Type Abstractions of Name-Passing Processes*

Lucia Acciai[1] and Michele Boreale[2]

[1] Laboratoire d'Informatique Fondamentale de Marseille, Université de Provence.
[2] Dipartimento di Sistemi e Informatica, Università di Firenze.
lucia.acciai@lif.univ-mrs.fr, boreale@dsi.unifi.it

Abstract. We study methods to statically approximate "first-order" process calculi (Pi, Join) by "propositional" models (CCS, BPP, Petri nets). We consider both open and closed behavior of processes. In the case of open behavior, we propose a type system to associate pi-calculus processes with restriction-free CCS types. A process is shown to be in simulation relation with each of its types, hence safety properties that hold of the types also hold of the process. We refine this approach in the case of closed behavior: in this case, types are BPP processes. Sufficient conditions are given under which a minimal BPP type can be computed that is bisimilar to a given process. These results are extended to the Join calculus using place/transition Petri nets as types.

1 Introduction

The behavior of large, possibly distributed programs that heavily rely on reference-passing is generally difficult to comprehend, both intuitively and formally. Process calculi like Pi [18] and Join [7] posses "first-order" features, like value passing and dynamic name creation, difficult to recast into well-analyzable operational formats. This situation should be contrasted with that found in "propositional" formalisms like CCS and Petri nets, that enjoy simpler and more tractable operational models, studied and utilized for decades (the terminology "first order" and "propositional" is not standard in the present context, and should be taken with a grain of salt.)

For many purposes it may be sufficient to take an abstract view of name-passing processes, hopefully easy to recast into propositional terms. Imagine one specifies a context Γ associating values and free names of a pi-process P with *tags* drawn from a finite set. Tags might represent particular events an external observer is interested in. Different names/values can possibly be collapsed onto the same tag (e.g., different values could be mapped to a unique tag representing their type). Imagine further that P's code specifies how to make such associations at run-time for newly generated names. If one observes P "through" Γ, that is, dynamically maps values/names to tags in transition labels as prescribed, one obtains an abstract process P_Γ. The latter is operationally described by a possibly infinite, yet simpler propositional transition system. In many cases, it may be sufficient to further limit one's attention to the *closed* behavior of P_Γ, that is, to communication actions suitably decorated with tags, such as the identity of

* The first author is supported by the French government research grant ACI TRALALA. The second author is supported by the EU within the FET-GC2 initiative, project SENSORIA.

the service that is called, types of the passed parameters and so on. In other words, P_Γ provides a bird's eye view of the system under observation, which is often sufficient to establish interesting properties of the system, typically safety ones.

The goal of this paper is to study means to *statically* computing finitary representations of P_Γ, or at least suitable approximations of P_Γ. The proposed methods will take the form of behavioral type systems for process calculi. Such systems can be used to assign a process P a propositional type T that, in general, over-approximates the behavior of P_Γ. This technique should in principle allow one to verify certain properties of T, being assured that the same properties also hold for P_Γ. In each of the considered type systems, the emphasis will be on keeping the class of types tractable. In particular, (bi)similarity and model-checking for interesting modal logics should be decidable for the given class. In other words, the aim here is laying a basis for property verification by a combination of type-checking and model-checking techniques. This approach is along the lines of the work on behavioral types by Igarashi and Kobayashi [10] .

More specifically, we start by introducing an asynchronous, "tagged" version of the pi-calculus and the notion of abstract process P_Γ (Section 2). In the first type system (Section 3), types are a class of asynchronous, restriction-less CCS processes, which we name CCS^-. Both bisimulation and model checking for interesting logics are decidable for CCS^- (e.g., by translation into Petri nets [5]). Our main result here shows that P_Γ is in simulation relation with the types inhabited by P; hence safety properties that hold for the types also holds for the abstract process P_Γ. The absence of restriction causes an obvious loss of precision in types; note however that in practice this can be remedied by hiding certain actions at the level of modal logic formulae. In the second system (Section 4), we focus on closed behavior. The goal is to obtain simpler and hopefully more efficient behavioral approximations, by getting rid of synchronization in types. We achieve this goal by directly associating each output action with an *effect* corresponding to the observable behavior that that output can trigger. Input actions have no associated effect, thus an inert type is associated to input processes. Types we obtain in this way are precisely *Basic Parallel Processes* (BPP, [4]): these are infinite-state processes for which, however, a wealth of decidability results exists [9,6]. We show that if we restrict ourselves to a class of pi-processes satisfying a generalized version of uniform receptiveness [26], a type can be computed that is bisimilar to P_Γ. These techniques will be illustrated using a concrete example (Section 5). We finally move to the Join calculus (Section 6) and generalize some of the results obtained for the pi-calculus. At the level of types, the main step is moving from BPP to the more general class of place/transition Petri nets, for which again interesting decidability results are known [9,6]. We end the paper with some remarks on related and further work (Section 7).

2 The asynchronous pi-calculus

2.1 Processes

Let $\mathcal{N}$, ranged over by $a, b, c, \ldots, x, y, \ldots$, be a countable set of *names* and $\mathcal{T}ag$, ranged over by $\alpha, \beta, \ldots$, be a set of *tags* disjoint from $\mathcal{N}$; we assume $\mathcal{T}ag$ also contains a distinct "unit" tag $()$. The set $\mathcal{P}$ of processes $P, Q, \ldots$ is defined as the set of terms generated by the following grammar:

$$P,Q ::= \overline{a}\langle b\rangle \mid \sum_{i\in I} a_i(b).P_i \mid \sum_{i\in I} \tau.P_i \mid \text{if } a=b \text{ then } P \text{ else } P \mid !a(b).P \mid (\nu a:\alpha)P \mid P|P\,.$$

This language is a variation on the asynchronous π-calculus. A non-blocking *output* of a name b along a is written $\overline{a}\langle b\rangle$. *Nondeterministic guarded summation* $\sum_{i\in I} a_i(b).P_i$ waits for a communication on a_i, for $i \in I$. An *internal choice* $\sum_{i\in I} \tau.P_i$ can choose to behave like any of the P_i via an invisible τ transition. *Conditional* if $a = b$ then P else Q behaves as P if a equals b, as Q otherwise. *Replication* $!a(b).P$ provides an unbounded number of copies of $a(b).P$. *Restriction* $(\nu a:\alpha)P$ creates (and assigns a tag α to) a new restricted name a with initial scope P. As usual, the *parallel composition* $P\,|\,Q$ represents concurrent execution of P and Q.

In an output action $\overline{a}\langle b\rangle$, name a is the *subject* and b the *object* of the action. Similarly, in a replicated input prefix $!a(b).P$ and in $\sum_{i\in I} a_i(b).P_i$, the names a and a_i for $i \in I$ are said to occur in *input subject position*. Binders and alpha-equivalence arise as expected and processes are identified up to alpha-equivalence. Substitution of a with b in an expression e is denoted by $e[^b/_a]$. In what follows, $\mathbf{0}$ stands for the empty summation $\sum_{i\in\emptyset} a_i(x).P_i$. We shall sometimes omit the object parts of input and output actions, when not relevant for the discussion; e.g. $\overline{a}$ stands for an output action with subject a and an object left unspecified. Similarly, we shall omit tag annotations, writing e.g. $(\nu a)P$ instead of $(\nu a:\alpha)P$, when the identity of the tag is not relevant.

2.2 Operational semantics

The (early) semantics of processes is given by the labelled transition system in Table 1. We let $\ell, \ell', \ldots$ represent generic elements of $\mathcal{N} \cup \mathcal{T}ag$. A transitions label μ can be a free output, $\overline{a}\langle b\rangle$, a bound output, $(\nu b:\alpha)\overline{a}\langle b\rangle$, an input, $a(b)$, or a silent move, $\tau\langle \ell, \ell'\rangle$. We assume a distinct tag ι for decorating *internal* transitions (arising from conditional and internal chioce; see Table 1) and often abbreviate $\tau\langle\iota,\iota\rangle$ simply as τ. In the following we indicate by $\mathrm{n}(\mu)$ the set of all names in μ and by $\mathrm{fn}(\mu)$, the set of free names of μ, defined as expected. The rules are standard, except for the extra book-keeping required by tag annotation of bound output and internal actions. In particular, in (RES-TAU) bound names involved in a synchronization are hidden from the observer and replaced by the corresponding tags. Note that if we erase the tag annotation from labels we get exactly the usual labelled semantics of asynchronous pi-calculus.

2.3 Γ-abstractions of processes

A *context* Γ is a finite partial function from names to tags, written $\Gamma = \{a_1:\alpha_1, \cdots, a_n:\alpha_n\}$, with distinct a_i. In what follows $\Gamma \vdash a:\alpha$ means that $a:\alpha \in \Gamma$. A *tag sorting system* $\mathcal{E}$ is a finite subset of $\{\alpha[\beta] \mid \alpha,\beta \text{ are tags and } \alpha \neq ()\}$. Informally, $\alpha[\beta] \in \mathcal{E}$ means that subject names associated with tag α can carry object names associated with tag β. In what follows, if $\alpha[\beta_1], \cdots, \alpha[\beta_n]$ are the only elements of $\mathcal{E}$ with subject α, we write $\alpha[\beta_1, \cdots, \beta_n] \in \mathcal{E}$.

A triple $(P,\Gamma,\mathcal{E})$, written $P_{\Gamma;\mathcal{E}}$, is called Γ-*abstraction* of P under $\mathcal{E}$. In what follows, we shall consider a fixed sorting system $\mathcal{E}$, and keep $\mathcal{E}$ implicit by writing P_Γ

$$\text{(OUT)}\ \overline{a}\langle b\rangle \xrightarrow{\overline{a}\langle b\rangle} \mathbf{0}$$

$$\text{(I-SUM)}\ \sum_{i\in I}\tau.P_i \xrightarrow{\tau} P_j,\quad j\in I \qquad \text{(G-SUM)}\ \sum_{i\in I}a_i(b_i).P_i \xrightarrow{a_j(c)} P_j[c/b_j],\quad j\in I$$

$$\text{(REP)}\ !a(c).P \xrightarrow{a(b)} P[b/c]\,|\,!a(c).P \qquad \text{(OPEN)}\ \frac{P \xrightarrow{\overline{b}\langle a\rangle} P' \quad b\neq a}{(\nu a:\alpha)P \xrightarrow{(\nu a:\alpha)\overline{b}\langle a\rangle} P'}$$

$$\text{(COM)}\ \frac{P \xrightarrow{\overline{a}\langle b\rangle} P' \quad Q \xrightarrow{a(b)} Q'}{P\,|\,Q \xrightarrow{\tau\langle a,b\rangle} P'\,|\,Q'} \qquad \text{(CLOSE)}\ \frac{P \xrightarrow{(\nu b:\beta)\overline{a}\langle b\rangle} P' \quad Q \xrightarrow{a(b)} Q'}{P\,|\,Q \xrightarrow{\tau\langle a,\beta\rangle} (\nu b:\beta)(P'\,|\,Q')}$$

$$\text{(IF-T)}\ \text{if } a=a \text{ then } P \text{ else } Q \xrightarrow{\tau} P \qquad \text{(IF-F)}\ \text{if } a=b \text{ then } P \text{ else } Q \xrightarrow{\tau} Q,\quad a\neq b$$

$$\text{(PAR)}\ \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q} \qquad \text{(RES)}\ \frac{P \xrightarrow{\mu} P' \quad a\notin \mathrm{n}(\mu)}{(\nu a:\alpha)P \xrightarrow{\mu} (\nu a:\alpha)P'}$$

$$\text{(RES-TAU)}\ \frac{P \xrightarrow{\tau\langle \ell_1,\ell_2\rangle} P' \quad a\in\{\ell_1,\ell_2\} \quad \ell=\ell_1[\alpha/a] \quad \ell'=\ell_2[\alpha/a]}{(\nu a:\alpha)P \xrightarrow{\tau\langle \ell,\ell'\rangle} (\nu a:\alpha)P'}$$

Symmetric rules not shown.

Table 1. Operational semantics of pi-calculus processes.

instead of $P_{\Gamma;\mathcal{E}}$. Next, we define a labeled transition system with process abstractions as states and transition labels λ, which can be output, $\overline{\alpha}\langle\beta\rangle$, input, $\alpha\langle\beta\rangle$ or annotated silent action, $\tau\langle\alpha,\beta\rangle$. The set of labels generated by this grammar is denoted by Λ. The labeled transition system is defined by the rules below. Here, μ_Γ denotes the result of substituting each $a\in \mathrm{fn}(\mu)\cap \mathrm{dom}(\Gamma)$ by $\Gamma(a)$ in μ. Informally, P_Γ represents the abstract behavior of P, once each concrete action μ has been mapped to an abstract action λ. Note that in both rule (A-OUT_N) and rule (A-INP_N) the context Γ grows with a new association $b:\beta$. In rule (A-INP_N), a tag for b is chosen among the possible tags specified in $\mathcal{E}$. Note that no type checking is performed by these rules, in particular (A-OUT_N) does not look up $\mathcal{E}$ to check that β can be carried by α.

$$\text{(A-OLD)}\ \frac{P \xrightarrow{\mu} P' \quad \mu ::= \tau\langle\ell,\ell'\rangle | a(b) | \overline{a}\langle b\rangle \quad \mathrm{n}(\mu)\subseteq \mathrm{dom}(\Gamma) \quad \lambda=\mu_\Gamma}{P_\Gamma \xrightarrow{\lambda} P'_\Gamma}$$

$$\text{(A-OUT}_\text{N}\text{)}\ \frac{P \xrightarrow{(\nu b:\beta)\overline{a}\langle b\rangle} P' \quad \Gamma\vdash a:\alpha}{P_\Gamma \xrightarrow{\overline{\alpha}\langle\beta\rangle} P'_{\Gamma,b:\beta}} \qquad \text{(A-INP}_\text{N}\text{)}\ \frac{P \xrightarrow{a(b)} P' \quad \Gamma\vdash a:\alpha \quad \alpha[\beta]\in\mathcal{E} \quad b\notin \mathrm{dom}(\Gamma)}{P_\Gamma \xrightarrow{\alpha\langle\beta\rangle} P'_{\Gamma,b:\beta}}$$

2.4 Simulation, bisimulation and modal logic

Let T any labelled transition system with labels in Λ. As usual, the *(strong) simulation* relation over T, written $\lesssim$, is the largest binary relation over states of T such that whenever $s_1\lesssim s_2$ and $s_1 \xrightarrow{\lambda} s_1'$ then there is a transition $s_2 \xrightarrow{\lambda} s_2'$ such that $s_1'\lesssim s_2'$. The

relation $\lesssim$ is easily seen to be a preorder. *(Strong) Bisimulation* over T, written $\sim$, is the largest binary relation over states of T such that both $\sim$ and $\sim^{-1}$ are simulations. The *closed* versions of simulation and bisimulation, written $\lesssim^{c}$ and $\sim^{c}$, respectively, are defined in a similar manner, but limited to silent transitions.

Next, we introduce a simple action-based modal logic that will help us to formulate concisely properties of processes and types. The logic is very simple and only serves to illustrate the approach presented in the paper. More precisely, we let $\mathcal{L}$ be given by $\phi ::= \text{true} \mid \langle A\rangle\phi \mid \langle\langle A\rangle\rangle\phi \mid \neg\phi \mid \phi \wedge \phi$, where $\emptyset \neq A \subseteq \Lambda$. These formulae are interpreted in the expected manner, in particular, a state s satisfies $\langle A\rangle\phi$, written $s \vDash \langle A\rangle\phi$, if there is a transition $s \xrightarrow{\lambda} s'$ with $\lambda \in A$ and $s' \vDash \phi$. The interpretation of modality $\langle\langle A\rangle\rangle\phi$ is similar, but the phrase "a transition $s \xrightarrow{\lambda} s'$ with $\lambda \in A$" is changed into "a sequence of transitions $s \xrightarrow{\sigma} s'$ with $\sigma \in A^*$". We shall make use of standard notational conventions, like abbreviating $\neg\langle A\rangle\neg\phi$ as $[A]\phi$, omitting a trailing "true", and so on. Note that $\mathcal{L}$ can be regarded as a fragment of the modal mu-calculus [27].

3 CCS$^-$ types for open behavior

In the first type system we propose, types are essentially CCS expressions whose behavior over-approximate the (abstract) process behavior.

3.1 CCS$^-$ types

The set $\mathcal{T}_{\text{CCS}}$ of types, ranged over by $\mathsf{T}, \mathsf{S}, \ldots$, is defined by the following syntax:

$$\mathsf{T} ::= \overline{\alpha}\langle\beta\rangle \mid \sum_{i\in I} \alpha_i\langle\beta_i\rangle.\mathsf{T}_i \mid \sum_{i\in I} \tau.\mathsf{T}_i \mid !\alpha\langle\beta\rangle.\mathsf{T} \mid \mathsf{T}|\mathsf{T}$$

where $\alpha, \alpha_i \neq ()$. The empty summation $\sum_{i\in\emptyset} \alpha_i\langle\beta_i\rangle.\mathsf{T}_i$ will be often denoted by nil, and $\mathsf{T}_1 \mid \cdots \mid \mathsf{T}_n$ will be often written as $\prod_{i\in\{1,\cdots,n\}} \mathsf{T}_i$. As usual, we shall sometimes omit the object part of actions when not relevant for the discussion or equal to the unit tag $()$, writing e.g. $\overline{\alpha}$ and $\tau\langle\alpha\rangle$ instead of $\overline{\alpha}\langle\beta\rangle$ and $\tau\langle\alpha,\beta\rangle$.Types are essentially asynchronous, restriction-free CCS processes over the alphabet of actions Λ. The standard operational semantics of CCS, giving rise to a labelled transition system with labels in Λ, is assumed (Table 2).

3.2 The typing rules

Let $\mathcal{E}$ be a fixed tag sorting system and Γ a context. Judgements of the type system are of the form $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$. The rules of the type system are presented in Table 3.

A brief explanation of some typing rules follows. In rule (T-OUT), the output process $\overline{a}\langle b\rangle$ gives rise to the action $\overline{a}\langle b\rangle_\Gamma = \overline{\alpha}\langle\beta\rangle$, provided this action is expected by the tag sorting system $\mathcal{E}$. The type T of an input process depends on $\mathcal{E}$: in (T-INP) all tags which can be carried by α, the tag associated with the action's subject, contribute to the definition of the summation in T as expected. In the case of (T-REP), summation is replaced by a parallel composition of replicated types, which is behaviorally –

$$\text{(C-OUT)}\ \overline{\alpha}\langle\beta\rangle \xrightarrow{\overline{\alpha}\langle\beta\rangle} \mathsf{nil} \qquad \text{(C-GSUM)}\ \textstyle\sum_{i\in I}\alpha_i\langle\beta_i\rangle.\mathsf{T}_i \xrightarrow{\alpha_j\langle\beta_j\rangle} \mathsf{T}_j,\quad j\in I$$

$$\text{(C-ISUM)}\ \textstyle\sum_{i\in I}\tau.\mathsf{T}_i \xrightarrow{\tau} \mathsf{T}_j,\quad j\in I \qquad \text{(C-REP)}\ !\alpha\langle\beta\rangle.\mathsf{T} \xrightarrow{\alpha\langle\beta\rangle} \mathsf{T}\,|\,!\alpha\langle\beta\rangle.\mathsf{T}$$

$$\text{(C-COM)}\ \frac{\mathsf{T} \xrightarrow{\overline{\alpha}\langle\beta\rangle} \mathsf{T}' \quad \mathsf{S} \xrightarrow{\alpha\langle\beta\rangle} \mathsf{S}'}{\mathsf{T}|\mathsf{S} \xrightarrow{\tau\langle\alpha,\beta\rangle} \mathsf{T}'|\mathsf{S}'} \qquad \text{(C-PAR)}\ \frac{\mathsf{T} \xrightarrow{\lambda} \mathsf{T}'}{\mathsf{T}|\mathsf{S} \xrightarrow{\lambda} \mathsf{T}'|\mathsf{S}}$$

Symmetric rules for $|$ not shown.

Table 2. Operational semantics of CCS$^-$ types.

$$\text{(T-OUT)}\ \frac{\Gamma\vdash a:\alpha \quad \alpha[\beta]\in\mathcal{E} \quad \Gamma\vdash b:\beta}{\Gamma\vdash_{\mathcal{E}} \overline{a}\langle b\rangle : \overline{\alpha}\langle\beta\rangle}$$

$$\text{(T-INP)}\ \frac{\Gamma\vdash a:\alpha \quad \alpha[\beta_1,\cdots,\beta_n]\in\mathcal{E} \quad \forall i\in\{1,\cdots,n\}:\ \Gamma,b:\beta_i\vdash_{\mathcal{E}} P:\mathsf{T}_i}{\Gamma\vdash_{\mathcal{E}} a(b).P : \sum_{i\in\{1,\cdots,n\}} \alpha\langle\beta_i\rangle.\mathsf{T}_i}$$

$$\text{(T-REP)}\ \frac{\Gamma\vdash a:\alpha \quad \alpha[\beta_1,\cdots,\beta_n]\in\mathcal{E} \quad \forall i\in\{1,\cdots,n\}:\ \Gamma,b:\beta_i\vdash_{\mathcal{E}} P:\mathsf{T}_i}{\Gamma\vdash_{\mathcal{E}} !a(b).P : \prod_{i\in\{1,\cdots,n\}} !\alpha\langle\beta_i\rangle.\mathsf{T}_i}$$

$$\text{(T-GSUM)}\ \frac{|I|\neq 1 \quad \forall i\in I:\ \Gamma\vdash_{\mathcal{E}} a_i(b_i).P_i : \sum_{j\in J_i}\alpha_i\langle\beta_j\rangle.\mathsf{T}_{ij}}{\Gamma\vdash_{\mathcal{E}} \sum_{i\in I} a_i(b_i).P_i : \sum_{i\in I, j\in J_i}\alpha_i\langle\beta_j\rangle\mathsf{T}_{ij}} \qquad \text{(T-ISUM)}\ \frac{\forall i\in I:\ \Gamma\vdash_{\mathcal{E}} P_i:\mathsf{T}_i}{\Gamma\vdash_{\mathcal{E}} \sum_{i\in I}\tau.P_i : \sum_{i\in I}\tau.\mathsf{T}_i}$$

$$\text{(T-PAR)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \Gamma\vdash_{\mathcal{E}} Q:\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} P|Q:\mathsf{T}|\mathsf{S}} \qquad \text{(T-RES)}\ \frac{\Gamma,a:\alpha\vdash_{\mathcal{E}} P:\mathsf{T}}{\Gamma\vdash_{\mathcal{E}} (\nu a:\alpha)P:\mathsf{T}}$$

$$\text{(T-IF)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \Gamma\vdash_{\mathcal{E}} Q:\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} \text{if } a=b \text{ then } P \text{ else } Q : \tau.\mathsf{T}+\tau.\mathsf{S}} \qquad \text{(T-SUB)}\ \frac{\Gamma\vdash_{\mathcal{E}} P:\mathsf{T} \quad \mathsf{T}\lesssim\mathsf{S}}{\Gamma\vdash_{\mathcal{E}} P:\mathsf{S}}$$

Table 3. Typing rules for CCS$^-$ types.

up to strong bisimulation – the same as a replicated summation. Note that, concerning guarded summation, the case with a single input $|I|=1$, (T-INP), is kept separate from the case with $|I|\neq 1$, (T-GSUM), only for ease of presentation. In (T-IF), the behavior of a conditional process is approximated by a type that subsumes both branches of the if-then-else into an internal choice. The subtyping relation $\lesssim$ is the simulation preorder over $\mathcal{E}$, (T-SUB). The rest of the rules should be self-explanatory.

3.3 Results

The subject reduction theorem establishes an operational correspondence between the abstract behavior P_Γ and any type T that can be assigned to P under Γ.

Theorem 1 (subject reduction). $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$ *and* $P_\Gamma \xrightarrow{\lambda} P'_{\Gamma'}$ *imply that there is* T' *such that* $\mathsf{T} \xrightarrow{\lambda} \mathsf{T}'$ *and* $\Gamma' \vdash_{\mathcal{E}} P' : \mathsf{T}'$.

As a corollary, we obtain that T simulates P_Γ; thanks to Theorem 1, it is easy to see that the relation $\mathcal{R} = \{(P_\Gamma, \mathsf{T}) \,|\, \Gamma \vdash_{\mathcal{E}} P : \mathsf{T}\}$ is a simulation relation.

Corollary 1. *Suppose* $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$. *Then* $P_\Gamma \lesssim \mathsf{T}$.

A consequence of the previous result is that safety properties satisfied by a type are also satisfied by the processes that inhabit that type – or, more precisely, by their Γ-abstract versions. Consider the small logic defined in Section 2.4: let us say that $\phi \in \mathcal{L}$ is a *safety* formula if every occurrence of $\langle A \rangle$ and $\langle\langle A \rangle\rangle$ in ϕ is underneath an odd number of negations. The following proposition, follows from Corollary 1 and first principles.

Proposition 1. *Suppose* $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$ *and* ϕ *is a safety formula, with* $\mathsf{T} \models \phi$. *Then* $P_\Gamma \models \phi$.

As a final remark on the type system, consider taking out rule (T-SUB): the new system can be viewed as a (partial) function that for any P computes a minimal type for P, that is, a subtype of all types of P (just read the rules bottom-up).

In the examples we describe below, we shall consider a calculus enriched with polyadic communication and values: these extensions are easy to accommodate. Polyadic communications are written as $\tau\langle \alpha_1, \alpha_2, \cdots, \alpha_n \rangle$, where α_1 is the subject and $\alpha_2, \cdots, \alpha_n$ are the objects; we omit objects that correspond to the unit tag.

Example 1 (factorial). Consider the process F defined below, which is the usual RPC encoding of the factorial function, and the system S, where F is called with an actual parameter n, a result is received on a private channel r and then the received value is printed.

$$F \triangleq !f(n,r).\,\text{if } n = 0 \text{ then } \overline{r}\langle n \rangle \text{ else } (\nu r' : ret)\big(\overline{f}\langle n-1, r' \rangle \,|\, r'(m).\overline{r}\langle n * m \rangle\big)$$
$$S \triangleq (\nu r : ret)(\overline{f}\langle n, r \rangle \,|\, r(m).\overline{print}\langle m \rangle) \,|\, F \ .$$

Let $\mathcal{E} = \{fact[ret], ret[()], pr[()]\}$ and $\Gamma = \{f : fact, print : pr, n : ()\}$. It is not difficult to check that $\Gamma \vdash_{\mathcal{E}} F : \mathsf{T}_F$ and $\Gamma \vdash_{\mathcal{E}} S : \mathsf{T}_S$, where:

$$\mathsf{T}_F \triangleq !fact\langle ret \rangle.\big(\,\tau.\overline{ret} + \tau.(\overline{fact}\langle ret \rangle \,|\, ret.\overline{ret})\,\big) \qquad \mathsf{T}_S \triangleq \overline{fact}\langle ret \rangle \,|\, ret.\overline{pr} \,|\, \mathsf{T}_F$$

and that

$$\mathsf{T}_S \models \phi_1 \triangleq \neg\langle\langle \Lambda - \{fact\langle ret \rangle, ret\} \rangle\rangle\ \langle \overline{ret} \rangle\ \langle\langle \Lambda - \{fact\langle ret \rangle\} \rangle\rangle\ \langle \overline{fact}\langle ret \rangle \rangle$$
$$\mathsf{T}_S \models \phi_2 \triangleq \neg\langle\langle \Lambda - \{fact\langle ret \rangle, ret\} \rangle\rangle\ \langle \overline{pr} \rangle\ \langle\langle \Lambda - \{fact\langle ret \rangle\} \rangle\rangle\ \langle \overline{fact}\langle ret \rangle \rangle$$

meaning that no call at f can be observed after observing a return (ϕ_1) or a print (ϕ_2), that is, as expected, after receipt of 0 as argument, no other calls to f can be produced. Note that in both formulas we are forced to restrict certain actions so as to avoid interaction with the environment (e.g. in the first case we disallow action $fact\langle ret \rangle$): this is the price to pay for getting rid of restriction in types. Formulas ϕ_1 and ϕ_2 express safety properties, thus thanks to Proposition 1, we can conclude that $S_\Gamma \models \phi_1 \wedge \phi_2$.

$$(\text{BPP-INV})\ \frac{\alpha[\beta] \to \mathsf{T} \in \mathcal{E}}{\alpha[\beta] \xrightarrow{\tau\langle\alpha,\beta\rangle}_{\mathcal{E}} \mathsf{T}} \qquad (\text{BPP-INT})\ \textstyle\sum_{i\in I} \tau.\mathsf{T}_i \xrightarrow{\tau}_{\mathcal{E}} \mathsf{T}_j\ (j \in I)$$

$$(\text{BPP-PAR}_{\text{L}})\ \frac{\mathsf{T} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{T}'}{\mathsf{T} \parallel \mathsf{S} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{T}' \parallel \mathsf{S}} \qquad (\text{BPP-PAR}_{\text{R}})\ \frac{\mathsf{T} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{T}'}{\mathsf{S} \parallel \mathsf{T} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{S} \parallel \mathsf{T}'}$$

Table 4. Operational semantics of BPP types.

4 BPP types for closed behavior

We focus here on the closed behavior of abstract processes. Although the system in the previous section already takes into account closed behavior, it is possible in this case to obtain a "direct style" behavioral type system by getting rid of synchronization in types. This is achieved by directly associating each output action with an *effect*, corresponding to the observable behavior that that output can trigger. Input actions have no associated effect, thus an inert type is associated to input guarded processes. The types we obtain in this way are precisely Basic Parallel Processes (BPP, [4]). We show that if we restrict ourselves to a particular class of processes, notably to (a generalization of) uniform receptive processes [26], a bisimulation relation relates processes and their types.

4.1 BPP Types

The set $\mathcal{T}_{\text{BPP}}$ of BPP types, ranged over by $\mathsf{T}, \mathsf{S}, \ldots$, is defined by the following syntax:

$$\mathsf{T} ::= \alpha[\beta]\ (\textit{Invocation}) \mid \sum_{i\in I} \tau.\mathsf{T}_i\ (\textit{Internal Choice}) \mid \mathsf{T} \parallel \mathsf{T}\ (\textit{Interleaving})$$

where $\alpha \neq ()$. We consider an extended tag sorting system where each element $\alpha[\beta]$ is enriched with an effect T, written $\alpha[\beta] \to \mathsf{T}$. More precisely, we let $\mathcal{E}$ be a set of rules of the form $\{\alpha_i[\beta_i] \to \mathsf{T}_i \mid 1 \leq i \leq n\}$. This can be viewed as a set of rules defining a set of BPP processes. In particular, a process invocation $\alpha[\beta]$ activates the corresponding rule in $\mathcal{E}$; the rest of the syntax and operational semantics should be self-explanatory (Table 4). In what follows we write nil for $\sum_{i\in\emptyset} \tau.\mathsf{T}_i$, and often omit dummy nil's, writing e.g. $\mathsf{T}|\mathsf{nil}$ simply as T.

4.2 Typing rules

Again, we consider contexts Γ of the form $\{a_1 : \alpha_1, ..., a_n : \alpha_n\}$. The new type system is defined in Table 5. Derivable statements take now the form $\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T}$, where Γ and $\mathcal{E}$ are respectively a fixed context and extended tag sorting system. The parameter $\mathcal{E}'$ is used to keep track of rules of $\mathcal{E}$ actually used in the derivation: this extra parameter will be useful to formulate a condition under which a bisimulation relation, rather than simply a simulation, can be established between a type and its inhabiting processes.

A brief explanation of the typing rules follows. Rule (T-BPP-O) ensures that there are some effects associated to the output action. In (T-BPP-INP) and (T-BPP-REP),

$$\text{(T-BPP-INP)}\ \frac{\Gamma \vdash a : \alpha \quad \forall \beta_i \text{ s.t. } \alpha[\beta_i] \in \mathrm{dom}(\mathcal{E}) \quad \exists \mathsf{T}_i \text{ s.t. } \alpha[\beta_i] \to \mathsf{T}_i \in \mathcal{E} \text{ and:} \quad \Gamma, b : \beta_i \vdash_{\mathcal{E};\mathcal{E}_i} P : \mathsf{T}_i \quad \mathcal{E}' = \bigcup_i (\mathcal{E}_i \cup \{\alpha[\beta_i] \to \mathsf{T}_i\})}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} a(b).P : \mathsf{nil}}$$

$$\text{(T-BPP-REP)}\ \frac{\Gamma \vdash a : \alpha \quad \forall \beta_i \text{ s.t. } \alpha[\beta_i] \in \mathrm{dom}(\mathcal{E}) \quad \exists \mathsf{T}_i \text{ s.t. } \alpha[\beta_i] \to \mathsf{T}_i \in \mathcal{E} \text{ and:} \quad \Gamma, b : \beta_i \vdash_{\mathcal{E};\mathcal{E}_i} P : \mathsf{T}_i \quad \mathcal{E}' = \bigcup_i (\mathcal{E}_i \cup \{\alpha[\beta_i] \to \mathsf{T}_i\})}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} !a(b).P : \mathsf{nil}}$$

$$\text{(T-BPP-O)}\ \frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash b : \beta \quad \exists \mathsf{T} :\ \alpha[\beta] \to \mathsf{T} \in \mathcal{E}}{\Gamma \vdash_{\mathcal{E};\emptyset} \overline{a}\langle b \rangle : \alpha[\beta]} \qquad \text{(T-BPP-RES)}\ \frac{\Gamma, a : \alpha \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T}}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} (\nu a : \alpha) P : \mathsf{T}}$$

$$\text{(T-BPP-GSUM)}\ \frac{|I| \neq 1 \quad \forall i \in I :\ \Gamma \vdash_{\mathcal{E};\mathcal{E}_i} a_i(b_i).P_i : \mathsf{nil} \quad \mathcal{E}' = \bigcup_i \mathcal{E}_i}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} \sum_{i \in I} a_i(b_i).P_i : \mathsf{nil}}$$

$$\text{(T-BPP-ISUM)}\ \frac{\forall i \in I :\ \Gamma \vdash_{\mathcal{E};\mathcal{E}_i} P_i : \mathsf{T}_i \quad \mathcal{E}' = \bigcup_i \mathcal{E}_i}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} \sum_{i \in I} \tau.P_i : \sum_{i \in I} \tau.\mathsf{T}_i} \qquad \text{(T-BPP-SUB)}\ \frac{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T} \quad \mathsf{T} \lesssim^{\mathrm{c}} \mathsf{S}}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{S}}$$

$$\text{(T-BPP-PAR)}\ \frac{\Gamma \vdash_{\mathcal{E};\mathcal{E}_1} P : \mathsf{T} \quad \Gamma \vdash_{\mathcal{E};\mathcal{E}_2} Q : \mathsf{S} \quad \mathcal{E}' = \mathcal{E}_1 \cup \mathcal{E}_2}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P | Q : \mathsf{T} \parallel \mathsf{S}}$$

$$\text{(T-BPP-IF)}\ \frac{\Gamma \vdash_{\mathcal{E};\mathcal{E}_1} P : \mathsf{T} \quad \Gamma \vdash_{\mathcal{E};\mathcal{E}_2} Q : \mathsf{S} \quad \mathcal{E}' = \mathcal{E}_1 \cup \mathcal{E}_2}{\Gamma \vdash_{\mathcal{E};\mathcal{E}'} \text{if } a = b \text{ then } P \text{ else } Q : \tau.\mathsf{T} + \tau.\mathsf{S}}$$

Table 5. Typing rules for BPP.

$\mathrm{dom}(\mathcal{E})$ denotes the set of all elements $\alpha[\beta]$'s occurring in $\mathcal{E}$: hence all object tags β_i associated with the subject tag α are taken into account. For each of them, it is checked that the effects produced by the continuation process P are those expected by the corresponding rule in $\mathcal{E}$. As previously mentioned, input has no associated effect, hence the resulting type is nil. In (T-BPP-SUB), note that the subtyping relation is now the closed simulation preorder $\lesssim^{\mathrm{c}}$. The other rules are standard. In what follows we write $\Gamma \vdash_{\mathcal{E}} P$ if there exist $\mathcal{E}'$ and T such that $\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T}$.

4.3 Results

The results obtained in Section 3.3 can be extended to the new system.

Theorem 2 (main results on $\vdash_{\mathcal{E};\mathcal{E}'}$). *Suppose $\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T}$. Then: (a) $P_\Gamma \xrightarrow{\lambda} P'_{\Gamma'}$ implies that there are a T' and $\mathcal{E}'' \subseteq \mathcal{E}'$ such that $\mathsf{T} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{T}'$ and $\Gamma \vdash_{\mathcal{E};\mathcal{E}''} P' : \mathsf{T}'$; (b) $P_\Gamma \lesssim^{\mathrm{c}} \mathsf{T}$; (c) safety formulas satisfied by T are also satisfied by P_Γ.*

Example 2 (factorial). Consider the processes defined in Example 1, the same context Γ and a new system augmented with a (stub) printing service: $S' = S \mid !print(d)$. In what follows, we omit the unit tag and write e.g. $\alpha \to \beta$ for $\alpha[()] \to \beta[()]$ Consider the following extended tag sorting system

$$\mathcal{E} = \{fact[ret] \to (\tau.ret + \tau.fact[ret]),\ ret \to ret,\ ret \to pr,\ pr \to \mathsf{nil}\}\ .$$

Let $\Lambda^\tau \subseteq \Lambda$ be the set of communication labels (all labels of the form $\tau\langle\alpha, \tilde{\beta}\rangle$). It is not difficult to prove that $\Gamma \vdash_{\mathcal{E}} S' : fact[ret]$ (note that subtyping plays an essential role in this derivation). Moreover, it holds that $fact[ret] \vDash \phi'_1, \phi'_2$, where ϕ'_1 and ϕ'_2 are the versions of ϕ_1 and ϕ_2 defined in Example 1 with visible actions replaced by silent ones:

$$\phi'_1 \stackrel{\triangle}{=} \neg\langle\langle\Lambda^\tau\rangle\rangle\, \langle\tau\langle ret\rangle\rangle\, \langle\langle\Lambda^\tau\rangle\rangle\, \langle\tau\langle fact, ret\rangle\rangle \qquad \phi'_2 \stackrel{\triangle}{=} \neg\langle\langle\Lambda^\tau\rangle\rangle\, \langle\tau\langle pr\rangle\rangle\, \langle\langle\Lambda^\tau\rangle\rangle\, \langle\tau\langle fact, ret\rangle\rangle\ .$$

Formulas ϕ'_1 and ϕ'_2 express safety properties, hence thanks to Theorem 2, we can conclude the analog of what shown in Example 1: $S'_\Gamma \vDash \phi'_1 \wedge \phi'_2$.

In several cases, the simulation preorder relating processes and their types is unnecessarily over-approximating. The rules for conditional and subtyping are obvious source of over-approximation, as well as the presence in $\mathcal{E}$ of dummy rules that are not actually used in type-checking the process: these are sources of "fake" transitions in types, that is, transitions with no correspondence in processes. A subtler problem is raised by input prefixes. Input prefixes correspond to rules in $\mathcal{E}$: but while an input prefix may never become available, and a (non-replicated) input disappears upon synchronization, the corresponding rules in $\mathcal{E}$ are always available and may give rise to fake transitions in types. In the rest of the section we show that, for processes enjoying a certain "uniform receptiveness" condition with respect to Γ (Definition 1), a bisimilarity relation between processes and types can be established. In this case, the abstract process and its type satisfy the same properties.

Let us introduce some extra notation and terminology first. In what follows, $\equiv$ will denote the standard structural congruence in pi-calculus (see e.g. [18]), while $\mathrm{out}(P)$ (resp. $\mathrm{inp}(P)$) will denote the set of free names occurring in some output (resp. input) action in P. Moreover, we define $\Gamma^{-1}(\alpha)$ as the set $\{a | \Gamma \vdash a : \alpha\}$ and define contexts as $C ::= (\nu a : \alpha)C \mid P|C \mid a(b).C \mid [\,]$. A process P is *input-local* if for every action prefix $a(x).Q$ in P, replicated or not, it holds $x \notin \mathrm{inp}(Q)$. Finally, a *receptor* is a process of the form $(\nu a)\left(\sum_{i\in I} a_i(x).P_i \mid \prod_{j\in J} !a(x).Q_j\right)$ such that $a \notin \mathrm{inp}(P_i, Q_j)$ for each i, j.

The definition below has a simple explanation: each tag should correspond to a unique receptor, and the latter should be immediately available to any potential sender. This somehow generalizes Sangiorgi's uniform receptiveness [26]. In particular, it is straightforward to modify the type system in [26] so that well-typed processes are uniform receptive in our sense. A more general technique for proving Γ-uniform receptiveness in concrete cases is given by its co-inductive definition: that is, finding a relation that satisfies the conditions listed in the definition and contains the given Γ and P. Below, we use $\to$ as an abbreviation of $\stackrel{\tau\langle\alpha,\beta\rangle}{\longrightarrow}$ for some α and β.

Definition 1 (Γ-uniform receptiveness). *Let $\theta : \alpha \mapsto R_\alpha$ be a function from tags to receptors. We let $\rhd_\theta$ be the largest relation over contexts and input-local processes such that whenever $\Gamma \rhd_\theta P$ then:*

1. *for each* α *such that* $\Gamma^{-1}(\alpha) \cap \mathrm{out}(P) \neq \emptyset$ *it holds that*
   (a) $\forall a \in \Gamma^{-1}(\alpha)$ *there are* R *and* Q *such that* $(\nu a)P \equiv (\nu a)(R|Q)$ *with* $a \notin \mathrm{inp}(Q)$ *and* $(\nu a)R = R_\alpha$*;*
   (b) *whenever* $P \equiv C[(\nu a : \alpha)P']$ *there are* R' *and* Q' *such that* $P' \equiv R'|Q'$ *with* $a \notin \mathrm{inp}(Q')$ *and* $(\nu a)R' = R_\alpha$*;*
2. *whenever* $P \equiv (\nu a : \alpha)P'$ *with* $a \in \mathrm{out}(P')$ *then* $\Gamma, a : \alpha \rhd_\theta P'$*;*
3. *whenever* $P \to P'$ *then* $\Gamma \rhd_\theta P'$.

We write $\Gamma \rhd P$*, and say* P *is* Γ-uniform receptive*, if* $\Gamma \rhd_\theta P$ *for some* θ.

In what follows, we write $\Gamma \vdash^{-}_{\mathcal{E}} P : \mathsf{T}$ if $\Gamma \vdash_{\mathcal{E};\mathcal{E}'} P : \mathsf{T}$ is derived without using rules (T-BPP-SUB) and (T-BPP-IF), and $\mathcal{E} = \mathcal{E}'$.

Theorem 3. *Suppose* P *is* Γ*-uniform receptive and that* $\Gamma \vdash^{-}_{\mathcal{E}} P : \mathsf{T}$*. Then* $P_\Gamma \sim^{\mathrm{c}} \mathsf{T}$.

5 An extended example

A simple printing system is defined where users are required to authenticate for being allowed to print. Users are grouped into trusted and untrusted, which are distinguished by two groups of credentials: $\{c_i \mid i \in I\}$ (also written $\tilde{c}_i$) for trusted and $\{c_j \mid j \in J\}$ (also written $\tilde{c}_j$) for untrusted, with $\tilde{c}_i \cap \tilde{c}_j = \emptyset$. Process A is an authentication server that receives from a client its credential c, a return channel r and an error channel e and then sends both r and e to a credential-handling process T. If the client is untrusted, T produces an error, otherwise a private connection between the client and the printer is established, by creating a new communication link k and passing it to C. C simulates the cumulative behavior of all clients: nondeterministically, it tries to authenticate by using credential c_l, for an $l \in I \cup J$, and waits for the communication link with the printer, on the private channel r, and for an error, on the private channel e. After printing, or receiving an error, C's execution restarts.

We expect that every printing request accompanied by trusted credentials will be satisfied, and that every print is preceded by an authentication request.

$$
\begin{aligned}
\mathit{Sys} &\triangleq (\nu a : \mathit{aut}, \tilde{c}_i : \mathit{ok}, \tilde{c}_j : \mathit{nok}, M : (), \mathit{print} : \mathit{pr})\big(T \mid C \mid A \mid !\mathit{print}(d)\big)\\
T &\triangleq \textstyle\prod_{i\in I} !c_i(x,e).(\nu k : \mathit{key})\big(\overline{x}\langle k\rangle \mid k(d).\overline{\mathit{print}}\langle d\rangle\big) \mid \prod_{j\in J} !c_j(x,e).\overline{e}\\
A &\triangleq !a(c,r,e).\overline{c}\langle r,e\rangle\\
C &\triangleq (\nu i : \mathit{iter})\Big(\bar{i} \mid !i.(\nu r : \mathit{ret}, e : \mathit{err})\big(\textstyle\sum_{l\in I\cup J} \tau.\overline{a}\langle c_l, r, e\rangle \mid r(z).((\overline{z}\langle M\rangle \mid \bar{i}) + e.\bar{i})\Big)
\end{aligned}
$$

*Example 3 (*CCS^{-} *types).* Consider the tag sorting system

$$
\begin{aligned}
\mathcal{E} = \{&\mathit{aut}[\mathit{ok}, \mathit{ret}, \mathit{err}], \mathit{aut}[\mathit{nok}, \mathit{ret}, \mathit{err}], \mathit{ok}[\mathit{ret}, \mathit{err}],\\
&\mathit{nok}[\mathit{ret}, \mathit{err}], \mathit{ret}[\mathit{key}], \mathit{pr}[()], \mathit{err}[], \mathit{key}[()], \mathit{iter}[]\} \ .
\end{aligned}
$$

It is easy to prove that $\emptyset \vdash_{\mathcal{E}} Sys : \mathsf{T}_T \mid \mathsf{T}_A \mid \mathsf{T}_C \mid !pr = \mathsf{T}$, where

$$\mathsf{T}_T \triangleq !ok\langle ret, err\rangle.\left(\overline{ret}\langle key\rangle \mid key.\overline{pr}\right) \mid !nok\langle ret, err\rangle.\overline{err}$$
$$\mathsf{T}_A \triangleq !aut\langle ok, ret, err\rangle.\overline{ok}\langle ret, err\rangle \mid !aut\langle nok, ret, err\rangle.\overline{nok}\langle ret, err\rangle$$
$$\mathsf{T}_C \triangleq \overline{iter} \mid !iter.\left((\tau.\overline{aut}\langle ok, ret, err\rangle + \tau.\overline{aut}\langle nok, ret, err\rangle) \mid (ret\langle key\rangle.(\overline{key}|\overline{iter}) + err.\overline{iter})\right).$$

Furthermore, it holds that

$$\mathsf{T} \vDash \phi_3 \triangleq \neg\langle\langle \Lambda - \{nok\langle ret, err\rangle, aut\langle nok, ret, err\rangle, \tau\langle aut, nok, ret, err\rangle\}\rangle\rangle\ \langle\overline{err}\rangle$$
$$\mathsf{T} \vDash \phi_4 \triangleq \neg\langle\langle \Lambda - \{ok\langle ret, err\rangle, aut\langle ok, ret, err\rangle, \tau\langle aut, ok, ret, err\rangle\}\rangle\rangle\ \langle\overline{pr}\rangle$$

that is, *error* is always generated by an authentication request containing untrusted credentials, and every *pr* action is preceded by a successful authentication request. Both formulas express safety properties, hence Proposition 1 ensures that are both satisfied by the abstract process $Sys_{\emptyset}$.

*Example 4 (*BPP *types).* Consider the system *Sys* previously defined; in this example we show that a BPP type for *Sys*, which allow a more precise analysis of the system can be obtained. Consider the following extended tag sorting system

$$\begin{aligned}\mathcal{E} = \{\ & aut[ok, ret, err] \to ok[ret, err],\quad ok[ret, err] \to ret[key],\ err \to iter,\\ & aut[nok, ret, err] \to nok[ret, err],\ nok[ret, err] \to err,\quad key \to pr,\quad pr \to \mathsf{nil},\\ & ret[key] \to (key \mid iter),\quad iter \to \tau.aut[ok, ret, err] + \tau.aut[nok, ret, err]\}.\end{aligned}$$

First, it is easy to see that $\emptyset \vdash^{-}_{\mathcal{E}} Sys : iter$. Moreover it is not difficult to prove, by coinduction, that *Sys* is $\emptyset$-uniform receptive. Hence from

$$iter \vDash \phi_3' \triangleq [\![\Lambda^{\tau}]\!]\ [\tau\langle aut, ok, ret, err\rangle]\ \langle\langle\Lambda^{\tau}\rangle\rangle\ \langle\tau\langle pr\rangle\rangle$$
$$iter \vDash \phi_4' \triangleq [\![\Lambda^{\tau}]\!]\ [\tau\langle aut, nok, ret, err\rangle]\ \langle\langle\Lambda^{\tau}\rangle\rangle\ \langle\tau\langle err\rangle\rangle$$

and Theorem 3, $Sys_{\emptyset} \vDash \phi_3' \wedge \phi_4'$, that is, every authentication request accompanied by trusted credentials will be followed by a *pr*, while every untrusted request will be followed by an *err*.

6 Join calculus and Petri nets types

We extend the treatment of the previous section to the Join calculus [7]. We shall only consider the case of *closed* behavior; the open case requires some more notational burden and we leave it for an extended version of the paper. The essential step we have to take, at the level of types, is now moving from BPP to Petri nets. Technically, this leap is somehow forced by the presence of the join pattern construct in the calculus. In the context of infinite states transition systems [6,9], the leap corresponds precisely to moving from rewrite rules with a single nonterminal on the LHS (BPP) to rules with multisets of nonterminals on the LHS (PN).

Processes	$P,Q ::=$	$\overline{a}\langle b\rangle$	*Output*
		$\mid$ def D in P	*Definition*
		$\mid P \mid P$	*Parallel*
Definitions	$D ::=$	$J \triangleright P$	*Pattern*
		$\mid D \wedge D$	*Conjunction*
Patterns	$J ::=$	$a^{\alpha}(b)$	*Input pattern*
		$\mid J \parallel J$	*Join pattern*

Table 6. Syntax of the Join calculus.

6.1 Processes and types

The syntax of the calculus is given in Table 6. Note that we consider a "pure" version of the calculus without conditionals. Adding if-then-else is possible, but again implies some notational burden at the level of types (notably, simulating an internal choice operator with Petri nets), which we prefer to avoid. Over this language, we presuppose the standard notions of binding, alpha-equivalence, structural congruence $\equiv$ and (tag-annotated) reduction semantics $\xrightarrow{\mu}$, where $\mu ::= \tau\langle(\ell_1,\ell'_1),\ldots,(\ell_n,\ell'_n)\rangle$. Here $(\ell_1,\ell'_1),\ldots,(\ell_n,\ell'_n)$ must be regarded as a *multiset* of pairs (subject, object). For a context Γ, the transitional semantics $\xrightarrow{\lambda}$ of the abstract processes P_Γ is defined as expected, where $\lambda = \tau\langle(\alpha_1,\beta_1),\ldots,(\alpha_n,\beta_n)\rangle$. A type is a multiset $\mathsf{T} ::= \alpha_1[\beta_1],\ldots,\alpha_n[\beta_n]$ that is, "," is an associative and commutative operator with the empty multiset as unit. Types will play the role of markings in a Petri net. We consider a tag sorting system $\mathcal{E}$ containing elements of the form $\mathsf{T} \to \mathsf{S}$, to be interpreted as transitions of a Petri net. More precisely, we shall fix a set of rules $\mathcal{E} = \{\mathsf{T}_i \to \mathsf{S}_i \mid 1 \le i \le n\}$, where the following *uniformity* condition is satisfied by $\mathcal{E}$: let $\mathrm{dom}(\mathcal{E})$ be the set of all $\alpha[\beta]$'s occurring in $\mathcal{E}$; then for every α, β and β' such that $\alpha[\beta],\alpha[\beta'] \in \mathrm{dom}(\mathcal{E})$, if $\alpha[\beta],\mathsf{T} \to \mathsf{S} \in \mathcal{E}$ then also $\alpha[\beta'],\mathsf{T} \to \mathsf{S}' \in \mathcal{E}$ for some S'. The operational semantics of types is then defined by the rules below, which make it clear that $\mathcal{E}$ is a Petri net, and a type T is a marking of this net.

$$\text{(J-T-COM)}\ \frac{\alpha_1[\beta_1],\ldots,\alpha_n[\beta_n] \to \mathsf{T} \in \mathcal{E}}{\alpha_1[\beta_1],\cdots,\alpha_n[\beta_n] \xrightarrow{\tau\langle(\alpha_1,\beta_1),\ldots,(\alpha_n,\beta_n)\rangle} \mathsf{T}} \qquad \text{(J-T-PAR)}\ \frac{\mathsf{T} \xrightarrow{\lambda} \mathsf{T}'}{\mathsf{T},\mathsf{S} \xrightarrow{\lambda} \mathsf{T}',\mathsf{S}}$$

6.2 Typing rules and results

The typing rules are defined in Table 7. The rules generalizes as expected those of Section 4. In the rules, we use a function $\mathrm{tags}(D)$ that extracts tags associated with

$$\text{(T-J-DEF)}\ \frac{\Gamma \vdash_{\mathcal{E}} D : \mathsf{nil} \quad \Gamma, \mathrm{tags}(D) \vdash_{\mathcal{E}} P : \mathsf{T}}{\Gamma \vdash_{\mathcal{E}} \mathrm{def}\ D\ \mathrm{in}\ P : \mathsf{T}} \qquad \text{(T-J-PAR)}\ \frac{\Gamma \vdash_{\mathcal{E}} P : \mathsf{T} \quad \Gamma \vdash_{\mathcal{E}} Q : \mathsf{S}}{\Gamma \vdash_{\mathcal{E}} P | Q : \mathsf{T}, \mathsf{S}}$$

$$\text{(T-J-CON)}\ \frac{\Gamma \vdash_{\mathcal{E}} D_i : \mathsf{nil} \quad i = 1, \ldots, n}{\Gamma \vdash_{\mathcal{E}} D_1 \wedge \cdots \wedge D_n : \mathsf{nil}} \qquad \text{(T-J-SUB)}\ \frac{\Gamma \vdash_{\mathcal{E}} P : \mathsf{T} \quad \mathsf{T} \lesssim^{\mathrm{c}} \mathsf{S}}{\Gamma \vdash_{\mathcal{E}} P : \mathsf{S}}$$

$$\text{(T-J-OUT)}\ \frac{\Gamma \vdash a : \alpha \quad \Gamma \vdash b : \beta \quad \alpha[\beta] \in \mathrm{dom}(\mathcal{E})}{\Gamma \vdash_{\mathcal{E}} \overline{a}\langle b \rangle : \alpha[\beta]}$$

$$\text{(T-J-PAT)}\ \frac{\begin{array}{c} J = a_1{}^{\alpha_1}(c_1) \parallel \cdots \parallel a_n{}^{\alpha_n}(c_n) \quad \forall \beta_{k_1}, \ldots, \beta_{k_n}\ \mathrm{s.t.}\ \alpha_i[\beta_{k_i}] \in \mathrm{dom}(\mathcal{E}) \quad \exists \mathsf{T}_k\ \mathrm{s.t.} \\ \alpha_1[\beta_{k_1}], \ldots, \alpha_n[\beta_{k_n}] \to \mathsf{T}_k \in \mathcal{E} \wedge \Gamma, \mathrm{tags}(D), c_1 : \beta_{k_1}, \ldots, c_n : \beta_{k_n} \vdash_{\mathcal{E}} P : \mathsf{T}_k \\ \mathcal{E}' = \bigcup_k \{\alpha_1[\beta_{k_1}], \ldots, \alpha_n[\beta_{k_n}] \to \mathsf{T}_k\} \end{array}}{\Gamma \vdash_{\mathcal{E}} J \triangleright P : \mathsf{nil}}$$

Table 7. Typing rules for the Join calculus and Petri nets.

definitions, as follows:

$$\begin{array}{ll} \mathrm{tags}(D_1 \wedge \cdots \wedge D_n) = \bigcup_{i=1,\ldots,n} \mathrm{tags}(D_i) & \mathrm{tags}(J \triangleright P) = \mathrm{tags}(J) \\ \mathrm{tags}(J_1 \parallel \cdots \parallel J_n) = \bigcup_{i=1,\ldots,n} \mathrm{tags}(J_i) & \mathrm{tags}(a^{\alpha}(c)) = \{a : \alpha\} \end{array} .$$

We have the following result.

Theorem 4. *Suppose* $\Gamma \vdash_{\mathcal{E}} P : \mathsf{T}$. *Then: (a) if* $P_\Gamma \xrightarrow{\lambda} P'_{\Gamma'}$ *then there exists a* T' *such that* $\mathsf{T} \xrightarrow{\lambda}_{\mathcal{E}} \mathsf{T}'$ *and* $\Gamma' \vdash_{\mathcal{E}} P' : \mathsf{T}'$*; and (b)* $P_\Gamma \lesssim^{\mathrm{c}} \mathsf{T}$.

7 Conclusions and related works

We have proposed methods for statically abstracting propositional approximations of first-order process calculi (pi-calculus and Join). These methods take the form of behavioral type systems. Correspondingly, three classes of types have been considered: restriction-free CCS, BPP and Petri nets. Concerning type reconstruction, we give methods to compute minimal types under certain assumptions on processes, but leave the development of explicit type inference algorithms for future work.

In Igarashi and Kobayashi's work [10], types are restriction-free CCS processes and output prefixes are allowed. Roughly, types are obtained from pi-processes by replacing any bound subject with the corresponding tag, and turning each object into a CCS-annotation describing the behavior of the prefix continuation. Depending on the actual instantiation of this framework, the type checking algorithm of [10] may need to call analysis procedures that check run-time properties of types (well-formedness). Our work is mostly inspired by [10], but we try to simplify its approach by considering an asynchronous version of the calculus and by extracting a tag-wise, rather than channel-wise, behavior of processes. On one hand, this simplification leads to some loss of information, which prevents us from capturing certain liveness properties such as race-

and deadlock-freedom. On the other hand, it allows us to make the connection between different kinds of behavior (open/closed) and different type models (CCS/BPP) direct and explicit. As an example, in the case of BPP we can spell out reasonably simple conditions under which the type analysis is "precise" (Γ-uniform receptiveness). Also, our approach naturally carries over to the Join calculus, by moving to Petri nets types.

The papers [23,3] present type systems inspired by [10]. The main difference between these works and Igarashi and Kobayashi's, is that behavioral types here are more precise than in [10], because described by using full CCS. An open (in the sense of [25]) version of simulation is used as a subtyping relation. Undecidability of (bi)simulation on CCS with restriction is somehow bypassed by providing an ad-hoc assume-guarantee principle to discharge safety checks at name restriction in a modular way.

Igarashi and Kobayashi' type system was inspired by work on the analysis of various properties of concurrent programs, notably linearity [12] and deadlock-freedom [11,13,14]. Nowadays, the list of such properties has grown, so as to include several forms of refined lock-freedom and resource usage analyses [15,16,17].

Concerning the Join calculus, previous work on type systems [8,21] proposed a functional typing à la ML and a type system à la Hindley/Milner. The analogy between Join and Petri nets was first noticed in [1] and then in [22]. In [2], Buscemi and Sassone classify join processes by comparison with different classes of Petri nets. Four distinct type systems are proposed, that give rise to a hierarchy comprising four classes of join processes. These classes are shown to be encodable respectively into place/transition Petri nets, Colored nets, Reconfigurable nets and Dynamic nets. While only the last class contains all join terms, note that only place/transition Petri nets are actually "propositional" and enjoy effective analysis techniques. In other words, the emphasis of [2] is on assessing expressiveness of Join rather than on computing tractable approximations of processes.

More loosely related to our work, there is a strong body of research on behavioral types for object calculi. Notably, [20,19] put forward behavioral types for object interfaces; these types are based on labelled transition systems that specify the possible sequences of calls at available methods (services). Our extended tag sorting systems are reminiscent of this mechanism. Similarly, in [24], a behavioral typing discipline for TyCO, a name-passing calculus of concurrent objects, is introduced. Types are defined by using graphs and the type compatibility relation is a bisimulation. This work is related to Yoshida's paper [28], where *graph types* are used for proving full abstraction of translations from sorted polyadic pi-terms into monadic ones.

References

1. A. Asperti and N. Busi. Mobile Petri Nets. Technical Report UBLCS 96-10, Università di Bologna, 1996.
2. M. G. Buscemi and V. Sassone. High-Level Petri Nets as Type Theories in the Join Calculus. In *Proc. of FoSSaCS*, *LNCS*, 2030:104–120, 2001.
3. S. Chaki, S. K. Rajamani and J. Rehof. Types as Models: Model Checking Message-Passing Programs. In *Proc. of POPL*, ACM press, pp. 45-57, 2002.
4. S. Christensen, Y. Hirshfeld and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In *Proc. of CONCUR*, *LNCS*, 715:143–157, 1993.

5. S. Christensen, Y. Hirshfeld and F. Moller. Decidable Subsets of CCS. *The Computer Journal*, 37(4):233–242, 1994.
6. J. Esparza. More Infinite Results. *Current trends in Theoretical Computer Science: entering the 21st century*, pp.480-503, 2001.
7. C. Fournet and G. Gouthier. The Reflexive Chemical Abstract Machine and the Join Calculus. In *Proc. of POPL*, ACM press, pp. 372-385, 1996.
8. C. Fournet, C. Laneve, L. Maranget and D. Remi. Implicit Typing à la ML for the join-calculus. In *Proc. of CONCUR*, *LNCS*, 1243:196–212, 1997.
9. Y. Hirshfeld and F. Moller. Decidability Results in Automata and Process theory. *LNCS*, 1043:102–148, 1996.
10. A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. In *Proc. of POPL*, ACM press, pp.128-141, 2001. Full version appeared in *TCS*, 311(1-3):121–163, 2004.
11. N. Kobayashi. A partially deadlock-free typed process calculus. In *Proc. of LICS*, 1997. Full version in *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.
12. N. Kobayashi, B.C. Pierce and D.N. Turner. Linearity and the Pi-Calculus. In *Proc. of POPL*, 1996. Full version in *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
13. N. Kobayashi, S. Saito and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR*, *LNCS*, 1877:489–503, 2000. Full version appeared as Technical Report TR00-01, Department of Information Science, University of Tokyo, January 2000.
14. N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.
15. N. Kobayashi. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informartica*, 42(4-5): 291–347, 2005.
16. N. Kobayashi. A New Type System for deadlock-Free Processes. In *Proc. of CONCUR*, *LNCS*, volume 4137, 2006.
17. N. Kobayashi, K. Suenaga and L. Wischik. Resource Usage Analysis for the Pi-Calculus. In *Proc. of VMCAI*, *LNCS*, 3855:298–312, 2006.
18. R. Milner. The polyadic π-calculus: a tutorial. Tec.Rep. LFCS report ECS-LFCS-91-180, 1991. Also in *Logic and Algebra of Specification*, Springer-Verlag, pp.203-246, 1993.
19. E. Najm, A. Nimour and J. B. Stefani. Infinite types for distributed object interfaces. In *Proc. of FMOODS*, *IFIP Conference Proceedings*, volume 139, 1999.
20. E. Najm and A. Nimour. Explicit behavioral typing for object interfaces. In *Proc. of ECOOP Workshops*, *LNCS*, volume 1743, 1999.
21. M. Odersky, C. Zenger, M. Zenger and G. Chen. A Functional View of Join. Technical Report, ACRC-99-016, University of South Australia, 1999.
22. M. Odersky. Functional Nets. In *Proc. of ESOP*, *LNCS*, 1782:1–25, 2000.
23. S. K. Rajamani and J. Rehof. A Behavioral Module System for the Pi-Calculus. In *Proc. of SAS*, *LNCS*, 2126:375–394, 2001.
24. A Ravara and V. T. Vasconcelos. Behavioral types for a calculus of concurrent objects. In *Proc. of Euro-Par*, *LNCS*, 1300:554–561, 1997.
25. D. Sangiorgi. A theory of bisimulation for π-calculus. *Acta Informatica*, 33, 1996.
26. D. Sangiorgi. The name discipline of uniform receptiveness. In *Proc. of ICALP*, 1997. *TCS*, 221(1-2):457–493, 1999.
27. C. Stirling. Modal Logics for Communicating Systems. *TCS*, 49(2-3):311–347, 1987.
28. N. Yoshida. Graph types for monadic mobile processes. In *Proc. of FST/TCS*, *LNCS*, 1180:371–386, 1996.