]}(\Omega) = (\emptyset, \emptyset, \emptyset)$ and each place corresponding to a redex in θ contains a control marker \circ_λ .*

The most important part of the agreement is that each pi-graph rewrite is matched by *exactly* two (abstract) Petri net occurrences: (1) the enabling of an observation, and (2) its discharge by the reset transition.

Lemma 2. *The rewrite $\beta_1; \gamma_1; \kappa_1 \vdash \pi[\theta] \xrightarrow{\alpha} \beta_2; \gamma_2; \kappa_2 \vdash \pi[\theta']$ can be inferred by rule μ of the semantics iff there exists a sequence of occurrences $M_1^{\pi[\theta]}[t_1 : \rho_1 > M'_2 : \rho_2 > M_2^{\pi[\theta']}$ such that $U(t_1) = \mu$, $U(t_2) = \text{reset}$ and $\text{label}(M'_2(\Omega)) = \alpha$*

with $\text{label}(\omega, \varphi, \delta) \stackrel{\text{def}}{=} \varphi(\delta)$ if $\omega = i$, $\overline{\varphi}(\delta)$ if $\omega = o$, ω otherwise

Proof. For the if part, we must take each rule of Table 2 and give its interpretation in terms of the translated Petri net. The translation has been designed to closely follow the rules, and thus all steps are almost direct and very similar. For the sake of concision, we only detail the case of the input prefix. The main hypothesis is a transition of the form:

$$\beta_1; \gamma_1; \kappa_1 \vdash \pi \left[\boxed{a(b)}.P \right] \xrightarrow{\beta_1(a)(\beta_1(b))} \text{gc}((\beta_1 \triangleleft \Delta \mapsto \text{next}_t(\kappa_1)?, \gamma_1, \text{in}(\kappa_1))) \vdash \pi[a(b).\boxed{P}]$$

In Figure 4 we illustrate the two corresponding occurrences in the translated Petri net. The place named s_1 is the translation of the input prefix; initially this place is a redex since it contains the control marker \circ_i . The place s_2 is the first subterm of the continuation process P , its marking contains the control marker \emptyset , i.e. it is inactive. The global context Γ contains $(\beta_1, \gamma_1; \kappa_1)$ and there is no observation in Ω . In this situation the [i-fresh] transition can be fired, which leads to the second marking described in the figure. At the low-level, the [i-fresh] occurrence replaces the control marker \circ_i by a continuation marker \bullet . However,

this continuation is consumed by the anonymous transition between s_1 and s_2 , which activates s_2 . We only retain at the abstract level the state with s_1 marked \emptyset (inactive) and s_2 marked with the control marker \circ_λ (depending on the exact nature of s_2 , it can be \circ , \circ_i or \circ_o). The observation is captured by the marking of Ω . The context is also updated accordingly in the marking of Γ . Finally, the reset transition is fired to discharge the observation.

The only-if part consists in identifying each high-level occurrence of the Petri net to a corresponding pi-graph rule. There is no ambiguity since the Petri net provides the name of the rule (the label of the fired transition). Moreover, any observation must be discharged by the reset transition and thus the two steps are indeed atomic \square

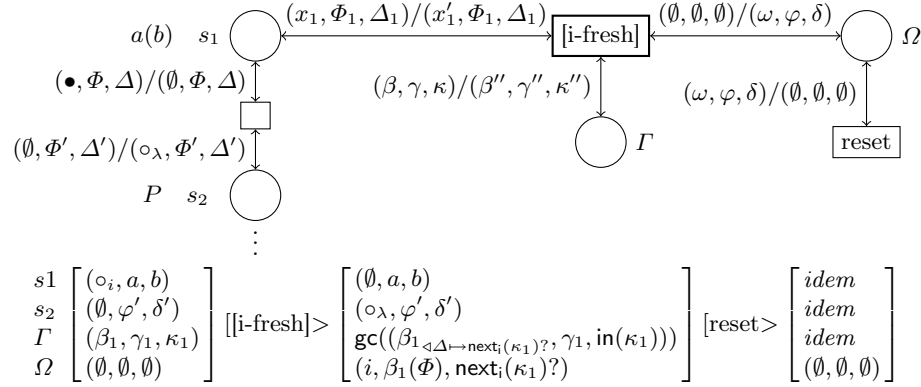


Fig. 4. Illustration of Lemma 2.

In the remainder we may forget about the particular subterm that is rewritten in the two-steps sequences identified by Lemma 2. In consequence, we simply denote a high-level occurrence as $M_1^\pi[\alpha > M_2^{\pi'}]$. This allows a more general restatement of the previous Lemma as follows.

Lemma 3. $\Gamma_1 \vdash \pi \xrightarrow{\alpha} \Gamma_2 \vdash \pi'$ iff $M_1^\pi[\alpha > M_2^{\pi'}]$

The notion of abstracted transition (Definition 6) can then be reinterpreted.

Definition 9. An **abstracted occurrence** $M_1[\alpha \gg M_n]$ is produced iff there is a sequence $M_1[\epsilon > M_2 \dots M_{n-2}[\epsilon > M_n[\alpha > M_{n+1}(\alpha \neq \epsilon)]]$ with $\forall i, 1 \leq i < n, \exists j, i < j \leq n$ such that the i -th transition produces a redex token absorbed by the j -th one.

This leads to the most important theorem of the study, as follows:

Theorem 2. (agreement)

$\Gamma_1 \vdash \pi_1 \xrightarrow{\alpha} \Gamma_n \vdash \pi_n$ iff $M_1^{\pi_1}[\alpha \gg M_n^{\pi_n}]$

Proof. For the only if part an abstracted transition (in the pi-graph) is assumed. By Lemma 3 we obtain directly a sequence of high-level occurrences (in the translated Petri net) verifying the conditions for an abstracted occurrence, as described above. Since Lemma 3 works both ways, the if part derives easily \square

5 Related and future works

In a previous work [5], we study the syntactic translation of a variant of the pi-calculus into high-level Petri nets with read arcs. The supported language has unrestricted recursion and the match operator, but no mismatch. These two features clearly impact the translation scheme in terms of complexity, together with the non-deterministic choice. By introducing a “transition-friendly” intermediate calculus – namely the pi-graphs – we are able to provide a *much* simpler translation scheme to lower-level (but still coloured) Petri nets (without read arcs). The construct of iterator is preferred to general recursion in the pi-graphs. As demonstrated in [8], this provides a guarantee of finiteness by construction on the semantic side, a property only enjoyed for recursion-free processes in [5].

There are also various semantic translations of pi-calculi into Petri nets. Despite the different philosophy of these approaches based on reachability analysis, many interesting points of comparison remain. A prominent example is [4], that investigates the translation of a variant of the pi-calculus into low-level P/T nets. The supported language has general recursion but no match nor mismatch. The translation produces P/T nets whose places correspond to sub-processes (so-called *fragments*) in a normal structural form said restricted. The transitions are either *reactions* between reachable fragments or communications through public channels. The (interleaving) *reduction semantics* of the considered pi-calculus variant can be “fully retrieved” in the P/T nets, i.e. the formalisms *agree* in our own terms. Our translation reaches a similar agreement but in terms of *transition semantics*, a more complex setting. Indeed, without the match (and mismatch) and considering the reduction semantics only, our translation would be greatly simplified (e.g. no need for name partitions or causal clocks) and it would be interesting to see how far we get from P/T nets in this case.

Since the language of [4] has general recursion, it is possible to generate infinite systems. However, the semantic property of *structural stationarity* captures an interesting (albeit undecidable) subclass of potentially infinite systems that can be characterized finitely (by ensuring that there are only a finite number of reachable fragments up to structural congruence). In contrast the pi-graphs cannot express processes with e.g. infinite control, which is a design choice because we require a guarantee of finiteness *by construction*.

Related semantic translations are proposed in [2, 3], which interpret the (non-) interleaving and causal semantics for a pi-calculus in terms of P/T nets with inhibitor arcs. The translation provides an interpretation of the early transitions, whereas our interleaving semantics are closer to the late semantics. The advantage of the latter is that there is no need to study all the possible substitutions for the names generated by the environment. The authors of [3] identify

the *finite net processes* that may only generate a bounded number of restricted names, a property that is not required in the pi-graphs thanks to the garbage collection of inactive names. A similar principle is proposed in the *history dependent automata* (HDA) framework [11]. 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 [10] and the support for both match and mismatch.

There are three main directions that seem worth studying based on the presented work. First, the proposed translation scheme has quite a *modular* structure with the term nets on the one side (interpretation of the syntax) and the context net on the other side (interpretation of the semantics). Variants of the context net (thus, of the semantics) could be considered to investigate, as an alternative to the interleaving semantics, more concurrent/causal strategies. The Petri nets resulting from the translations are coloured but, we believe, with a lot of symmetry involved, which means efficient unfoldings could be produced for verification purpose. Finally, we plan to study a variant of the iterator construct - a form of *replication* - that would allow multiple simultaneous iterations to run in parallel.

References

1. Milner, R.: Communicating and Mobile Systems: The π -Calculus. Cambridge University Press (1999)
2. Busi, N., Gorrieri, R.: A Petri net semantics for π -calculus. In: CONCUR. Volume 962 of Lecture Notes in Computer Science., Springer (1995) 145–159
3. Busi, N., Gorrieri, R.: Distributed semantics for the pi-calculus based on Petri nets with inhibitor arcs. J. Log. Algebr. Program. **78** (2009) 138–162
4. Meyer, R., Gorrieri, R.: On the relationship between π -calculus and finite place/transition Petri nets. In: CONCUR. Volume 5710 of Lecture Notes in Computer Science., Springer (2009) 463–480
5. Devillers, R., Klaudel, H., Koutny, M.: A compositional Petri net translation of general π -calculus terms. Formal Asp. Comput. **20** (2008) 429–450
6. Berger, M.: An interview with Robin Milner. <http://www.informatics.sussex.ac.uk/users/mfb21/interviews/milner/> (2003)
7. Peschanski, F., Bialkiewicz, J.A.: Modelling and verifying mobile systems using pi-graphs. In: SOFSEM. Volume 5404 of Lecture Notes in Computer Science., Springer (2009) 437–448
8. 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
9. Busi, N., Gabbriellini, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: ICALP. Volume 3142 of Lecture Notes in Computer Science., Springer (2004) 307–319
10. Degano, P., Priami, C.: Causality for mobile processes. In: ICALP. Volume 944 of Lecture Notes in Computer Science., Springer (1995) 660–671
11. Montanari, U., Pistore, M.: History-dependent automata: An introduction. In: SFM. Volume 3465 of Lecture Notes in Computer Science., Springer (2005) 1–28