Probabilistic asynchronous π-calculus*

Oltea Mihaela Herescu and Catuscia Palamidessi

Dept. of Comp. Sci. and Eng., The Pennsylvania State University
University Park, PA 16802-6106 USA
{herescu,catuscia}@cse.psu.edu

Abstract. We propose an extension of the asynchronous π-calculus with a notion of random choice. We define an operational semantics which distinguishes between probabilistic choice, made internally by the process, and nondeterministic choice, made externally by an adversary scheduler. This distinction will allow us to reason about the probabilistic correctness of algorithms under certain schedulers. We show that in this language we can solve the electoral problem, which was proved not possible in the asynchronous π-calculus. Finally, we show an implementation of the probabilistic asynchronous π-calculus in a Java-like language.

1 Introduction

The π-calculus ([5, 6]) is a very expressive specification language for concurrent programming, but the difficulties in its distributed implementation challenge its candidature to be a canonical model of distributed computation. Certain mechanisms of the π-calculus, in fact, require solving a problem of distributed consensus.

The asynchronous π-calculus ([3, 2]), on the other hand, is more suitable for a distributed implementation, but it is rather weak for solving distributed problems ([9]).

In order to increase the expressive power of the asynchronous π-calculus we propose a probabilistic extension, π_{pa}, based on the probabilistic automata of Segala and Lynch ([12]). 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 prevent" the process from achieving its goal. Similar models were presented in [14] and [15].

Next we show an example of distributed problem that can be solved with π_{pa}, namely the election of a leader in a symmetric network. It was proved in [9] that such problem cannot be solved with the asynchronous π-calculus. We propose an algorithm for the solution of this problem, and we prove that it is correct, i.e. that the leader will eventually be elected, with probability 1, under every possible scheduler. Our algorithm is reminiscent of the algorithm used in

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

[10] for solving the dining philosophers problem, but in our case we do not need the fairness assumption. Also, the fact that we give the solution in a language provided with a rigorous operational semantics allows us to give a more formal proof of correctness.

Finally, we define a "toy" distributed implementation of the π_{pa}-calculus into a Java-like language. The purpose of this exercise is to prove that π_{pa} is a reasonable paradigm for the specification of distributed algorithms, since it can be implemented without loss of expressivity.

The novelty of our proposal, with respect to other probabilistic process algebras which have been defined in literature (see, for instance, [13]), is the definition of the parallel operator in a CCS style, as opposed to the SCCS style. Namely, parallel processes are not forced to proceed simultaneously. Note also that for general probabilistic automata it is not possible to define the parallel operator ([11]), or at least, there is no natural definition. In π_{pa} the parallel operator can be defined as a natural extension of the non probabilistic case, and this can be considered, to our opinion, another argument in favor of the suitability of π_{pa} for distributed implementation.

2 Preliminaries

In this section we recall the definition of the asynchronous π-calculus and the definition of probabilistic automata. We consider the *late* semantics of the π-calculus, because the probabilistic extension of the late semantics is simpler than the eager version.

2.1 The asynchronous π-calculus

We follow the definition of the asynchronous π-calculus given in [1], except that we will use recursion instead of the replication operator, since we find it to be more convenient for writing programs. It is well known that recursion and replication are equivalent, see for instance [4].

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

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

The basic actions are $x(y)$, which represents the *input* of the (formal) name y from channel x, $\bar{x}y$, which represents the *output* of the name y on channel x, and τ, which stands for any silent (non-communication) action.

The process $\sum_i \alpha_i.P_i$ represents guarded choice on input or silent prefixes, and it is usually assumed to be finite. We will use the abbreviations $\mathbf{0}$ (*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 $\mid$ are the *restriction* and the *parallel* operator, respectively. We adopt the convention that the prefix operator has priority wrt $+$ and $|$. The process $rec_X P$ represents a process X defined as $X \stackrel{\text{def}}{=} P$, where P may contain occurrences of X (recursive definition). We assume that all the 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 *free names* of P, i.e. those names which do not occur in the scope of any binder, are denoted by $fn(P)$. 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 free occurrences of x in P by y, possibly applying alpha-conversion in order to avoid capture.

The operational semantics is specified via a transition system labeled by *actions* $\mu, \mu' \ldots$. These are given by the following grammar:

$$\textit{Actions}\ \mu ::= x(y) \ \mid \ \bar{x}y \ \mid \ \bar{x}(y) \ \mid \ \tau$$

Essentially, we have all the actions from the syntax, plus 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. The bound names of an action μ, $bn(\mu)$, are defined as follows: $bn(x(y)) = bn(\bar{x}(y)) = \{y\}$; $bn(\bar{x}y) = bn(\tau) = \emptyset$. Furthermore, we will indicate by $n(\mu)$ all the *names* which occur in μ.

The rules for the late semantics are given in Table 1. The symbol $\equiv$ used in 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 following:

(i) $P \equiv Q$ if Q can be obtained from P by alpha-renaming, notation $P \equiv_\alpha Q$,
(ii) $P \mid Q \equiv Q \mid P$,
(iii) $rec_X P \equiv P[rec_X P/X]$,

Note that communication is modeled by handshaking (Rules Com and Close). The reason why this calculus is considered a paradigm for *asynchornous* communication is that there is no primitive *output prefix*, hence no primitive notion of continuation after the execution of an output action. In other words, the process executing an ouptut action will not be able to detect (in principle) when the corresponding input action is actually executed.

2.2 Probabilistic automata, adversaries, and executions

Asynchronous automata have been proposed in [12]. We simplify here the original definition, and tailor it to what we need for defining the probabilistic extension of the asynchronous π-calculus. 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

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

Sum	$\sum_i \alpha_i.P_i \xrightarrow{\alpha_j} P_j$
Out	$\bar{x}y \xrightarrow{\bar{x}y} \mathbf{0}$
Open	$\dfrac{P \xrightarrow{\bar{x}y} P'}{\nu y P \xrightarrow{\bar{x}(y)} P'} \quad x \neq y$
Res	$\dfrac{P \xrightarrow{\mu} P'}{\nu y P \xrightarrow{\mu} \nu y P'} \quad y \notin n(\mu)$
Par	$\dfrac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} \quad bn(\mu) \cap fn(Q) = \emptyset$
Com	$\dfrac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]}$
Close	$\dfrac{P \xrightarrow{\bar{x}(y)} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} \nu y(P' \mid Q')}$
Cong	$\dfrac{P \equiv P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv Q}{P \xrightarrow{\mu} Q}$

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

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 $\mathcal{T} \subseteq S \times \mathit{Prob}(A \times S)$. We call the elements of $\mathcal{T}$ *transition groups* (in [12] 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). If at most one transition group is allowed for each state, the automaton is called *fully probabilistic*. Figures 1 and 2 give examples of a probabilistic and a fully probabilistic automaton, respectively.

In [12] it is remarked that this notion of automaton subsumes and extends both the the *reactive* and *generative* models of probabilistic processes ([13]). In

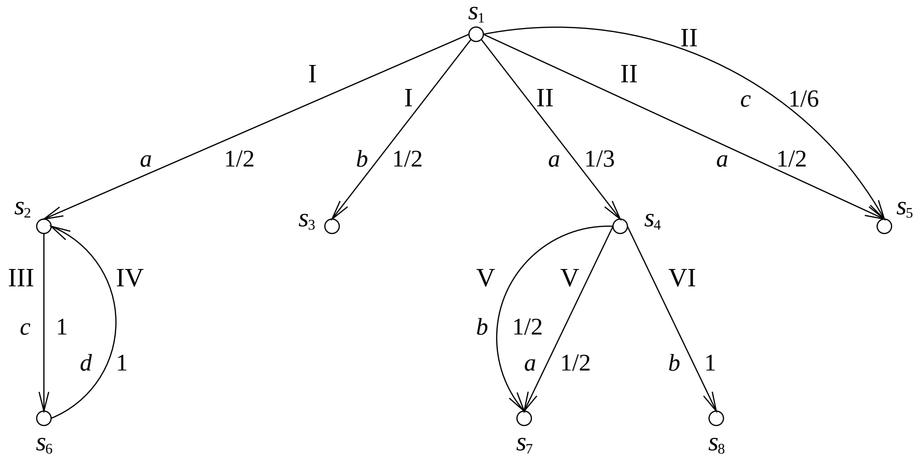

Fig. 1. Example of a probabilistic automaton M. The transition groups are labeled by I, II, ..., VI

particular, the generative model corresponds to the notion of fully probabilistic automaton.

We define now the notion of execution of an automaton under a *scheduler*, by adapting and simplifying the corresponding notion given in [12]. 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.

Given a probabilistic automaton $M = (S, \mathcal{T}, s_0)$, 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. Example: Figure 3 represents the tree obtained from the probabilistic automaton M of Figure 1.

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, ζ :

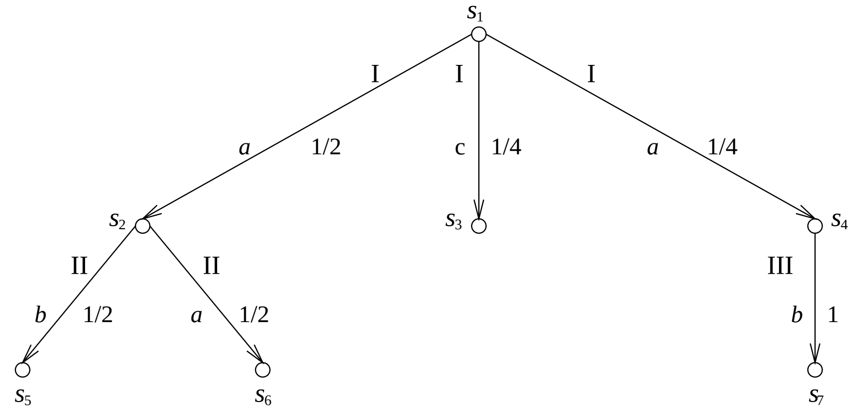

Fig. 2. A fully probabilistic automaton

$\mathit{nodes}(M) \to \mathit{Prob}(A \times S)$ such that $\zeta(n) = (X, \mathit{pb})$ implies $(\mathit{state}(n), (X, \mathit{pb})) \in \mathcal{T}$.

The *execution tree* of an automaton $M = (S, \mathcal{T}, s_0)$ under an adversary ζ, denoted by $\mathit{etree}(M, \zeta)$, is the tree obtained from $\mathit{tree}(M)$ by pruning all the arcs corresponding to transitions which are not in the group 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 $(n, (X', \mathit{pb}')) \in \mathcal{T}'$ iff $X' = \{(\mu, n') \mid (\mu, \mathit{state}(n')) \in X\}$ and $\mathit{pb}'(\mu, n') = \mathit{pb}(\mu, \mathit{state}(n'))$, where $(X, \mathit{pb}) = \zeta(n)$. Example: Figure 4 represents the execution tree of the automaton M of Figure 1, under an adversary ζ.

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 $\mathit{pb}(\xi) = \prod_i p_i$. 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 $\mathit{pb}(C_\xi) = \mathit{pb}(\xi)$. 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 $\mathit{pb}(\bigcup_{i \in I} C_i) = \sum_{i \in I} \mathit{pb}(C_i)$. 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 probablity 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.

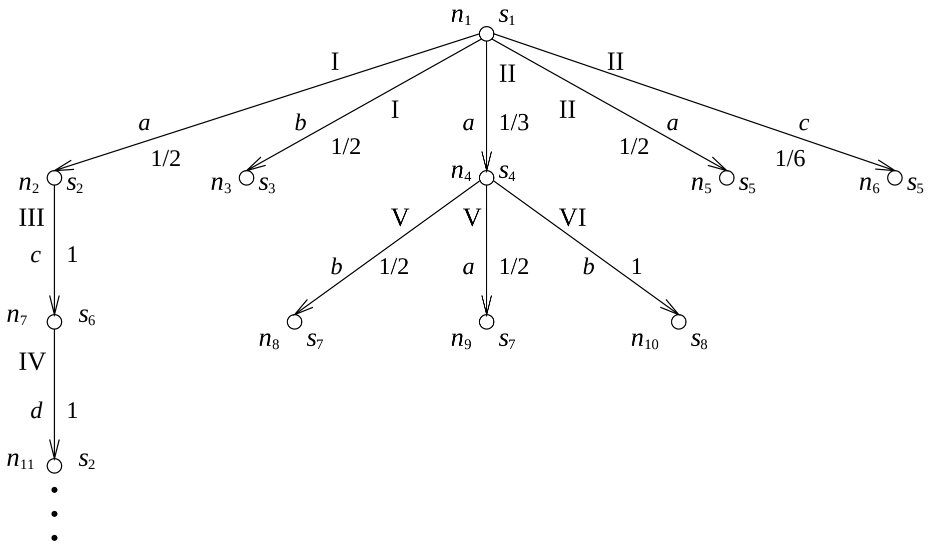

Fig. 3. $\mathit{tree}(M)$, where M is the probabilistic automaton M of Figure 1

3 The probabilistic asynchronous π-calculus

In this section we introduce the probabilistic asynchronous π-calculus (π_{pa}-calculus for short) and we give its operational semantics in terms of probabilistic automata.

The π_{pa}-calculus is obtained from the asynchronous π-calculus by replacing $\sum_i \alpha_i.P_i$ with the following *probabilistic choice operator*

$$\sum_i p_i \alpha_i.P_i$$

where the p_i's represents positive probabilities, i.e. they satisfy $p_i \in (0,1]$ and $\sum_i p_i = 1$, and the α_i's are input or silent prefixes.

In order to give the formal definition of the probabilistic model for π_{pa}, we find it 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 $s\,\{\xrightarrow[p_i]{\mu_i} s_i \mid i \in I \text{ and } \phi(i)\}$.

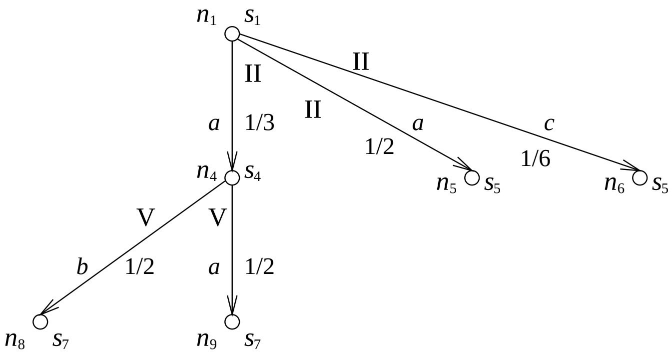

Fig. 4. etree(M, ζ), where M is the probabilistic automaton M of Figure 1, and (the significant part of) ζ is defined by $\zeta(n_1) = \text{II}$, $\zeta(n_4) = \text{V}$

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 impose some restrictions on the syntax of terms (see the caption of Table 2). In Appendix A we give an equivalent definition of the operational semantics without these restrictions.

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 orininal 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 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.

Example 1. Consider the processes $P = rec_X(1/2\ x(y).\mathbf{0} + 1/2\ \tau.X)$, $Q = \bar{x}y$ and define $R = P \mid Q$. The transition groups starting from R are:

$$R\,\{\xrightarrow[1/2]{x(y)} Q\ ,\ \xrightarrow[1/2]{\tau} R\} \qquad R\,\{\xrightarrow[1/2]{\tau} \mathbf{0}\ ,\ \xrightarrow[1/2]{\tau} R\} \qquad R\,\{\xrightarrow[1]{\bar{x}y} P\}$$

SUM	$\sum_i p_i \alpha_i.P_i \; \{\xrightarrow[p_i]{\alpha_i} P_i\}_i$	
OUT	$\bar{x}y \; \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$	
OPEN	$\dfrac{P \; \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P \; \{\xrightarrow[1]{\bar{x}(y)} P'\}}$	$x \neq y$
RES	$\dfrac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{\nu y P \; \{\xrightarrow[p'_i]{\mu_i} \nu y P_i\}_{i:y \notin fn(\mu_i)}}$	$\exists i.\; y \notin fn(\mu_i)$ and $\forall i.\; p'_i = p_i / \sum_{j:y \notin fn(\mu_j)} p_j$
PAR	$\dfrac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\mu_i} P_i \mid Q\}_i}$	
COM	$\dfrac{P \; \{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} P' \mid Q_i[y/z_i]\}_{i:\mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i:\mu_i \neq x(z_i)}}$	
CLOSE	$\dfrac{P \; \{\xrightarrow[1]{\bar{x}(y)} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p_i]{\tau} \nu y(P' \mid Q_i[y/z_i])\}_{i:\mu_i = x(z_i)} \cup \{\xrightarrow[p_i]{\mu_i} P \mid Q_i\}_{i:\mu_i \neq x(z_i)}}$	
CONG	$\dfrac{P \equiv P' \qquad P' \; \{\xrightarrow[p_i]{\mu_i} Q'_i\}_i \qquad \forall i.\; Q'_i \equiv Q_i}{P \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}$	

Table 2. The late-instantiation probabilistic transition system of the π_{pa}-calculus. In SUM we assume that all branches are different, namely, if $i \neq j$, then either $\alpha_i \neq \alpha_j$, or $P_i \not\equiv P_j$. Furthermore, in RES and PAR we assume that all bound variables are distinct from each other, and from the free variables.

Figure 5 illustrates the probabilistic automaton corresponding to R. The above transition groups are labeled by I, II and III respectively.

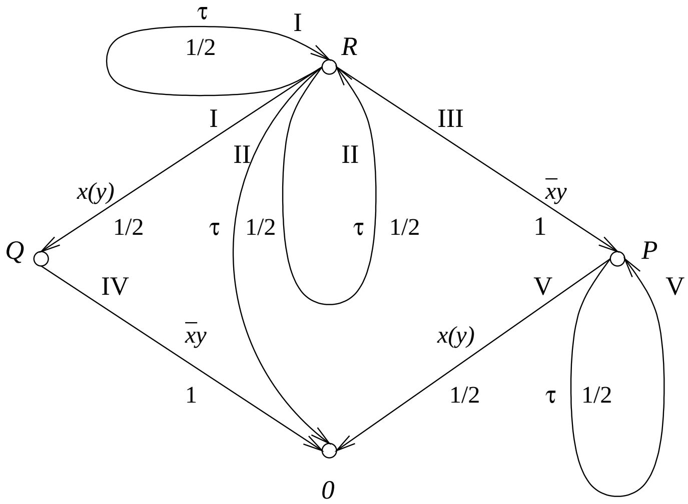

Fig. 5. The probabilistic automaton of Example 1

Example 2. Consider the processes P and Q of example 1 and define $R = (\nu x)(P \mid Q)$. In this case the transition groups starting from R are:

$$R\,\{\xrightarrow[1]{\tau} R\} \qquad R\,\{\xrightarrow[1/2]{\tau} \mathbf{0}\ ,\ \xrightarrow[1/2]{\tau} R\}$$

Figure 6 illustrates the probabilistic automaton corresponding to this new definition of R. The above transition groups are labeled by I and II respectively.

Next example shows that the expansion law does not hold in π_{pa}. This should be no surprise, since the choices associated to the parallel operator and to the sum, in π_{pa}, have a different nature: the parallel operator gives rise to nondeterministic choices of the scheduler, while the sum gives rise to probabilistic choices of the process.

Example 3. Consider the processes $R_1 = x(z).P \mid y(z).Q$ and $R_2 = p\ x(z).(P \mid y(z).Q) + (1-p)\ y(z).(x(z).P \mid Q)$. The transition groups starting from R_1 are:

$$R_1\,\{\xrightarrow[1]{x(z)} P \mid y(z).Q\} \qquad R_1\,\{\xrightarrow[1]{y(z)} x(z).P \mid Q\}$$

On the other hand, there is only one transition group starting from R_2, namely:

$$R_2\,\{\xrightarrow[p]{x(z)} P \mid y(z).Q\ ,\ \xrightarrow[1-p]{y(z)} x(z).P \mid Q\}$$

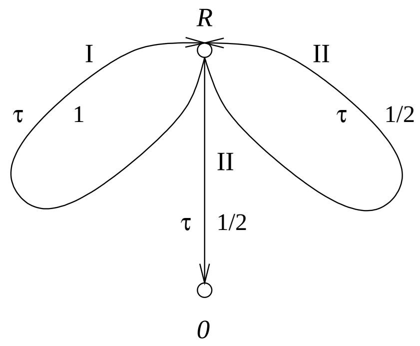

Fig. 6. The probabilistic automaton of Example 2

As announced in the introduction, the parallel operator is associative. This property can be easily shown by case analysis.

Proposition 1. *For every process P, Q and R, the probabilistic automata of $P \mid (Q \mid R)$ and of $(P \mid Q) \mid R$ are isomorphic, in the sense that they differ only for the name of the states (i.e. the syntactic structure of the processes).*

We conclude this section with a discussion about the design choices of π_{pa}.

3.1 The rationale behind the design of π_{pa}

In defining the rules of the operational semantics of π_{pa} we felt there was only one natural choice, with the exception of the rules Com and Close. For them we could have given a different definition, with respect to which the parallel operator would still be associative.

The alternative definition we had considered for Com was:

$$Com' \; \frac{P \; \{\xrightarrow[1]{\bar{x}y} P'\} \qquad Q \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}{P \mid Q \; \{\xrightarrow[p'_i]{\tau} P' \mid Q_i\}_{i:\mu_i = x(y)}} \; \begin{array}{l} \exists i.\; \mu_i = x(y) \text{ and} \\ \forall i.\; p'_i = p_i / \sum_{j:\mu_j = x(y)} p_j \end{array}$$

and similarly for Close.

The difference between Com and Com$'$ is that the latter forces the process performing the input action (Q) to perform only those actions that are compatible with the output action of the partner (P).

At first Com$'$ seemed to be a reasonable rule. At a deeper analysis, however, we discovered that Com$'$ imposes certain restrictions on the schedulers that, in a distributed setting, would be rather unnatural. In fact, the natural way of implementing the π_a communication in a distributed setting is by representing the input and the output partners as processes sharing a common channel. When the sender wishes to communicate, it puts a message in the channel. When the receiver wishes to communicate, it tests the channel to see if there is a message, and, in the positive case, it retrieves it. In case the receiver has a choice guarded by input actions on different channels, the scheduler can influence this choice by activating certain senders instead of others. However, if more than one sender has been activated, i.e. more than one channel contains data at the moment in which the receiver is activated, then it will be the receiver which decides internally which channel to select. COM models exactly this situation. Note that the scheduler can influence the choices of the receiver by selecting certain outputs to be premises in COM, and delaying the others by using PAR.

With COM′, on the other hand, when an input-guarded choice is executed, the choice of the channel is determined by the scheduler. Thus COM′ models the assumption that the scheduler can only activate (at most) one sender before the next activation of a receiver.

The following example illustrates the difference between COM and COM′.

Example 4. Consider the processes $P_1 = \bar{x}_1 y$, $P_2 = \bar{x}_2 z$, $Q = 1/3\ x_1(y).Q_1 + 2/3\ x_2(y).Q_2$, and define $R = (\nu x_1)(\nu x_2)(P_1 \mid P_2 \mid Q)$. Under COM, the transition groups starting from R are

$$R\,\{\xrightarrow[1/3]{\tau} R_1, \xrightarrow[2/3]{\tau} R_2\} \qquad R\,\{\xrightarrow[1]{\tau} R_1\} \qquad R\,\{\xrightarrow[1]{\tau} R_2\}$$

where $R_1 = (\nu x_1)(\nu x_2)(P_2 \mid Q_1)$ and $R_2 = (\nu x_1)(\nu x_2)(P_1 \mid Q_2)$. The first group corresponds to the possibility that both $\bar{x}_1$ and $\bar{x}_2$ are available for input when Q is scheduled for execution. The other groups correspond to the availability of only $\bar{x}_1$ and only $\bar{x}_2$ respectively.

Under COM′, on the other hand, the only possible transition groups are

$$R\,\{\xrightarrow[1]{\tau} R_1\} \qquad R\,\{\xrightarrow[1]{\tau} R_2\}$$

Note that, in both cases, the only possible transitions are those labeled with τ, because $\bar{x}_1$ and $\bar{x}_2$ are restricted at the top level.

4 Solving the electoral problem in π_{pa}

In [9] it has been proved that, in certain networks, it is not possible to solve the leader election problem by using the asynchronous π-calculus. The problem consists in ensuring that all processes will reach an agreement (elect a leader) in finite time. One example of such network is the system consisting of two symmetric nodes P_0 and P_1 connected by two internal channels x_0 and x_1 (see Figure 7).

In this section we will show that it is possible to solve the leader election problem for the above network by using the π_{pa}-calculus. Following [9], we will assume that the processes communicate their decision to the "external word" by using channels o_0 and o_1.

The reason why this problem cannot be solved with the asynchronous π-calculus is that a network with a leader is not symmetric, and the asynchronous π-calculus is not able to force the initial symmetry to break. Suppose for example

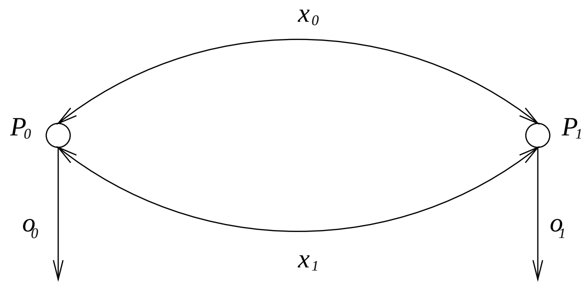

Fig. 7. A symmetric network $\mathrm{P} = \nu x_0 \ \nu x_1 (P_0 \mid P_1)$. The restriction on x_0, x_1 is made in order to enforce synchronization.

that P_0 would elect itself as the leader after performing a certain sequence of actions. By symmetry, and because of lack of synchronous communication, the same actions may be performed by P_1. Therefore P_1 would elect itself as leader, which means that no agreement has been reached.

We propose a solution based on the idea of breaking the symmetry by repeating again and again certain random choices, until this goal has been achieved. The difficult point is to ensure that it will be achieved with probability 1 *under every possible scheduler*.

Our algorithm works as follows. Each process performs an output on its channel and, in parallel, tries to perform an input on both channels. If it succeeds, then it declares itself to be the leader. If none of the processes succeeds, it is because both of them perform exactly one input (thus reciprocally preventing the other from performing the second input). This might occur because the inputs can be performed only sequentially[1]. In this case, the processes have to try again. The algorithm is illustrated in Table 3.

In the algoritm, the selection of the first input is controlled by each process with a probabilistic blind choice, i.e. a choice whose branches are prefixed by a silent (τ) action. This means that the process commits to the choice of the channel *before* knowing whether it is available. It can be proved that this commitment is essential for ensuring that the leader will be elected with probability 1 under every possible adversary scheduler. The distribution of the probabilities, on the contrary, is not essential. This distribution however affects the efficiency (i.e. how soon the synchronization protocol converges). It can be shown that it is better to split the probability as evenly as possible (hence 1/2 and 1/2).

After the first input is performed, a process tries to perform the second input. What we would need at this point is a *priority choice*, i.e. a construct that selects the first branch if the prefix is enabled, and selectes the second branch

[1] In the π_{pa}-calculi and in most process algebra there is no primitive for simultaneous input action. Nestmann has proposed in [7] the addition of such construct as a way of enhancing the expressive power of the asynchronous π-calculus. Clearly, with this addition, the solution to the electoral problem would be immediate.

$$
\begin{aligned}
P_i \;=\; & \bar{x}_i\langle t\rangle \\
& \mid rec_X\,(\,1/2\;\tau.x_i(b).\;\mathit{if}\;b \\
& \qquad\qquad \mathit{then}\;(\,(1-\varepsilon)\;x_{i\oplus 1}(b).(\bar{o}_i\langle i\rangle \mid \bar{x}_i\langle f\rangle) \\
& \qquad\qquad\qquad + \\
& \qquad\qquad\qquad \varepsilon\;\tau.(\bar{x}_i\langle t\rangle \mid X)) \\
& \qquad\qquad \mathit{else}\;\bar{o}_i\langle i\oplus 1\rangle\;) \\
& \qquad + \\
& \qquad 1/2\;\tau.x_{i\oplus 1}(b).\;\mathit{if}\;b \\
& \qquad\qquad \mathit{then}\;(\,(1-\varepsilon)\;x_i(b).(\bar{o}_i\langle i\rangle \mid \bar{x}_{i\oplus 1}\langle f\rangle) \\
& \qquad\qquad\qquad + \\
& \qquad\qquad\qquad \varepsilon\;\tau.(\bar{x}_{i\oplus 1}\langle t\rangle \mid X)) \\
& \qquad\qquad \mathit{else}\;\bar{o}_i\langle i\oplus 1\rangle\;)
\end{aligned}
$$

Table 3. A π_{pa} solution for the electoral problem in the symmetric network of Figure 7. Here $i \in \{0,1\}$ and $\oplus$ is the sum modulo 2.

otherwise. With this construct the process would perform the input on the other channel when it is available, and backtrack to the initial situation otherwise. Since such construct does not exists in the π-calculi, we use probabilities as a way of approximating it. Thus we do not guarantee that the first branch will be selected for sure when the prefix is enabled, but we guarantee that it will be selected with probability close to 1: the symbol ε represents a very small positive number. Of course, the smallest ϵ is, the more efficient the algorithm is.

When a process, say P_0, succeeds to perform both inputs, then it declares itself to be the leader. It also notifies this decision to the other process. For the notification we could use a different channel, or we may use the same channel, provided that we have a way to communicate that the output on such channel has now a different meaning. We follow this second approach, and we use boolean values **t** and **f** for messages. We stipulate that **t** means that the leader has not been decided yet, while **f** means that it has been decided. Notice that the symmetry is broken exactly when one process succeeds in performing both inputs.

In the algorithm we make use of the 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 [8], these features (booleans and if-then-else) can be translated into the asynchronous π-calculus, and therefore in π_{pa}.

Correctness of the algorithm

We prove now that the algorithm is correct, namely that the probability that a leader is eventually elected is 1 under every scheduler.

In the following we use pairs to denote the τ transitions corresponding to the execution of the blind choice. A pair (i, j) will mean that process i has selected channel j. We will call such transitions *random draws.*

Definition 1. *A sequence $d_1, d_2, \ldots, d_n, \ldots$ of random draws is alternated iff $\forall\, k$, if $d_k = (i, j)$ then $d_{k+1} = (i, j)$ or $d_{k+1} = (i \oplus 1, j \oplus 1)$.*

Note that a sequence is alternated iff for every two draws $(i, j), (i', j')$ if $i = i'$ then $j = j'$.

For the proof, we are going to consider, at first, a modified algorithm where the inner choice $((1 - \varepsilon) \ldots + \varepsilon \ldots)$ is replaced by a priority choice.

Lemma 1. *Consider an execution fragment ξ of the process $\nu x_0\ \nu x_1(P_0 \mid P_1)$ and the algorithm of Table 3 modified by using the priority choice. Let $d_1, d_2, \ldots, d_n$ be the sequence of random draws in ξ. Assume that, for some $k < n$, $d_k = (i, j)$, $d_{k+1} = \ldots = d_{n-1} = (i \oplus 1, j \oplus 1)$, and $d_n = (i \oplus 1, j)$. Then, under every adversary, all the executions in the cone of ξ terminate with the election of a leader, and they contain no more random draws.*

Proof (Sketch) If at a certain point both processes have committed to the same channel, then only one of them will be able to perform the input action on that channel, whereas the other one is blocked waiting to perform an input action on the same channel. The process that is able to make the input action will therefore be able to make the second input action too and will become the leader. The other process will finally be enabled to make the input on the channel on which it was blocked, and will receive the notification that the other process has become the leader. Neither processes select the recursive branch and therefore no more random draws are made. □

Lemma 2. *The probability that a sequence of random draws of length n is alternated is $1/2^{n-1}$.*

Proof Obvious, by induction on n, and by the observation that the random draws are independent. □

Proposition 2. *Consider the process $\nu x_0\ \nu x_1(P_0 \mid P_1)$ and the algorithm of Table 3 modified by using the priority choice. The probability of the executions which contain (at least) n random draws, for $n \geq 2$, is at most $1/2^{n-2}$ under every adversary.*

Proof By Lemma 1 the first $n-1$ random draws must be alternated (otherwise the leader would have been elected earlier). By Lemma 2 such alternated sequence has probability $1/2^{n-2}$. Note that the maximum probability $1/2^{n-2}$ corresponds to the worst possible case of an adversary which tries to delay the election of a leader as much as possible, by scheduling the processes in such a way that a process tries to perform the second input only when the channel is not available. □

We are now ready to prove the correctness of our algorithm.

Proposition 3. *Consider the process $\nu x_0\ \nu x_1(P_0 \mid P_1)$ and the algorithm of table 3 (with no modifications). The probability of the executions which contain (at least) n random draws, for $n \geq 2$, is at most*

$$\frac{(1+\varepsilon)^{n-2}}{2^{n-2}}$$

under every adversary.

Proof The proof proceeds like in the proof of Proposition 2, with the exception that we need to consider also the possibility that a leader is *not* elected after a draw which breaks the alternation. Such event occurs with probability ε. The probability that an execution contains $n-1$ draws where the alternation is violated k times is therefore

$$\frac{1}{2^{n-2}} \frac{(n-2)!}{k!(n-2-k)!} \varepsilon^k$$

The sum of these probability for all possible values of k is

$$\frac{1}{2^{n-2}} \sum_{k=0}^{n-2} \frac{(n-2)!}{k!(n-2-k)!} \varepsilon^k \;=\; \frac{(1+\varepsilon)^{n-2}}{2^{n-2}}$$

□

As a consequence of this proposition we finally obtain the correctness of our algorithm:

Theorem 1. *Consider the process $\nu x_0\ \nu x_1(P_0 \mid P_1)$ and the algorithm of table 3 (without modifications). The probability that the leader is eventually elected is* 1 *under every adversary.*

Proof An execution does not elect a leader only if it is infinite and contains an infinite number of random draws. By Proposition 3 the probability of the execution fragments which contain at least n random draws is at most

$$\frac{(1+\varepsilon)^{n-2}}{2^{n-2}}$$

ince $\varepsilon < 1$, this probability converges to 0 for $n \to \infty$. □

We conclude this section with the observation that, if we modify the blind choice to be a choice prefixed with the input actions which come immediately afterward, then the above theorem would not hold anymore. In fact, we can define a scheduler which selects the processes in alternation, and which suspends a process, and activates the other, immediately after the first has made a random choice and performed an input. The latter will be forced (because of the guarded choice) to perform the input on the other channel. Then the scheduler will proceed with the first process, which at this point can only backtrack. Then it will schedule the second process again, which will also be forced to backtrack, and so on. Since all the choices of the processes are obligated in this scheme, the scheduler will produce an infinite (unsuccessful) execution with probability 1.

5 Implementation of π_{pa} in a Java-like language

In this section we propose an implementation of the *synchronization-closed* π_{pa}-calculus, namely the subset of π_{pa} consisting of processes in which all occurrences of communication actions $x(y)$ and $\bar{x}y$ are under the scope of a restriction operator νx. This means that all communication actions are forced to synchronize.

The implementation is written in a Java-like language following the idea outlined in Section 3.1. It is compositional wrt all the operators, and distributed, i.e. homomorphic wrt the parallel operator.

Channels are implemented as one-position buffers, namely as objects of the following class:

```
class Channel {
        Channel message;
        boolean isEmpty;

        public void Channel() {
                isEmpty = true;
        }

        public synchronized void send(Channel y) {
                while (!isEmpty)  wait();
                isEmpty = false;
                message = y;
                notifyAll();
        }

        public synchronized GuardState test_and_receive() {
                GuardState s = new GuardState();
                if (! isEmpty) { s.test = true;
                                 s.value = message;
                                 isEmpty = true;
```

```
                              return s; }
              else { s.test = false;
                     s.value = null;
                     return s; }
       }
}

class GuardState {
        public boolean test;
        public Channel value;
}
```

The methods `send` and `test_and_receive` are used for implementing the output and the input actions respectively. They are both *synchronized*, because the test for the emptyness (resp. non-emptyness) of the channel, and the subsequent placement (resp. removal) of a datum, must be done atomically.

Note that, in principle, the receive method could have been defined dually to the send method, i.e. read and remove a datum if present, and suspend (wait) otherwise. This definition would work for input prefixes which are not in the context of a choice. However, it does not work for input guarded choice. In order to simulate correctly the behavior of the input guarded choice, in fact, we should check continuously for input events, until we find one which is enabled. Suspending when one of the input guards is not enabled would be incorrect. Our definition of `test_and_receive` circumvent this problem by reporting a failure to the caller, instead of suspending it.

Given the above representation of channels, the π_{pa}-calculus can be implemented by using the following encoding $[\![\cdot]\!]$:

Probabilistic choice

$$[\![\sum_{i=1}^{m} p_i x_i(y).P_i + \sum_{i=m+1}^{n} p_i \tau.P_i]\!] =$$

```
{ boolean choice = false;
  GuardState s = new GuardState();
  float x;
  Random gen = new Random();
  while (!choice) {
     x = 1 - gen.nextFloat();  % nextFloat() returns a real number in [0,1)

     if (0 < x <=  p1 )
          { s = x1.test_and_receive();
            if (s.test) { y = s.value; [[ P1 ]]
                          choice = true; }
          }
     ...
```

```
        if (p_1 + p_2 + ... + p_{m-1}  < x <=  p_1 + p_2 + ... + p_m)
             { s = xm.test_and_receive();
               if (s.test) { y = s.value; [[ P_m ]]
                             choice = true; }
             }
        if (p_1 + p_2 + ... + p_m  < x <= p_1 + p_2 + ... + p_{m+1})
             { [[ P_{m+1} ]]
               choice = true; }
        ...
        if (p_1 + p_2 + ... + p_{n-1}  < x <= p_1 + p_2 + ... + p_n)
             { [[ P_n ]]
               choice = true; }
}
```

Note that with this implementation, when no input guards are enabled, the process keeps performing internal (silent) actions instead of suspending.

Output action

$\llbracket \bar{x}y \rrbracket =$ `{x.send(y);}`

Restriction

$\llbracket \nu x P \rrbracket =$ `{Channel x = new Channel();` $\llbracket P \rrbracket$ `}`

Parallel If our language is provided with a parallel operator, then we can just have a homomorphic mapping:

$$\llbracket P_1 \mid P_2 \rrbracket = \llbracket P_1 \rrbracket \mid \llbracket P_2 \rrbracket$$

In Java, however, there is no parallel operator. In order to mimic it, a possibility is to define a new class for each process we wish to compose in parallel, and then create and start an object of that class:

```
class processP1 extends Thread {
          public void run() {
                   [[ P_1 ]]
          }
}
```

$\llbracket P_1 \mid P_2 \rrbracket =$ `{ new processP1.start();` $\llbracket P_2 \rrbracket$ `}`

Recursion Remember that the process $rec_X P$ represents a process X defined as $X \stackrel{\mathrm{def}}{=} P$, where P may contain occurrences of X. For each such process, define the following class:

```
class X {
      static public void exec() {
                $[\![ P ]\!]$
      }
}
```

Then define:

$[\![rec_X P]\!] =$ `{ X.exec(); }`

$[\![X]\!] =$ `{ X.exec(); }`

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. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.
2. Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.
3. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.
4. Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.
5. 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.
6. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.
7. Uwe Nestmann. On the expressive power of joint input. In Catuscia Palamidessi and Ilaria Castellani, editors, *EXPRESS '98: Expressiveness in Concurrency*, volume 16.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B.V., 1998.
8. Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996)*, volume 1119 of *Lecture Notes in Computer Science*, pages 179–194. Springer-Verlag, 1996. Full version to appear in *Information and Computation*.
9. Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–265, Paris, France, 1997.
10. Michael O. Rabin and Daniel Lehmann. On the adantages 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 *Proceedings of POPL'81*, pages 133-138.

11. Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.
12. Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995. An extended abstract appeared in *Proceedings of CONCUR '94*, LNCS 836: 22–25.
13. Rob J. van Glabbeek, Scott A. Smolka, and Bernhard Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.
14. Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 327–338, Portland, Oregon, 1985. IEEE Computer Society Press.
15. Wang Yi and Kim G. Larsen. Testing probabilistic and nondeterministic processes. In *Proceedings of the 12th IFIP International Symposium on Protocol Specification, Testing and Verification*, Florida, USA, 1992. North Holland.

Appendix A

Table 4 presents an equivalent transition system for the π_{pa}-calculus where no assumptions on the bound variables are made. Note that the side condition on the rule SUM is necessary for treating cases like $1/2\ x(y).\mathbf{0} + 1/2\ x(y).\mathbf{0}$. This condition could be eliminated by assuming that the transition groups are multiset instead than sets.

$$\textsc{Sum} \quad \sum_i p_i\alpha_i.P_i\ \{\xrightarrow[p'_i]{\alpha_i} P_i\}_i \qquad p'_i = p_i / \sum_{j:\alpha_j=\alpha_i\ P_j\equiv P^i} p_j$$

$$\textsc{Out} \quad \bar{x}y\ \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$$

$$\textsc{Open} \quad \frac{P\ \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P\ \{\xrightarrow[1]{\bar{x}(y)} P'\}} \qquad x \neq y$$

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

$$\textsc{Par} \quad \frac{P\ \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{P \mid Q\ \{\xrightarrow[p_i]{\mu_i} P_i \mid Q\}_i} \qquad \forall i.\ bn(\mu_i) \cap fn(Q) = \emptyset$$

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

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

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

Table 4. Alternative formulation of the probabilistic transition system for the π_{pa}-calculus.