A RANDOMIZED DISTRIBUTED ENCODING OF THE π-CALCULUS WITH MIXED CHOICE*

Catuscia Palamidessi and Oltea Mihaela Herescu
Dept. of Comp. Sci. and Eng., The Pennsylvania State University
{ catuscia,herescu } @cse.psu.edu

Abstract We consider the problem of encoding the π-calculus (more precisely, the version of the π-calculus with mixed choice) into the asynchronous π-calculus via a *uniform* translation preserving a *reasonable* semantics. Although it has been shown that this is not possible with an exact encoding, we suggest a randomized approach using a probabilistic extension of the asynchronous π-calculus, and we show that our solution is correct with probability 1 under any proper adversary wrt a notion of testing semantics. This result establishes the basis for a distributed and symmetric implementation of mixed choice which, differently from previous proposals in literature, does not rely on assumptions on the relative speed of processes and is robust to attacks of *proper* adversaries.

1. Introduction

In [15] it has been shown that the π-calculus is strictly more expressive than the asynchronous π-calculus, in the sense that it is not possible to encode the first into the latter in a *uniform* way while preserving a *reasonable* semantics. Uniform essentially means homomorphic wrt the parallel and the renaming operators, and reasonable means sensitive to livelocks (divergencies) and to the visible actions. This result is due to the fact that in the π-calculus we can define an algorithm for solving the leader election problem in a *symmetric* network (i.e. a network of isomorphic processes), while this is not possible in the asynchronous π-calculus.

The picture changes if we decide to be content with an encoding that does not preserve *exactly* a reasonable semantics, but preserves it with probability 1. To this purpose, we consider as target language a probabilistic extension of the asynchronous π-calculus, π_{pa} ([4]), based on the probabilistic automata of Segala and Lynch ([22]). The characteristic of this model is that it distinguishes between probabilistic and nondeterministic behavior. The first is associated with the random choices of the process, while the second is related to the arbitrary decisions of an external scheduler. This separation allows us to reason about adverse conditions, i.e. schedulers that "try to sabotage" the encoding, by forcing the translated processes to loop. We propose an

* Work supported by the NSF-POWRE grant EIA-0074909.

encoding that is robust wrt a large class of adversary schedulers: they can make use of all the information about the state and the history of the system, including the result of the past random choices of the processes. The only assumption we need is that the scheduler treats the output action of the asynchronous π-calculus "properly", i.e. as a message that should eventually become available to the reader.

In order to prove the correctness of the encoding we develop a π_{pa} extension of the notion of testing semantics ([14, 2]). This semantics is sensitive to divergencies and deadlocks, hence it is "reasonable" in the sense of [15]. We will show that our encoding is correct in the sense that translated processes preserve, under any proper adversary, and with probability 1, the may and must conditions with respect to each translated observer. There have been other notions of testing semantics developed for probabilistic automata or similar systems, see [8, 7, 21], however, those notions are formalized as orderings among probabilistic processes, and as such they would not be suitable to formulate the correctness of the encoding, which needs to be stated as a correspondence between processes of different kind (non-probabilistic, π, and probabilistic, π_{pa}). It is worth noting that we could not use bisimulation, barbed bisimulation, or coupled simulation either, not even in their weak and asynchronous versions, because these semantics are on one hand "too concrete" for the kind of translation developed here (see Section 4 for more details), and, on the other hand, they are not sensitive to divergencies.

The interest in considering π_{pa} as target language lies on the fact that it can be implemented in a distributed way, i.e. without using centralized control or shared memory. In fact, like in the asynchronous π-calculus, the output actions are not allowed to have a continuation, hence they can be mapped naturally into asynchronous communication, which is the only form of communication available in a distributed architecture. A uniform translation of π_{pa} into a distributed Java-like machine is illustrated in [4]. The condition of uniformity on the encodings of π into π_{pa} and of π_{pa} into Java ensure that the distribution and symmetry are preserved, thus we can argue that our results provide an approach to the distributed and symmetric implementation of the π-calculus.

The distributed implementation of mixed choice, also known as *the binary interaction problem*, has been widely investigated, as well as the more general *multiway interaction problem*. Most of the proposed solutions are asymmetric, see for instance [10, 17, 23], and most of them rely on an ordering among the identifiers of the processes (or equivalently among the nodes of the connection graph). The only symmetric solutions that have been proposed are, not surprisingly, randomized ([3, 9, 19]). They also rely on assumptions about the relative speed of the processes during the phase in which the processes attempt to establish communication (or equivalently, on particular restrictions on the scheduler). The solution proposed in [12] can also be considered as belonging to the category of randomized approaches, although the randomization is not used explicitly by the process, but assumed implicitly in the scheduler. In the full version of this paper ([16]) we show that the algorithms of [3, 9, 19] do not work, when the processes proceed at independent speed, by giving an example of network and adversary for which any attempt of synchronization produces a livelock. The relation with [12] is discussed in Section 3.

To our knowledge, our proposal is the first symmetric solution to the binary interaction problem which makes no assumptions about the relative speed of the processes and it is robust wrt any proper adversary. We also regard as a pleasant feature of our encoding the fact that it does not require the fairness assumption on the scheduler. Most of the randomized algorithms for coordination of distributed processes do require fairness, including the one in [18], but the implementations of concurrent programming languages (for instance Java) usually do not guarantee a fair scheduling policy.

2. Preliminaries

In this section we recall the definition of the π-calculus with mixed choice, of the probabilistic asynchronous π-calculus, and of probabilistic automata.

2.1. The π-calculus with mixed choice

We consider the variant of the π-calculus presented in [20]. The main difference with the original version ([11]) is the absence of a matching operator, and the replacement of free choice with a construct for mixed choice. In our presentation, we will use recursion instead of the replication operator, as we find it more convenient for writing programs.

Consider a countable set of *channel names*, $x, y, \ldots$, and a countable set of *process names* $X, Y, \ldots$. The set of prefixes, $\alpha, \beta, \ldots$, and the set of π-calculus processes, $P, Q, \ldots$, are defined by the following syntax:

$$\begin{array}{llcl} \textit{Prefixes} & \alpha & ::= & x(y) \mid \bar{x}y \mid \tau \\ \textit{Processes} & P & ::= & \sum_i \alpha_i.P_i \mid \nu x P \mid P \mid P \mid X \mid \textit{rec}_X P \end{array}$$

Prefixes represent the basic actions of processes: $x(y)$ is the *input* of the (formal) name y from channel x; $\bar{x}y$ is the *output* of the name y on channel x; τ stands for any silent (non-communication) action.

The process $\sum_i \alpha_i.P_i$ represents guarded choice and it is usually assumed to be finite. We will use the abbreviations **0** (*inaction*) to represent the empty sum, $\alpha.P$ (*prefix*) to represent sum on one element only, and $P + Q$ for the binary sum. The symbols νx and $|$ are the *restriction* and the *parallel* operator, respectively. The process $\textit{rec}_X P$ represents a process X defined as $X \stackrel{\text{def}}{=} P$, where P may contain occurrences of X (recursive definition). We assume that all occurrences of X in P are prefixed.

The operators νx and $y(x)$ are x-*binders*, i.e. in the processes $\nu x P$ and $y(x).P$ the occurrences of x in P are considered *bound*, with the usual rules of scoping. The *alpha-conversion* of bound names is defined as usual, and the renaming (or substitution) $P[y/x]$ is defined as the result of replacing all occurrences of x in P by y, possibly applying alpha-conversion to avoid capture.

The operational semantics is specified via a transition system labeled by *actions* $\mu, \mu' \ldots$, which represent either a prefix or the *bound output* $\bar{x}(y)$. This is introduced to model *scope extrusion*, i.e. the result of sending to another process a private (ν-bound) name. In the following, we will use *fn* and *bn* to denote the set of free and

$$\textsc{Sum} \quad \sum_i \alpha_i.P_i \xrightarrow{\alpha_j} P_j \qquad\qquad \textsc{Res} \quad \frac{P \xrightarrow{\mu} P'}{\nu y P \xrightarrow{\mu} \nu y P'} \quad y \notin names(\mu)$$

$$\textsc{Open} \quad \frac{P \xrightarrow{\bar{x}y} P'}{\nu y P \xrightarrow{\bar{x}(y)} P'} \quad x \neq y \qquad\qquad \textsc{Par} \quad \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \quad bn(\mu) \cap fn(Q) = \emptyset$$

$$\textsc{Com} \quad \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]} \qquad \textsc{Close} \quad \frac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y(P' \mid Q')}$$

$$\textsc{Cong} \quad \frac{P \equiv P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv Q}{P \xrightarrow{\mu} Q}$$

Table 1. The late-instantiation transition system of the π-calculus.

bound names, respectively, in processes and actions. We will also use *names* to denote both kind of names.

In literature there are two main definitions for the transition system of the π-calculus: the *early* system and the *late* system. Here we choose to present the second one because it is more refined, hence more challenging for obtaining positive embedding results.

The rules for the late semantics are given in Table 1. The symbol $\equiv$ used in Rule CONG stands for *structural congruence*, a form of equivalence which identifies "statically" two processes and which is used to simplify the presentation. We assume this congruence to satisfy the associative monoid rules for $|$, alpha-conversion, $rec_X P \equiv P[rec_X P/X]$, and $(\nu x P) \mid Q \equiv \nu x(P \mid Q)$ if $x \notin fn(Q)$.

2.2. Probabilistic automata, adversaries, and executions

Asynchronous automata have been proposed in [22]. Here we consider a variant suitable for π_{pa}. The main difference is that we consider only discrete probabilistic spaces, and that the concept of deadlock is simply a node with no out-transitions.

A discrete probabilistic space is a pair (X, pb) where X is a set and pb is a function $pb : X \to (0, 1]$ such that $\sum_{x \in X} pb(x) = 1$. Given a set Y, we define

$$Prob(Y) = \{(X, pb) \mid X \subseteq Y \text{ and } (X, pb) \text{ is a discrete probabilistic space}\}.$$

Given a set of states S and a set of actions A, a *probabilistic automaton* on S and A is a triple $(S, \mathcal{T}, s_0)$ where $s_0 \in S$ (initial state) and $\mathcal{T} \subseteq S \times Prob(A \times S)$. We call the elements of $\mathcal{T}$ *transition groups* (in [22] they are called *steps*). The idea behind this model is that the choice between two different groups is made nondeterministically and possibly controlled by an external agent, e.g. a scheduler, while the transition within the same group is chosen probabilistically and it is controlled internally (e.g. by a probabilistic choice operator). An automaton in which at most one transition group is allowed for each state is called *fully probabilistic*.

We define now the notion of execution of an automaton under a *scheduler*, by adapting and simplifying the corresponding notion given in [22]. A scheduler can be seen as a function which solves the nondeterminism of the automaton by selecting, at each moment of the computation, a transition group among all the ones allowed in the present state. Schedulers are sometimes called *adversaries*, thus conveying the idea of an external entity playing "against" the process. A process is *robust* wrt a certain class of adversaries if it gives the intended result for each possible scheduling imposed by an adversary in the class. Clearly, the reliability of an algorithm depends on how "smart" the adversaries of this class can be. We will assume that an adversary can decide the next transition group depending not only on the current state, but also on the whole history of the computation till that moment, including the random choices made by the automaton.

In the following, we assume M to be the probabilistic automaton $(S, \mathcal{T}, s_0)$.

We define $\mathit{tree}(M)$ as the tree obtained by unfolding the transition system, i.e. the tree with a root n_0 labeled by s_0, and such that, for each node n, if $s \in S$ is the label of n, then for each $(s, (X, pb)) \in \mathcal{T}$, and for each $(\mu, s') \in X$, there is a node n' child of n labeled by s', and the arc from n to n' is labeled by μ and $pb(\mu, s')$. We will denote by $\mathit{nodes}(M)$ the set of nodes in $\mathit{tree}(M)$, and by $\mathit{state}(n)$ the state labeling a node n.

An *adversary* for M is a function ζ that associates to each node n of $\mathit{tree}(M)$ a transition group among those which are allowed in $\mathit{state}(n)$. More formally, $\zeta : \mathit{nodes}(M) \rightarrow \mathit{Prob}(A \times S)$ is an adversary for M iff $\zeta(n) = (X, pb)$ implies $(\mathit{state}(n), (X, pb)) \in \mathcal{T}$.

The *execution tree* of M under an adversary ζ, denoted by $\mathit{etree}(M, \zeta)$, is the tree obtained from $\mathit{tree}(M)$ by pruning all the groups of transitions which are not selected by ζ. More formally, $\mathit{etree}(M, \zeta)$ is a fully probabilistic automaton $(S', \mathcal{T}', n_0)$, where $S' \subseteq \mathit{nodes}(M)$, n_0 is the root of $\mathit{tree}(M)$, and $(n, (X', pb')) \in \mathcal{T}'$ iff $X' = \{(\mu, n') \mid (\mu, \mathit{state}(n')) \in X\}$ and $pb'(\mu, n') = pb(\mu, \mathit{state}(n'))$, where $(X, pb) = \zeta(n)$.

An *execution fragment* ξ is any path (finite or infinite) from the root of $\mathit{etree}(M, \zeta)$. The notation $\xi \leq \xi'$ means that ξ is a prefix of ξ'. If ξ is $n_0 \xrightarrow[p_0]{\mu_0} n_1 \xrightarrow[p_1]{\mu_1} n_2 \xrightarrow[p_2]{\mu_2} \ldots$, the *probability* of ξ is defined as $pb(\xi) = \prod_i p_i$. If ξ is maximal, then it is called *execution*. We denote by $\mathit{exec}(M, \zeta)$ the set of all executions in $\mathit{etree}(M, \zeta)$.

We define now a probability on certain sets of executions, following a standard construction of Measure Theory. Given an execution fragment ξ, let $C_\xi = \{\xi' \in \mathit{exec}(M, \zeta) \mid \xi \leq \xi'\}$ (*cone* with prefix ξ). Define $pb(C_\xi) = pb(\xi)$. Let $\{C_i\}_{i \in I}$ be a countable set of disjoint cones (i.e. I is countable, and $\forall i, j.\, i \neq j \Rightarrow C_i \cap C_j = \emptyset$). Then define $pb(\bigcup_{i \in I} C_i) = \sum_{i \in I} pb(C_i)$. It is possible to show that pb is well defined, i.e. two countable sets of disjoint cones with the same union produce the same result for pb. We can also define the probability of an empty set of executions as 0, and the probability of the complement of a certain set of executions as the complement wrt 1 of the probability of the set. The closure of the cones wrt the empty set, the countable union, and the complementation generates what in Measure Theory is known as a σ-field.

2.3. The probabilistic asynchronous π-calculus

In this section we recall the definition of π_{pa} ([4]). This calculus is a probabilistic extension of the asynchronous π-calculus ([6]), whose fundamental feature is the absence of the output prefix construct, namely the continuation after an output action. This is why the calculus is called "asynchronous", although communication is modeled by handshaking. We consider a version of the asynchronous π-calculus which admits the input-guarded choice (see for instance [1]), in contrast to the original version which is choiceless. In [13] it is proved that this difference is irrelevant wrt expressiveness.

The novelty of π_{pa} is that each branch of the choice is associated with a probability. The grammar is as follows:

$$\begin{array}{llll} \textit{Prefixes} & \alpha & ::= & x(y) \ \mid \ \tau \\ \textit{Processes} & P & ::= & \bar{x}y \ \mid \ \sum_i p_i \alpha_i . P_i \ \mid \ \nu x P \ \mid \ P \mid P \ \mid \ X \ \mid \ rec_X P \end{array}$$

In the *probabilistic choice operator* $\sum_i p_i \alpha_i . P_i$, the p_i's represent positive probabilities, i.e. they satisfy $p_i \in (0, 1]$ and $\sum_i p_i = 1$.

In order to give the formal definition of the probabilistic model for π_{pa}, it is convenient to introduce the following notation for representing transition groups: given a probabilistic automaton $(S, \mathcal{T}, s_0)$ and $s \in S$, we write

$$s \, \{ \xrightarrow[p_i]{\mu_i} s_i \mid i \in I \}$$

iff $(s, (\{(\mu_i, s_i) \mid i \in I\}, pb)) \in \mathcal{T}$ and $\forall i \in I \ p_i = pb(\mu_i, s_i)$, where I is an index set. When I is not relevant, we will use the simpler notation $s \, \{ \xrightarrow[p_i]{\mu_i} s_i \}_i$. We will also use the notation $s \, \{ \xrightarrow[p_i]{\mu_i} s_i \}_{i:\phi(i)}$, where $\phi(i)$ is a logical formula depending on i, for the set $s \, \{ \xrightarrow[p_i]{\mu_i} s_i \mid i \in I \text{ and } \phi(i) \}$.

The operational semantics of a π_{pa} process P is defined as a probabilistic automaton whose states are the processes reachable from P and the $\mathcal{T}$ relation is defined by the rules in Table 2. In order to keep the presentation simple, we assume that all branches in SUM are different, namely, if $i \neq j$, then $\alpha_i . P_i \not\equiv \alpha_j . P_j$. Furthermore, in RES and PAR we assume that all bound variables are distinct from each other, and from the free variables. We also assume that the transition groups that are interderivable using CONG are identified.

The SUM rule models the behavior of a choice process. Note that all possible transitions belong to the same group, meaning that the transition is chosen probabilistically by the process itself. RES models restriction on channel y: only the actions on channels different from y can be performed and possibly synchronize with an external process. The probability is redistributed among these actions. PAR represents the interleaving of parallel processes. All the transitions of the processes involved are made possible, and they are kept separated in the original groups. In this way we model the fact that the selection of the process for the next computation step is determined by a scheduler. In fact, choosing a group corresponds to choosing a process. COM models

$$\textsc{Sum} \quad \textstyle\sum_i p_i \alpha_i . P_i \; \{\xrightarrow[p_i]{\alpha_i} P_i\}_i \qquad\qquad \textsc{Out} \quad \bar{x}y \; \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$$

$$\textsc{Open} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P \; \{\xrightarrow[1]{\bar{x}(y)} P'\}} \quad x \neq y \qquad\qquad \textsc{Par} \quad \frac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\mu_i} P_i \mid Q\}_i}$$

$$\textsc{Res} \quad \frac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{\nu y P \; \{\xrightarrow[p_i']{\mu_i} \nu y P_i\}_{i : y \notin fn(\mu_i)}} \quad \begin{array}{l} \exists i.\; y \notin fn(\mu_i) \text{ and} \\ \forall i.\; p_i' = p_i / \sum_{j : y \notin fn(\mu_j)} p_j \end{array}$$

$$\textsc{Com} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} P' \mid Q_i[y/z_i]\}_{i : \mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i : \mu_i \neq x(z_i)}}$$

$$\textsc{Close} \quad \frac{P \; \{\xrightarrow[1]{\bar{x}(y)} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} \nu y(P' \mid Q_i[y/z_i])\}_{i : \mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i : \mu_i \neq x(z_i)}}$$

$$\textsc{Cong} \quad \frac{P \equiv P' \qquad P' \; \{\xrightarrow[p_i]{\mu_i} Q_i'\}_i \qquad \forall i.\; Q_i' \equiv Q_i}{P \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}$$

Table 2. The late-instantiation probabilistic transition system of the π_{pa}-calculus. In PAR we assume that if the argument of μ_i is bound then it does not occur free in Q.

communication by handshaking. The output action synchronizes with all matching input actions of a partner, with the same probability of the input action. The other possible transitions of the partner are kept with the original probability as well. CLOSE is analogous to COM, the only difference is that the name being transmitted is private to the sender. OPEN works in combination with CLOSE like in the standard (asynchronous) π-calculus. The other rules, OUT and CONG, should be self-explanatory.

3. Encoding π into π_{pa}

In this section we define a uniform, compositional translation from π to π_{pa}. For the sake of simplicity we assume that the same channel cannot be used as both input and output guard in the same choice construct.

The main difficulty of course consists in encoding the choice operator. We follow an idea used by Nestmann in [12], which consists in associating a lock l, initially set to *true*, to each choice, and then launch a parallel process for each branch. A process P corresponding to an input branch will try to get both its lock (local lock, l) and the partner's lock (remote lock, r). When P succeeds, it tests the locks: if they are both *true* (meaning that P has won the competition) then P sets the locks to *false* so that all the other processes can *abort*, sends a positive acknowledgment (*true*) to the partner, and proceeds with its continuation. The partner also proceeds when it receives the positive acknowledgment. If the local lock is *false* then P aborts. If the remote lock is *false* then P tells the partner to abort by sending it a negative acknowledgment (*false*).

The problem with the algorithm in [12] is that processes might loop forever in the attempt to get both locks. If the initial situation is symmetric, then it is possible to define a scheduler (even a fair one) which always selects the processes in the same order, and never breaks the symmetry. In [12] it is assumed that the scheduler itself has a random behavior, i.e. it selects at random which process to execute next (and in a way totally independent from the history of the system). As argued in the introduction, we believe that it is important to consider stronger schedulers.

In order to make the algorithm robust with respect to every scheduler (under an assumption of "proper" behavior that will be explained later), we enhance it with a randomized choice made internally by the processes involved in the synchronization. The idea is similar to the one used by Lehmann and Rabin for solving the dining philosophers problem ([18]). The forks, in this case, are the locks. The idea is to let the process choose randomly the first lock, and wait on it until it becomes available. When the connection graph (where the forks are the nodes and the philosophers are the arcs) is a simple ring, like in the classic dining philosophers case, this algorithm is deadlock and livelock free with probability 1 under any fair adversary ([18]).

The problem of the mixed choice however still presents a complication: the connection graph may be rather complex, and in [5] it has been shown that the classic algorithm of Lehmann and Rabin does not work for general graphs. For instance, it does not work when a node is part of two or more cycles. In order to cope with this problem, we associate to each choice containing output guards an additional lock h. The processes corresponding to the input branches will first have to compete for the lock h of the partner. Thus, at most one output branch for each choice will be involved at a time in an interaction attempt. This ensures that there will be no connected cycles in the graph representing the interaction attempts, and we will see that this condition is sufficient for the correctness of the algorithm.

In the encoding we make use of some syntactic sugar: we assume polyadic communication (i.e. more than one parameter in the communication actions), boolean values **t** and **f** and an if-then-else construct, which is defined by the structural rules

$$\mathit{if}\ \mathbf{t}\ \mathit{then}\ P\ \mathit{else}\ Q \equiv P \qquad \mathit{if}\ \mathbf{f}\ \mathit{then}\ P\ \mathit{else}\ Q \equiv Q$$

As discussed in [13], these features can be translated into π_a. The encoding of π into π_{pa} is defined in Table 3. Note that all the operators are translated homomorphically except for the choice. In the encoding of the choice, l represents the principal lock (corresponding to a fork in the algorithm of Lehmann and Rabin), h represents the auxiliary lock (for ensuring that no more than one output branch for each choice will be involved simoultaneously in an interaction attempt). In the encoding of the input prefix, l represents the local principal lock, and r represents the remote principal lock. The name a is used to send an acknowledgment to the partner.

Note that in the encoding of the input-prefix the top-level choice, which represents the arbitrary choice of the first principal lock, is a blind choice $(1/2\,\tau\ldots + 1/2\,\tau\ldots)$. This means that the process commits to a lock *before* knowing whether such lock is available. It can be proved that this commitment is essential for the termination of

$$
\begin{array}{rcl}
[\![\nu x P]\!] & = & \nu x [\![P]\!] \\[1ex]
[\![P_1 \mid P_2]\!] & = & [\![P_1]\!] \mid [\![P_2]\!] \\[1ex]
[\![X]\!] & = & X \\[1ex]
[\![rec_X P]\!] & = & rec_X [\![P]\!] \\[1ex]
\left[\!\!\left[\begin{array}{l} \sum_i \alpha_i.P_i \\ + \\ \sum_j \tau.Q_j \\ + \\ \sum_k \beta_k.R_k \end{array} \right]\!\!\right] & = & \nu l\ (\bar{l}\mathbf{t} \mid \nu h\ (\bar{h} \mid \prod_i [\![\alpha_i.P_i]\!]_{lh}) \ \mid\ \prod_j [\![\tau.Q_j]\!]_l \ \mid\ \prod_k [\![\beta_k.R_k]\!]_l) \\[1ex]
[\![\bar{x}y.P]\!]_{rh} & = & \nu a\ (\bar{x}\langle r,a,h,y\rangle \mid a(b).\ \mathit{if}\ b\ \mathit{then}\ [\![P]\!]\ \mathit{else}\ \mathbf{0}) \\[1ex]
[\![\tau.Q]\!]_l & = & l(b).(\bar{l}\mathbf{f} \mid\ \mathit{if}\ b\ \mathit{then}\ [\![Q]\!]\ \mathit{else}\ \mathbf{0}) \\[1ex]
[\![x(y).R]\!]_l & = & rec_X (x(r,a,h,y).h.rec_Y (\begin{array}[t]{l} 1/2\ \tau.l(b_L).((1-\varepsilon)\ r(b_R).B + \varepsilon\ \tau.(\bar{l}b_L \mid Y)) \\ + \\ 1/2\ \tau.r(b_R).((1-\varepsilon)\ l(b_L).B + \varepsilon\ \tau.(\bar{r}b_R \mid Y))\ ) \end{array} \\
\text{where} & & \\
B & = & \begin{array}[t]{llll} & \mathit{if}\ b_L \wedge b_R & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{t} \mid [\![R]\!] \\ \mathit{else} & \mathit{if}\ b_L & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{t} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{f} \mid X \\ \mathit{else} & \mathit{if}\ \quad\ \ b_R & \mathit{then} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{t} \mid \bar{x}\langle r,a,h,y\rangle \\ & & \mathit{else} & \bar{h} \mid \bar{l}\mathbf{f} \mid \bar{r}\mathbf{f} \mid \bar{a}\mathbf{f} \end{array}
\end{array}
$$

Table 3. The encoding of π into π_{pa}. In the translation of the mixed choice, the α_i's represent output actions, and the β_k's represent input actions. ε stands for a real number in $[0,1)$.

the algorithm. The distribution of the probabilities, on the contrary, is not essential for termination. However, this distribution affects the efficiency, i.e. how soon the synchronization protocol will converge. It can be proved that in this choice it is better to split the probability as evenly as possible, hence $1/2$ and $1/2$.

Once the process has obtained the first principal lock, the idea is that it should try to get the second one. If it succeeds, then it should test the locks and proceeds accordingly to the results of the tests as explained at the beginning of this section. Otherwise, it should release both locks and go back to the beginning of the inner loop, where it will make another random draw for selecting the first lock. This conditional behavior would need a priority choice to be expressed, namely a choice in which the first branch would always be selected whenever the corresponding guard is enabled. Such construct does not exist in the (asynchronous) π-calculus, and its introduction would make the semantics rather complicated (although it would be easy to implement it in a language like Java). To overcome the problem, we use a probabilistic choice $((1-\epsilon)\ldots+\epsilon\ldots)$ to approximate a priority choice. Of course, the smaller ϵ is, the tighter the approximation is.

4. Correctness of the encoding

In order to assess the correctness of the translation of π into π_{pa}, we consider a probabilistic extension of the notion of testing semantics proposed in [14, 2]. This extension has the advantage of being probabilistically "reasonable", i.e. sensitive to deadlocks and livelocks with non-null probability. Furthermore, in testing semantics all communications are internalized (except the one used by the observer to declare *success*), and this spares us from the problem, discussed in [12], which arises with semantics like bisimulation, barbed bisimulation, and coupled simulation, even in their weak and asynchronous versions. The kind of encoding that we use for choice cannot be correct wrt these semantics, due to their sensitiveness to the output capabilities. In fact, in the original process the output guards which are not chosen disappear after the choice is made. In the translation, however, a choice is mapped into the parallel composition of the branches, hence an output guard which is not able to interact with a partner will remain present even after some other branch wins the competition, thus causing the presence of a residual output barb. However these barbs are "garbage" by definition, not able to synchronize with any other process at this point (at least, not according to the synchronization protocol of the translated process), so they should not be counted. This sensitivity to the synchronization capabilities is exactly what testing semantics features, differently from bisimulation semantics.

Let us recall briefly the key concepts of the testing semantics for the π-calculus. An *observer* O is a π-calculus process able to perform a special action ω, denoting success. Usually ω is seen as an output action, but it does not really matter. We assume this action to be different from all those performed by tested processes. Given a π-calculus process P and an observer O, an *interaction* between P and O is a maximal (finite or infinite) sequence of τ transitions starting from $P \mid O$:

$$P \mid O = Q_0 \xrightarrow{\tau} Q_1 \xrightarrow{\tau} Q_2 \xrightarrow{\tau} \ldots$$

Maximal means that the sequence is either infinite, or the last state is not able to make any further τ transition.

We say that P **may** O iff there exists an interaction such that $Q_i \xrightarrow{\omega}$ for some i. We say that P **must** O iff for every interaction there exists i such that $Q_i \xrightarrow{\omega}$. Finally, P is *testing equivalent* to Q, notation $P \simeq Q$, if for every observer O, P **may** O iff Q **may** O, and P **must** O iff Q **must** O.

In order to state the correctness of the embedding, we need to extend the notion of testing to the π_{pa}-calculus. We propose the following extension, which, we believe, captures the spirit of testing semantics.

4.1. Testing semantics for the π_{pa}-calculus

The natural extension to π_{pa} of the concept of interaction between a process P and an observer O is an execution starting from $P \mid O$, under some adversary ζ, and consisting only of arcs labeled by τ. An interaction is *successful* if it passes trough a state in which an ω step can be performed.

Our intended notion of successful must testing is that the probability that an interaction be successful is 1. To this end, we need to consider the probability of successful executions *relatively* to those executions which are interactions.

In the sequel we denote by νP the process $\nu x_1 \ldots \nu x_n P$, where $x_1, \ldots, x_n$ are all the free names occurring in P. With a slight abuse of notation, we denote the execution tree of the automaton generated by P under the adversary ζ as $etree(P, \zeta)$, and the set of its branches (executions) as $exec(P, \zeta)$.

Let P be a π_{pa} process and let O be a π_{pa} observer. An interaction ξ between P and O is an element of $exec(\nu(P|O), \zeta)$. Given an interaction ξ of the form:

$$\nu(P \mid O) = Q_0 \xrightarrow[p_0]{\tau} Q_1 \xrightarrow[p_1]{\tau} Q_2 \xrightarrow[p_2]{\tau} \ldots,$$

we say that ξ is *successful* if there exist i and p such that $Q_i \xrightarrow[p]{\omega}$. We denote by $sexec(\nu(P|O), \zeta)$ the set $\{\xi \in exec(\nu(P|O), \zeta) \mid \xi \textit{ is successful}\}$.

The following property is fundamental for defining our notion of testing for π_{pa}:

Proposition 1 *Given an adversary ζ, the set $sexec(\nu(P|O), \zeta)$ can be obtained as a countable union of disjoint cones.*

As a consequence of this proposition, the probability of $sexec(\nu(P|O), \zeta)$ is well defined (cfr. Section 2.2).

Definition 2 Let $\mathcal{A}$ be a class of adversaries. Let P, Q be π_{pa} processes and O be a π_{pa} observer.

- P $\mathbf{may}_{\mathcal{A}}$ O iff there exists an adversary $\zeta \in \mathcal{A}$ s.t. $pb(sexec(\nu(P|O), \zeta)) > 0$.
- P $\mathbf{must}_{\mathcal{A}}$ O iff for every adversary $\zeta \in \mathcal{A}$, $pb(sexec(\nu(P|O), \zeta)) = 1$.
- $P \simeq_{\mathcal{A}} Q$ iff for every O, P $\mathbf{may}_{\mathcal{A}}$ O iff Q $\mathbf{may}_{\mathcal{A}}$ O and P $\mathbf{must}_{\mathcal{A}}$ O iff Q $\mathbf{must}_{\mathcal{A}}$ O.

4.2. Correctness of the encoding wrt testing semantics

First of all, we need to make precise what class of adversaries our algorithm can cope with. Clearly, we wish this class to be as large as possible. Yet, we cannot allow just *any* adversary. The problem is related to the output actions: a malicious adversary that never schedules $\bar{l}b_L$ or $\bar{r}b_R$ in the definition of $[\![x(y).P]\!]_l$ will make it impossible for the process to get the lock and therefore will force it to loop forever.

In the intended meaning of the asynchronous π-calculus, however, these actions represent messages rather than processes. The idea is that they are "sent" when they reach the top-level in a parallel context, and are "received" when the handshaking with the corresponding input action takes place. Thus it is reasonable to assume that the scheduler will not delay forever the reception of a message, i.e. if an output action is in parallel with a process able to execute the corresponding input action, then the handshaking will eventually take place.

Definition 3 An adversary ζ for P is *proper* if, whenever P evolves into a process of the form $\nu x_1 \ldots \nu x_k(P_1| \ldots |P_n)$, in which one of the P_i's is an output action on one of the channels $x_1 \ldots x_k$, if ζ selects infinitely often a parallel process ready to execute the corresponding input action, then P_i will eventually be scheduled for handshaking. Namely, P_i will be in the premise of a COM or CLOSE rule. We will denote by $\mathcal{P}$ the class of *proper* adversaries.

Note that the above definition is weaker than the notion of fair scheduler, which requires that *any* process which is ready infinitely often will eventually be scheduled for execution. Clearly, the fairness assumption would be sufficient for our encoding, however it is not necessary. This may seem surprising, since the solution to the dining philosophers proposed in [18] *requires* fairness. However, a careful analysis of the algorithm in [18] reveals that the fairness assumption is used *only* because a philosopher who has committed to a fork enters a *busy waiting* loop, and it remains in the loop until the fork becomes available. An unfair scheduler, hence, could keep scheduling always the same philosopher in a busy waiting loop, thus generating a livelock. If the busy wait is replaced by a suspension command (obliging the scheduler to select another process) then the fairness assumption is not be necessary ([16]).

It is important to note that π_{pa} (like most process algebra) has a suspension mechanism associated with the communication actions: if a process can proceed only by performing a handshaking, then the process will suspend until the partner is ready. Furthermore the semantics of π_{pa} ensures that a scheduler is obliged to select processes which are not suspended. Note that in $[\![x(y).P]\!]_l$ (Table 3) the acquisition of h (auxiliary lock) and of the first lock are done by input prefixes (with no alternatives) and therefore they will suspend if the locks are unavailable. It is easy to implement such suspension mechanism in a language like Java by using the `wait()` and `notify()` primitives.

Theorem 4 *For every π-calculus process P, and every observer O*

(i) P **may** O *iff* $[\![P]\!]\ \mathbf{may}_{\mathcal{P}}\ [\![O]\!]$

(ii) P **must** O *iff* $[\![P]\!]\ \mathbf{must}_{\mathcal{P}}\ [\![O]\!]$

An important ingredient of the proof of the above result is that, at any point of the execution of $[\![P]\!]$, in the graph representing the interaction attempts all cycles are disconnected (i.e. they are not connected to each other by any path). It has been shown in [5] that this is a necessary condition for the algorithm of [18] to be livelock-free, even under the fairness hypothesis.

The following corollary, which is an immediate consequence of the above theorem, states correctness in the standard process algebra sense.

Corollary 5 *For every π-calculus processes P and Q, if $[\![P]\!] \simeq_{\mathcal{P}} [\![Q]\!]$ then $P \simeq Q$.*

Note that the viceversa (full abstraction) does not hold: This is due to the fact that, if we allow arbitrary observers in π_{pa}, then we can distinguish $[\![P]\!]$ and $[\![Q]\!]$ by using observers which interact directly with their actions, i.e. without following the synchronization protocol enforced by the algorithm.

References

[1] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[2] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1995.

[3] N. Francez and M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proc. of FOCS*, pages 373–379, 1980.

[4] Oltea Mihaela Herescu and Catuscia Palamidessi. Probabilistic asynchronous π-calculus. In *Proc. of FOSSACS*, LNCS 1784, pages 146–160. Springer-Verlag, 2000.

[5] Oltea Mihaela Herescu and Catuscia Palamidessi. On the generalized dining philosophers problem. In *Proc. of PODC*, pages 81–89, 2001.

[6] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP*, LNCS 512, pages 133–147. Springer-Verlag, 1991.

[7] Bengt Jonsson, Kim G. Larsen, and Wang Yi. Probabilistic extensions of process algebras. In J. Bergstra, A. Ponse, and S. Smolka, Eds., *Handbook of Process Algebras*. Elsevier. 2001.

[8] Bengt Jonsson and Wang Yi. Compositional testing preorders for probabilistic processes. In *Proc. of LICS*, pages 431–441, 1995. IEEE Computer Society Press.

[9] Yuh-Jzer Joung and Scott A. Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):137–149, 1998.

[10] F. Knabe. A distributed protocol for channel-based communications with choice. *Computers and Artificial Intelligence*, 12(5):475–490, 1993.

[11] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 & 41–77, 1992.

[12] Uwe Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000.

[13] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Journal of Information and Computation*, 163:1–59, 2000.

[14] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, 1984.

[15] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. To appear in *Mathematical Structures in Computer Science*. A preliminary version appeared in *Proc. of POPL'97*.

[16] Catuscia Palamidessi and Oltea Mihaela Herescu. A randomized distribuited encoding of the π-calculus with mixed choice. Internal Report. Postscript available at `http://www.cse.psu.edu/~catuscia/papers/Prob_impl/report.ps`.

[17] Joachim Parrow and Peter Sjodin. Multiway synchronization verified with coupled simulation. In *Proc. of CONCUR*, LNCS 630, pages 518–533. Springer-Verlag, 1992.

[18] Michael O. Rabin and Daniel Lehmann. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 20, pages 333–352. Prentice Hall, 1994. An extended abstract appeared in the *Proc. of POPL'81*, pages 133-138.

[19] John H. Reif and Paul G. Spirakis. Real-time synchronization of interprocess communications. *ACM Trans. on Programming Languages and Systems*, 6(2):215–238, 1984.

[20] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(1,2):235–274, 1996.

[21] Roberto Segala. Testing probabilistic automata. In *Proc. of CONCUR*, LNCS 1119, pages 299–314, 1996. Springer-Verlag.

[22] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[23] Yih-Kuen Tsay and Rajive L. Bagrodia. Fault-tolerant algorithms for fair interprocess synchronization. *IEEE Trans. on Parallel and Distributed Systems*, 5(7):737–748, 1994.