The Receptive Distributed π-Calculus

(Extended Abstract)

Roberto M. Amadio
Université de Provence, Marseille

Gérard Boudol
INRIA, Sophia-Antipolis

Cédric Lhoussaine
Université de Provence, Marseille

Abstract

In this paper we study an asynchronous distributed π-calculus, with constructs for localities and migration. We show that a simple static analysis ensures the receptiveness of channel names, which, together with a simple type system, guarantees a local deadlock-freedom property, that we call message deliverability. This property states that any migrating message will find an appropriate receiver at its destination locality. We argue that this distributed, receptive calculus is still expressive enough, by giving a series of example illustrating the "receptive style" of programming we have. Finally we show that our calculus contains the π_1-calculus, up to weak asynchronous bisimulation.

1. Introduction

In this paper we study a simplified version of Hennessy and Riely's *distributed π-calculus* Dπ [10]. This is a calculus based on the asynchronous, polyadic π-calculus, involving explicit notions of *locality* and *migration*, that is code movement from a location to another. In this model communication is purely local, so that messages to remote ressources (i.e. receivers, in π-calculus terminology) must be explicitly routed. In such a model – as opposed to other ones like the JOIN calculus [8, 9] or the π_{1l}-calculus [2] where messages transparently go through the network to reach their (unique) destination – the problem arises of how to avoid local deadlocks. More precisely, we would like to avoid the situation where a message migrates to, or stays in a locality where no receiver will ever be available.

To solve this problem, we must ensure that a resource will be available when requested. An obvious way to achieve this is to enforce *receptiveness* – following Sangiorgi's terminology [13] – of any (private) channel name, that is the property that at any time a process is able of offering an input on that channel name. However, the kind of receptiveness we are looking for is neither the "uniform", nor the "linear" one described in [13]. Indeed, requiring each (private) name to be either uniform or linear receptive would result in a dramatic loss of expressive power. Then we are seeking for a formalization of something like the (unique) naming of a persistent "object", which may encapsulate a state – we also wish to ensure the desirable property of unicity of receivers, that has been advocated by Fournet and Gonthier in [8] as a requirement for a language to be implementable in a distributed manner.

Work partially supported by the RNRT project MARVEL.

It is actually not very difficult to design a simple inference system for statically checking processes, in such a way that private names are (uniquely) receptive inside their scope: the system is a refinement of the one of π_{1l}, where we impose that recursive processes have a unique input parameter. Then for instance an input cannot be nested inside another, unless they are input on the same name. This is not enough to entail the local deadlock-freedom property however, since we not only have channel names, but also location names, and other names, called keys, that may be compared for equality but cannot be used as communication channels – the latter are crucially needed to retain the expressive power, as we shall see. In particular, the restriction operation $(\nu u)P$ of the π-calculus must be refined, because we do not demand a location name or a key to be receptive – there is no notion of a receiver for such a name – and still we want such a name to be sometimes restricted in scope. Types seem to be the obvious solution for this particular problem.

The type system we consider in this paper appears to be a simplified version of the simple type system of [10]: we use location types, that record the names and types of channel that may be used to communicate inside a locality, and we use "located types" for channels that are sent together with their location, instead of the existential types of [10]. We show the usual "subject reduction property", which holds for the system for checking receptiveness too. Then we are able to prove that typed receptive processes do not run into the kind of local deadlocks we wished to avoid. We call this property *message deliverability*, since it states that any message will find an appropriate receiver in its destination locality.

The issue of avoiding deadlocks by statically inferring some properties has already been addressed. For instance, in [4], a type system intended to prevent deadlocks is proposed for the blue calculus, where the types are formulae of Hennessy-Milner Logic with recursion. Kobayashi also studied in [11, 14] means to avoid deadlocks in the π-calculus. His approach is quite different from ours, however: he uses a sophisticated type system where types involve time tags and where typing contexts record an ordering on the usage of names, whereas we only use a very simple information – the set of receiver's names. Since we regard receivers as passive entities, providing resources for names, we are not seeking to avoid the situation where no message is sent to a receiver – in this situation the resource provider should be garbage collected.

The question arises whether the nice properties of our receptive distibuted π-calculus are obtained at the expense of its expressivity. Indeed, receptiveness is both a strong safety property and a quite heavy constraint. We therefore illustrate, by a series of examples, the "receptive style" of programming we need to adopt; this tends to be quite similar to the object oriented style, where a receiver (in the π-calculus sense) is a named persistent "object", reacting to messages according to some protocol, where the keys play the rôle of "selectors". We also establish a result that proves that expressive power is not lost: we show that the π_1-calculus of Amadio [2], in which the JOIN calculus can be encoded [3], may be translated into our receptive π_1-calculus, in a way which is fully abstract with respect to asynchronous bisimulation.

2. Distributed Processes.

In this section we introduce our calculus of distributed processes, which may be seen as a simplification of the distributed π-calculus of Hennessy and Riely [10], and as an extension of the π_1-calculus of Amadio [2, 3]. This is basically the asynchronous, polyadic π-calculus, with some primitives for spatial distribution of processes and migration.

As usual, we assume given a denumerable set $\mathcal{N}$ of (simple) *names*, ranged over by a, b, $c,\ldots$ We will informally distinguish a subset of names, that we denote $\ell, \ell', \ldots,$ which are supposed to name *localities* – formally, there will be different kinds of types for names. Then the processes may also send and receive *compound names*, which are pairs of simple names that we write $a@\ell$, meaning "the (channel) name a used at location ℓ". We use u, v, $w,\ldots$ to denote simple or compound names, and a (possibly empty) vector of such names is written $\vec{u}$. We shall often use the operation $\_@\ell$ on (compound) names, defined by

$$u@\ell = \begin{cases} a@\ell & \text{if } u = a \\ u & \text{if } u = a@\ell' \end{cases}$$

This operation is extended to sets of names in the obvious way (e.g. $N@L = \{\, u@\ell \mid u \in N \ \&\ \ell \in L \,\}$). We use $A, B, \ldots$ to denote process identifiers, belonging to a denumerable set $\mathcal{P}$, disjoint from $\mathcal{N}$. These are parametric on names, and we shall write a recursive call as $A(a;\vec{u})$. The intended meaning is that the name a is the only one that may – and must – be used as an input channel in the body of a recursive definition of A. The syntax of distributed processes involves *types*, but we defer any other consideration about types to the next section. Then the grammar for terms is as follows, where τ is a type:

$$\begin{array}{rcl} u,\ v \ldots & ::= & a \mid a@\ell \\ w & ::= & u \mid a : \tau \\ P,\ Q,\ R \ldots & ::= & \overline{a}\vec{u} \mid a(\vec{u}).P \mid (P \mid Q) \mid [a = b]P, Q \mid (\nu w)P \\ & \mid & A(a;\vec{u}) \mid (\mathsf{rec}\, A(a;\vec{u}).P)(a';\vec{u'}) \mid [\ell :: P] \end{array}$$

In an input $a(\vec{u}).P$, all the names occurring in $\vec{u}$ are assumed to be distinct, and similarly in $(\mathsf{rec}\, A(a;\vec{u}).P)$ all the parameters (including the components of the compound names in $\vec{u}$) are distinct. For any w, we define its subject $\mathsf{subj}(w)$ as follows: $\mathsf{subj}(a) = a = \mathsf{subj}(a@\ell) = \mathsf{subj}(a : \tau)$. In $(\nu w)P$, the subject of w is the only name occurring in w which is bound. We make the standard assumption that recursion is *guarded*, that is in $(\mathsf{rec}\, A(a;\vec{u}).P)(a';\vec{u'})$ all recursive calls to A in P occur under an input guard. We shall say that P is *closed* if it does not contain any free process identifier.

The need for compound names is explained by Hennessy and Riely in [10], although they use more sophisticated "dependent names"; we will see later examples showing how to use such names. In our syntax we do not distinguish "the process located at ℓ" from "the process P moving towards ℓ", written respectively $\ell[\![P]\!]$ and $\ell :: P$ in [10], and $\{P\}\ell$ and $\mathsf{spawn}(\ell, P)$ in [2]: both are denoted by $[\ell :: P]$, a construction introduced in [5] – where migration was not considered however. Another difference with [10] is that our underlying π-calculus is asynchronous, and we use explicit recursion instead of replication.

In order to define a notion of bisimulation, we describe the *behaviour* of processes by means of a labelled transition system for the calculus. That is, processes perform *actions* and evolve to new processes in doing so, which is denoted $P \xrightarrow{\alpha} P'$. The set of actions is given by:

$$\alpha ::= \tau \mid u\vec{v} \mid (\nu\vec{w})\overline{u}\vec{v}$$

meaning respectively internal communication (there should be no confusion between τ as a type and τ as an action), input of names $\vec{v}$ on the name u – if $u = a@\ell$, this means input on channel a at location ℓ – and output of names, some of them possibly being private. We also call subject of α, denoted $\mathsf{subj}(\alpha)$, the name u whenever $\alpha = u\vec{v}$ or $\alpha = (\nu\vec{w})\overline{u}\vec{v}$. We define the sets $\mathsf{fn}(\alpha)$ and $\mathsf{bn}(\alpha)$ of names occurring respectively free and bound in the action α in the obvious way (recall that in $(\nu a@\ell)$, a is bound while ℓ is free), and $\mathsf{nm}(\alpha)$ denotes the union of these two sets.

The rules of the transition system, given in Figure 1, extend the usual ones for the π-calculus. In the rule (out), the term $\mathbf{0}$ is $(\nu a)((\mathsf{rec}\, A(a;).a().A(a;))(a))$, that is a process which cannot perform any action. In the rules (in) and (rec), it is understood that the substitution operation involves some pattern matching, e.g. $[a@b/x@y]P$ is $[a/x, b/y]P$. In the rule (ext), the action α must be an output action (otherwise $(\nu w)\alpha$ would not be an action). There are also rules symmetric to (cm) and (cp), which are omitted. In the rule (cg), the equivalence $=_\alpha$ stands for syntactic identity up

$$(out)\quad \frac{}{\overline{a}\vec{u} \xrightarrow{\overline{a}\vec{u}} \mathbf{0}} \qquad\qquad (in)\quad \frac{}{a(\vec{u}).P \xrightarrow{a\vec{v}} [\vec{v}/\vec{u}]P}$$

$$(ext)\quad \frac{P \xrightarrow{\alpha} P'}{(\nu w)P \xrightarrow{(\nu w)\alpha} P'}\ \mathsf{subj}(w) \in \mathsf{fn}(\alpha) - \mathsf{nm}(\mathsf{subj}(\alpha)) \quad (\nu)\quad \frac{P \xrightarrow{\alpha} P'}{(\nu w)P \xrightarrow{\alpha} (\nu w)P'}\ \mathsf{subj}(w) \notin \mathsf{nm}(\alpha)$$

$$(cm)\quad \frac{P \xrightarrow{(\nu\vec{w})\overline{u}\vec{v}} P'\ ,\ Q \xrightarrow{u\vec{v}} Q'}{P \mid Q \xrightarrow{\tau} (\nu\vec{w})(P' \mid Q')}\ \mathsf{subj}(\vec{w}) \cap \mathsf{fn}(Q) = \emptyset$$

$$(cp)\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset \qquad\qquad (@)\quad \frac{P \xrightarrow{\alpha} P'}{[\ell :: P] \xrightarrow{\alpha@\ell} [\ell :: P']}$$

$$(m_t)\quad \frac{P \xrightarrow{\alpha} P'}{[a = a]P, Q \xrightarrow{\alpha} P'} \qquad\qquad (m_f)\quad \frac{Q \xrightarrow{\alpha} Q'}{[a = b]P, Q \xrightarrow{\alpha} Q'}\ a \neq b$$

$$(rec)\quad \frac{[\mathsf{rec}\, A(a;\vec{u}).P/A, a';\vec{u'}/a;\vec{u}]P \xrightarrow{\alpha} P'}{(\mathsf{rec}\, A(a;\vec{u}).P)(a';\vec{u'}) \xrightarrow{\alpha} P'} \qquad\qquad (cg)\quad \frac{P =_\alpha P' \quad P' \xrightarrow{\alpha} Q' \quad Q' =_\alpha Q}{P \xrightarrow{\alpha} Q}$$

Figure 1: Labelled transition system

to renaming of bound names. In the rule (@) for located processes, we extend the operation $\_@\ell$ to actions, as follows:

$$\alpha@\ell = \begin{cases} u@\ell\,\vec{v} & \text{if } \alpha = u\vec{v} \\ \overline{u@\ell}\,\vec{v} & \text{if } \alpha = \overline{u}\vec{v} \\ (\nu u@\ell)(\alpha'@\ell) & \text{if } \alpha = (\nu u)\alpha' \\ (\nu a : \tau)(\alpha'@\ell) & \text{if } \alpha = (\nu a : \tau)\alpha' \end{cases}$$

where in the last two cases we use α-conversion to ensure that $\ell \neq \mathsf{subj}(u)$ or $\ell \neq a$. As one can see, any action of the located process $[\ell :: P]$ is located – that is, its subject is a compound name –, but not necessarily at ℓ: if for instance this was an action of a sub-process located at ℓ', then its location is not modified, since $\alpha@\ell'@\ell = \alpha@\ell'$, as one can easily check. As a matter of fact, it is easy to see that the following equation holds, where equality is the usual strong bisimilarity:

$$[\ell' :: [\ell :: P]] \sim [\ell :: P]$$

This means that our operational semantics expresses the fact that distributed systems have a flat structure, like in [2, 10], but unlike in the distributed JOIN calculus [9] or the Ambient calculus [7] where "domains", or "sites", may be nested. Moreover, there is only one "domain" with a given name, that is

$$[\ell :: P] \mid [\ell :: Q] \sim [\ell :: P \mid Q]$$

This makes a difference with the model of Ambients. These equalities allow us to interpret $[\ell :: P]$ as "the process P migrating to ℓ", since for instance $[\ell' :: [\ell :: P] \mid Q] \mid [\ell :: R]$ behaves in the same way as $[\ell' :: Q] \mid [\ell :: P \mid R]$. Finally one can see that in the communication rule (cm) the two complementary actions must share the same subject u. This means that communication may only occur on the same channel (a if $u = a$ or $u = a@\ell$), at the same location (ℓ if $u = a@\ell$). In other words, unlike in [2, 9] where messages can transparently go through domains to reach the corresponding receiver, we have the "go and communicate" semantics of the Dπ-calculus, which is also the one of [7]. Typically, no transition can arise from $[\ell :: \overline{a}\vec{u}] \mid [\ell' :: a(\vec{v}).P]$ if $\ell \neq \ell'$, and one has to explicitly move and meet in order to communicate, like in $[\ell :: [\ell' :: \overline{a}\vec{u}]] \mid [\ell' :: a(\vec{v}).P]$.

3. A Simple Type System.

In the polyadic π-calculus, the names are normally used according to some typing discipline, and the processes are checked to obey this discipline. We will do the same here, requiring the processes to conform to some typing assumption about names, since this will be needed to ensure the message deliverability property we are seeking for. The types we use generalize the usual sorts of the π-calculus, and in particular channel sorts will be types. We also have to assign types to location names, and to compound names. The idea here is that a location type should record the names and types of channels on which communication is possible inside a locality. Therefore, a location type is just a typing context of a certain kind, as in [10], while the type of a located channel is a "located channel type" $\gamma^@$ – which is simpler than an existential type, as used in [10]. We also introduce a type val, for names that may be tested for equality, but are not used as communication channels or location names – we allow the latter to be compared too. Then the types (for names) are as follows:

$$\begin{array}{rcll}
\tau,\ \sigma\ldots & ::= & \zeta \mid \psi \mid \gamma^@ & \textit{types} \\
\zeta & ::= & val \mid \gamma & \textit{values and channel types} \\
\gamma,\ \delta\ldots & ::= & Ch(\tau_1,\ldots,\tau_n) & \textit{channel types} \\
\psi,\ \phi\ldots & ::= & \{a_1:\gamma_1,\ldots,a_n:\gamma_n\} & \textit{location types}
\end{array}$$

where in location types the a_i's are distinct names, and the order of items $a_i : \gamma_i$ is irrelevant. The typing judgements have the form $\Psi \vdash_\ell P$, meaning that P, when located at ℓ, uses the names as prescribed by the typing context Ψ. A *typing context* Ψ is a pair (Ψ_{typ}, Ψ_{loc}) of mappings from a finite subset $\mathsf{dom}(\Psi)$ of $\mathcal{N} \cup \mathcal{P}$, satisfying the following constraints – where we write $\ell \notin \Psi$ to mean that ℓ is neither in the domain of Ψ nor in the types assigned by Ψ:

1. Ψ_{typ} assigns to each $x \in \mathsf{dom}(I)$ a value or channel type (i.e. a ζ type).
2. Ψ_{loc} assigns to each $x \in \mathsf{dom}(I)$ a finite set of names, such that $\ell \in \Psi_{loc}(x) \ \Rightarrow\ \ell \notin \Psi$.
3. if $\Psi_{typ}(x) = val$ or $x \in \mathcal{P}$ then $\Psi_{loc}(x) = \emptyset$.

A context Ψ such that $\mathsf{dom}(\Psi) = \{x_1,\ldots,x_n\}$ with $\Psi_{typ}(x_i) = \zeta_i$ and $\Psi_{loc}(x_i) = L_i$ will be written:

$$x_1@L_1 : \zeta_1, \ldots, x_n@L_n : \zeta_n$$

This means that the name x_i is used at the locations contained in L_i, uniformly with type ζ_i. We abbreviate $x@\{\ell\} : \gamma$ as $x@\ell : \gamma$, and we simply write $x_i : \zeta_i$ if $L_i = \emptyset$; such an assumption $x@\emptyset : \zeta$ is only actually used whenever $\zeta = val$, otherwise it only serves to maintain some consistency about the typing of names. In the typing rules we use the notation Ψ, Φ for the context defined as follows:

$$\begin{array}{rcl}
\mathsf{dom}(\Psi,\Phi) & = & \mathsf{dom}(\Psi) \cup \mathsf{dom}(\Phi) \\
(\Psi,\Phi)_{typ}(x) & = & \left\{\begin{array}{ll} \Psi_{typ}(x) & \text{if } x \in \mathsf{dom}(\Psi) \\ \Phi_{typ}(x) & \text{if } x \in \mathsf{dom}(\Phi) \end{array}\right. \\
(\Psi,\Phi)_{loc}(x) & = & \Psi_{loc}(x) \cup \Phi_{loc}(x)
\end{array}$$

$$\frac{}{a : val \vdash_\ell a : val} \qquad \frac{}{a@\ell : \gamma \vdash_\ell a : \gamma} \qquad \frac{}{\psi@\ell \vdash_{\ell'} \ell : \psi}$$

$$\frac{}{a@\ell : \gamma \vdash_{\ell'} a@\ell : \gamma^{@}} \qquad \frac{\Psi \vdash_\ell \vec{u} : \vec{\tau} \quad , \quad \Phi \vdash_\ell \vec{v} : \vec{\sigma}}{\Psi\,,\,\Phi \vdash_\ell \vec{u} : \vec{\tau}\,,\, \vec{v} : \vec{\sigma}}$$

Figure 2: Type system for names

$$\frac{\Psi \vdash_\ell \vec{u} : \vec{\tau}}{a@\ell : Ch(\vec{\tau})\,,\,\Psi\,,\,\Phi \vdash_\ell \overline{a}\vec{u}} \qquad \frac{a@\ell : Ch(\vec{\tau})\,,\,\Psi\,,\,\Phi \vdash_\ell P \quad , \quad \Psi \vdash_\ell \vec{u} : \vec{\tau}}{a@\ell : Ch(\vec{\tau})\,,\,\Phi \vdash_\ell a(\vec{u}).P} \qquad \frac{\Psi \vdash_\ell P \;,\; \Psi \vdash_\ell Q}{\Psi \vdash_\ell P \mid Q}$$

$$\frac{\Psi \vdash_\ell P \;,\; \Psi \vdash_\ell Q}{\Psi \vdash_\ell [a = b]P, Q}\;(*) \qquad \frac{a@\ell : \gamma\,,\,\Psi \vdash_\ell P}{\Psi \vdash_\ell (\nu a)P} \qquad \frac{a@\ell : \gamma\,,\,\Psi \vdash_{\ell'} P}{\Psi \vdash_{\ell'} (\nu a@\ell)P} \qquad \frac{a : val\,,\,\Psi \vdash_\ell P}{\Psi \vdash_\ell (\nu a : val)P}$$

$$\frac{\psi@\ell\,,\,\Psi \vdash_{\ell'} P}{\psi\,,\,\Psi \vdash_{\ell'} (\nu\ell : \psi)P} \qquad \frac{\Psi \vdash_\ell P}{\Psi \vdash_{\ell'} [\ell :: P]}\;\ell' \notin \Psi \qquad \frac{\Psi \vdash_\ell \vec{u} : \vec{\tau}}{a@\ell : \gamma\,,\,A : Ch(\gamma, \vec{\tau}),\,\Psi\,,\,\Phi \vdash_\ell A(a; \vec{u})}$$

$$\frac{a@\ell : \gamma\,,\,A : Ch(\gamma, \vec{\tau})\,,\,\Psi\,,\,\Phi \vdash_\ell P \quad , \quad \Psi \vdash_\ell \vec{u} : \vec{\tau} \quad , \quad \Psi' \vdash_\ell \vec{u'} : \vec{\tau}}{a'@\ell : \gamma\,,\,\Psi'\,,\,\Phi \vdash_\ell (\mathsf{rec}\, A(a; \vec{u}).P)(a'; \vec{u'})}$$

$(*)$ $\quad \Psi_{typ}(a) = val = \Psi_{typ}(b)$ or $a, b \notin \Psi$.

Figure 3: Type system for terms

Notice that this context is undefined if $\Psi_{typ}(x) \neq \Phi_{typ}(x)$ for some $x \in \mathsf{dom}(\Psi, \Phi)$, or if there exist $x \in \mathsf{dom}(\Psi, \Phi)$ and $\ell \in (\Psi_{loc}(x) \cup \Phi_{loc}(x))$ such that $\ell \in \mathsf{dom}(\Psi, \Phi)$ or ℓ occurs in some type assigned by Ψ or Φ. For instance, $x@\emptyset : \zeta, \Psi$ is only defined if $\Psi_{typ}(x) = \zeta$ whenever $x \in \mathsf{dom}(\Psi)$. This operation Ψ, Φ involves implicit contractions, if $\mathsf{dom}(\Psi) \cap \mathsf{dom}(\Phi) \neq \emptyset$. To state the typing rules for $\Psi \vdash_\ell P$, we first need to introduce a system for establishing sequents of the form

$$\Psi \vdash_\ell u_1 : \tau_1, \ldots, u_n : \tau_n$$

that is for computing the types assigned to (value, location, compound) names by the context Ψ at the current location ℓ. The rules for inferring these judgements are given in Figure 2, where we use the notation $\psi@\ell$ for $a_1@\ell : \gamma_1, \ldots, a_n@\ell : \gamma_n$ if $\psi = \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$. Notice that we always have for instance $\vdash_{\ell'} \ell : \{\}$, since $\{\}@\ell : \gamma$ is the empty context. The typing rules for processes are collected in Figure 3, where we use the following:

Convention. *In the rules for the binding constructs, that is input, restriction and recursion, we implicitly have the usual condition that the bound names do not occur in the resulting context. That is, if the context contains an item* $a@L : \zeta$ *and if* u *is bound, then* $a \notin \mathsf{nm}(u)$, $\mathsf{nm}(u) \cap L = \emptyset$

and no name of $\mathsf{nm}(u)$ *occurs in* ζ*; moreover, if the conclusion is* $\Psi \vdash_\ell Q$ *then* $\ell \notin \mathsf{nm}(u)$*. In the case of recursion, the recursive identifier* A *must not appear in* Ψ', Φ.

Let us comment the rules of the system. We see that a simple name may only be used if it is of type *val*, or if it is a channel located at the current location, or if it is a location name. In this latter case, its type is the collection of channel names, together with their type, located at that location. A compound name may be used at any locality, but with a located type. The rules for the output and the input constructs are quite standard; let us just notice that the rule for output involves a form of subtyping for localities (*cf.* [10]): for instance, if $\psi = \{b : \gamma\}$, the judgement

$$a@\ell' : Ch(\psi)\,,\; b@\ell : \gamma\,,\; c@\ell : \delta \vdash_{\ell'} \overline{a}\ell$$

is valid, although the type of ℓ given by the context, namely $\{b : \gamma, c : \delta\}$, is slightly more generous than the one carried by the channel a. This form of subtyping is not allowed in input, since the received names cannot occur in the typing context. The only point to note about typing parallel composition is that the component must agree on the type of names they use, since they share the same context. In the rule for name comparison, we see that we can compare both names of value type and location names since $\ell \notin \Psi$ is true if ℓ is used as a locality in Ψ. There are four cases for typing a restriction $(\nu w)P$ – the rules follow the same pattern as the ones for typing names. In the last one, for $(\nu\ell : \psi)P$, we keep the typing context ψ – that is, more accurately $a_1@\emptyset : \gamma_1, \ldots, a_n@\emptyset : \gamma_n$ if $\psi = \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$ –, to ensure that the names $a_1, \ldots, a_n$ will not be misused by a concurrent process for instance. To type $[\ell :: P]$ one must be able to type P at locality ℓ, while the resulting current locality of $[\ell :: P]$ is arbitrary. Finally in the rules for recursive processes, we note that, to ensure that the parameters are used in a consistent way, we assign to the process identifier a (channel) type.

The terms of the ordinary asynchronous π-calculus with recursion are obviously terms of our distributed calculus, not involving any locality-based feature (compound names, location types, located processes). For typing these terms we use only the types given by

$$\kappa ::= val \;\mid\; Ch(\kappa_1, \ldots, \kappa_n)$$

Then we get the usual typing of these terms (see [15] for instance) by transforming the judgements $\Delta \vdash P$ into $\Psi \vdash_\ell P$ where ℓ is a fresh name, and

$$\Psi = \{\, a : val \mid a : val \in \Delta \,\} \cup \{\, a@\ell : \kappa \mid \kappa \neq val \;\&\; a : \kappa \in \Delta \,\}$$

forbidding the use of the rules for ℓ, $a@\ell$, $(\nu a@\ell)P$ and $(\nu\ell : \psi)P$ from the system. Our type system is very close to the simple type system for Dπ presented in [10], except that we use located types $\gamma^@$ instead of existential types (there are also other minor differences, in the presentation of the system for instance, since we do not distinguish "threads" from "distributed systems", and we assign channel and value types, and localities, to names instead of assigning location types).

Our main result about typing is the standard "subject reduction property". This result is needed to establish the message deliverability property.

Theorem (Preservation of Typing) 3.1. *If* $\Psi \vdash_\ell P$ *and* $P \xrightarrow{\alpha} P'$ *then*

(i) *if* $\alpha = u\vec{v}$ *then* $\Psi = a@\ell' : Ch(\vec{\tau}), \Phi$ *with* $a@\ell' = u$ *or* $a = u$ *and* $\ell' = \ell$*, and* $\Psi', \Psi \vdash_\ell P'$ *where* $\Psi' \vdash_{\ell'} \vec{v} : \vec{\tau}$,

(ii) *if* $\alpha = (\nu\vec{w})\overline{u}\vec{v}$ *then* $\Psi = a@\ell' : Ch(\vec{\tau}), \Psi', \Phi$ *with* $a@\ell' = u$ *or* $a = u$ *and* $\ell' = \ell$*, and there exists* Ψ'' *such that* $\Psi', \Psi'' \vdash_{\ell'} \vec{v} : \vec{\tau}$ *and* $\Psi', \Psi'', \Phi \vdash_\ell P'$,

(iii) *if* $\alpha = \tau$ *then* $\Psi \vdash_\ell P'$.

$$\frac{}{\Vdash \overline{a}\vec{u}} \qquad \frac{a \Vdash P}{a \Vdash a(\vec{u}).P}\ a \notin \mathsf{nm}(\vec{u}) \qquad \frac{I \Vdash P\ ,\ I' \Vdash Q}{I\ ,\ I' \Vdash (P \mid Q)}\ I \cap I' = \emptyset$$

$$\frac{u, I \Vdash P}{I \Vdash (\nu u)P} \qquad \frac{I \Vdash P}{I \Vdash (\nu a : val)P} \qquad \frac{\mathsf{dom}(\psi)@\ell\ ,\ I \Vdash P}{I \Vdash (\nu \ell : \psi)P} \qquad \frac{I \Vdash P\ ,\ I \Vdash Q}{I \Vdash [a = b]P, Q}$$

$$\frac{}{a \Vdash A(a;\vec{u})} \qquad \frac{a \Vdash P}{a' \Vdash (\mathsf{rec}\, A(a;\vec{u}).P)(a';\vec{u'})} \qquad \frac{I \Vdash P}{I@\ell \Vdash [\ell :: P]}\ \{\, a \mid a, a@\ell \in I\,\} = \emptyset$$

Figure 4: Well-formed terms

Proof Sketch: we first establish that the type system admits weakening, is stable by injective renaming, and is such that

$$\Psi \vdash_\ell (\mathsf{rec}\, A(a;\vec{u}).P)(a';\vec{u'}) \ \Rightarrow\ \Psi \vdash_\ell [(\mathsf{rec}\, A(a;\vec{u}).P)/A, a'/a, \vec{u'}/\vec{u}]P$$

Then we proceed by induction on the inference of $P \xrightarrow{\alpha} P'$, and by examination of the last rule used to infer $\Psi \vdash_\ell P$ – there is only one such rule ❏

In a next section we will see examples of typed terms.

4. Interfaces and Receptive Processes.

It is easily seen that the type system does not guarantee the kind of safety property we are looking for: we can obviously type non-receptive processes, and we can also type processes sending a message that will never find a corresponding receiver, like for instance $(\nu a)\overline{a}$. Then we introduce an inference system for checking "well-formedness" of processes. Basically, to be well-formed a process must not contain nested inputs on different names. Moreover, any input must be involved in a recursive process that makes the receiver reincarnate itself, possibly in a different state, after being consumed. In addition, we shall impose, as in π_1 [2], that there is a unique receiver for each name, that is, two parallel components of a process are not allowed to receive on a same name. Last but not least, we demand that to restrict the scope of a name of channel type, we know for a fact that a resource is provided – that is, a receiver exists – for that name.

The well-formed processes are defined as the ones for which a statement $I \Vdash P$, that is "*P is well-formed with interface I*", can be proved. In this statement the interface I is a finite set of names on which a process may perform an input. We present the rules for well-formedness in Figure 4, where we use the same convention as for the typing regarding the binding constructs, and where, as usual, a set $I = \{u_1, \ldots, u_n\}$ is represented as the list $u_1, \ldots, u_n$ of its elements, and union is written I, I'.

Let us comment on these rules. In the rule for input $a(u_1, \ldots, u_n).P$, the interface of P is assumed to be $\{a\}$ (that is, P appears to offer inputs on a, and only on that name), while the received names, occurring in $u_1, \ldots, u_n$, are not in the interface of P. Therefore a well-formed process satisfies the constraint that a received name must not be used as an input channel, a contraint that characterizes the "local" (asynchronous, polyadic) π-calculus of [12]. For instance, $\overline{a}(b) \mid a(c).(\cdots c(\vec{d}).P \cdots)$ is not well-formed. Assuming "locality" is important for our purpose, since otherwise the environment of a process like $(\nu a)(\overline{a}(\vec{u}) \mid \overline{b}a)$ could create a receiver for a, and message deliverability would become very difficult to check. Note also that we only accept $(\nu w)P$ if w is a name such that there is a receiver on its subject in P, or if w is a typed value or location name. In the latter case, where $w = \ell : \psi$, if $\psi = \{a_1 : \gamma_1, \ldots, a_n : \gamma_n\}$, then we require a receiver to be provided at ℓ for each name $a_1, \ldots, a_n$. Then the location type ψ describes the set of resources available to an agent at the location ℓ. Observe also that the only way to introduce a name a in the interface is to introduce a process identifier, which will be involved in a guarded recursion, where the input guarding the recursive call must be on the same name a – thanks to the rule for input.

The rule for parallel composition in Figure 4 could be relaxed to allow the intersection of I and I' to be non-empty – and simultaneously we would not impose any condition for $I@\ell \Vdash [\ell :: P]$. In this case, we can still guarantee that messages are deliverable although not necessarily to a unique receiving process. We require the unicity of the receivers because it seems a sensible assumption both from a specification and an implementation point of view (*cf.* object-oriented models, and [8]). Moreover, this property is useful to establish that we do not loose expressive power, see the last section of the paper.

We may observe that for the terms of the ordinary π-calculus, we also get a notion of well-formedness $I \Vdash P$ where I is a set of simple names, simply by forbidding the rules for $(\nu\ell : \psi)P$ and $[\ell :: P]$. The system we get is a refinement of the one introduced for the π_1-calculus in [2]. Our first result about well-formedness is, again, a "subject reduction property":

Proposition (Preservation of Well-Formedness) 4.1. *If $I \Vdash P$ and $P \xrightarrow{\alpha} P'$ then*
(i) *$I \Vdash P'$ if $\alpha = \tau$ or $\alpha = u\vec{v}$,*
(ii) *if $\alpha = (\nu\vec{w})\overline{u}\vec{v}$ and $J = \{ u \mid \exists i.\ w_i = u \} \cup \{ a@\ell \mid \exists i.\ w_i = \ell : \psi \ \&\ a \in \mathsf{dom}(\psi) \}$ then $I, J \Vdash P$.*

Proof Sketch: the proof is similar to the one of the Theorem 3.1 (and similar to the one of the same result in [2]) ❑

We can now establish the receptiveness property for our distributed calculus. To state this property, let us define the predicate $P \downarrow u$, meaning "P may perform an input with subject u", that is:

$$P \downarrow u \quad \Leftrightarrow_{\text{def}} \quad \exists\vec{v}\,\exists P'.\ P \xrightarrow{u\vec{v}} P'$$

Proposition (Receptiveness) 4.2. *Let P be a closed well-formed term. Then:*
(i) *if $I \Vdash P$ then $P \downarrow u$ iff $u \in I$,*
(ii) *if $P \downarrow u$ and $P \xrightarrow{\alpha} P'$ then $P' \downarrow u$.*

Proof Sketch: (i)($\Rightarrow$) We proceed by induction on the inference of $P \xrightarrow{\alpha} P'$.
(i)($\Leftarrow$) We show this implication for P a well-formed term (not necessarily closed) where free occurrences of process identifiers are guarded, by induction on the inference of $I \Vdash P$.
(ii) We use the first point and the previous proposition ❑

An immediate consequence of this proposition is that we statically know the locality of each receiver in a term: for a public name, it is indicated in the interface – the locality of a receiver on a is the current one if $a \in I$, and is ℓ if $a@\ell \in I$ (as we shall see with the "distributed object" example below, the same channel may have several receivers, though in different locations). For a subterm $(\nu u)P$ of a well-formed process, we know that u is in the interface of P. Note however that in $(\nu a@\ell)P$, the location name ℓ is free, and may be received as a parameter from a message, see our last example of a "migrating cell" in the next section.

This result suggests the denomination "*distributed (asynchronous) receptive π-calculus, with unique receivers*", in short the Dπ_1^r-calculus, for the set of well-formed closed processes, which is closed by labelled transitions. Similarly we call π_1^r, that is "*the receptive π_1-calculus*", the sub-calculus where we do not use any locality based feature. The calculus obviously inherits the standard definitions of – barbed, strong ($\sim$), weak, asynchronous, and so on – bisimulation.

We now turn to the issue of message delivery. We aim at showing that, if a message is sent (at some locality) on a channel of a known scope in a well-formed and typed process, then the process contains a receiver for this message. Let us denote by $P \downarrow \overline{u}$ the fact that P performs an output action with subject u, and let us define

$$P \simeq Q \quad \Leftrightarrow_{\text{def}} \quad \begin{cases} P \sim Q & \text{and} \\ \forall \Psi, \ell.\ \Psi \vdash_\ell P \ \Leftrightarrow\ \Psi \vdash_\ell Q & \text{and} \\ \forall I.\ I \Vdash P \ \Leftrightarrow\ I \Vdash Q \end{cases}$$

As a preliminary result, we show that any process that is able to send a message is equivalent to another having a special form:

Lemma 4.3. *$P \downarrow \overline{a}$ iff $P \simeq (\nu \vec{w})(\overline{a}\vec{v} \mid R)$ for some $\vec{w}$, $\vec{v}$ and R with $a \notin \mathsf{subj}(\vec{w})$. $P \downarrow \overline{a@\ell}$ iff $P \simeq (\nu \vec{w})([\ell :: \overline{a}\vec{v}] \mid R)$ for some $\vec{w}$, $\vec{v}$ and R with $a \notin \mathsf{subj}(\vec{w})$.*

Now we can prove our main result, where $\rightarrow$ stands for $\xrightarrow{\tau}$:

Theorem (Message Deliverability) 4.4. *Let P be a closed well-formed and typed process with $I \Vdash P$ and $\Psi \vdash_\ell P$. If $P \xrightarrow{*} P'$ then*

(i) *if $P' \simeq (\nu \vec{w})(\overline{a}\vec{v} \mid R)$ with $a \in I \cup \mathsf{subj}(\vec{w})$ then $R \downarrow a$ or $R \downarrow a@\ell$.*

(ii) *if $P' \simeq (\nu \vec{w})([\ell' :: \overline{a}\vec{v}] \mid R)$ with $a@\ell' \in I$ or $a \in \mathsf{subj}(\vec{w})$ then $R \downarrow a@\ell'$ or $\ell' = \ell$ and $R \downarrow a$.*

Proof Sketch: by the Theorem 3.1(iii) and the Proposition 4.1(i), we have $\Psi \vdash_\ell P'$ and $I \Vdash P'$. We then examine how $(\nu \vec{w})(\overline{a}\vec{v} \mid R)$ or $(\nu \vec{w})([\ell' :: \overline{a}\vec{v}] \mid R)$ can be typed; we proceed by induction on the length of $\vec{w}$ to show that, in the first case, $I, I' \Vdash (\overline{a}\vec{v} \mid R)$ with $a \in I, I'$ or $a@\ell \in I, I'$, and, in the second case, $I, I' \Vdash ([\ell' :: \overline{a}\vec{v}] \mid R)$ with $a@\ell' \in I, I'$ or $\ell' = \ell$ and $a \in I, I'$. We conclude using the Proposition 4.2(i) $\square$

Note that this result does not hold for untyped terms. For instance we have $\Vdash (\nu a : val)\overline{a}$ or $\Vdash (\nu a : \emptyset)\overline{a}$, and these terms contain a message that cannot be delivered.

5. Receptive Distributed Programming.

In this section we give some examples to illustrate the style of "programming" that one must adopt to conform to the receptive discipline. One may wonder for instance how we could "program" non-receptive agents such as the "one-slot buffer" that repeatedly waits for some data on a given channel and then sends it on another channel. In the synchronous π-calculus, this process may be written:

$$\big(\mathsf{rec}\, B(a;b).a(c).\overline{b}(c).B(a;b)\big)(a;b)$$

Clearly, this cannot be written so easily in $\mathrm{D}\pi_1^r$. Let us begin with some notational conventions and preliminary examples. First, we shall write a recursive process $(\mathsf{rec}\, A(a;\vec{u}).P)(a;\vec{u})$ that does not introduce new parameters simply as $\mathsf{rec}\, A(a;\vec{u}).P$. The *identity* agent is the following:

$$Id_a \ =_{\text{def}}\ \mathsf{rec}\, A(a;).a(\vec{u}).(\overline{a}\vec{u} \mid A(a;))$$

This term is well-formed, with interface $\{a\}$ (provided that $a \notin \mathsf{nm}(\vec{u})$), and has the typing $a@\ell : Ch(\vec{\tau}) \vdash_\ell Id_a$. We note that, up to asynchronous strong bisimulation – whose definition is given in the next section – this receptive process that sends back all messages it receives is equivalent to the process that just ignores them:

$$T_a \ =_{\text{def}}\ \mathsf{rec}\, A(a;).a(\vec{u}).A(a;)$$

This agent has also the same typing and interface as Id_a. We write $(\nu a)P$ as an abbreviation for $(\nu a)(P \mid Id_a)$ whenever a is not free in P. This is a well-formed process, with interface I, if $I \Vdash P$.

We write $\overline{a}(\vec{u}, \_, \vec{u'})$ for $(\nu c)(\overline{a}(\vec{u}, c, \vec{u'}) \mid \mathit{Id}_c)$ and $a(\vec{u}, \_, \vec{u'}).P$ for $a(\vec{u}, c, \vec{u'}).P$, where c does not occur in P. In these abbreviations we could use T_a instead of Id_a. We recall the *input once* construct of [2], given by

$$a(\vec{u}){:}P \ =_{\text{def}} \ a(\vec{u}).(P \mid \mathit{Id}_a)$$

It is easy to see that this term has the same typing as $a(\vec{u}).P$, and that $a \Vdash a(\vec{u}){:}P$ if $\Vdash P$ and $a \notin \mathsf{nm}(\vec{u})$. Again, this could be defined using T_a instead of Id_a. The "replicated input" construct of [2] is given by

$$a(\vec{u}) \triangleright P \ =_{\text{def}} \ \mathsf{rec}\, A(a;).a(\vec{u}).(P \mid A(a;))$$

This has the same typing and well-formedness as the input once construct – that is the same typing as input, and $a \Vdash a(\vec{u}) \triangleright P$ if $\Vdash P$ and $a \notin \mathsf{nm}(\vec{u})$. From this we can define replication, as usual:

$$!P \ =_{\text{def}} \ (\nu a)(\overline{a} \mid a() \triangleright (\overline{a} \mid P))$$

This is well-formed if P has an empty interface. In the examples we also use the standard notation $\mathsf{if}\ a = b\ \mathsf{then}\ P\ \mathsf{else}\ Q$ – and more generally $\mathsf{if}\ C\ \mathsf{then}\ P\ \mathsf{else}\ Q$ – for $[a = b]P, Q$.

Example 1: Buffers

We may program the one-slot buffer as follows: first it inputs on a a message that is supposed to convey a data to store in the buffer (if this is not the case the message is ignored, i.e, it is resent), and then on the *same* channel it receives a request for extracting the contents of the buffer, which is delivered on a private return channel (again, if this protocol is violated this second message is ignored, though obviously something more elaborate could be done in a more synchronous version of the buffer).

$$\begin{array}{rl} \mathit{Buff}_1(a) = \mathsf{rec}\, B(a;).a(k_1,x,y).\mathsf{if}\ k_1 \neq \mathsf{put} & \\ \quad \mathsf{then}\ \overline{a}(k_1,x,y) \mid B(a;) & \\ \quad \mathsf{else\ rec}\, B_1(a;x).a(k_2,z,y).\mathsf{if}\ k_2 \neq \mathsf{get} & \\ \quad\quad \mathsf{then}\ \overline{a}(k_2,z,y) \mid B_1(a;x) & \\ \quad\quad \mathsf{else}\ \overline{y}(x) \mid B(a;) & \end{array}$$

The requests for reading and writing the buffer are respectively $(\nu c)(\overline{a}(\mathsf{get}, \_, c) \mid c(x){:}P)$ and $\overline{a}(\mathsf{put}, b, \_)$.

As one can see, the buffer is now a kind of "agent", that is a process which is invoked by its name a and reacts according to some internal protocol. The reader can check that $a \Vdash \mathit{Buff}_1(a)$, and that the typing judgement $a@\ell : \mathit{Ch}(\mathit{val}, \tau, \mathit{Ch}(\tau)) \vdash_\ell \mathit{Buff}_1(a)$ is valid. To build a "two-slots buffer" from this one, we may proceed as usual, putting together two one-slot buffers with a private communication between them. However, to write this we need to refine our previous program, because we need to explicitly indicate the keys used, defining $\mathit{Buff}_1(a; \mathsf{put}, \mathsf{get})$ – in the obvious way. Then the two-slots buffer is:

$$(\nu z_0, z_1 : \mathit{val})\big(\ \mathit{Buff}_1(a; \mathsf{put}, z_0) \ \mid\ !(\nu c)(\overline{a}(z_0, \_, c) \mid c(x){:}\overline{a}(z_1, x, \_)) \ \mid\ \mathit{Buff}_1(a; z_1, \mathsf{get})\,\big)$$

Example 2: Mutual Exclusion

Synchronization can be "programmed" in the π-calculus, a typical example being mutual exclusion between tasks enforced by the use of a *lock*, as follows:

$$\begin{array}{l} (\nu l)\big(\,\overline{l} \ \mid\ !\ \mathsf{task}_1(\vec{u}).l().(\ \cdots\ \overline{l}\,) \\ \quad\quad \vdots \\ \quad \mid\ !\ \mathsf{task}_n(\vec{v}).l().(\ \cdots\ \overline{l}\,)\,\big) \end{array}$$

This violates both the unique receiver property and receptiveness. In the receptive style, the lock is represented by a process that receives on a unique channel l messages carrying a value lock or unlock and a return channel r:

$$\begin{array}{rcl}\mathsf{Lock}(l) & = & \mathsf{rec}\, A(l;).l(k,r).\mathsf{if}\ k \neq \mathsf{lock}\\ & & \qquad\qquad\quad \mathsf{then}\ \overline{l}(k,r) \mid A(a;)\\ & & \qquad\qquad\quad \mathsf{else}\ \overline{r} \mid \mathsf{rec}\, A'(l;).a(k',s).\mathsf{if}\ k' \neq \mathsf{unlock}\\ & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ \overline{a}(k',s) \mid A'(l;)\\ & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ A(l;)\end{array}$$

and a task is now written

$$\mathsf{task}(\vec{u}).(\nu r)\big(\,\overline{l}(\mathsf{lock},r) \ \mid\ r():(\ \cdots\ \overline{l}(\mathsf{unlock},\_))\big)$$

Again, the lock is now a persistent agent that has an identity l, and reacts according to its own protocol, governed by the keys it receives.

Example 3: Exporting Access to a Public or Private Object

In this example we consider a distributed process which at locality ℓ exports (privately) its location name to a distinct locality ℓ', while at ℓ' it learns the name ℓ and uses it to move and ask some service. Such a process may be for instance, denoting by $obj(b)$ a recursive process offering resources on b – that is $b \Vdash obj(b)$:

$$P = (\nu\ell : \{b : Ch()\})\big([\ell :: \ obj(b) \mid [\ell' :: \overline{a}\ell\,]\,]\big) \ \mid\ [\ell' :: a(\ell):[\ell :: \overline{b}]\,]$$

Let us see how to type such a term. If we let $\gamma = Ch()$, $\psi = \{b : \gamma\}$ and $\delta = Ch(\psi)$, then we have $b@\ell : \gamma \vdash_{\ell'} \ell : \psi$, and therefore the following inference is valid

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell} \overline{b}}}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell'} [\ell :: \overline{b}]}}{a@\ell' : \delta\,,\ \Psi \vdash_{\ell'} a(\ell):[\ell :: \overline{b}]}}{a@\ell' : \delta\,,\ \Psi \vdash_{\ell''} [\ell' :: a(\ell):[\ell :: \overline{b}]]}$$

We also have, again using $b@\ell : \gamma \vdash_{\ell_1} \ell : \psi$ to type $\overline{a}\ell$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell} obj(b)} \qquad \dfrac{\dfrac{}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell'} \overline{a}\ell}}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell} [\ell_1 :: \overline{a}\ell]}}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell} obj(b) \mid [\ell' :: \overline{a}\ell]}}{a@\ell' : \delta, b@\ell : \gamma\,,\ \Psi \vdash_{\ell''} [\ell :: obj(b) \mid [\ell' :: \overline{a}\ell]]}}{a@\ell' : \delta\,,\ b : \gamma\,,\ \Psi \vdash_{\ell''} (\nu\ell : \psi)[\ell :: obj(b) \mid [\ell' :: \overline{a}\ell]]}$$

We conclude that the process P agrees with the typing context $a@\ell' : Ch(\{b : Ch()\})\,,\ b : Ch()\,,\ \Psi$ – at any location. In this example, the "object" $obj(b)$ is publicly known, outside its own location. One can imagine a variant of this situation, where the object is only known within its locality. Then one should export the name of this object together with its location, and this may be written:

$$P = [\ell :: \ (\nu b)(obj(b) \mid [\ell' :: \overline{a}(b@\ell)])\,] \ \mid\ [\ell' :: a(b@\ell):[\ell :: \overline{b}]]$$

We let the reader check that this term is typable with $a@\ell' : Ch(\gamma^{@})$.

Example 4: Remote Procedure Call

In this example, a client process, located at ℓ, wants to call a service named a, located at ℓ', with arguments $\vec{d}$, and waits for results – this is a situation similar to *remote procedure call* (the dual of this, that is *remote evaluation*, is trivially expressed in our language, by the primitive operation $[\ell :: P]$). Then the client creates a private return channel at its own location, and sends it, as the subject of a compound name indicating the return locality, together with the data $\vec{d}$. We write this as follows, assuming that the data $\vec{d}$ and the returned results are values:

$$Q = (\nu r)(\, [\ell' :: \overline{a}(\vec{d}, r@\ell)] \;\mid\; r(\vec{v}){:}P\,)$$

This is a well-formed term, with an empty interface, if $\Vdash P$. Then it is not difficult to check that, if we let $\gamma = Ch(\vec{\tau})$, the following is a valid inference:

$$\frac{\Phi \vdash_\ell \vec{v} : \vec{\tau} \quad , \quad \Phi\, ,\, a@\ell' : Ch(\overrightarrow{val}, \gamma^{@})\, ,\, \vec{d} : val\, ,\, \Psi \vdash_\ell P}{a@\ell' : Ch(\overrightarrow{val}, \gamma^{@})\, ,\, \vec{d} : val\, ,\, \Psi \vdash_\ell Q}$$

The service, receiving the data $\vec{d}$ and $r@\ell$, is supposed to compute some values and send them back to the client. This is done in the form of a migrating message $[\ell :: \overline{r}\vec{v}]$. As one can see from the type $Ch(\overrightarrow{val}, \gamma^{@})$ of its name, the service does not need to know anything about the location of the client.

Example 5: An Object Server

Suppose we have written a generic object $obj(b; \vec{d})$, with name b and state parameters $\vec{d}$, and that we would like to create at some location ℓ_0 a server for that kind of objects, which will deliver instances of it on requests on a name s – this is similar to *code on demand*. Then we write this as follows, assuming that the parameters $\vec{d}$ are values:

$$s(c@\ell, \vec{d}) \triangleright [\ell :: (\nu b)(\overline{c}b \mid obj(b; \vec{d}))]$$

A client at site ℓ_1 that wants to acquire its own instance of the object, and use it in the process P will be written:

$$(\nu c)(\, [\ell_0 :: \overline{s}(c@\ell_1, \vec{d})] \;\mid\; c(b){:}P\,)$$

We let the reader check that, if we can type $obj(b; \vec{d})$ with type ζ for b, then these terms are typable with $s@\ell_0 : Ch(\, Ch(\zeta)^{@}, \overrightarrow{val})$.

Example 6: A Distributed Object

In this example we imagine an "object" which is located both at ℓ_0 and ℓ_1 (with the same name), and has a shared, possibly distributed state. More specifically, we program a distributed "button", named a. "Pushing the button" at ℓ_i means sending a message to the channel a at ℓ_i. Pushing twice the button at either ℓ_0 or ℓ_1 results in the emission of a message c at ℓ. In particular, pushing once at ℓ_0 and once at ℓ_1 will produce an emission. This condition obviously forces some cooperation between the two locations. We seek a solution that will allow emission even when one of the two locations stops working. This condition rules out centralized solutions where emissions are always decided at one location. On the other hand, some degree of asymmetry seems desirable to rule out a circular situation where each location tries to "borrow a push" from the other location in order to produce an emission. In the presented solution, the button at location ℓ_0 is allowed to borrow from the one at ℓ_1, but not the other way round. The distributed button relies on a private value b – to borrow a push:

$$P = (\nu b : val)\big([\ell_0 :: P_0] \mid [\ell_1 :: P_1]\big)$$

where

$$\begin{array}{rcl} P_0 & = & \mathsf{rec}\, A(a;).a(i).\big([\ell_1 :: \overline{a}b] \mid a(i).(A(a;) \mid [\ell :: \overline{c}])\big) \\ P_1 & = & \mathsf{rec}\, A'(a;).a(i).\mathsf{if}\ i = b\ \mathsf{then}\ (A'(a;) \mid \overline{a}b) \\ & & \qquad\qquad\qquad \mathsf{else}\ a(i).\mathsf{if}\ i = b\ \mathsf{then}\ (A'(a;) \mid [\ell_0 :: \overline{a}b]) \\ & & \qquad\qquad\qquad\qquad\qquad \mathsf{else}\ (A'(a;) \mid [\ell :: \overline{c}]) \end{array}$$

It is easy to see that these terms are well-formed, with interface $\{a\}$, and therefore $a@\ell_0, a@\ell_1 \Vdash P$, and typable with $a@\{\ell_0, \ell_1\} : Ch(val)$. This solution could be refined. For instance, the processes may want to check that the request to borrow received by the process at ℓ_1 corresponds to the *current* request to borrow from the process at ℓ_0 (if it is not the case then the offer to lend is declined). Similar, but more complicated techniques could be applied for the programming of a distributed channel manager or, more generally, of a distributed memory.

Notice that to fix the scope of the name of such a distributed object, we should extend the syntax to allow $(\nu a@L)P$, where L is a non-empty finite set of (location) names. It is then straightforward to modify the type system and the system for well-formedness to deal with this extension.

Example 7: A Migrating Cell

In the same style as for the buffer, we could program a *cell* (see [15]), which we may read without destroying its content. To make this example more interesting, we suppose that the cell not only responds to requests for reading and writing, but also to instruction to *migrate* to some given location. The name c of the cell has type $\gamma = Ch(val, loc, val, Ch(val)^{@}, Ch()^{@})$, where the first argument is the key for reading, writing and migrating, the second argument, of locality type $loc = \{\}$, is the destination locality in case of migration, the third argument is the content of the cell, and the last two are return channels, for readers and writers respectively. To maintain some consistency, the cell moved at a new location bears a new name, which is used to forward messages arriving at the location it originates from. The migrating cell $\mathsf{MigCell}(c;x)$ has a parameter x which is its initial value; it is the following term:

$$\begin{array}{l} \mathsf{rec}\, A(c;x).c(k,\ell,x',y@\ell',z@\ell'').\,[k = \mathsf{read}]\,([\ell' :: \overline{y}x] \mid A(c;x))\,, \\ \qquad\qquad\qquad\qquad\qquad\qquad [k = \mathsf{write}]\,([\ell'' :: \overline{z}] \mid A(c;x'))\,, \\ \qquad\qquad\qquad\qquad\qquad\qquad [k = \mathsf{migr}]\,(\nu c'@\ell)([\ell :: A(c';x)] \mid \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{rec}\, A'(c;x).c(\vec{u}).(A'(c;x) \mid [\ell :: \overline{c'}\vec{u}]))\,, \\ \qquad\qquad\qquad\qquad\qquad\qquad A(c;x) \end{array}$$

In case a read operation, that is $\overline{c}(\mathsf{read}, \_, \_, y@\ell', \_)$, has to be processed, the cell returns its current value x on the channel y to the reader at location ℓ'. In case of a write operation $\overline{c}(\mathsf{write}, \_, x', \_, z@\ell'')$, the cell is updated with the new value x', and an acknowledgement z is sent back to the writer. Finally a migration message $\overline{c}(\mathsf{migr}, \ell, \_, \_, \_)$ causes the creation of a new cell c' at the destination location ℓ, while a “forwarder” $\mathsf{rec}\, A'(c;x).c(\vec{u}).(A'(c;x) \mid [\ell :: \overline{c'}\vec{u}])$ is set up at the current locality.

The reader can check that $\mathsf{MigCell}(c;x)$ is both well-formed, with interface $\{c\}$, and typable, with c of type γ at the current location, and x of type val. In this version, the remote cell can only be accessed by messages in the location it originated from. We could also make the new cell available at its new location, by sending its name on a public channel a, that is writing $[\ell :: \overline{a}c' \mid A(c';x)]$

instead of $[\ell :: A(c'; x)]$. However, to type this new version, we need recursive types. These are introduced by adding type variables and recursion, as follows:

$$\tau ::= \cdots \mid t \mid \mu t.\tau$$

These are used in the type system as usual, that is, identifying a recursive type with the (infinite, regular) tree it determines (notice that there is also an implicit notion of equality of location types, that may be regarded as record types). Now if we let $\delta = \mu t.Ch(val, \{a : Ch(t)\}, val, Ch(val)^{@}, Ch()^{@})$ we can type the new migrating cell with c of type δ, assuming that a is available with type $Ch(\delta)$ at all known localities.

As we have seen, in many examples we have written named agents – i.e. receivers – that look like "objects". Then we could have used explicit constructs for them, like the ones of TyCO [16], that is $a?\{k_1 = (\vec{u}_1)P_1, \ldots, k_n = (\vec{u}_n)P_n\}$ instead of $a(k, \vec{u})[k = k_1]P_1, \ldots [k = k_n]P_n, Q$, with corresponding messages $\overline{a}.k_i(\vec{v})$ in place of $\overline{a}(k_i, \vec{v})$. We would then use record types, with the advantage that the types of argument vectors need not be uniformly the same, but depend on the key invoked. In other words, it would be interesting to apply our receptive approach to TyCO.

6. Encoding the π_1-Calculus.

In this section, we show that there is a translation from the π-calculus to $\mathrm{D}\pi_1^r$. It is shown in [3] that the joined input of the join-calculus [8] can be defined in the π_1-calculus up to weak asynchronous bisimulation. On the other hand, it has been shown in [8] that there is a fully abstract translation of the asynchronous π-calculus in the join-calculus. Therefore, if we can translate the π_1-calculus in the $\mathrm{D}\pi_1^r$-calculus – or rather in the π_1^r-calculus –, we can reasonably claim that the nice properties of the $\mathrm{D}\pi_1^r$-calculus are not obtained at the expense of its expressive power. In this informal argument, we are making the reasonable assumption that the translation from the asynchronous π-calculus to the join-calculus can be extended to include a conditional operator on *values*. We now give such a translation, that we show to be fully abstract with respect to a refined form of asynchronous bisimulation [1, 2], defined as follows:

Definition (Asynchronous Bisimulation) 6.1. *A symmetric relation $\mathcal{S}$ is an asychronous bisimulation if $P\,\mathcal{S}\,Q$ implies*

(i) *there exists I such that $I \Vdash P$ and $I \Vdash Q$, and*

(ii) *if $P \xrightarrow{\tau} P'$ then $Q \xrightarrow{\tau} Q'$ for some Q' such that $P'\,\mathcal{S}\,Q'$,*

(iii) *if $P \xrightarrow{(\nu\vec{w})\overline{u}\vec{v}} P'$ with $u \notin I$ and $\mathsf{subj}(\vec{w}) \cap \mathsf{fn}(Q) = \emptyset$ then $Q \xrightarrow{(\nu\vec{w})\overline{u}\vec{v}} Q'$ for some Q' such that $P'\,\mathcal{S}\,Q'$,*

(iv) *if $P \xrightarrow{u\vec{v}} P'$ then either $Q \xrightarrow{u\vec{v}} Q'$ with $P'\,\mathcal{S}\,Q'$, or $Q \xrightarrow{\tau} Q'$ with $P'\,\mathcal{S}\,(Q' \mid R)$ where $R = \overline{a}\vec{v}$ if $u = a$ and $R = [\ell :: \overline{a}\vec{v}]$ if $u = a@\ell$.*

We denote with $\sim_a$ the greatest asychronous bisimulation. The notion of weak asychronous bisimulation is obtained by replacing everywhere transitions with weak transitions. We denote with $\approx_a$ the greatest weak asychronous bisimulation.

As a source calculus, we will consider the π_1-calculus with guarded recursion. There is no loss of generality here, as unguarded recursion can be replaced by guarded recursion up to bisimulation. For instance, $Idle_a = \mathsf{rec}\, A(a;).A(a;)$ is weakly bisimilar to the *identity* agent Id_a that we already introduced. We shall also assume that we distinguish, in the π_1-calculus, channel names from value names (as we did for $\mathrm{D}\pi_1^r$), and we restrict the ability to compare names to the latter. Again, we do not loose expressive power since, for instance, we can still encode joined input. Actually the embedding of the join-calculus into the π_1-calculus does not need the conditional operator on values, whereas the translation from the π_1-calculus to the $\mathrm{D}\pi_1^r$-calculus will make an essential use of it. For simplicity, we are also assuming that in the π_1-calculus restriction always acts on names of channel sort.

The idea of the encoding is quite standard and simple (see [2, 3, 8] for similar encodings). We turn any message on a channel a into a request to a *channel manager* $CM(a)$ for a, sending the arguments of the message together with a key out. Symmetrically, we turn any input on a into a request to $CM(a)$, sending a key in and a private return channel to actually receive something. The channel manager will filter the messages according to the keys, and act as appropriate. This is similar to what one would do in implementing communication for a language like the π-calculus – see for instance [6, 15] –, except that one would exploit elaborate data structures like "pools" or queues to manage the input and output requests in a more realistic way. However, there is an attack on this encoding which compromises abstraction: the environment can send a request for input to the channel manager. We then authenticate the requests for input by introducing a restricted key in_a for every channel manager (of a) which is known only by the process that can actually input on the channel a. In our encoding a message carrying any other key than in_a is regarded as a request for output (except inside internal choice, see below).

We shall use the notational conventions introduced in the previous section. In addition, we use process identifiers and recursive definitions such as $(\mathsf{rec}\,A(;\vec{b}).P)(;\vec{b'})$ which do *not* have an input parameter. These can be defined as $(\nu a'')(\mathsf{rec}\,A(a;\vec{b}).(Id_a \mid P'))(a'';\vec{b'})$ where the process P' is P where every free occurrence of $A(;\vec{b'})$ is replaced with $(\nu a')A(a';\vec{b'})$. We shall also need a kind of internal choice (similar to the one given in [2] by means of "booleans"), $P \oplus_a Q$ where a is a channel name of type $Ch(\vec{\tau}, val, val)$, and P and Q are such that $a \Vdash P, Q$. This is defined as follows – where $[x \neq y]P, Q$ is $[x = y]Q, P$, $c \neq c'$ and, as usual, all the introduced names are fresh:

$$\begin{aligned} P \oplus_a Q =_{\mathrm{def}} (\nu c : val)\big(\, & \overline{a}(\_, c, c) \mid (\nu c' : val)\overline{a}(\_, c, c') \mid \mathsf{rec}\,A(a;).a(\vec{u}, x, y). \\ & [x \neq c](\overline{a}(\vec{u}, x, y) \mid A(a;))\,, \\ & \mathsf{rec}\,A'(a;).a(\vec{u}, x', y').[x' \neq c](\overline{a}(\vec{u}, x', y') \mid A'(a;))\,, \\ & \qquad [x = y]P, Q\,\big) \end{aligned}$$

It is easy to see that this is a well-formed term, with interface $\{a\}$. In the following translation, the new names are assumed to be fresh. In particular, for every name a we assume a fresh name in_a (that is, not in the set of names of π_1). The name in_a is used as the key of the channel a. We translate a well-formed term $I \Vdash P$ (where P respects some sorting) of the π_1-calculus, where $I = \{a_1, \ldots, a_n\}$, in the following process of the $\mathrm{D}\pi_1^r$-calculus:

$$[\![I \Vdash P]\!] = (\nu\, in_{a_1} : val) \cdots (\nu\, in_{a_n} : val)(CM(a_1;) \mid \cdots \mid CM(a_n;) \mid [\![P]\!])$$

which turns out to be also well-formed in the context I. The type of a channel name, say $Ch(\vec{\tau})$, is transformed into $Ch(val, \vec{\tau}, Ch(\vec{\tau}), val, val)$: the first argument is the input/output key of the channel, then we have the arguments of the message to be delivered, followed by the type of the return channel to which they are actually sent, and then we have two keys for internal choice. The channel manager $CM(a;)$ is given by:

$$\begin{aligned} CM(a;) &= \mathsf{rec}\,A(a;).a(j, \vec{b}, s, c_1, c_1').a(i, \vec{d}, r, c_2, c_2'). \\ &\qquad L \oplus_a \mathsf{if}\ j = in_a\ \mathsf{then}\ L\ \mathsf{else}\ \mathsf{if}\ i \neq in_a\ \mathsf{then}\ L\ \mathsf{else}\ (\overline{r}\vec{b} \mid A(a;)) \qquad \text{where} \\ L &= A(a;) \mid \overline{a}(j, \vec{b}, s, c_1, c_1') \mid \overline{a}(i, \vec{d}, r, c_2, c_2') \end{aligned}$$

As one can see, the channel manager first expects a request for output, and then waits for a corresponding request for input. The definition of the channel manager follows a general method presented in [3] for the compilation of joined filtered inputs in the π_1-calculus. Note that the loop back process L allows to preempt any commitment concerning the message to be input. This is particularly helpful when trying to simulate the behaviour of the process $I \Vdash P$ with that of its translation. The process $[\![P]\!]$ is defined as follows:

$$\begin{array}{rcl}
[\![\overline{a}\vec{b}]\!] & = & \overline{a}(_,\vec{b},_,_,_) \\
[\![a(\vec{b}).P]\!] & = & (\nu r)(\overline{a}(in_a,_,r,_,_) \mid r(\vec{b}){:}[\![P]\!]) \\
[\![P \mid Q]\!] & = & ([\![P]\!] \mid [\![Q]\!]) \\
[\![[a=b]P,Q]\!] & = & [a=b][\![P]\!],[\![Q]\!] \\
[\![(\nu a)P]\!] & = & (\nu a)(\nu\, in_a : val)(CM(a;) \mid [\![P]\!]) \\
[\![(\mathsf{rec}\,A(\vec{a};\vec{b}).P)(\vec{a'};\vec{b'})]\!] & = & (\mathsf{rec}\,A(;\vec{a},\overrightarrow{in_a},\vec{b}).[\![P]\!])(;\vec{a'},\overrightarrow{in_{a'}},\vec{b'}) \\
[\![A(\vec{a};\vec{b})]\!] & = & A(;\vec{a},\overrightarrow{in_a},\vec{b})
\end{array}$$

As one could expect, the translation does not use any locality-based feature of the Dπ_1^r-calculus (compound names, location types, located processes). Moreover, the only receivers in the translation are the channel managers and the "input once" return channels. It is easy to see how the "roaming messages" are translated: the deadlocked situation where a message on a restricted name a cannot find a receiver is transformed into a process offerring a resource with no call to it, or into a livelock where other messages to a are perpetually resent. Our main result about this translation of the π_1-calculus into the receptive π_1-calculus is that it is fully abstract with respect to weak asynchronous bisimulation:

THEOREM 6.2. *Assume that $I \Vdash P$ and $I \Vdash Q$ in the π_1-calculus. Then*

$$P \approx_a Q \;\Leftrightarrow\; [\![I \Vdash P]\!] \approx_a [\![I \Vdash Q]\!]$$

The proof, which is to be found in the full version of the paper, goes through an analysis of the various actions that may be performed by $[\![I \Vdash P]\!]$. We then show that an exact simulation of the behaviour of π_1 processes is provided by the translation.

References

[1] R. AMADIO, I. CASTELLANI and D. SANGIORGI, *On bisimulations for the asynchronous calculus*, CONCUR'96, Springer Lect. Notes in Comp. Sci. 1119 (1996) 147-162.

[2] R. AMADIO, *An asynchronous model of locality, failure, and process mobility*, In Proc. COORDINATION'97, Springer Lect. Notes in Comp. Sci. 1282 (1997). Extended version appeared as Res. Report INRIA 3109.

[3] R. AMADIO, *On modeling mobility*, Journal of Theoretical Computer Science, to appear (1999).

[4] G. BOUDOL, *Typing the use of resources in a concurrent calculus*, Proc. ASIAN 97, Springer Lect. Notes in Comp. Sci. 1345 (1997) 239-253.

[5] G. BOUDOL, I. CASTELLANI, M. HENNESSY and A. KIEHN, *Observing Localities*, Theoretical Computer Science 114 (1993) 31-61.

[6] L. CARDELLI, *An implementation model of rendez-vous communication*, In Seminar on Concurrency, Springer Lect. Notes in Comp. Sci. 197 (1985) 449-457.

[7] L. CARDELLI and A. GORDON, *Mobile ambients*, In Proc. FoSSaCS, ETAPS 98, Springer Lect. Notes in Comp. Sci. 1378 (1998) 140-155.

[8] C. Fournet and G. Gonthier, *The reflexive CHAM and the join-calculus*, In Proc. ACM Principles of Prog. Lang. (1996) 372-385.

[9] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy, *A calculus of mobile agents*, In Proc. CONCUR'96, Springer Lect. Notes in Comp. Sci. 1119 (1996) 406-421.

[10] M. Hennessy and J. Riely, *Resource access control in systems of mobile agents*, Techn. Report 2/98, School of Cognitive and Computer Sciences, University of Sussex (1998).

[11] N. Kobayashi, *A partially deadlock-free typed process calculus*, ACM TOPLAS Vol. 20 No 2 (1998) 436-482.

[12] M. Merro and D. Sangiorgi, *On asynchrony in name-passing calculi*, In Proc. ICALP'98, Springer Lect. Notes in Comp. Sci. 1443 (1998) 856-867.

[13] D. Sangiorgi, *The name discipline of uniform receptiveness*, In Proc. ICALP'97, Springer Lect. Notes in Comp. Sci. 1256 (1997) 303-313.

[14] E. Sumii and N. Kobayashi, *A generalized deadlock-free process calculus*, In Proc. HLCL'98, Electronic Notes in Comp. Sci. vol. 16-3 (1998).

[15] D. Turner, *The Polymorphic Pi-Calculus: Theory and Implementation*, Ph. D. Thesis, University of Edinburgh (1995).

[16] V. Vasconcelos, *Typed Concurrent Objects* In Proc. ECOOP'94, Springer Lect. Notes in Comp. Sci. 821 (1994) 100-117.