An Asynchronous Model of Locality, Failure, and Process Mobility

Roberto M. Amadio

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An asynchronous model of locality, failure, and process mobility

Roberto M. Amadio

N° 3109

Février 1997

THÈME 1

ISSN 0249-6399

An asynchronous model of locality, failure, and process mobility

Roberto M. Amadio

Thème 1 — Réseaux et systèmes
Projet Meije

Rapport de recherche n ° 3109 — Février 1997 — 31 pages

Abstract: We present a model of distributed computation which is based on a fragment of the π-calculus relying on *asynchronous* communication. We enrich the model with the following features: the explicit distribution of processes to locations, the failure of locations and their detection, and the mobility of processes. Our contributions are two folds. At the specification level, we give a synthetic and flexible formalization of the features mentioned above. At the verification level, we provide original methods to reason about the bisimilarity of processes in the presence of failures.

Key-words: π-calculus. Bisimulation. Asynchronous communication. Locations. Models of distributed systems. Failures and failures' detection. Mobility of processes.

(Résumé : tsvp)

The author is a Professor at the *Laboratoire d'Informatique de Marseille*, and an external collaborator of the INRIA-Ecole des Mines Project *Meije*. He can be contacted at the following address: CMI, 39 rue Joliot-Curie, F-13453, Marseille, France. `mailto: amadio@gyptis.univ-mrs.fr`, `http://protis.univ-mrs.fr/~amadio/`. This work was partially supported by France Télécom, CTI-CNET 95-1B-182 Modélisation de Systèmes Mobiles, and by the French-Indian collaboration program CEFIPRA.

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – Télécopie : 04 93 65 77 65

Un modèle asynchrone de la localité, l'échec et la mobilité de processus

Résumé : Nous présentons un modèle de calcul réparti qui est basé sur un fragment du π-calcul avec communication asynchrone. Nous enrichissons le modèle pour représenter la distribution explicite de processus à locations, l'échec de locations et leur détection, et la mobilité de processus. Au niveau de la spécification, nous présentons une formalisation synthétique et flexible des aspects mentionnés ci-dessus. Au niveau de la vérification, nous décrivons des methodes originales pour raisonner sur la bisimilarité de processus en presence d'échec.

Mots-clé : π-calculus. Bisimulation. Communication asynchrone. Locations. Modèles de systèmes repartis. Échec et détection d'échec. Mobilité de processus.

1 Introduction

Traditional process calculi such as CCS and CSP lie their foundations on a reduced set of concepts and therefore do not provide direct support for the modeling of certain relevant aspects of systems such as the distribution of resources on different locations, the impact of failures on the behaviour of the system, the detection of failures, and the mobility of processes (the exact meaning of these terms will become clearer, as we progress in our discussion).

This paper pursues a research line initiated in [AP94], in which an explicit modeling of the features mentioned above is specified, and then a reduction to a more basic model is seeked.

In carrying on this program, we rely on a π-calculus formalism [AZ84, EN86, MPW92]. In first approximation, the π-calculus models systems of asynchronous processes which interact by message passing. The calculus embodies features such as dynamic process creation, dynamic channel creation, transmission of channel names, and a static scoping discipline. The blending of these features has led to a calculus which is quite expressive and close to programming issues, while having a tractable semantic theory.

We select a variety of the π-calculus as the basic model on which additional features are added. The advantage of this approach, is that notions and results can be inherited and stated, respectively, within the theory of the π-calculus. The disadvantage is that to understand this paper some knowledge of the π-calculus is required.

The variety of π-calculus which we consider is a fragment of the *asynchronous* π-calculus [HT91, Bou92]. In this calculus, the send of a message is non-blocking, that is a process can deliver a message without waiting for a receiving process (think of e-mail). This communication model implicitly relies on a non-bounded buffer in which messages can be stocked. Messages in the buffer can be reordered in arbitrary ways (the buffer does not obey a FIFO discipline).

We consider a fragment of the asynchronous π-calculus in which every channel name is associated with a *unique* (persistent) process which serves messages addressed to that name (communication becomes point-to-point). To emphasize the unicity of the receptor, we will refer to this fragment as the π_1-calculus. Technically, the π_1-calculus is formalized by means of a simple typing discipline which enjoys a suitable subject reduction property. We show that the π_1-calculus is sufficiently expressive to simulate the asynchronous π-calculus (with multiple receivers). We also observe that by restricting the syntax to "functional" processes, we can define an expressive sub-calculus where (internal) reduction is *confluent*.

Starting from the π_1-calculus, we specify in an incremental way the features we are interested in:

1. We explicitly distribute processes to *locations*. Locations are our unit of distribution and they can be generated dynamically.

2. Locations are also our unit of *failure*. A location can fail, entailing the failure of all processes running at it.

3. We specify an operator to *spawn* a process at a remote location. It is then possible to synthesize a *closure*, i.e. a process with an environment, at a location and start its execution at another location.

4. We specify an operator to *detect the failure* of a location.

There is a variety of choices to be made concerning the model of failures (halting, transient, byzantine,...), the exact kind of mobility of processes which is allowed and its impact on message routing, and the power of the failure detectors. We will not try to cover all possible combinations of these choices, instead we will study in depth a simple model while hinting to possible variations.

In first approximation, we will consider a system of asynchronous processes which interact by asynchronous message passing. Processes are distributed to locations which can stop (halting failure), they can spawn processes at remote locations under certain conditions which keep the routing problem simple, and they can consult a perfect oracle which will eventually say if a location has failed or not.

The rest of this paper is organised as follows. In sections 2 and 3, we define our model and illustrate its expressivity. In particular, in section 2 we present the π_1-calculus, and study its typing system (theorem 1), and in section 3, we incrementally define the π_{1l}-calculus as an enrichment of the π_1-calculus where locations, failures, mobility of processes, and failures detectors are explicitly modelled.

In section 4, we turn to semantic issues. Our goal is to develop techniques to prove the bisimilarity of processes. In particular, we study characterizations of contextual equivalences, and calculi translations. We define an adequate translation (theorem 2) from the π_{1l}-calculus to the π_1-calculus. Next, we characterise barbed equivalence (a contextual equivalence) for the π_1-calculus (theorem 3). The tool we use is a recently introduced notion of asynchronous bisimulation [ACS96]. We also show that there is a fragment of the π_{1l}-calculus for which the translation into the π_1-calculus is fully abstract, and we formalize the fact that in our model distribution is *transparent* in the absence of failures.

Finally in section 5, we consider related work and summarize our main achievements.

2 The asynchronous π_1-calculus

We start by considering a polyadic, asynchronous π-calculus whose processes are specified as follows (we often omit parentheses):

$$p ::= a(\vec{b}).p \mid \overline{a}\vec{b} \parallel p \mid p \parallel \mathbf{0} \mid \nu a\, p \mid (\mathtt{rec} A(\tilde{a}).p)(\tilde{b}) \parallel A(\tilde{a}) \mid [a=b]p, q \qquad (1)$$

We collect here some basic conventions. We denote with $a, b, \ldots$ channel names, with $\vec{a}, \vec{b}, \ldots$ vectors of channel names, with $\tilde{a}$ two vectors of channel names separated by a ";", say $\vec{a_1}; \vec{a_2}$ (either vector can be empty), and with $p, q, \ldots$ processes. The sets $fn(p)$, $bn(p)$ contain, respectively, the names free and bound in the process p. If $\vec{a}$ is a vector of names, we denote with $\{\vec{a}\}$ the corresponding set. If $\tilde{a} = \vec{a_1}; \vec{a_2}$ then we let $\tilde{a}_{io} = \vec{a_1}$, $\tilde{a}_o = \vec{a_2}$, and

$\{\tilde{a}\} = \{\vec{a_1}\} \cup \{\vec{a_2}\}$. Intuitively, in a recursive definition, we distinguish between the names $\tilde{a}_{io}$ that can be used in input and output, and the names $\tilde{a}_o$ that can be used in output only. Correspondingly, every process identifier A has two arities $ar_{io}(A)$ and $ar_o(A)$: $ar_{io}(A)$ is the number of parameters that can be used in input and output, whereas $ar_o(A)$ is the number of parameters that can be used only in output. In a well-formed process, actual and formal parameters agree, and all process identifiers are bound. In a recursive definition $(\mathtt{rec}A(\tilde{a}).p)(\tilde{b})$, we suppose that $fn(p) \subseteq \{\tilde{a}_{io}, \tilde{a}_o\}$. To define recursive processes, we will also rely on parametric equations as an equivalent notation. The equivalence $\equiv$ stands for syntactic identity up to renaming of bound names.

Sorts are defined as follows: $s ::= Ch(s_1, \ldots, s_n)$, where $n \geq 0$. A channel of sort $Ch(s_1, \ldots, s_n)$ can carry a tuple $c_1, \ldots, c_n$, where c_i has sort s_i, for $i = 1, \ldots, n$. We suppose that every name a has a sort s which we denote with $st(a)$, that there are infinitely many names for every sort, and that terms are well-sorted. The basic reduction rule of this calculus is:

$$\text{(cm)} \quad a(\vec{b}).p \mid \overline{a}\vec{c} \;\; \rightarrow [\vec{c}/\vec{b}]p \tag{2}$$

The behaviour of a process is completely described by a labelled transition system (lts), whose actions α are specified as follows:

$$\alpha ::= \tau \mid a\vec{b} \parallel \nu\{\vec{c}\}\,\overline{a}\vec{b} \tag{3}$$

In $\nu\{\vec{c}\}\,\overline{a}\vec{b}$, we suppose that $a \notin \{\vec{c}\} \subseteq \{\vec{b}\}$. Conventionally, we set $n(\alpha) = fn(\alpha) \cup bn(\alpha)$ where:

$$\begin{array}{lll} fn(\tau) = \emptyset & fn(a\vec{b}) = \{a\} \cup \{\vec{b}\} & fn(\nu\{\vec{c}\}\,\overline{a}\vec{b}) = \{a, \vec{b}\}\backslash\{\vec{c}\} \\ bn(\tau) = \emptyset & bn(\overline{a}\vec{b}) = \emptyset & bn(\nu\{\vec{c}\}\,\overline{a}\vec{b}) = \{\vec{c}\} \end{array} \tag{4}$$

The labelled transition system is specified in figure 1, following an early instantiation style. The notion of *weak* transition is defined as usual: $p \stackrel{\tau}{\Rightarrow} p'$ iff $p(\stackrel{\tau}{\rightarrow})^* p'$, and, for $\alpha \neq \tau$, $p \stackrel{\alpha}{\Rightarrow} p'$ iff $p \stackrel{\tau}{\Rightarrow} \cdot \stackrel{\alpha}{\rightarrow} \cdot \stackrel{\tau}{\Rightarrow} p'$.

The π_1-calculus The π_1-calculus is a *typed* version of the asynchronous π-calculus. A typing context Γ, is a set of names $\{a_1, \ldots, a_n\}$. In figure 2, we introduce a system to prove when a process p is well-typed in the context Γ. The typing rules rely on the following intuitions: (1) If $a \in \Gamma$ then there is *exactly* one (persistent) process that is allowed to receive on a. (2) Property (1) has to be preserved by labelled transitions. (3) Whenever we create a name, we have to make sure that a unique receiving process is associated to that name.

The typing rules apply to processes with free process identifiers, as to type a recursive definition we need to type a process where the related process identifier is free. The actual parameters of a recursive definition provide a kind of *declaration* of the channel names on which the defined process intends to perform input/output actions, and output actions, respectively. The typing system makes a "linear" use of the names in the context, and in this respect it has some points in common with other typing systems which have been proposed for the π-calculus (cf., e.g., [KPT96]). What appears to be original, is the handling

$$(in)\quad \frac{\cdot}{a(\vec{b}).p \stackrel{a\vec{c}}{\to} [\vec{c}/\vec{b}]p} \qquad (out)\quad \frac{\cdot}{\overline{a}\vec{b} \stackrel{\overline{a}\vec{b}}{\to} \mathbf{0}}$$

$$(out_{ex})\quad \frac{p \stackrel{\nu\{\vec{c}\}\,\overline{a}\vec{b}}{\to} p' \quad a \neq d \quad d \in \{\vec{b}\}\backslash\{\vec{c}\}}{\nu d\, p \stackrel{\nu\{d,\vec{c}\}\,\overline{a}\vec{b}}{\to} p'} \qquad (\nu)\quad \frac{p \stackrel{\alpha}{\to} p' \quad a \notin n(\alpha)}{\nu a\, p \stackrel{\alpha}{\to} \nu a\, p'}$$

$$(cp)\quad \frac{p \stackrel{\alpha}{\to} p' \quad bn(\alpha) \cap fn(q) = \emptyset}{p \mid q \stackrel{\alpha}{\to} p' \mid q} \qquad (cm)\quad \frac{p \stackrel{\nu\{\vec{c}\}\,\overline{a}\vec{b}}{\to} p' \quad q \stackrel{a\vec{b}}{\to} q' \quad \{\vec{c}\} \cap fn(q) = \emptyset}{p \mid q \stackrel{\tau}{\to} \nu\vec{c}(p' \mid q')}$$

$$(\mathtt{rec})\quad \frac{[\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p \stackrel{\alpha}{\to} p'}{(\mathtt{rec}A(\tilde{a}).p)(\tilde{b}) \stackrel{\alpha}{\to} p'} \qquad (cg)\quad \frac{p \equiv p' \quad p' \stackrel{\alpha}{\to} q' \quad q' \equiv q}{p \stackrel{\alpha}{\to} q}$$

$$(m_{\mathtt{t}})\quad \frac{p \stackrel{\alpha}{\to} p'}{[a = a]p, q \stackrel{\alpha}{\to} p'} \qquad (m_{\mathtt{f}})\quad \frac{q \stackrel{\alpha}{\to} q' \quad a \neq b}{[a = b]p, q \stackrel{\alpha}{\to} q'}$$

Figure 1: Labelled transition system for the asynchronous polyadic π-calculus

$$\frac{\cdot}{\emptyset \vdash 0} \qquad \frac{\cdot}{\emptyset \vdash \overline{a}\vec{b}}$$

$$\frac{\Gamma_1 \vdash p_1 \quad \Gamma_2 \vdash p_2 \quad \Gamma_1 \cap \Gamma_2 = \emptyset}{\Gamma_1 \cup \Gamma_2 \vdash p_1 \mid p_2} \qquad \frac{\Gamma \cup \{a\} \vdash p \quad a \notin \Gamma}{\Gamma \vdash \nu a\, p}$$

$$\frac{\Gamma \vdash p \quad a \in \Gamma \quad \Gamma \cap \{\vec{b}\} = \emptyset}{\Gamma \vdash a(\vec{b}).p} \qquad \frac{\{\tilde{a}_{io}\} \vdash p \quad \sharp\{\tilde{b}_{io}\} = ar_{io}(A)}{\{\tilde{b}_{io}\} \vdash (\mathtt{rec}A(\tilde{a}).p)(\tilde{b})}$$

$$\frac{\sharp\{\tilde{a}_{io}\} = ar_{io}(A)}{\{\tilde{a}_{io}\} \vdash A(\tilde{a})} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash [a = b]p, q}$$

Figure 2: Typing rules for the π_1-calculus

of the input prefix and of the recursive definitions. Note that in a recursive definition we require that the number of distinct actual io-parameters equals the io-arity of the process identifier ($\sharp\{\vec{b}\} = ar_{io}(A)$). Hence, the typing under a process identifier is performed under the hypothesis that all actual io-parameters are distinct.

We can show that typing is preserved by labelled transitions. Typing contexts are not affected by labelled transitions but in the case of output extrusion. We note that a context never shrinks, this is because the π_1-calculus always keeps a trace of the running processes, even when they are virtually terminated as in the process $Idle(a)$ (cf. figure 3). This design decision entails that if two processes are typed with respect to the same context, then this property is preserved by labelled transitions. This fact simplifies the definition of bisimulation (cf. definition 8).

Theorem 1 (subject reduction) *If* $\Gamma \vdash p$ *and* $p \xrightarrow{\alpha} p'$ *then* $\Gamma \cup bn(\alpha) \vdash p'$.

Proof of theorem 1 Let σ be a name substitution which is the identity almost everywhere. We say that σ is injective on a context Γ, if σ restricted to Γ is injective. We write $\sigma\Gamma$ for $\{\sigma a \mid a \in \Gamma\}$, and σp for the application of the substitution σ to the process p.

Lemma 1 *If* $\Gamma \vdash p$, *and* σ *is an injective substitution on* Γ, *then* $\sigma\Gamma \vdash \sigma p$.

Proof. By induction on the proof of $\Gamma \vdash p$. For instance, we consider the case where the last rule applied is:

$$\frac{\{\tilde{a}_{io}\} \vdash p \quad \sharp\{\tilde{b}_{io}\} = ar_{io}(A)}{\{\tilde{b}_{io}\} \vdash (\mathtt{rec}A(\tilde{a}).p)(\tilde{b})}$$

Let σ be injective on $\{\tilde{b}_{io}\}$. Without loss of generality, we may suppose that the bound names $\tilde{a}$ have been renamed so that $\sigma\tilde{a} = \tilde{a}$. Then $\sigma(\mathtt{rec}A(\tilde{a}).p) = (\mathtt{rec}A(\tilde{a}).p)$, as $fn(p) \subseteq \{\tilde{a}\}$. Since σ is injective on $\{\tilde{b}_{io}\}$, we have $\sharp(\sigma\{\tilde{b}_{io}\}) = ar_{io}(A)$. Hence, we can conclude $\sigma\{\tilde{b}_{io}\} \vdash \sigma(\mathtt{rec}A(\tilde{a}).p)(\tilde{b})$. QED

Lemma 2 *If* $\{\tilde{a}_{io}\} \vdash p$ *and* $\sharp\{\tilde{b}_{io}\} = ar_{io}(A)$ *then* $\{\tilde{b}_{io}\} \vdash [\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p$.

Proof. From a proof of $\{\tilde{a}_{io}\} \vdash p$ we can obtain a proof of $\{\tilde{b}_{io}\} \vdash [\tilde{b}/\tilde{a}]p$ by lemma 1. We consider the leaves of the related proof tree having the shape:

$$\frac{\sharp\{\tilde{c}_{io}\} = ar_{io}(A)}{\{\tilde{c}_{io}\} \vdash A(\tilde{c})}$$

If we replace each leaf of this type with a proof whose root has the shape:

$$\frac{\sharp\{\tilde{c}_{io}\} = ar_{io}(A) \quad \{\tilde{a}_{io}\} \vdash p}{\{\tilde{c}_{io}\} \vdash (\mathtt{rec}A(\tilde{a}).p)(\tilde{c})}$$

We obtain a proof of $\{\tilde{b}_{io}\} \vdash [\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p$. QED

We can now prove theorem 1 by induction on the derivation of the transition $p \xrightarrow{\alpha} p'$ and analysis of the last typing rule applied. For instance suppose the transition rule is:

$$\frac{[\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p \xrightarrow{\alpha} p'}{(\mathtt{rec}A(\tilde{a}).p)(\tilde{b}) \xrightarrow{\alpha} p'}$$

Then the last typing rule applied is:

$$\frac{\{\vec{a}_{io}\} \vdash p \quad \sharp\{\tilde{b}_{io}\} = ar_{io}(A)}{\{\tilde{b}_{io}\} \vdash (\mathtt{rec}A(\tilde{a}).p)(\tilde{b})}$$

By lemma 2, we derive $\{\tilde{b}_{io}\} \vdash [\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p$. By inductive hypothesis, we can conclude $\{\tilde{b}_{io}\} \cup bn(\alpha) \vdash p'$. QED

Barbed bisimulation We provide some insight on the way π_1-processes can be observed. For the time being, we will just introduce a notion of barbed bisimulation which is sufficient to argue about the *adequacy* of various encodings. In section 4, we will develop a notion of (asynchronous) bisimulation for the π_1-calculus based on the lts in figure 1.

As for the asynchronous π-calculus (cf. [HT91, ACS96]), we should suppose that only output actions are visible. Intuitively, since communication is asynchronous the observer has no way of knowing when an input action is carried on (we refer to [HT91, ACS96] for a more extended discussion). There is also an additional hypothesis that should be made, namely we suppose that an output action is visible only if the corresponding receptor is not defined in the observed process (otherwise the resulting process would not be well-typed). The context Γ tells us exactly which are the receptors defined in the process p. Hence if $\Gamma \vdash p$, then we can only observe output commitments on names which are not in Γ.

Definition 1 (commitment) *Suppose* $\Gamma \vdash p$. *We write* $p \downarrow \overline{a}$ *if* $p \xrightarrow{\alpha} p'$, $\alpha \equiv \nu\{\vec{c}\}\,\overline{a}\vec{b}$, *and* $a \notin \Gamma$. *We also write* $p \Downarrow \overline{a}$ *if* $p \stackrel{\tau}{\Rightarrow} p'$ *and* $p' \downarrow \overline{a}$.

Definition 2 (barbed bisimulation) *A symmetric relation* S *on* π_1*-terms is a strong barbed bisimulation if whenever* pSq *the following holds:*

(1) *If* $p \downarrow \overline{a}$ *then* $q \downarrow \overline{a}$.

(2) *If* $p \xrightarrow{\tau} p'$ *then* $q \xrightarrow{\tau} q'$ *and* $p'Sq'$.

Let $\overset{\bullet}{\sim}$ *be the largest barbed bisimulation. The notion of weak barbed simulation is obtained by replacing everywhere the commitment* $\downarrow$ *with* $\Downarrow$, *and the reduction* $\xrightarrow{\tau}$ *with* $\stackrel{\tau}{\Rightarrow}$. *We denote with* $\overset{\bullet}{\approx}$ *the largest weak barbed bisimulation.*

Derived operators Our next goal is to provide evidence for the expressivity of the π_1-calculus. Towards this end, we introduce in figure 3 a few *derived* operators which allow for a more handy notation. For each operator, we show the derived typing and (internal) reduction rules. In the following, we give some intuition, and state some properties of these operators.

- The process $Idle(\vec{a})$ can be regarded as a process which declares the channels $\vec{a}$ for input/output but never actually uses them.
- Using the idle process, we can type a process that *receives only once* on a channel.
- The replicated input operator is particularly interesting. The process $a(\vec{b}) \triangleright p$ (if we had π-calculus replication, we could write this process as $!(a(\vec{b}).p)$) can be thought as a *functional* or *stateless* process. This feature can be formalised as follows.

Definition 3 (π_{1f}-calculus) *Let the* π_{1f}*-calculus (f for functional) be the subcalculus of the* π_1*-calculus in which we allow input prefix and recursion only as macro expansions of processes of the shape* $a(\vec{b}) \triangleright p$.

Let $\equiv_1$ be a structural equivalence which includes besides α-renaming, the laws for the commutation of restriction with restriction and parallel composition, and the laws for the associativity and commutativity of parallel composition.

Proposition 1 (confluence) *In the* π_{1f}*-calculus,* τ*-reduction is confluent modulo* $\equiv_1$.

PROOF HINT. We note that the π_{1f}-calculus is closed under τ reduction. Given a term of the π_{1f}-calculus, two distinct reductions superpose when two messages are addressed to the same channel, as in $C[\overline{a}\vec{b} \mid \overline{a}\vec{b'} \mid a(\vec{c}) \triangleright p]$. It is immediately checked that the two reductions commute. QED

We note that the typing rules forbid the nesting of replicated inputs on free names. Indeed, this would break the property that each channel has at most one receiver. Nevertheless, the π_{1f}-calculus is still quite expressive. For instance, one can adequately encode the simply

Idle

$$Idle(\vec{a}) \equiv (\mathtt{rec}A(\vec{b};).A(\vec{b};))(\vec{a};)$$

$$\frac{\sharp\{\vec{a}\} = ar_{io}(A)}{\{\vec{a}\} \vdash Idle(\vec{a})}$$

Input once

$$a(\vec{b}) : p \equiv a(\vec{b}).(p \mid Idle(a))$$

$$\frac{\Gamma \vdash p \quad a \notin \Gamma}{\Gamma \cup \{a\} \vdash a(\vec{b}) : p} \qquad a(\vec{b}) : p \mid \overline{a}\vec{c} \to [\vec{c}/\vec{b}]p \mid Idle(a)$$

Replicated input

$$a(\vec{b}) \triangleright p \equiv (\mathtt{rec}A(a;\vec{a'}).a(\vec{b}).(A(a;\vec{a'}) \mid p))(a;\vec{a'})$$

$fn(p) \subseteq \{\vec{b}\} \cup \{\vec{a'}\} \cup \{a\}$, which are pairwise disjoint sets.

$$\frac{\emptyset \vdash p \quad a \notin \{\vec{b}\}}{\{a\} \vdash a(\vec{b}) \triangleright p} \qquad a(\vec{b}) \triangleright p \mid \overline{a}\vec{c} \to a(\vec{b}) \triangleright p \mid [\vec{c}/\vec{b}]p$$

Booleans

$$(\mathtt{if}\ c\ \mathtt{then}\ p\ \mathtt{else}\ q) \equiv [c_1 = c_2]p, q$$

$$\overline{a}\mathtt{t}, \vec{b} \equiv \nu c_1\, (\overline{a}c_1, c_1, \vec{b} \mid Idle(c_1))$$
$$\overline{a}\mathtt{f}, \vec{b} \equiv \nu c_1\, \nu c_2\, (\overline{a}c_1, c_2, \vec{b} \mid Idle(c_1, c_2))$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash \mathtt{if}\ c\ \mathtt{then}\ p\ \mathtt{else}\ q}$$

Internal choice

$$p \oplus q \equiv \nu a\, (a(c) : \mathtt{if}\ c\ \mathtt{then}\ p\ \mathtt{else}\ q \mid \overline{a}\mathtt{t} \mid \overline{a}\mathtt{f})$$

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \oplus q} \qquad p \oplus q \to \cdot \sim_a p \quad p \oplus q \to \cdot \sim_a q$$

Link

$$a \mapsto b \equiv Link(a;b) = a(c) \triangleright \nu d\, (\overline{b}d \mid Link(d;c))$$

$$\frac{\cdot}{\{a\} \vdash a \mapsto b} \quad (\overline{a}c \mid a \mapsto b) \to a \mapsto b \mid \nu d\, (\overline{b}d \mid d \mapsto c)$$

Figure 3: Derived operators

typed call-by-value λ-calculus with a ground type o as follows (cf. [ALT95]), where we note $\emptyset \vdash \langle M \rangle d$.

$$\begin{cases} \langle o \rangle & = Ch() \\ \langle \sigma \to \tau \rangle & = Ch(\langle \sigma \rangle, Ch(\langle \tau \rangle)) \\ \langle a \rangle d & = \overline{d} a \\ \langle \lambda a.M \rangle d & = \nu c\,(\overline{d} c \mid c(a, d') \triangleright \langle M \rangle d') \\ \langle MN \rangle d & = \nu d'\, \nu d''\, (\langle M \rangle d' \mid d'(c) \triangleright (\langle N \rangle d'' \mid d''(c') \triangleright \overline{c}(c', d))) \end{cases}$$

Roughly, one can think of the join calculus [FG96] as the π_{1f}-calculus extended with the *join operator*. The join operator allows to receive two (or more) messages as an atomic operation. This feature is essential in programming non-functional processes, in particular using the join, one can represent a variant of the *channel manager* described in the following figure 5 (which can be understood as a process with two states).

• Boolean values `t` and `f` are coded as a pair of fresh names (equal for `t` and distinct for `f`). We use `bool` as an abbreviation for $Ch(), Ch()$ (which is a list of sorts). If c is a pair, we denote with c_1 the first component and with c_2 the second. An `if_then_else_` operator can then be simulated relying on the matching operator. Using the `if_then_else_`, we can code an internal choice operator (the equivalence $\sim$ stands for strong bisimulation and will be defined in 8). It is possible to code the `if_then_else_` and the internal choice operators without using the matching operator, however in this case the typing rules are less general.

Another possibility, is to remove the matching operator and introduce a rule to type (a simulation of) the `if_then_else_`. In this case, internal choice can still be defined, but matching is not definable. Indeed, it can be shown that contexts without matching have less discriminating power. In a calculus without matching, what matters of a name is not its identity, but the visible activity one can generate by sending a message to it. It is easy to immagine situations in which two distinct names generate the same activity, and therefore cannot be distinguished from an external observer.

Definition 4 (π_1^--calculus) *Let the π_1^--calculus be fragment of the π_1-calculus without matching.*

• We can translate the π_1^--calculus into a sub-calculus where all transmitted names are new, or equivalently, the transmission of free names is forbidden. The translation relies on the *link operator* [San95] which can be used to simulate a free input with a bound input. The idea is to replace the message $\overline{a}b$, where b is free, with the process $\nu c\,(\overline{a}c \mid c \mapsto b)$. The link $c \mapsto b$ forwards messages addressed to c, to the channel b, and recursively replaces a free output with a bound output, hence introducing another link process. The translation acts as follows on the output, and does essentially nothing on the other processes of the π_1^--calculus, that is $[\![p \mid p']\!] = [\![p]\!] \mid [\![p']\!]$, etcetera.

$$[\![\overline{a}b]\!] = \nu c\,(\overline{a}c \mid c \mapsto b) \tag{5}$$

In this paper, we will not study the formal properties of this translation.

$$\langle \overline{a} b_1, \ldots, b_n \rangle = \nu c\, (\overline{a} c \mid c(d).(\overline{d} b_1 \mid \cdots \mid c(d).(\overline{d} b_{n-1} \mid c(d) : \overline{d} b_n) \cdots))$$

$$\langle a(b_1, \ldots, b_n).p \rangle = a(c).(\nu d\, \overline{c} d \mid d(b_1).(\overline{c} d \mid \cdots \mid d(b_{n-1}).(\overline{c} d \mid d(b_n) : \langle p \rangle) \cdots))$$

Figure 4: From the polyadic to the unsorted monadic π_1-calculus

$$\begin{array}{lll}
\langle a(\vec{b}).p \rangle & = \nu c\, (\overline{a_i} c \mid c(\vec{b_o}) : \langle p \rangle) & \\
\langle \overline{a} \vec{b} \rangle & = \overline{a_o} \vec{b_o} & \\
\langle \nu a\, p \rangle & = \nu a_i\, \nu a_o\, (\langle p \rangle \mid CM(a_i, a_o)) & \\
 & \quad CM(a_i, a_o) = a_o(\vec{b_o}).a_i(c). & (\overline{a_o} \vec{b_o} \mid \overline{a_i} c \mid CM(a_i, a_o)) \oplus \\
 & & (\overline{c} \vec{b_o} \mid CM(a_i, a_o)) \\
\langle p \mid q \rangle & = \langle p \rangle \mid \langle q \rangle &
\end{array}$$

Figure 5: From a π-calculus with multiple receivers to the π_1-calculus

Another useful translation, is the one into a monadic π-calculus where all transmitted vectors of names have length one. In the monadic calculus, we assume that all names have a sort s satisfying the recursive equation $s = Ch(s)$. By analogy with the untyped λ-calculus, we call this the *unsorted* monadic π_1-calculus. We observe that the translation presented in [Bou93] from the polyadic to the monadic asynchronous π-calculus can be typed in our framework. We outline the translation in figure 4. Note that there are more refined sorting disciplines which can be defined on the monadic calculus, and in which we can still sort the translation above. One obvious solution, is to introduce a "sum" sort and assign to the channel d in the translation the sort $Ch(s_1 + \cdots + s_n)$, where s_i are the sorts assigned to the names b_i.

Translating the asynchronous π-calculus A test for the expressivity of the π_1-calculus is its ability to simulate a calculus where a channel can have multiple receivers. As source language, we consider the core of an asynchronous polyadic π-calculus. The translation is presented in figure 5. We suppose that for every channel a with sort s of the source calculus there is a pair of names a_i, a_o (i for input and o for output) in the π_1-calculus such that a_i has sort $Ch(s)$ and a_o has sort s. Since we cannot have several receivers on the same channel, we associate to every (restricted) channel a channel manager $CM(a_i, a_o)$, which continuously receives input/output requests and matches them if possible. We note that $\emptyset \vdash \langle p \rangle$.

A first rough relationship between the source and target calculus can be stated by supposing that in the source calculus we consider processes such that: (i) all input names are restricted (so that the commitment $\overline{a}_i c$ in the translation are hidden), and (ii) input parameters cannot be used as the subject of an input action. The notion of barbed bisimulation is adapted in a straightforward way to this asynchronous π-calculus.

It is possible to give decidable conditions that guarantee properties (i-ii), for instance see the read/write sorting discipline in [PS93]. Moreover, property (ii) is not so restrictive since Boreale [Bor96] has defined an adequate translation from an asynchronous π-calculus into an asynchronous π-calculus satifsying condition (ii).

Proposition 2 *Let p, p' be processes of the asynchronous π-calculus satisfying properties (i) and (ii). Then: $p \dot{\approx} p'$ iff $\langle p \rangle \dot{\approx} \langle p' \rangle$.*

PROOF HINT. We observe that:

(1) If $p \xrightarrow{\tau} p'$ then $\langle p \rangle (\xrightarrow{\tau})^4 \langle p' \rangle$.

(2) $p \downarrow \overline{a}$ iff $\langle p \rangle \downarrow \overline{a_o}$.

In simulating a communication on the channel a the process $CM(a_o, a_i)$ takes four steps: it performs two inputs on a_o and a_i respectively then it performs an internal choice and finally communicates to the receiver the actual parameters. The last action is deterministic and does not rise any difficulty. The first two actions are potentially non-deterministic, to avoid this phenomena of *gradual commitment*, the channel manager can always preempt the communication by means of an internal choice. By using this preemption mechanism, we concentrate all non-determinism on the internal choice. We can then consider the first two and the fourth communications as *administrative*. We write $q \stackrel{\tau}{\Rightarrow}_{ad} q'$, if q reduces to q' by means of administrative reductions. Let $Pr_1 = \{q \mid \exists p(\langle p \rangle \stackrel{\tau}{\Rightarrow} q)\}$. We define a relation $\mathcal{R}$ between well formed π-terms and π_1-terms in Pr_1 as follows:

$$p \mathcal{R} q \text{ if } \exists q' \, (\langle p \rangle \stackrel{\tau}{\Rightarrow}_{ad} q' \text{ and } q \stackrel{\tau}{\Rightarrow}_{ad} q') \tag{6}$$

We show that processes which are $\mathcal{R}$-related have corresponding commitments and their internal reductions can be kept in lockstep. QED

3 An enriched π_1-calculus

We extend the syntax of the π_1-calculus in order to model the distribution of processes to locations, the failure of a location, the spawning of a process at a remote location, and the detection of a failure.

Language We start by defining the language of *configurations*. A configuration is a "solution" in which we can find processes running at a location, messages, and locations.

- A process p running at a location a is denoted with $\{p\}a$. New channels and new processes that might be created during the computation of p are located in a. To create processes at remote locations, a special operator $\mathtt{spawn}(\_)$ is applied.
- Messages (m) can be output particles ($\overline{a}\vec{b}$), stop of a location a ($\mathtt{stop}(a)$), spawning of a process p at a location a ($\mathtt{spawn}(a,p)$), and testing of a location a, with a return on b_1 if the location is running, and on b_2 otherwise ($\mathtt{ping}(a,b_1,b_2)$).
- We associate to every location name a *location process* which receives routing, $\mathtt{stop}(\_)$, $\mathtt{spawn}(\_)$, and $\mathtt{ping}(\_)$ messages. To this end, we introduce a new sort $\mathtt{loc}$, and a specific way of creating a location process which receives on a name a of sort $\mathtt{loc}$ ($Loc_T(a)$, where $T \in \{R,S\}$, R for run, and S for stop). Location names are just names of sort $\mathtt{loc}$, in particular location names are transmissible values. The typing rules will be extended to location processes as well. In this way, we will guarantee that for every location name there is at most one location process. We refer the reader to [AP94] for an alternative presentation in which the information about the status of the locations is maintained in a context.

Formally, we define the following syntactic categories. The languages for sorts and processes, include the respective languages defined for the π_1-calculus.

$$\left\{\begin{array}{lll} \text{sort} & s ::= & \mathtt{loc} \mid Ch(s_1,\ldots,s_n) \quad (n \geq 0) \\ \text{process} & p ::= & a(\vec{b}).p \mid p \mid p \mid \mathbf{0} \parallel \nu a\, p \mid (\mathtt{rec}A(\tilde{a}).p)(\tilde{b}) \parallel A(\tilde{a}) \parallel \\ & & [a=b]p,q \parallel m \mid l \\ \text{configuration} & r ::= & \{p\}a \mid m \mid l \mid r \mid r \parallel \mathbf{0} \mid \nu a\, r \\ \text{message} & m ::= & \overline{a}\vec{b} \parallel \mathtt{stop}(a) \mid \mathtt{spawn}(a,p) \parallel \mathtt{ping}(a,b_1,b_2) \\ \text{location process} & l ::= & Loc_T(a) \quad T \in \{R,S\} \end{array}\right. \tag{7}$$

Reduction rules Next we define a few reduction rules which specify the possible interactions between the components of the solution. It is particularly appealing that all the rules share the same pattern: reduction happens when a message (possibly decorated with the name of its location) meets its destination.

$$\left\{\begin{array}{lll} (\text{cm}) & \overline{a}\vec{c} \mid \{a(\vec{b}).p\}a' & \rightarrow \{[\vec{c}/\vec{b}]p\}a' \\ (\mathtt{stop}) & \mathtt{stop}(a) \mid Loc_R(a) & \rightarrow Loc_S(a) \\ (\text{route}) & \{m\}a \mid Loc_R(a) & \rightarrow m \mid Loc_R(a) \\ (\mathtt{spawn}) & \mathtt{spawn}(a,p) \mid Loc_R(a) & \rightarrow \{p\}a \mid Loc_R(a) \\ (\mathtt{ping_t}) & \mathtt{ping}(a,b_1,b_2) \mid Loc_R(a) & \rightarrow \overline{b}_1 \mid Loc_R(a) \\ (\mathtt{ping_f}) & \mathtt{ping}(a,b_1,b_2) \mid Loc_S(a) & \rightarrow \overline{b}_2 \mid Loc_S(a) \end{array}\right. \tag{8}$$

We describe the operational intuition behind these rules:

(cm) Processes are decorated with the location where they run. In the absence of failures, this decoration is *transparent* (cf. proposition 6), in particular to send a message to a process, we do not need to know its location. Later, we will add a few structural equivalences (equations (10)) to ease the manipulation of the decorations.

(`stop`) When a running location process $Loc_R(a)$ meets a stop message $\texttt{stop}(a)$ it becomes a stopped location process $Loc_S(a)$, and stays in that state for ever (halting failure). One should note the dual use of the stop command: it can be employed either to program the halt of a location, or to model the potential failure of a location.

(**route**) Once a location has stopped, all processes running at that location should be *virtually* stopped for an external observer. We model this requirement, by blocking the routing of the messages at location a: *a process that cannot route its messages is as good as a stopped one.* On the other hand, a process running at a failed location keeps receiving messages as stated by rule (cm). Since communication is asynchronous and messages are addressed to a unique process, we can never observe this receiving activity. Of course, it would be possible to actually stop all processes running at a failed location, as it is done in [AP94], however in a model based on asynchronous communication, this is a needless complication.

(`spawn`) One should wonder if this extension is really necessary. Indeed, one alternative would be to stick to the π-calculus tradition of transmitting names only. In this case, we could imagine that each location is equipped with a sort of interpreter (a "universal π-calculus machine") which by some protocol receives a description of the process to run (as a sequence of channel names), and runs it locally. While this solution is theoretically possible, it would make the modeling of process mobility in distributed systems particularly heavy. It is a widespread belief that, in order to perform formal verification, the model has to abstract from inessential details. A model in which we have to take into account the details of the interpreter would probably defy formal treatment. The modeling solution which we adopt instead, is that of enriching the calculus with a $\texttt{spawn}(a, p)$ operator that allows to start the execution of the process p at the location a. Hence, in our model the transmission of processes is regarded as a primitive and atomic operation whose implementation is left unspecified. An important restriction on the transmission of processes, will be described next in the context of the typing rules.

(`ping`) The systems we model are fully asynchronous, a few non-trivial problems can be solved in this framework in the presence of failures. For instance, the algorithm for renaming in an asynchronous environment described in [ABND$^+$90]. On the other hand, there are problems, consensus being the most famous [FLP95], which cannot be solved in a fully asynchronous framework in the presence of failures. In order to cope with this limitation, the asynchronous model has been enriched in a number of ways including randomization, partial synchrony hypotheses, and failure detectors. We refer to [CT96, CHT96] for an up-to-date discussion of these issues. The approach we follow here, is to enrich our model with a failure detector $\texttt{ping}(\_)$ which eventually allows any process to know if a location runs or not. This solution can be integrated with little effort into our model. On the other hand, the handling of time or probabilities would require a major revision.

Variations on failures, and failures detectors Halting failure is probably the simplest form of failure considered in the literature. More complex failures include transient failures and byzantine failures (see [Lyn96, Tel95]). It is easy to adapt our model to represent transient failures: simply allow a process location to go from a stopped state to a run state. The representation of byzantine failures, requires a formalization of the notion of "arbitrary behaviour". This is a specification issue which we will not address in this paper.

Chandra and Toueg [CT96] have proposed a classification of the power of failure detectors. In their work, we find n asynchronous processes which interact by reliable point-to-point communication channels. At most $n-1$ processes are subject to an halting failure. Every process, maintains a local view of the failures that have occurred in the system. Let $F(t)$ be the collection of processes which have failed at time t, and $H(p)(t)$ be collection of processes which the process p suspects have failed at time t (t ranges over the natural numbers). Roughly, failure detectors are classified according to the convergence properties of the functions $H(p)$ to the function F.

We hint to a representation of these concepts in our model. We distribute n processes on n distinct locations, and we state that at most $n-1$ locations can fail (to say this operationally, we use the process in equation 11).

We suppose that every process maintains locally a list of processes suspected to have failed. This list represents the local view $H(p)(t)$. Initially, this list is empty, and it is periodically updated by using the `ping(_)` operation, which should be regarded as a way to query an oracle.

In our formalization, we have postulated the existence of an oracle which never gives misleading answers. We can then fulfill the following requirements:

(1) Every failed process is eventually suspected by every (correct) process.

(2) A correct process is never suspected by some process.

In Chandra and Toueg terminology, this is called a *perfect failure detector*. By the results in op. cit., there is an algorithm which solves the consensus problem using a perfect failure detectors and tolerates up to $n-1$ faults.

Formally, (1) and (2) are properties of the *runs* of the system (in our terminology, a run is a sequence of internal reductions). In this respect, it should be noted that our implementation of a perfect failure detector relies on a fairness hypothesis, otherwise one can build runs where the answer of the oracle is never received.

Chandra and Toueg consider weakenings of condition (2). Accordingly, one can define oracles whose answers are less and less reliable. For instance, consider the combination of condition (1) and condition ($2'$):

($2'$) Eventually, no correct process is suspected.

To weaken our model, we add a state "fuzzy run" $Loc_{FR}(a)$ and a state "fuzzy stop" $Loc_{FS}(a)$. These states behave as the states "run" and "stop", respectively, but for the fact that they give arbitrary answers to `ping(_)` messages. We also add internal transitions from fuzzy run to run, and from fuzzy stop to stop, so that, under a fairness hypothesis, answers will eventually become reliable.

$$\frac{\Gamma \vdash p \quad st(a) = \texttt{loc}}{\Gamma \vdash \{p\}a} \qquad \frac{\Gamma \cup \{a\} \vdash p \quad a \notin \Gamma \quad st(a) = \texttt{loc}}{\Gamma \vdash \nu a\, r}$$

$$\frac{st(a) = \texttt{loc}}{\{a\} \vdash Loc_T(a)} \qquad \frac{st(a) = \texttt{loc}}{\emptyset \vdash \texttt{stop}(a)}$$

$$\frac{st(a) = \texttt{loc} \quad \emptyset \vdash p}{\emptyset \vdash \texttt{spawn}(a, p)} \qquad \frac{st(a) = \texttt{loc} \quad st(b_1) = st(b_2) = Ch()}{\emptyset \vdash \texttt{ping}(a, b_1, b_2)}$$

Figure 6: Additional typing rules for the π_{1l}-calculus

To summarize, there is a space of models of failure and failure detection, which can be formalized and studied *within* our framework. The formalizations differ in the definition of the location process, and may rely on a fairness hypothesis. In this paper, we concentrate on the model which enjoys the simplest formalization.

Typing rules The typing rules for processes and configurations are obtained by adding the rules in figure 6 to those in figure 2. We allow the creation and transmission of new location names. As for channels, whenever we create a new location name a, we have to associate with it a location process ($Loc_T(a)$). We omit the rules for typing the parallel composition or restriction of configurations. These rules are shaped after the corresponding rules for processes.

The main point to note is the restriction on the rule for $\texttt{spawn}(\_)$: the spawned process is typed in the empty context. In this way, we make sure that by spawning we are not moving a process which can receive on some name, from a location to another. If we would allow this, we would break the property that *each channel name can be seen as an absolute physical address which does not change during the computation.*

Upon relaxing this hypothesis, one has to address two problems: at the implementation level one has to develop routing algorithms which adapt to changes in the network topology in the presence of failures, at the specification level one has to find an abstract description of the properties guaranteed by the routing algorithm. To the author's knowledge, there is no satisfying analysis of these issues. An attempt at defining a programming language where processes can migrate while keeping their identity has been recently proposed in [FGL+96], however that paper does not analyse the implementation level.

Labelled transition system The reduction rules 8, can be rephrased as labelled transitions, by including "location signals" among the actions:

$$\alpha ::= \tau \parallel a\vec{b} \mid \nu\{\vec{c}\}\,\overline{a}\vec{b} \mid a_T \mid \overline{a}_T \quad T \in \{R, S\} \tag{9}$$

Labelled transitions are defined on configurations and they are displayed in figure 7. Besides renaming of bound names we assume the following structural equivalences:

$$\begin{array}{ll} \{p \mid q\}a \equiv \{p\}a \mid \{q\}a & \{\nu b\, p\}a \equiv \nu b\, \{p\}a \ (a \neq b) \\ \{Loc_T(a')\}a \equiv Loc_T(a') & \end{array} \tag{10}$$

The rules specified for the π_1-calculus are trivially extended, moreover we add the labelled transitions for the location processes and the new messages.

Remark 1 *It should be noted that a blind application of labelled transitions can bring a configuration in a "meaningless" configuration. For instance, a transition* $\cdot \stackrel{\overline{a}_S}{\to} \cdot \stackrel{\overline{a}_S}{\to} \cdot$ *can never arise in our model since we can stop a location at most once. Another example in this vein is the transition* $\cdot \stackrel{\overline{a}_S}{\to} \cdot \stackrel{a_R}{\to} \cdot$*, where we stop a location, and then the location says it is running.*

Proposition 3 (subject reduction) *If* $\Gamma \vdash r$ *and* $r \stackrel{\alpha}{\to} r'$ *then* $\Gamma \cup bn(\alpha) \vdash r'$.

Proof. By adapting the proof of theorem 1.

Representing a migrating stack Having completed the formalization of our model, we illustrate its expressive power by a few examples. We start by programming a "migrating stack", that is a stack with standard operations $push, pop, empty$ which is enriched with a *move* operation allowing its migration from a location to another. This should suggest that the typing restriction on `spawn`$(\_)$ is not too severe.

First, we describe an implementation of a stack over an unspecified domain of values D, next, we enrich this implementation with a *move* operation. The process $Stack(a)$ supports operations $op \in \{empty, pop, push(d)\}$. $Stack(a)$ receives on a the request for an operation whose result is returned on k. It then forwards the request to $Cell(c; c', d)$ which is actually the first element of the stack. $Cell(c; c', d)$ contains a data d (which is nil if it is the last element of the stack) and a pointer c' to the next $Cell$. The cell returns the result r of the operation along the channel b. The stack updates its pointer to the internal representation of the stack and returns the result r_2 on the channel k. In the following, we are ignoring sorting issues in order to make the program shorter. For instance, the result r_2 might be either a boolean, or a data, or nil, or an acknowledgement signal. The program could be correctly sorted by doing some case analysis.

$$St(a, b; c) = a(op, k).(\overline{c}\, op, b \mid b(r_1, r_2).(\overline{k} r_2 \mid St(a, b; r_1)))$$

$$\begin{array}{ll} Cell(c; c', d) = c(op, b). & \\ \ [op = empty] & \texttt{if } d = nil \texttt{ then } (\overline{b}c, \texttt{t} \mid Cell(c; c', d)) \texttt{ else } (\overline{b}c, \texttt{f} \mid Cell(c; c', d)) \\ \ [op = pop] & \texttt{if } d = nil \texttt{ then } (\overline{b}c, nil \mid Cell(c; c', d)) \texttt{ else } (\overline{b}c', d \mid Idle(c)) \\ \ [op = push(d')] & \nu c''\, (Cell(c''; c, d') \mid \overline{b}c'', \_ \mid Cell(c; c', d)) \end{array}$$

To create a stack $Stack(a)$, which can be accessed through the name a, we run the process $Create(a)$ which is defined as follows:

$$Create(a) \quad = \nu b\, \nu c\, (St(a, b; c) \mid Cell(c; \_, nil))$$

$$(in)\quad \frac{\cdot}{\{a(\vec{b}).p\}d \xrightarrow{a\vec{c}} \{[\vec{c}/\vec{b}]p\}d} \qquad (out)\quad \frac{\cdot}{\overline{a}\vec{b} \xrightarrow{\overline{a}\vec{b}} \mathbf{0}}$$

$$(out_{ex})\quad \frac{r \xrightarrow{\nu\{\vec{c}\}\,\overline{a}\vec{b}} r' \quad a \neq d \quad d \in \{\vec{b}\}\backslash\{\vec{c}\}}{\nu d\, r \xrightarrow{\nu\{d,\vec{c}\}\,\overline{a}\vec{b}} r'} \qquad (\nu)\quad \frac{r \xrightarrow{\alpha} r' \quad a \notin n(\alpha)}{\nu a\, r \xrightarrow{\alpha} \nu a\, r'}$$

$$(cp)\quad \frac{r \xrightarrow{\alpha} r_1 \quad bn(\alpha) \cap fn(r') = \emptyset}{r \mid r' \xrightarrow{\alpha} r_1 \mid r'} \qquad (cm)\quad \frac{r \xrightarrow{\nu\{\vec{c}\}\,\overline{a}\vec{b}} r_1 \quad r' \xrightarrow{a\vec{b}} r_1' \quad \{\vec{c}\} \cap fn(r') = \emptyset}{r \mid r' \xrightarrow{\tau} \nu\vec{c}(r_1 \mid r_1')}$$

$$(\mathtt{rec})\quad \frac{\{[\mathtt{rec}A(\tilde{a}).p/A, \tilde{b}/\tilde{a}]p\}d \xrightarrow{\alpha} r}{\{(\mathtt{rec}A(\tilde{a}).p)(\tilde{b})\}d \xrightarrow{\alpha} r} \qquad (cg)\quad \frac{r \equiv r' \quad r' \xrightarrow{\alpha} r_1' \quad r_1' \equiv r_1}{r \xrightarrow{\alpha} r_1}$$

$$(m_{\mathtt{t}})\quad \frac{\{p\}d \xrightarrow{\alpha} r}{\{[a=a]p,q\}d \xrightarrow{\alpha} r} \qquad (m_{\mathtt{f}})\quad \frac{\{q\}d \xrightarrow{\alpha} r \quad a \neq b}{\{[a=b]p,q\}d \xrightarrow{\alpha} r}$$

$$(Loc_{RR})\quad \frac{\cdot}{Loc_R(a) \xrightarrow{a_R} Loc_R(a)} \qquad (Loc_{RS})\quad \frac{\cdot}{Loc_R(a) \xrightarrow{a_S} Loc_S(a)}$$

$$(Loc_{SS})\quad \frac{\cdot}{Loc_S(a) \xrightarrow{a_S} Loc_S(a)} \qquad (\mathtt{stop})\quad \frac{\cdot}{\mathtt{stop}(a) \xrightarrow{\overline{a}_S} \mathbf{0}}$$

$$(\mathtt{spawn})\quad \frac{\cdot}{\mathtt{spawn}(a,p) \xrightarrow{\overline{a}_R} \{p\}a} \qquad (route)\quad \frac{\cdot}{\{m\}a \xrightarrow{\overline{a}_R} m}$$

$$(\mathtt{ping}_{\mathtt{t}})\quad \frac{\cdot}{\mathtt{ping}(a,b_1,b_2) \xrightarrow{\overline{a}_R} \overline{b}_1} \qquad (\mathtt{ping}_{\mathtt{f}})\quad \frac{\cdot}{\mathtt{ping}(a,b_1,b_2) \xrightarrow{\overline{a}_S} \overline{b}_2}$$

$$(cm_T)\quad \frac{r \xrightarrow{\overline{a}_T} r' \quad r_1 \xrightarrow{a_T} r_1'}{r \mid r_1 \xrightarrow{\tau} r' \mid r_1'}$$

Figure 7: Lts for the π_{1l}-calculus

To create a stack at a remote location n, we can use the process $rCreate$. Upon creation, we receive on the name k a name to access the stack.

$$rCreate(;n,k) \quad = \mathtt{spawn}(n, \nu a\,(\overline{k}a \mid Create(a)))$$

Now to migrate $Stack(a)$ to location n, it is enough to create a new stack at location n, say $Stack(a')$, and transfer the contents of $Stack(a)$ to $Stack(a')$. This is a simple loop that *pops* elements from $Stack(a)$ and *pushes* them in $Stack(a')$. In order to preserve the order of the elements in the stack we can use an intermediate stack.

The name a' is returned to the process which has requested the *move* operation only when the transfer is completed, hence the *move* operation can be considered as an *atomic* operation. Formally, the migrating stack is obtained by adding an extra branch to the definition of $St(a)$ entailing the execution of the procedure sketched above. Following requests to $Stack(a)$ can be handled in a variety of ways, for instance: (1) $Stack(a)$ becomes an idle process, (2) $Stack(a)$ returns an exception, or (3) $Stack(a)$ forwards requests to $Stack(a')$.

Representing failures There is a twist in the representation of failures. If we introduce the message $\mathtt{stop}(d)$ in the system description, then at any point in the computation the system can reach a configuration where the location d is stopped. Moreover, under a fairness constraint, it will eventually reach this configuration. If we want to leave open the possibility that a location may also run for ever, then it is more appropriate to introduce the process $\mathtt{maystop}(d) = \mathtt{stop}(d) \oplus \mathbf{0}$. The following example, will highlight the difference between $\mathtt{stop}(d)$ and $\mathtt{maystop}(d)$ in a concrete case.

More generally, we can represent the fact that at most m locations out of n locations $d_1, \ldots, d_n$ may stop ($m \leq n$) by considering the following process:

$$\nu c, c_1, \ldots, c_n\,(\Pi_{i=1..n}(\overline{c}c_i \mid c_i : \mathtt{maystop}(d_i)) \mid c(a_1) \ldots c(a_m) : \Pi_{i=1..m}\overline{a_i}) \qquad (11)$$

Our approach should be contrasted with the one taken (in a CCS context) by Janowsky [Jan95]. He defines the notion of bisimulation in such a way that an equivalence which is shown to hold for a number of faults n, will also hold for m faults, $0 \leq m \leq n$. While this may save some work at the verification level, it requires the introduction of a notion of bisimulation *ad hoc*.

Representing a system resilient to failures We give an example of a system in which the `ping` operator is used to monitor two resources which may fail. More precisely, the system is composed by a user U which relies on two resources R_1 and R_2 to emit an observable signal on b. A fourth process M, monitors the activity of R_1 and R_2, so that when the resource R_i ($i = 1, 2$) fails it is replaced by a new one. Formally, the system is described in the π_{1l}-calculus by the system of equations in figure 8. The specification is expressed by the requirement that $Sys(;b)$ is weakly bisimilar to $Spec(;b)$ (formally, $Sys(;b)$ and $Spec(;b)$ do not belong to the language of configurations, they are just abbreviations for the left-hand-side of the equation).

$$
\begin{array}{ll}
R_1(;a_1) & = \nu c\,(\overline{a_1}c \mid c : R_1(;a_1)) \\
R_2(;a_2) & = \nu c\,(\overline{a_2}c \mid c : R_2(;a_2)) \\
 & \\
U(a_1,a_2;b) & = a_1(c).(\overline{c} \mid a_2(c).(\overline{c} \mid \overline{b} \mid U(a_1,a_2;b))) \\
 & \\
M_1(;a_1,a_2,d_1,d_2) & = \nu e_1\,\nu e_2\,(\mathtt{ping}(d_1,e_1,e_2) \mid \\
 & \quad e_1 : M_2(;a_1,a_2,d_1,d_2) \mid \\
 & \quad e_2 : \nu d_1\,(\mathtt{spawn}(R_1(;a_1),d_1) \mid Loc_R(d_1) \mid \mathtt{maystop}(d_1) \mid M_2(;a_1,a_2,d_1,d_2))) \\
 & \\
M_2(;a_1,a_2,d_1,d_2) & = \nu e_1\,\nu e_2\,(\mathtt{ping}(d_2,e_1,e_2) \mid \\
 & \quad e_1 : M_1(;a_1,a_2,d_1,d_2) \mid \\
 & \quad e_2 : \nu d_2\,(\mathtt{spawn}(R_2(;a_1),d_2) \mid Loc_R(d_2) \mid \mathtt{maystop}(d_2) \mid M_1(;a_1,a_2,d_1,d_2))) \\
 & \\
Sys(;b) & = \nu a_1,a_2,d_1,d_2,d,d' \\
 & \quad (\{U(a_1,a_2;b)\}d \mid Loc_R(d) \mid \{M_1(;a_1,a_2,d_1,d_2)\}d' \mid Loc_R(d') \mid \\
 & \quad \{R_1(;a_1)\}d_1 \mid Loc_R(d_1) \mid \mathtt{maystop}(d_1) \\
 & \quad \{R_2(;a_2)\}d_2 \mid Loc_R(d_2) \mid \mathtt{maystop}(d_2)\ ) \\
 & \\
S(;b) & = \overline{b} \mid S(;b) \\
Spec(;b) & = \nu d\,(\{S(;b)\}d \mid Loc_R(d))
\end{array}
$$

Figure 8: Example of system resilient to failures

An automatic verification, can be carried on with a standard tool such as the Concurrency Workbench modulo a suitable representation of $Sys(;b)$ as a *finite control* CCS process. To this end, we model failure by introducing a *switch*. A resource can internally choose to switch off (this mimicks failure) and the monitor can switch it on again (this mimicks the creation of a new resource).

This example illustrates the difference between `stop(_)` and `maystop(_)`. Suppose that we program the monitor in such a way that it waits for the failure of, say, the resource R_1 before checking the failure of the resource R_2. Then, if we use `maystop(_)` to model failure, the user U is stuck if R_1 never fails and R_2 fails. On the other hand, if we model failure with `stop(_)`, we are assured that the monitor will take appropriate action to allow U to progress.

4 Tools to reason about equivalence

There is a simple translation $\lceil\_\rceil$ from the π_{1l}-calculus to the π_1-calculus. We are interested in this translation as a way of reducing verification problems for the π_{1l}-calculus to verification problems for the π_1-calculus (cf. [AP94]). The translation (bi-)simulates the π_{1l}-calculus in the π_1-calculus. A fortiori it has nothing to do with the way a program of the π_{1l}-calculus would actually be executed. Every name a of sort $st(a)$, is translated into the same name with sort $\lceil st(a)\rceil$, where:

$$\lceil Ch(s_1,\ldots,s_n)\rceil = Ch(\lceil s_1\rceil,\ldots,\lceil s_1\rceil) \quad \lceil \mathtt{loc}\rceil = Ch(\mathtt{bool},\mathtt{bool},Ch(),Ch())$$

The translation of configurations is displayed in figure 9, where we use a `case` statement (which can be easily coded with a nesting of `if_then_else_`'s) to make the control of the location process clearer. The translation of configurations relies on an auxiliary translation of processes which is parametric in a location name. This name represents the location where the process is running.

Definition 5 (complete configuration) *Let $\Gamma \vdash r$ be a well typed configuration. We say that the configuration r is* complete *if $r \stackrel{\tau}{\Rightarrow} r'$ and $a \notin \Gamma$ implies that r' cannot perform a transition with label $\overline{a}_T$.*

Intuitively, in a complete configuration all locations mentioned in the configuration have been defined and therefore transitions labelled with $\overline{a}_T$ are not visible. Let $\Gamma \vdash r$ be a complete configuration. This property is preserved by internal reduction, hence we can introduce a relation of barbed bisimulation on the π_{1l}-calculus, commitment being defined as follows. Let $\Gamma \vdash r$ be a complete configuration, then $r \downarrow \overline{a}$ if $a \notin \Gamma$, and $r \stackrel{\nu\{\vec{c}\}\overline{a}\vec{b}}{\rightarrow} \cdot$.

Definition 6 *A symmetric relation S on well-typed, complete configurations is a strong barbed bisimulation if whenever rSr' the following holds:*

(1) *If $r \downarrow \overline{a}$ then $r' \downarrow \overline{a}$.*

To spawn or to route	$\mathtt{t},\mathtt{t}$
To ping	$\mathtt{t},\mathtt{f}$
To stop	$\mathtt{f},\mathtt{f}$

Codes for the signals to the location process

$$
\begin{array}{ll}
\lceil m \rceil d & = \nu c\,(\overline{d}\,\mathtt{t},\mathtt{t},c,c \mid c : \lceil m \rceil) \\
\lceil \overline{a}\vec{b} \rceil & = \overline{a}\vec{b} \\
\lceil \mathtt{spawn}(d,p) \rceil & = \nu c\,(\overline{d}\,\mathtt{t},\mathtt{t},c,c \mid c : \lceil p \rceil d) \\
\lceil \mathtt{ping}(d,c,c') \rceil & = \overline{d}\,\mathtt{t},\mathtt{f},c,c' \\
\lceil \mathtt{stop}(d) \rceil & = \overline{d}\,\mathtt{f},\mathtt{f},\_,\_ \\
 & \\
\lceil Loc_T(a) \rceil & = L_T(a;) \quad \text{where} \\
L_R(a;) & = a(b,b',c,c').\ \mathtt{case}\ (b,b',c,c')\ \mathtt{of} \\
 & \quad (\mathtt{t},\mathtt{t},c,c) : \overline{c} \mid L_R(a;) \\
 & \quad (\mathtt{t},\mathtt{f},c,c') : \overline{c} \mid L_R(a;) \\
 & \quad (\mathtt{f},\mathtt{f},\_,\_) : L_S(a;) \\
 & \\
L_S(a;) & = a(b,b',c,c').\ \mathtt{case}\ (b,b',c,c')\ \mathtt{of} \\
 & \quad (\mathtt{t},\mathtt{f},c,c') : \overline{c'} \mid L_S(a;) \\
 & \quad (\_,\_,\_,\_) : \overline{a}b,b',c,c' \mid L_S(a;)
\end{array}
$$

$$
\begin{array}{llll}
\lceil \nu a\, r \rceil & = \nu a \lceil r \rceil & \lceil \nu a\, p \rceil c & = \nu a \lceil p \rceil c \quad a \neq c \\
\lceil p \mid q \rceil c & = \lceil p \rceil c \mid \lceil q \rceil c & \lceil r \mid r' \rceil & = \lceil r \rceil \mid \lceil r' \rceil \\
\lceil \mathbf{0} \rceil & = \mathbf{0} & \lceil \mathbf{0} \rceil c & = \mathbf{0} \\
 & & & \\
\lceil a(\vec{b}).p \rceil c & = a(\vec{b}).\lceil p \rceil c \quad c \notin \{\vec{b}\} & \lceil (\mathtt{rec}A(\tilde{a}).p)(\tilde{b}) \rceil c & = (\mathtt{rec}A(\tilde{a}).\lceil p \rceil c)(\tilde{b}) \quad c \notin \{\tilde{a}\} \\
\lceil [a=b]p,q \rceil c & = [a=b]\lceil p \rceil c, \lceil q \rceil c & \lceil \{p\}c \rceil & = \lceil p \rceil c
\end{array}
$$

Figure 9: Translating the π_{1l}-calculus into the π_1-calculus

(2) *If* $r \xrightarrow{\tau} r_1$ *then* $r' \xrightarrow{\tau} r'_1$ *and* $r_1 S r'_1$.

Let $\overset{\bullet}{\sim}_l$ *be the largest barbed bisimulation. The notion of weak barbed simulation is obtained by replacing everywhere the commitment* $\downarrow$ *with* $\Downarrow$ *and the reduction* $\xrightarrow{\tau}$ *with* $\overset{\tau}{\Rightarrow}$. *We denote with* $\overset{\bullet}{\approx}_l$ *the largest weak barbed bisimulation.*

Theorem 2 (adequacy) *Let* r, r' *be complete well-typed configurations. Then:*

$$r \overset{\bullet}{\approx}_l r' \quad \textit{iff} \quad \lceil r \rceil \overset{\bullet}{\approx} \lceil r' \rceil$$

Proof of theorem 2 In the following we work up to the structural congruence which is generated by the associative and commutative laws for parallel composition, the identity law of $\mathbf{0}$ w.r.t. parallel composition, the laws for the commutation of restriction with restriction and parallel composition, the law for the unfolding of recursive definition, and the following equations:

$$\begin{array}{ll} \nu\vec{a}\, Idle(\vec{a}) & = \mathbf{0} \\ \texttt{if t then } p \texttt{ else } q & = p \\ \texttt{if f then } p \texttt{ else } q & = q \end{array}$$

Lemma 3 *Let* r *be a complete well-typed configuration. Then:*

(1) $r \downarrow \overline{a}$ *iff* $\lceil r \rceil \downarrow \overline{a}$.

(2) *If* $r \xrightarrow{\tau} r'$ *then* $\lceil r \rceil \xrightarrow{\tau}_{\leq 2} \lceil r' \rceil$.

Proof hint. (1) We analyse the transitions $r \overset{\nu\{\vec{c}\}\overline{a}\vec{b}}{\longrightarrow} \cdot$, and $\lceil r \rceil \overset{\nu\{\vec{c}\}\overline{a}\vec{b}}{\longrightarrow} \cdot$. Commitments on (the translation of) location names are impossible by the hypothesis that r is complete.

(2) The reductions (cm), (`stop`), and (`ping`) are simulated in one step. The reductions ($route$) and (`spawn`) are simulated in two steps. The second step is a reduction of the shape:

$$\nu b\, (b : p \mid \overline{b}) \xrightarrow{\tau} p \quad b \notin fn(p) \tag{12}$$

QED

We call the reductions of type 12 *administrative*. These reductions are normalizing and confluent. Roughly, reductions and commitments in the π_{1l}-calculus and in the π_1-calculus are in one-to-one correspondence modulo administrative reductions. As mentioned in lemma 3, the simulating term may need one extra administrative reduction in order to conform to the shape of the translation of the reduced term in the source calculus. Toward the formalisation of this idea, we define a set $Pr_l = \{p \mid \exists r \text{ complete } (\lceil r \rceil \overset{\tau}{\Rightarrow} p)\}$.

On the processes in Pr_l, we can determine the administrative reductions, for instance by a suitable annotation of the restrictions. We write $p \xrightarrow{\tau}_{ad} p'$ if $p \xrightarrow{\tau} p'$ and the reduction is administrative. We also use $\overset{\tau}{\Rightarrow}_{ad}$ to indicate zero or more administrative reductions.

We define a binary relation $\mathcal{R}$ between complete configurations and processes in Pr_l as follows:

$$r \mathcal{R} p \quad \text{iff} \quad p \overset{\tau}{\Rightarrow}_{ad} \lceil r \rceil \tag{13}$$

We note that it is not possible to perform an administrative reduction starting from $\lceil r \rceil$, so $\lceil r \rceil$ plays the role of a normal form.

Lemma 4 *In the following let r be a complete configuration and $p \in Pr_l$. Then:*

(1) $r \mathcal{R} \lceil r \rceil$.

(2) *If $r \downarrow \overline{a}$ and $r \mathcal{R} p$ then $p \Downarrow \overline{a}$.*

(3) *If $p \downarrow \overline{a}$ and $r \mathcal{R} p$ then $r \downarrow \overline{a}$.*

(4) *If $r \mathcal{R} p$ and $r \xrightarrow{\tau} r'$ then $p \stackrel{\tau}{\Rightarrow}_{ad} \cdot \xrightarrow{\tau} \cdot \stackrel{\tau}{\Rightarrow}_{ad} \lceil r' \rceil$.*

(5) *If $r \mathcal{R} p$ and $p \xrightarrow{\tau}_{ad} p'$ then $r \mathcal{R} p'$.*

(6) *If $r \mathcal{R} p$, $p \xrightarrow{\tau} p'$, and we are not in the previous case, then $r \xrightarrow{\tau} r'$ and $p' \stackrel{\tau}{\Rightarrow}_{ad} \lceil r' \rceil$.*

We can now conclude the proof by showing as a direct application of lemma 4, that the relation $\mathcal{R} \circ \stackrel{\bullet}{\approx} \circ \mathcal{R}^{-1}$ is a barbed bisimulation for the π_{1l}-calculus and the relation $\mathcal{R}^{-1} \circ \stackrel{\bullet}{\approx}_l \circ \mathcal{R}$ is a barbed bisimulation for the π_1-calculus. QED

Bisimulation for the π_1-calculus We undertake a deeper study of equivalence for the π_1-calculus. It is well known that barbed bisimulation fails to be a congruence, in particular it is not preserved by parallel composition. We can refine barbed bisimulation by asking preservation under certain contexts. In particular, we require preservation under parallel composition and call the resulting equivalence barbed equivalence.

Definition 7 (barbed equivalence) *We define a relation $\sim_b$ of barbed equivalence between well typed processes as follows: $p \sim_b p'$ iff for each q, such that $p \mid q$ and $p' \mid q$ are well-typed, $p \mid q \stackrel{\bullet}{\sim} p' \mid q$ holds. The notion of weak barbed equivalence $\approx_b$ is obtained by replacing $\stackrel{\bullet}{\sim}$ with $\stackrel{\bullet}{\approx}$.*

In the following, whenever we compose two processes we implicitly suppose that their composition is well-typed. We also note that if $p \approx_b p'$, then there is a context Γ which types both processes. Suppose $\Gamma \vdash p$, $\Gamma' \vdash p'$ and $a \in \Gamma \backslash \Gamma'$, then $p \mid \overline{a}$ cannot be barbed bisimilar to $p' \mid \overline{a}$ as the second commits on $\overline{a}$ while the first does not. For instance, it can be shown that $Idle(a)$ is barbed equivalent to $a(\vec{b}) \triangleright \overline{a}\vec{b}$ but it is *not* barbed equivalent to $\mathbf{0}$.

In this section, we show that barbed equivalence can be characterised by a suitable (asynchronous) bisimulation over the labelled transition system. This supports the view that the π_1-calculus is not only an *expressive* calculus, but it has also a "tractable theory" of equivalence (at least in the sense the π-calculus has one!). For the sake of simplicity we will work with the monadic unsorted π_1-calculus (cf section 2). Following standard notation [MPW92], we write the action $\nu\{b\}\,\overline{a}b$ as $\overline{a}(b)$.

In defining the commitment relation, we have been careful to observe only those output commitments which relate to free channels whose receiver is not defined in the observed process. Following this idea, we introduce a restricted form of labelled transition. Let

the function cmt be defined on actions as follows: $cmt(\tau) = cmt(ab) = \emptyset$ and $cmt(\overline{a}b) = cmt(\overline{a}(b)) = \{a\}$. The rule (cp) in the lts described in figure 1, is then replaced by:

$$(cp_{tp}) \quad \frac{p \xrightarrow{\alpha} p' \quad bn(\alpha) \cap fn(q) = \emptyset \quad \Gamma \vdash q \quad cmt(\alpha) \cap \Gamma = \emptyset}{p \mid q \xrightarrow{\alpha} p' \mid q} \tag{14}$$

Whenever we speak of transitions of typed processes, we will apply the rule (cp_{tp}). We can now define a notion of (asynchronous) bisimulation over the restricted lts. The following definition follows quite closely [ACS96] modulo some type constraints.

Definition 8 (bisimulation) *A symmetric relation S on typed processes is a bisimulation if $p\,S\,q$ implies:*

(1) *There is a context Γ such that $\Gamma \vdash p$ and $\Gamma \vdash q$.*

(2) *If $p \xrightarrow{\alpha} p'$, $bn(\alpha) \cap fn(q) = \emptyset$, and α is not an input action, then $q \xrightarrow{\alpha} q'$ and $p'\,S\,q'$.*

(3) *If $p \xrightarrow{ab} p'$ then either $q \xrightarrow{ab} q'$ and $p'\,S\,q'$, or $q \xrightarrow{\tau} q'$ and $p'\,S\,(q' \mid \overline{a}b)$.*

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

It is shown in [ACS96] that weak asynchronous bisimulation is preserved by all operators of the asynchronous π-calculus but matching. In particular, the fact that asynchronous bisimulation preserves parallel composition, suffices to show that asynchronous bisimulation implies barbed equivalence. This is stated as follows (in the weak case).

Proposition 4 *If $p \approx_a p'$ then (1) for each q, $p \mid q \approx_a p' \mid q$, and (2) $p \approx_b p'$.*

In the other direction, we obtain the following result which relies on a proof technique introduced in [ACS96].

Definition 9 *Let us fix a decidable structural equivalence relation. A lts is* image finite *(w.r.t. weak transitions), if for any process p and action α the set $\{p' \mid p \stackrel{\alpha}{\Rightarrow} p'\}$ is finite up to the structural equivalence relation. We say that a process p is image finite if the lts formed of the processes reachable from p by labelled transitions is image finite.*

Image finite processes include "finite control" processes and therefore represent an interesting class. In the case of strong transitions, all processes of the π_1-calculus turn out to be image finite.

Theorem 3 (characterisation) *(1) If $p \sim_b q$ then $p \sim_a q$. (2) If p, q are image finite and $p \approx_b q$, then $p \approx_a q$.*

Proof. Let $\mathcal{F}$ be the monotone operator over $\mathcal{P}(Pr \times Pr)$ associated to the definition of asynchronous bisimulation. Suppose $\approx_a^0 = Pr \times Pr$, $\approx_a^{k+1} = \mathcal{F}(\approx_a^k)$, and $\approx_a^\omega = \bigcap_{k<\omega} \approx_a^k$. It is well-known that on an image finite lts the operator $\mathcal{F}$ preserves co-directed sets. In

particular, $\mathcal{F}(\approx_a^\omega) = \approx_a^\omega$. It follows that on image finite processes $\approx_a = \approx_a^\omega$. We show that $p \approx_b q$ implies $p \approx_a^\omega q$. From the previous remark the theorem follows.

- We fix some notation. Let L_i, L_o, L_i', L_o' denote finite disjoint sets of names. We use L, L' as an abbreviation for $L_i \cup L_o$, $L_i' \cup L_o'$, respectively. If $L' = \{a_1, \ldots a_p\}$ then $\nu L'\, p \equiv \nu a_1 \ldots \nu a_p\, p$.

- We define a collection of tests $R(n, L) \equiv_{abr} R(n, L_i, L_o)$ depending on $n \in \omega$ and L finite set of channel names, and such that $L_o \vdash R(n, L_i, L_o)$. Intuitively, $R(n, L_i, L_o)$ tests a process p that may receive on L_i, that is $L_i \vdash p$, and may send on L_o, that is $L_o = fn(p) \backslash L_i$. We show by induction on n that:

$$\begin{array}{l}\exists L, L'\,(L \supseteq fn(p \mid q), L' \subseteq L \text{ and } \nu L'\,(p \mid R(n, L)) \stackrel{\bullet}{\approx} \nu L'\,(q \mid R(n, L))))\\ \text{implies } p \approx_a^n q.\end{array}$$

- If the property above holds then we can conclude the proof by observing:

$$\begin{array}{ll} p \approx_b q & \Rightarrow \forall r\,(p \mid r \stackrel{\bullet}{\approx} q \mid r)\\ & \Rightarrow \forall n \in \omega\,(p \mid R(n, L) \stackrel{\bullet}{\approx} q \mid R(n, L)) \quad \text{with } L = fn(p \mid q),\ L' = \emptyset\\ & \Rightarrow \forall n \in \omega\,(p \approx_a^n q)\\ & \Rightarrow p \approx_a^\omega q\end{array}$$

- We define the tests $R(n, L)$. If $X = \{p_1, \ldots, p_n\}$ is a set of processes, then $\oplus X$ is an abbreviation for $p_1 \oplus \cdots \oplus p_n$. We suppose that the collection of channel names Ch has been partitioned in two infinite well-ordered set Ch' and Ch''. In the following we have $L' \subseteq L \subset_{finite} Ch''$. We also assume the following sequences of distinct names in Ch':

$$\begin{array}{l}\{b_n, b_n' \mid n \in \omega\}\\ \{c_n^\beta \mid n \in \omega \text{ and } \beta \in \{\tau, aa', a, \overline{a}a', \overline{a} \mid a, a' \in Ch''\}\}\\ \{c_n'^\beta \mid n \in \omega \text{ and } \beta \in \{aa', a \mid a, a' \in Ch''\}\}\\ \{d_n^\beta \mid n \in \omega \text{ and } \beta \in \{a \mid a \in Ch''\}\}\\ \{e_n \mid n \in \omega\}\end{array}$$

The test $R(n, L)$ is defined by induction on n as follows, where we pick a'' to be the first name in the well-ordered set $Ch'' \backslash L$. When emitting or receiving a name which is not in L we work up to injective substitution to show that $P \approx_a^n Q$. In the following whenever we write, e.g., $\overline{b}_n \oplus \cdots$, we actually mean $(\overline{b}_n \mid Idle(L_o)) \oplus \cdots$ The processes $Idle(L_o)$ have to be added to have a correct typing.

$$R(0, L_i, L_o) = \overline{b}_0 \oplus \overline{b'}_0$$

$$\begin{array}{l} R(n, L_i, L_o) = \overline{b}_n \oplus \overline{b'}_n \quad \oplus \quad (\text{for } n > 0)\\ \quad (\overline{c}_n^{\tau} \oplus R(n-1, L_i, L_o)) \ \oplus\\ \quad \oplus \{\overline{c}_n^{\overline{a}a'} \oplus (\overline{a}a' \mid R(n-1, L_i, L_o)) \mid a \in L_i, a' \in L\} \ \oplus\\ \quad \oplus \{\overline{c}_n^{\overline{a}} \oplus \nu a''\,(\overline{a}a'' \mid R(n-1, L_i, L_o \cup \{a''\})) \mid a \in L_i\} \ \oplus\\ \quad \oplus \{\overline{c}_n^{aa'} \oplus a(a'').(\overline{c'}_n^{aa'} \oplus ([a'' = a']\overline{d}_n^{a'} \oplus R(n-1, L_i, L_o))) \mid a \in L_o, a' \in L\} \ \oplus\\ \quad \oplus \{\overline{c}_n^{a} \oplus a(a'').(\overline{c'}_n^{a} \oplus (\oplus\{[a'' = a']\overline{d}_n^{a'} \mid a' \in L\} \oplus \overline{e}_n \oplus R(n-1, L_i \cup \{a''\}, L_o)) \mid a \in L_o\}\end{array}$$

• We suppose $n > 0$, $\nu L'\,(p \mid R(n, L)) \stackrel{\bullet}{\approx} \nu L'\,(q \mid R(n, L))$, and $p \stackrel{\alpha}{\Rightarrow} p'$. We proceed by case analysis on the action α to show that q can match the action α (in the asynchronous sense). QED

Full abstraction and transparency We concentrate on a non-trivial set of configurations which is defined as follows.

Definition 10 *A* location closed *configuration is a configuration where transitions of the shape a_T or $\overline{a}_T$ are not observable, and such that this property is preserved by labelled transitions.*

Of course, location closed configurations are complete configurations. Many systems resilient to failures, including the one described in section 3, can be formalized within this fragment. On location closed configurations, the translation described in figure 9 turns out to be *fully abstract*. Intuitively, the translation of a location closed configuration can interact with the environment without revealing any information about the internal representation of locations.

To state our result, one has to adapt the definition 8 of bisimulation so that it relates location closed configurations to processes of the π_1-calculus. By a little abuse of notation, we still indicate with $\approx_a$ the related greatest weak bisimulation.

Proposition 5 (full abstraction) *Let r be a location closed configuration. Then $r \approx_a \lceil r \rceil$.*

PROOF HINT. Internal actions are related as in theorem 2. Input-output actions turn out to be in one-to-one correspondence. In establishing this correspondence, one has to take into account the sort translation. For instance, an input action $a\vec{b}$, where $st(a) = s$, is related to an input action $a\vec{b}$, where $st(a) = \lceil s \rceil$. QED

We conclude this section, with a formalization of the idea that in the absence of failure, the distribution of processes is *transparent*. Given a location closed configuration r, $er_l(r)$ is either (i) a process of the π_1-calculus where all the information on locations has been erased, or (ii) undefined if the configuration contains stopped locations, or `stop(_)` messages. The formal definition of the function $er_l(\_)$, on its domain of definition, is given in figure 4.

Proposition 6 (transparency) *Let r be a location closed configuration. If $er_l(r)$ is defined, then $r \approx_a er_l(r)$.*

PROOF HINT. Input-output transitions of r and $er_l(r)$ are in one-to-one correspondence. The internal transitions of r, for routing, spawning, and pinging (cf. rules 8), simply disappear in $er_l(r)$. QED

$er_l(\mathtt{loc})$	$= Ch()$	$er_l(Ch(s_1,\ldots,s_n))$	$= Ch(er_l(s_1),\ldots,er_l(s_n))$
$er_l(Loc_R(a))$	$= Idle(a)$	$er_l(\mathtt{spawn}(a,p))$	$= er_l(p)$
$er_l(\mathtt{ping}(a,b_1,b_2))$	$= \overline{b}_1$	$er_l(\overline{a}\vec{b})$	$= \overline{a}\vec{b}$
$er_l(\{p\}a)$	$= er_l(p)$	$er_l(\nu a\, r)$	$= \nu a\, er_l(r)$
$er_l(a(\vec{b}).p)$	$= a(\vec{b}).er_l(p)$	$er_l(p \mid p')$	$= er_l(p) \mid er_l(p')$
$er_l(\nu a\, p)$	$= \nu a\, er_l(p)$	$er_l(r \mid r')$	$= er_l(r) \mid er_l(r')$
$er_l(A(\tilde{a}))$	$= A(\tilde{a})$	$er_l((\mathtt{rec}A(\tilde{a}).p)(\tilde{b}))$	$= (\mathtt{rec}A(\tilde{a}).er_l(p))(\tilde{b})$
$er_l(\mathbf{0})$	$= \mathbf{0}$	$er_l([a=b]p,q)$	$= [a=b]er_l(p), er_l(q)$

Figure 10: Location erasure

5 Related work and achievements

In previous work [AP94], we have developed a formal framework which models the distributed module of the Facile programming language [TLP+93]. The Facile communication model is quite powerful as it includes guarded choice, synchronous communication, and multiple receivers. A synchronous communication (ignoring choice) may require a synchronization between processes distributed to three different locations: the location of the sender, the location of the receiver, and the location of the channel manager (which is a process which has to resolve concurrent requests for reading or writing on a channel). This complexity limits the manageability of the distributed model.

The work on the join calculus [FG96], suggested that a simplification of the communication primitive (asynchronous communication with a unique receiver) could considerably simplify reasoning about a system where failures can occur. Technically, the π_1-calculus can be regarded as a way to capture some basic features of the join calculus [FG96], e.g. unicity of the receptor, by imposing a type discipline rather than by modifying the π-calculus. One advantage of this approach, is that it is possible to reuse technical insights already developed for the (asynchronous) π-calculus [ACS96] such as labelled transition system, and proof techniques based on bisimulation.

Incidentally, the π_1-calculus can also be seen as a way to make the communication primitives of the π-calculus closer to those of *object-oriented* programming languages, where interaction arises when an object calls the method of another *uniquely determined* object. Indeed this computational paradigm was a main source of inspiration in the design of the typing system of the π_1-calculus. Unfortunately, the term *object* is overloaded with meaning, and for this reason we have replaced it with the more neutral *process*.

Besides Facile, other programming languages which address (some of) the issues of locations, failures, and process mobility include CML [Rep91], Erlang [AWWV96], Java [AG97], Pict [PT96], Obliq [Car95], Oz [Smo95], and Telescript [Mag97]. As Facile, they lack a

complete formal definition, and a fortiori any serious technique to reason about program equivalence.

The definition and analysis of systems where failures can occur, has also been the subject of a number of studies in the *distributed algorithms* community in the last decade [Lyn96, Tel95]. In these studies, a system is roughly the (asynchronous) product of a *finite* number of labelled transition systems. The way the labelled transition systems are generated is either ignored or informally specified. It follows that it is impossible to speak seriously about issues such as process equivalence, model-checking, scoping, and process mobility.

To summarize the state of the art, we can say that programming languages lack a formal semantics, and models in the distributed algorithm community lack the right level of intensionality. Our proposal sits between the two. We have not tried to create a theory from scratch, but we have set this theory in an appropriate and well-understood model (the π-calculus). Our framework is close to programming issues (scoping, process mobility...), it is flexible enough to be adapted to different models of failure, failure detection, and process mobility, and it has a tractable theory of process equivalence.

Acknowledgements In carrying on this work, I relied on previous joint work with Ilaria Castellani, Sanjiva Prasad, and Davide Sangiorgi. I also benefited from discussions with Gérard Boudol and members of the PARA project at INRIA-Rocquencourt.

References

[ABND+90] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of ACM*, 37:524–548, 1990.

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *CONCUR 96, Springer Lect. Notes in Comp. Sci. 1119*, Pisa, 1996. Extended version appeared as INRIA-RR 2913, available at `http://protis.univ-mrs.fr/~amadio/`.

[AG97] K. Arnold and J. Gosling. *The Java programming language.* Addison-Wesley, 1997.

[ALT95] R. Amadio, L. Leth, and B. Thomsen. From a concurrent λ-calculus to the π-calculus. In *Proc. Foundations of Computation Theory 95, Dresden.* Springer Lect. Notes in Comp. Sci. 965, 1995.

[AP94] R. Amadio and S. Prasad. Localities and Failures (Extended Summary). In *Proceedings of* 14^{th} *FST and TCS Conference, FST-TCS'94*, volume 880 of *Springer Lect. Notes in Comp. Sci.*, pages 205–216, 1994. Extended version available at `http://protis.univ-mrs.fr/~amadio/`.

[AWWV96] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent programming in Erlang.* Prentice Hall, 1996.

[AZ84] E. Astesiano and E. Zucca. Parametric channels via label expressions in CCS. *Theoretical Computer Science*, 33:45–64, 1984.

[Bor96] M. Boreale. On the expressiveness of internal mobility in name passing calculi. In *CONCUR 96, Springer Lect. Notes in Comp. Sci. 1119*, Pisa, 1996.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, Sophia-Antipolis, 1992.

[Bou93] G. Boudol. Some chemical abstract machines. In *Springer Lect. Notes in Comp. Sci. 803: Proceedings of REX School*. Springer-Verlag, 1993.

[Car95] L. Cardelli. Obliq: a language with distributed scope. In *Proc. POPL*, 1995.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of ACM*, 43(4):685-722, 1996.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225-267, 1996.

[EN86] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical report, DAIMI PB-208, Computer Science Department, Aarhus, Denmark, 1986.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. *Proc. POPL*, 1996.

[FGL+96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR 96, Springer Lect. Notes in Comp. Sci. 1119*, Pisa, 1996.

[FLP95] M Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consenus with one faulty process. *Journal of ACM*, 32:374-382, 1995.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. *Proc. ECOOP 91, Geneve*, 1991.

[Jan95] T. Janowski. *Bisimulation and fault-tolerance.* PhD thesis, University of Warwick, 1995.

[KPT96] N. Kobayashi, B. Pierce, and D. Turner. Linearity in the π-calculus. *Proc. POPL*, 1996.

[Lyn96] N. Lynch. *Distributed algorithms.* Kaufmann, 1996.

[Mag97] General Magic. The telescript programming guide. `http://www.genmagic.com/Develop/Telescript/tsdocs.html`, 1997.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Process, Parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *LICS*, 1993.

[PT96] B. Pierce and D. Turner. Pict: a programming language based on the π-calculus. U. Cambridge, 1996.

[Rep91] J. Reppy. CML: A higher-order concurrent language. In *Proc. ACM-SIGPLAN 91, Conf. on Prog. Lang. Design and Impl.*, 1991.

[San95] D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. In *Proc Tapsoft 95*. Springer Lect. Notes in Comp. Sci. 915, 1995.

[Smo95] G. Smolka. The Oz programming model. In *Proc. Computer science today, Springer Lect. Notes in Comp. Sci. 1000*, 1995.

[Tel95] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 1995.

[TLP+93] B. Thomsen, L. Leth, S. Prasad, T.M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile Antigua release programming guide. Technical Report ECRC-93-20, ECRC, Munich, December 1993. Available at `ftp.ecrc.de`.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399