A randomized encoding of the π-calculus with mixed choice*

Catuscia Palamidessi

INRIA and LIX, École Polytechnique, Rue de Saclay, 91128 Palaiseau, France

Email: catuscia@lix.polytechnique.fr

URL: http://www.lix.polytechnique.fr/~catuscia/

Oltea Mihaela Herescu

IBM Austin Research Lab, 11501 Burnet Road, Austin, TX 78758, USA

Email: olteamh@us.ibm.com

Abstract

We consider the problem of encoding the π-calculus with mixed choice into the asynchronous π-calculus via a *uniform* translation while 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 it is robust to attacks of *proper* adversaries.

1 Introduction

At the end of the Eighties, with the Internet starting to be widely diffused, and with computing architectures becoming increasingly more distributed, there has been a profound change in the area of Concurrency Theory. The classic process calculi, of which CCS ([1]) is one of the most prominent representatives, were inadequate to express the new phenomena and problems that were becoming more and more important, like the dynamic reconfiguration of networks and the security of communication. To cope with these new needs, novel constructs and calculi started to be developed. In 1989 Milner, Parrow and Walker proposed the π-calculus ([2]), that rapidly became popular. In essence, the idea was to enhance CCS by adding mechanisms for passing channel names and for scope extrusion. These new features allow for a simple and elegant description of link mobility (reconfiguration of the logical communication structure) and privacy of communication channels.

*This research has been supported by Projet Rossignol of ACI Sécurité Informatique (Ministère de la recherche et nouvelles technologies).

The π-calculus had inherited from CCS the construct of choice and synchronous communication. Soon it became clear that there was a gap between these primitives and the world of highly distributed systems, where communication is essentially asynchronous and agreement on choices is often difficult to achieve. Shortly after the advent of the π-calculus, [3] and [4] independently proposed an asynchronous version of it, that differed from the original one for the absence of the output prefix (which justifies the name "asynchronous") and for the choice operator.

The asynchronous π-calculus became quickly popular, not only because it seemed more adequate to describe distributed systems and easier to implement, but also because both [3] and [4] provided (independently) elegant encodings of the output prefix, thus proving that synchronous communication could be compiled into the asynchronous π-calculus. Some years later Nestmann and Pierce proved that also the input-guarded choice can be encoded into the asynchronous π-calculus ([5]). This result had a considerable impact, to the point that several authors afterwards have considered presentations of the asynchronous π-calculus containing the input-guarded choice as a primitive (see for instance [6]).

The question of the possibility of encoding the (full) choice operator, however, remained open until 1997, when Palamidessi proved that, under certain conditions, this encoding is impossible ([7]). The conditions are the *uniformity* of the encoding and the preservation of a *reasonable* semantics. The terms "uniform" and "reasonable" have been introduced in [7], but they correspond to standard concepts in Distributed Computing. Uniform means homomorphic with respect to the parallel and the renaming operators, and it amounts to requiring that the translation preserve the degree of distribution and of symmetry of the original system[1]. Reasonable means that the translation should preserve the intended observables on every computation, in particular, it should not introduce livelocks (aka divergences)[2]

The negative result is based on the fact that in the π-calculus we can define an algorithm for solving the leader election problem in a symmetric network, while this is not possible in the asynchronous π-calculus. The crucial point is that in the latter it is not possible to break the initial symmetry of the system, and thus it is not possible to reach a state in which one of the processes is a "leader" while the others are not. In the π-calculus, on the contrary, the symmetry can be broken by using the choice construct. In [8] it is shown that the additional expressive power of the the π-calculus is due exactly to the mixed choice operator, which is characterized by the presence of both input and output guards. Homogeneous choices, i.e. choices with only input or only output guards, can be encoded uniformly without introducing divergences. Note that also the encoding of input-guarded choice provided in [5] respects these conditions.

The additional power provided by mixed choice is useful to solve typical distributed problems which involve agreement. On the other hand, while mixed choice is difficult to implement, the asynchronous π-calculus can be implemented in a fully distributed way (see, for instance, [9]). So, an encoding of mixed choice into the asynchronous π-calculus would be interesting not only from a theoretical point of view, but also from a practical one. Since the negative result of [7] depends crucially on the conditions described above, it is natural to consider how significant these conditions are, and whether we could obtain an encoding by relaxing them a bit.

It is clear that uniformity is related to some useful properties: Maintaining

[1] Distributed means that there is neither centralized control nor shared memory. (Strong) symmetry, when the communication graph form a ring, means that processes are identical, except for the names of the channels, and that are initially in the same state. (Some authors use the term symmetry to refer to *weak* symmetry, where the initial states may be different.) The general definition for arbitrary graphs is more complicated, the interested reader can find it in [7].

[2] There are various definitions of livelock in literature. The one we consider here corresponds to the notion of divergence, and it is the most common.

the degree of distribution is nice because any centralized coordination is a potential source of bottlenecks. Symmetry is nice because it guarantees “equal opportunities” to processes, and because it makes it easier to compose systems and to reason modularly about them. In this paper we focus on exploring an approach that guarantees complete uniformity. However, in our opinion, this condition is not always crucial: It all depends on what kind of features we consider important for a given application. It is worthwhile to investigate also solutions that renounce to full distribution and symmetry in exchange of other advantages, like for instance efficiency. In [8] the interested reader can find a discussion about various encodings that, although not uniform, can be considered satisfactory from a practical point of view.

Livelock freedom is considered crucial in the community of Distributed Algorithms. The principle is that a system that may diverge should not be considered equivalent to a system that guarantees success (or progress) in all possible runs. This attitude contrasts with the situation in the area of Concurrency Theory, where there is a tendency (at least, by a part of the community) to consider livelock freedom as a concept of secondary importance. This may be due to the influence on the community of the weak bisimulation semantics, which is insensitive to silent loops[3]. This tendency to disregard livelock is a bit surprising in the opinion of the authors, given that certain paradigmatic problems coming from the Distributed Algorithms world, like the Dining Philosophers, are well-known and considered important benchmarks also by the Concurrency Theory community. The Dining Philosophers problem would not be a problem at all if livelocks were considered harmless!

Our position on this matter is intermediate between the two extremes above: In our view livelocks matter, *unless we can guarantee that they will be very unlikely.* More precisely, we accept livelocks provided that they will occur with probability 0. This sentence expresses the basic principle of the paper. In summary, we are interested in defining an encoding of the π-calculus with mixed choice into (a variant of) the asynchronous π-calculus (with input-guarded choice). We want to respect the condition of uniformity, but we are ready to weaken the condition of “preserving a reasonable semantics” into “preserving it with probability 1”.

In order to define an encoding with the above described features, we consider as target language a probabilistic extension of the asynchronous π-calculus, π_{pa} ([11]), 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, notably schedulers that “may try to sabotage” the translated processes by forcing them to loop. We propose an encoding that is robust with respect to 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 Section 5.2 we define formally this notion and we argue that it is a reasonable requirement.

In order to prove the correctness of the encoding we consider testing semantics ([13, 14]). This semantics is sensitive to divergences and visible actions, hence it is “reasonable” in the sense of [7]. We will develop a notion of testing semantics

[3]Note however that Milner, who introduced the notion of weak bisimulation in concurrency, did not deny the importance of livelock freedom: he only said that, instead of making the notions of weak bisimulation more complicated, one can prove the absence of livelocks by using other methods [1]. Furthermore, to be fair, there are some works considering versions of weak bisimulations which take divergence into account, notably [10].

suitable for π_{pa}, and we will show that our encoding is correct in the sense that translated processes preserve and reflect, 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 [15, 16, 17], 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, asynchronous versions, because these semantics are "too concrete" for the kind of translation developed here (see Section 5 for more details). On the other hand, they are not sensitive to divergences, so we would anyway have to consider other additional semantic conditions.

The interest in considering π_{pa} as target language lies in the fact that it can be implemented in a distributed and symmetric way relatively easily: 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 proposal for a uniform translation of π_{pa} into a distributed Java-like machine is illustrated in [11]. The condition of uniformity on the encodings of π into π_{pa} and of π_{pa} into Java ensure that distribution and symmetry are preserved, thus we can argue that our results provide an approach to the symmetric and distributed implementation of the π-calculus.

The distributed implementation of mixed choice, also called *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 [18, 19, 20], 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 ([21, 22, 23]). 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)[4]. This is what in distributed computing is called "partial synchrony hypothesis". As for the solution proposed in [8], it 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 Section 3 we will argue that the algorithms of [21, 22, 23] do not work when the processes proceed at independent speed, by showing an example of network and adversary for which any attempt to synchronize produces a livelock. The relation with [8] is discussed in Section 4. 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 (full asynchrony) and it is robust with respect to any proper adversary. Full asynchrony is a natural hypothesis in case of distributed systems. As for the robustness with respect to adversaries, its importance is well described in [24]: "*We allow for the possibility of an adversary scheduler since we assume that the interactions we describe* [...] *are only the visible part of an iceberg of complex relations about which we do not know and that we are not willing to study. We are to assume that the worst may happen, which is a very sound principle of system design.*".

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

[4]In the case of [21] it is not clear to us what is the exact assumption. The authors phrase it as follows: "[We assume that] the behavior of a waiting process does not depend on the choice of partners made by other processes".

[24], but the implementations of concurrent programming languages (for instance Java) usually do not guarantee a fair scheduling policy.

The rest of the paper is organized as follows: Next section recalls the basic notions and definitions about the π-calculus, the probabilistic automata, and the probabilistic asynchronous π-calculus. Section 3 illustrates an example of binary interaction problem and discusses why the solutions in [21, 22, 23] do not work when we remove the assumption of partial synchrony. In Section 4 we define a uniform, compositional encoding from the π-calculus with mixed choice into the probabilistic asynchronous π-calculus. In Section 5 we define a probabilistic extension of the testing semantics and we show the correctness of the encoding. In Section 6 we briefly discuss the properties of the encoding.

Part of the material of this paper appeared already in [25]. More precisely, [25] contains most of the technical definitions and the statements of the main results. The parts which appear only in this paper are: the first half of the introduction, parts of the informal explanations, some auxiliary definitions, the lemmata, the proofs, and the entire Section 3.

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 [26]. The main difference with respect to the original version ([2, 27]) is 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:

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

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 $\mathbf{0}$ (*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. The construct $[x=y]$ is the *match* operator: the process $[x=y]\,P$ is the process that behaves like P if x and y are the same name, otherwise it suspends. The process $rec_X P$ represents a process X defined as $X \stackrel{\mathrm{def}}{=} P$, 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

Sum	$\sum_i \alpha_i.P_i \xrightarrow{\alpha_j} P_j$		Res	$\dfrac{P \xrightarrow{\mu} P'}{\nu y P \xrightarrow{\mu} \nu y P'}$	$y \notin names(\mu)$
Open	$\dfrac{P \xrightarrow{\bar{x}y} P'}{\nu y P \xrightarrow{\bar{x}(y)} P'}$	$x \neq y$	Par	$\dfrac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}$	$bn(\mu) \cap fn(Q) = \emptyset$
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	$\dfrac{P \equiv P' \quad P' \xrightarrow{\mu} Q' \quad Q' \equiv Q}{P \xrightarrow{\mu} Q}$		Rec	$\dfrac{P[rec_X P/X] \xrightarrow{\mu} P'}{rec_X P \xrightarrow{\mu} P'}$	

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

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 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, which induce two different semantics: the *early* and the *late* bisimulation semantics ([27]). Here we present the second one. We had made this choice because the late semantics is more refined, hence more challenging, in principle, for obtaining positive embedding results. In practice however the choice between the two semantics does not make any difference for the correctness of our embedding.

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 standard rules: associative monoid rules for $|$, the commutativity of the summands for Σ, the alpha-conversion, and the following:

- $[x = x]\, P \equiv P$,
- $(\nu x\, P) \mid Q \equiv \nu x\, (P \mid Q)$ if $x \notin fn(Q)$,
- $\nu x\, 0 \equiv 0$,
- $\nu x\, \nu x\, P \equiv \nu x\, P$,
- $\nu x\, \nu y\, P \equiv \nu y\, \nu x\, P$,
- $\nu x\, x(y).P \equiv \nu x\, \bar{x}y.P \equiv 0$.

2.2 Probabilistic automata, adversaries, and executions

Asynchronous automata have been proposed in [12]. 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 finite or countable 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 the sets of all probabilistic spaces on Y as

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

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 [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). An automaton in which there is at most one transition group for each state is called *fully probabilistic*. Figures 1 and 2 give examples of a probabilistic and a fully probabilistic automaton, respectively.

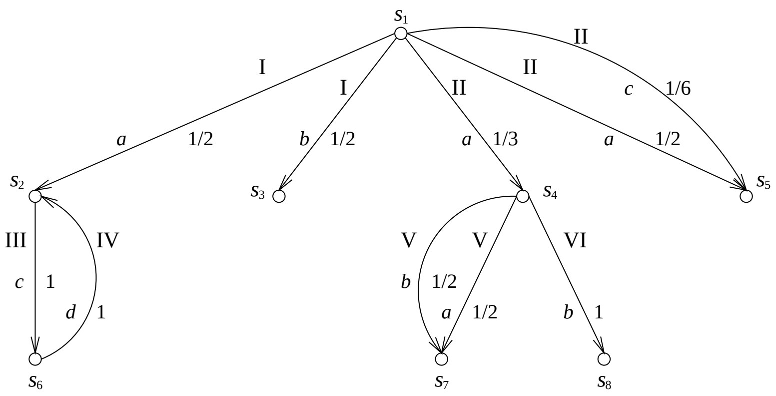


Figure 1: Example of a probabilistic automaton M. The transition groups are labeled by I, II, ..., VI

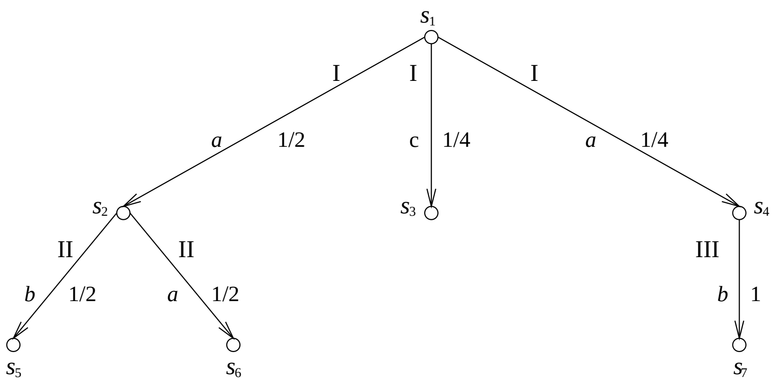


Figure 2: A fully probabilistic automaton

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 that 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* with respect to a certain class of adversaries if it achieves its 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 $tree(M)$ 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 $nodes(M)$ the set of nodes in $tree(M)$, and by $state(n)$ the state labeling a node n. Example: Figure 3 represents the tree obtained from the probabilistic automaton M of Figure 1.

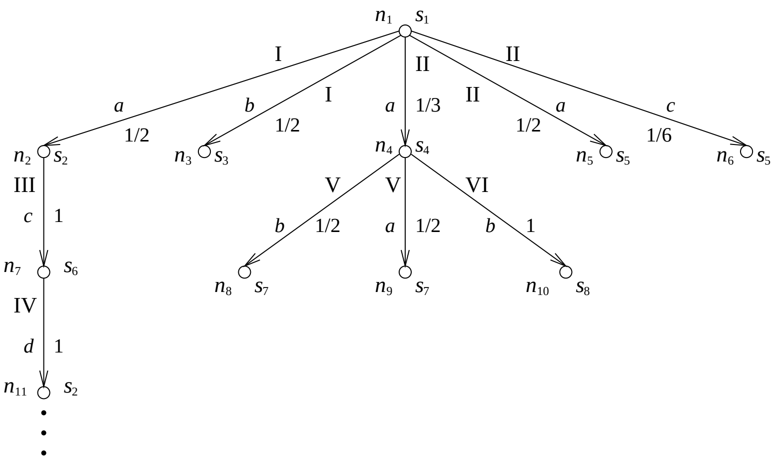


Figure 3: $tree(M)$, where M is the probabilistic automaton M of Figure 1

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

The *execution tree* of an automaton $M = (S, \mathcal{T}, s_0)$ under an adversary ζ, denoted by $etree(M, \zeta)$, is the tree obtained from $tree(M)$ by pruning all the arcs corresponding to transitions which are not in the group selected by ζ. More formally, $etree(M, \zeta)$ is a fully probabilistic automaton $(S', \mathcal{T}', n_0)$, where $S' \subseteq nodes(M)$, n_0 is the root of $tree(M)$, and $(n, (X', pb')) \in \mathcal{T}'$ iff $X' = \{(\mu, n') \mid (\mu, state(n')) \in X$ and n' is a child of n in $tree(M)\}$ and $pb'(\mu, n') = pb(\mu, state(n'))$, where $(X, pb) = \zeta(n)$. If $(n, (X', pb')) \in \mathcal{T}'$, $(\mu, n') \in X'$, and $pb'(\mu, n') = p$, we will use sometime the notation $n \xrightarrow[p]{\mu} n'$. 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 $etree(M, \zeta)$. 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 $exec(M, \zeta)$ the set of all executions in $etree(M, \zeta)$.

We define now a probability on certain sets of executions, following a standard construction of Measure Theory. Given an execution fragment ξ, let $C_\xi = \{\xi' \in exec(M, \zeta) \mid \xi \leq \xi'\}$ (*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 $pb(\bigcup_{i \in I} C_i) = \sum_{i \in I} pb(C_i)$. Two countable sets of disjoint cones with the same union produce the same result for pb, so pb is well defined. Further, we define the probability of an empty set of executions as 0, and the probability of the complement of a certain set of executions, with respect to the all executions as the

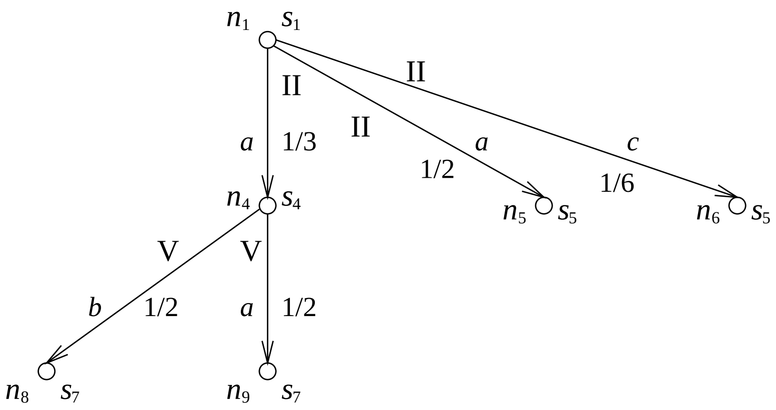


Figure 4: etree(M, ζ), where M is the probabilistic automaton M of Figure 1, and (the significant part of) ζ is defined by $\zeta(n_1) = \text{II}$, $\zeta(n_4) = \text{V}$

complement with respect to 1 of the probability of the set. The closure of the cones (plus the empty set) under countable unions and 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} ([11]). This calculus is a probabilistic extension of the asynchronous π-calculus ([3, 4]), whose fundamental feature is the absence of the output prefix construct. This is why the calculus is called "asynchronous". We use the presentation of the asynchronous π-calculus which contains the input-guarded choice as a primitive, in contrast to the original version which is choiceless. As explained in the introduction, the difference is irrelevant with respect to expressiveness.

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

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

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 $\sum_i p_i = 1$, and the α_i's are input or silent prefixes.

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

Sum	$\sum_i p_i\alpha_i.P_i \; \{\xrightarrow[p_i]{\alpha_i} P_i\}_i$	Out	$\bar{x}y \; \{\xrightarrow[1]{\bar{x}y} \mathbf{0}\}$
Res	$\dfrac{P \; \{\xrightarrow[p_i]{\mu_i} P_i\}_i}{\nu y P \; \{\xrightarrow[p'_i]{\mu_i} \nu y P_i\}_{i:y\not\in fn(\mu_i)}}$		$\exists i.\ y \not\in fn(\mu_i)$ and $\forall i.\ p'_i = p_i / \sum_{j:y\not\in fn(\mu_j)} p_j$
Open	$\dfrac{P \; \{\xrightarrow[1]{\bar{x}y} P'\}}{\nu y P \; \{\xrightarrow[1]{\bar{x}(y)} P'\}} \quad x \neq y$	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	$\dfrac{P \equiv P' \qquad P' \; \{\xrightarrow[p_i]{\mu_i} Q'_i\}_i \qquad \forall i.\ Q'_i \equiv Q_i}{P \; \{\xrightarrow[p_i]{\mu_i} Q_i\}_i}$		
Rec	$\dfrac{P[rec_X P/X] \; \{\xrightarrow[p_i]{\mu_i} P'_i\}_i}{rec_X P \; \{\xrightarrow[p_i]{\mu_i} P'_i\}_i}$		

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.

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 $\alpha_i.P_i \not\equiv \alpha_j.P_j$[5]. Furthermore, in Res and Par we assume that all bound variables are distinct from each other, and from the free variables.

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 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

[5]Without this assumption we would need to define transition groups to be multisets instead of sets, see for instance [28].

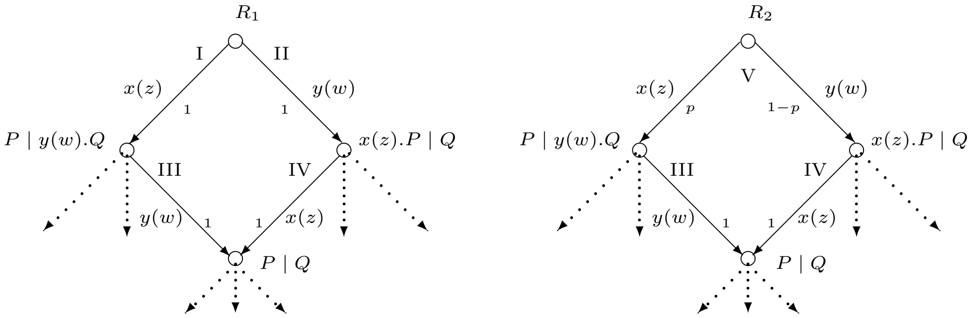


Figure 5: The probabilistic automata R_1 and R_2 of Example 2.1. The transition groups from R_1 are labeled by I and II respectively. The transition group from R_2 is labeled by V.

other rules, OUT and CONG, should be self-explanatory.

Concerning the structural equivalence used in CONG, we assume the same rules as for the π-calculus (cfr. 2.1), except for the last one, the rule for collecting garbage, which is replaced by the following two:

- $\nu x\,((p\,x(y).P + \sum_i p_i\,\alpha_i.P_i) \mid R) \equiv \nu x\,((\sum_i p_i/(1-p)\,\alpha_i.P_i) \mid R)$ if R does not contain a complementary output action on x.
- $\nu x\,(\bar{x}y \mid R) \equiv \nu x R$ if R does not contain a complementary input action on x.

Example 2.1 Let $R_1 = x(z).P \mid y(w).Q$ and $R_2 = p\ x(z).(P \mid y(w).Q) + (1 - p)\ y(w).(x(z).P \mid Q)$. The transition groups starting from R_1 are:

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

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

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

Figure 5 illustrates the probabilistic automata corresponding to R_1 and R_2.

Previous 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.

3 An example of the binary interaction problem

In order to illustrate the difficulty of implementing the mixed choice construct in presence of full asynchrony and adversary schedulers, we show that the algorithm of [23], which is correct and livelock free under certain partial synchrony conditions, may give rise to a livelock if we remove these conditions. Similar examples can be constructed for [21, 22]. We will discuss about the latter ones at the end of this section.

We start by recalling the algorithm proposed by [23], restricted to the case of binary (aka two-way) interaction. In the algorithm, each possible binary interaction

```
while trying do     {
        randomly choose an interaction; let X be the associated variable
        if TEST&SET(X, dec, inc) = 1
                then  participate to the interaction
                else  {    wait(t);
                           if TEST&SET(X, no_op, dec) = 0
                                then  participate to the interaction
                                /* else try another interaction */
                      }
}
```

Table 3: The algorithm for binary interaction proposed in [23]. The notations *inc*, *dec*, and *no_op* mean, respectively: add 1, subtract 1, and no operation.

is associated to a variable that ranges over 0 and 1. The variable can be accessed only by the processes interested in the interaction, via a test-and-set function of the following kind:

$\texttt{TEST\&SET}(X, op, op')$

which means: Read the value of X. If it is 0, then apply op to X. Otherwise, apply op'. In both cases, return the value of X before the operation. These actions (read and set) are meant to be executed *atomically*, i.e. as an indivisible sequence. Originally, all variables are set to 0.

The code executed by each process interested in interacting is shown in Table 3. The idea is that each process P ready to interact chooses randomly one of the possible interactions, and tests the corresponding variable X. If the value of X is 0 then P sets X to 1 and waits for a time t (timeout). Then P tests X again. If the value has changed (to 0), meaning that the partner has chosen the same interaction, then the interaction is started. Otherwise, P resets X to 0 and tries a new interaction, possibly with a different partner. On the contrary, if at the first test X was 1, then it means that the partner is willing to interact. In this case P sets X to 0 to signal to the partner its positive response, and starts the interaction.

We show now that the algorithm can produce a livelock. Consider a network consisting of three parallel processes A, B, and C, connected in the way illustrated in Figure 6.1. The scheduler selects a process, say A. Assume, without loss of generality, that A chooses the interaction A–B. Then A sets the corresponding variable to 1 (Figure 6.3) and waits. At this point, the scheduler selects the process *that does not have any adjacent variable set to* 1, in this case C. Assume, without loss of generality, that C chooses the interaction C–B, sets the corresponding variable to 1 (Figure 6.5) and waits. At this point, since we are are not constrained by any assumption about the relative execution time, we can assume that the scheduler has been very slow in executing C, so that the timeout of A has expired. Then A is selected again, and it must reset the variable of the interaction A–B to 0 and go back to the initial state (Figure 6.6). The same is done with C (Figure 6.7). Finally, the scheduler selects the process which has not been executed so far, in this case B. (This last step is to ensure fairness, i.e. that we have a livelock even in presence of the fairness assumption.) The final situation, represented in Figure

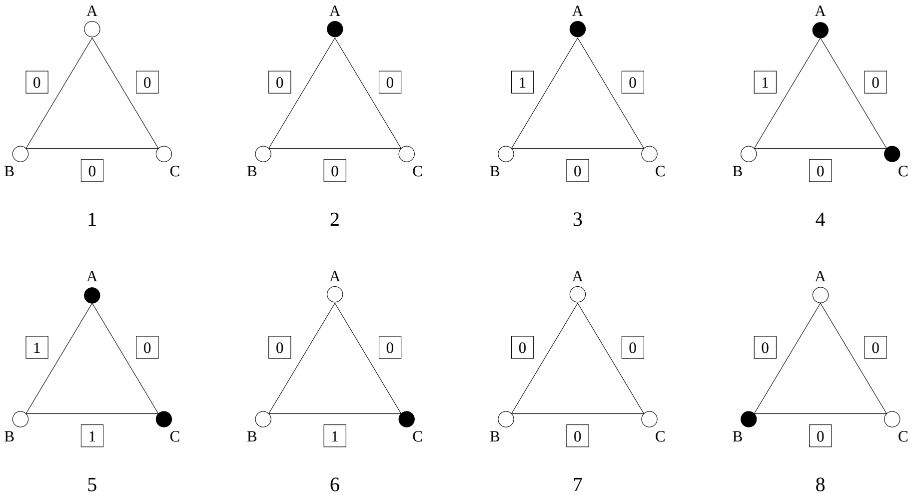


Figure 6: A case of livelock for the algorithm in Table 3. The states at the beginning of Lines 3, 5 and 6, in which the process has chosen the interaction to attempt, are represented with a filled circle. The states at the beginning of Line 1, 2 and 8 are represented with a white circle. Lines 4 and 7 are never reached.

6.8, is symmetric to Figure 6.2, hence these operations can be repeated again and again, thus creating a livelock. Note that, under the assumption of partial synchrony adopted in [23], the counterexample does not hold because the execution of C cannot be arbitrarily long, hence we cannot guarantee the transition from the state in Figure 6.5 to the one in Figure 6.6.

We now consider the algorithms given in [21] and [22]. The one in [21] is essentially the same as the algorithm that we have just discussed. In fact the authors of [23] have followed the idea of [21] for the binary case. The new contribution of [23] is the extension of the algorithm to the multiway case.

As for the algorithm given in [22], the basic idea is the same, but there is a difference that, according to the authors, ensures a better performance. The difference consists in the fact that when a process checks for the presence of a partner willing to interact, it checks for a set of them, not just one. In [22] is is shown that the greatest performance is obtained in correspondence of a certain cardinality m for this set.

More precisely, the algorithm works as follows: at each turn, the process chooses randomly a value b in $\{0, 1\}$. Depending on the value of b, the process proceeds in two different ways. In the first case, the process checks m interaction possibilities, chosen randomly. If the partner of one of these has expressed interest for the same interaction (by setting to 1 the corresponding variable X), then the interaction takes place. In the second case, the process chooses randomly an interaction, sets the corresponding variable to 1, and waits, until the timeout t expires, for the variable to be set back to 0 by the partner, in which case the interaction takes place.

The algorithm is given in Table 4. We have slightly modified the original formulation in order to avoid explaining all the variables and primitives used in [22]. These modifications, however, are inessential. It is easy to see that the example in Table 6 constitutes a case of livelock also for this algorithm.

```
while trying do    {
        randomly choose b ∈ {0,1}
        if b = 0
                then  for    i = 1  to m   do  {
                                 randomly choose an interaction;
                                 let X be the associated variable;
                                 if  TEST&SET(X, dec, no_op) = 1
                                        then  participate to the interaction;
                             }
                else   {        randomly choose an interaction;
                                 let X be the associated variable;
                                 X := 1;;
                                 while X = 1 do  wait(t);
                                 if      TEST&SET(X, no_op, dec) = 0
                                        then  participate to the interaction;
                       }
}
```

Table 4: The algorithm for binary interaction proposed in [22]. We have slightly changed the formulation so to be able to reuse the primitives explained for the previous algorithm instead of introducing new ones.

4 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. This assumption makes the encoding simpler, and could be guaranteed, for example, by a suitable type system.

The main difficulty of course consists in encoding the choice operator. We follow an idea used by Nestmann in [8], which consists in associating a lock l, initially set to *true*, to each choice construct, and then launch a parallel process for each branch of the choice. 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 [8] 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 order to avoid this problem, in [8] 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).

In contrast to [8], we assume that a scheduler is given nondeterministically and that it is arbitrary (except for an assumption of "proper" behavior that will be explained later). Then, in order to make the algorithm robust with respect to every scheduler, 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 ([24]). The forks, in this case, are the locks. The idea is to let the process choose randomly the first lock, and wait until it becomes available. This algorithm has been proved deadlock and livelock free with probability 1 under any fair adversary for the case in which the connection graph is a ring ([24])[6]. The connection graph is defined as the graph whose nodes are the forks and whose edges are the philosophers.

The problem of the mixed choice however still presents a complication: in general the connection graph resulting from the translation of a π-calculus process may be more complicated than a simple ring, and in [29] it has been shown that the classic algorithm of Lehmann and Rabin does not work for general graphs. More precisely, it works only if the graph does not contain two distinct cycles connected by a path. This condition is both necessary and sufficient. In order to cope with this problem, we make sure that, even though the connection graph may have a general structure, only a subset of the processes are allowed to compete for the "forks" at a time, and that these processes form a subgraph that respects the above condition. To this purpose, we associate to each choice containing output guards an additional lock h. The processes corresponding to the input branches then 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. We will see that this property is sufficient to ensure the condition above (absence of connecting paths among distinct cycles), 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 $\mathbf{t}$ and $\mathbf{f}$ and an if-then-else construct, which is defined by the structural rules

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

As discussed in [5], these features can be encoded into π_a. For instance, polyadic communication can be translated into the monadic one by first passing a new private channel, and then performing a series of communications, one for each parameter, on that channel. Note that it is necessary that the parameters be sent in the same order in which they are expected by the receiver. In a calculus provided with output prefix this would be immediate, in π_a it takes some ingeniousness, but it can be done (see [5] for details). Note that one needs also to assume that the polyadic source is well-typed, i.e. that there will never be mismatch between the number of parameters (arity) in two complementary communication actions. In the translated process, in fact, such a mismatch could introduce deadlock (a communication sequence could be started but not terminated). This is not a problem for the purposes of this paper, however, because it is easy to see that all the polyadic actions performed in the encoding respect the arity.

The encoding of π into π_{pa} is defined in Table 5. We remark 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 simultaneously in an interaction attempt). In the encoding of the input prefix, l represents the local principal

[6] In [24] the authors also assume that the adversary cannot decide its strategy on the basis of the future random choices, although it may have complete visibility of the past, including the past random choices. This assumption is implicit in our notion of adversary as defined in Section 2.2.

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

Table 5: 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 small positive real number (smaller than 1). The names l, h, a and r are fresh.

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 l, 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, and will wait for its availability. This commitment is essential for the termination of the algorithm: If the process checked fist for the availability of the lock, then it would be easy to construct a scheduler, even a fair one, that induces a livelock. We illustrate a possible such scheduler in the case of a ring. First, the scheduler selects the a process P_0 and lets it choose the first lock, say, the right one l_0. Then the scheduler selects the adjacent process P_1 immediately to the right of l_0. This second process cannot choose l_0, hence it will have to eventually choose its right lock l_1. Then the scheduler selects the next process to the right, P_2, and so on, until all processes are in the situation of holding their right lock. At this point, the scheduler selects P_0 again. P_0 will try the second lock and fail, then it will release r_0, then will try again to get the first lock, and since the one to its left is still unavailable, it will have to choose l_0 again. Then the scheduler will select P_1, and so on.

The distribution of the probabilities on the top-level choice, on the contrary, is not essential for termination. However, such distribution would probably affect the efficiency, i.e. how soon the synchronization protocol will converge. We conjecture that in the top-level choice it is best 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[7]. In fact, one would have to use either a transition system with negative premises, see for instance [30], or enrich the transition relation with guesses of offers from the environment, see [31]. 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.

5 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 [13, 14]. 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 [8], which arises with semantics like bisimulation, barbed bisimulation, and coupled bisimulation, even in their weak and asynchronous versions. The kind of encoding that we use for choice cannot be correct with respect to 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 an action (input or output, it does not matter. For economy of notation we will assume it to be an input) on a special name ω. We assume this name to be different from all those occurring in 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 $P \mid O$:

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

[7]The kind of priority choice that we need here, with an input guard on the first branch and with no guard in the second branch, would however be easy to implement directly in a language like, for instance, Java.

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 $Q_i \xrightarrow{\omega}$. Note that, in both cases, we only need to reach a state Q_i where the action ω is enabled, we don't need to execute it. Reaching such a Q_i is regarded as *success*. 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.

5.1 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 may testing (resp. must testing) is that the probability that an interaction be successful is positive (resp. is 1). To this end, we need to consider the probability of successful executions *relatively* to those executions which are interactions. We will use the standard mathematical notation for *relative probability*: $pb(A|B)$ represents the probability that the event A happens, given that the event B happens.

This notion can be formalized in two different, but equivalent ways:

- Define an interaction as a branch of the execution tree of $P \mid O$ under some ζ, with the property that all arcs of the branch are labeled by τ or ω. Then define the *relative probability* $pb(\xi \textit{ is successful} \mid \xi \textit{ is an interaction})$ in the standard way, namely as

  $$pb(\xi \textit{ is an interaction} \wedge \xi \textit{ is successful})/pb(\xi \textit{ is an interaction}).$$

  Note that by definition $pb(\xi \textit{ is an interaction} \wedge \xi \textit{ is successful}) = pb(\xi \textit{ is successful})$.

- Restrict the execution tree to contain only arcs labeled by τ and by ω. This can be done by closing the initial process $P \mid O$ on all the free names except ω. Then define the relative probability $pb(\xi \textit{ is successful} \mid \xi \textit{ is an interaction})$ as the probability of the successful branches of this tree.

The first solution is more elegant, but it is formally more complicated since it involves computing two measures in the execution tree. Hence we follow the second approach.

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 $Q_i \xrightarrow[p]{\omega}$. Additionally, ξ is *minimal successful* if its last state is the first Q_i in ξ such that $Q_i \xrightarrow[p]{\omega}$. We denote by $sexec(\nu(P|O), \zeta)$ the set $\{\xi \in exec(\nu(P|O), \zeta) \mid \xi \textit{ is successful}\}$, and by $msexec(\nu(P|O), \zeta)$ the set $\{\xi \in exec(\nu(P|O), \zeta) \mid \xi \textit{ is minimal successful}\}$.

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

Proposition 5.1 *Given an adversary ζ, the set $sexec(\nu(P|O), \zeta)$ can be obtained as a countable union of disjoint cones.*

Proof For every $\xi \in sexec(\nu(P|O), \zeta)$, let $min(\xi)$ be the prefix of ξ which ends at the first i such that $Q_i \xrightarrow[p]{\omega}$. Let us consider the set of executions $\mathcal{C}(\nu(P|O), \zeta)$ defined as $\mathcal{C}(\nu(P|O), \zeta) = \{C_{min(\xi)} \mid \xi \in sexec(\nu(P|O), \zeta)\}$. Clearly we have $\mathcal{C}(\nu(P|O), \zeta) = \{C_\xi \mid \xi \in msexec(\nu(P|O), \zeta)\}$, and that $\mathcal{C}(\nu(P|O), \zeta)$ is a set of disjoint cones (see Section 2.2 for the definition of cone) Furthermore, $\cup_{C \in \mathcal{C}(\nu(P|O), \zeta)} C = sexec(\nu(P|O), \zeta)$.

Countability follows from the fact that $etree(\nu(P|O), \zeta)$ is finitely branching. □

As a consequence of this proposition, the probability of $sexec(\nu(P|O), \zeta)$ is well defined (cfr. Section 2.2), and can be computed by adding the probabilities of the cones of the partition $\mathcal{C}(\nu(\mathcal{P}|\mathcal{O}, \zeta))$:

$$pb(sexec(\nu(P|O), \zeta)) = \Sigma_{C \in \mathcal{C}(\nu(P|O), \zeta)}\, pb(C) = \Sigma_{\xi \in msexec(\nu(P|O), \zeta)}\, pb(\xi).$$

Note that a minimal successful ξ is always finite, hence its probability can be computed as the (finite) product of the probabilities of its steps.

Definition 5.2 Let $\mathcal{A}$ be a class of adversaries. Let P, Q be π_{pa} processes and O be a π_{pa} observer.

(i) $P\ \mathbf{may}_{\mathcal{A}}\ O$ iff there exists an adversary $\zeta \in \mathcal{A}$ for P s.t. $pb(sexec(\nu(P|O), \zeta)) > 0$.

(ii) $P\ \mathbf{must}_{\mathcal{A}}\ O$ iff for all adversaries $\zeta \in \mathcal{A}$ for P, $pb(sexec(\nu(P|O), \zeta)) = 1$.

(iii) $P \simeq_{\mathcal{A}} Q$ 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 $Q\ \mathbf{must}_{\mathcal{A}}\ O$.

Note that, although $P\ \mathbf{must}_{\mathcal{A}}\ O$ implies $P\ \mathbf{may}_{\mathcal{A}}\ O$ (for $\mathcal{A} \neq \emptyset$), must-equivalence does not imply may-equivalence. Hence it makes sense to require both in the definition of $\simeq_{\mathcal{A}}$. As an example, consider $P = \bar{x} \mid rec_X \tau.X$ and $Q = \bar{y} \mid rec_X \tau.X$. We have that, for every O, and for a class $\mathcal{A}$ containing unfair adversaries, $P\ \cancel{\mathbf{must}}_{\mathcal{A}}\ O$ and $Q\ \cancel{\mathbf{must}}_{\mathcal{A}}\ O$. Hence P and Q are must-equivalent, but obviously they are not may-equivalent, in fact $P\ \mathbf{may}_{\mathcal{A}}\ x.\omega$ while $Q\ \cancel{\mathbf{may}}_{\mathcal{A}}\ x.\omega$.

5.2 Correctness of the encoding with respect to 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. This condition reflects what is called "reliable point-to-point communication" in the field of Distributed Algorithms, and it is reflected in Part (i) of the following definition. Part (ii) is due to a technical subtlety in the definition of testing semantics.

Definition 5.3 An adversary ζ for P is *proper* if, whenever P evolves into a process of the form $\nu x_1 \ldots \nu x_k(P_1 | \ldots | P_n)$, then

(i) if P_i is an output action $\bar{y}x_i$, P_j is of the form $p\, y(z).Q + \ldots$, and ζ selects infinitely often P_j, then P_i and P_j will eventually be scheduled for handshaking. Namely, P_i, P_j will be in the premise of a COM or CLOSE rule.

(ii) if P_i is of the form $\bar{a}(\mathbf{t})$ and P_j is of the form $(a(b)\ \mathit{if}\ b\ \mathit{then}\ \omega\ \mathit{else}\ \mathbf{0})$, then P_i and P_j will eventually be scheduled for handshaking.

We will denote by $\mathcal{P}$ the class of *proper* adversaries.

Note that the the above definition coincides with (weak) fairness for the processes in (ii), but in general it is weaker than fairness. For instance, it does not prevent, that in $\bar{x} \mid x.P \mid rec_X \tau.X$ the process $\bar{x} \mid x.P$ be delayed forever: it will be delayed, in fact, under a proper adversary which never selects neither $\bar{x}$ nor $x.P$. The above definition, therefore, is considerably weaker than the notion of fair scheduler, which requires that *any* process which is ready infinitely often will eventually be scheduled for execution. The situation that we want to avoid, with the notion of proper adversary, is that in $\bar{x} \mid rec_X(1-\epsilon)x.P + \epsilon\tau.X$ the synchronization on x be systematically disregarded by the scheduler (remember that the scheduler, in principle, could do that, because it can choose between COM (or CLOSE) and PAR). A proper adversary dealing repeatedly with an input on x, in a situation in which $\bar{x}$ is available, should eventually choose COM (or CLOSE). In other words, we want that the execution in which the synchronization on x never happens, in $\bar{x} \mid rec_X(1-\epsilon)x.P + \epsilon\tau.X$, have probability 0. Note that in an implementation in which we would use directly priority guards, instead of the above approximation, we would not need the restriction to proper adversaries.

Clearly, the fairness assumption would *a fortiori* be sufficient for our encoding, however it is not necessary. This may seem surprising, since the solution to the dining philosophers proposed in [24] *requires* fairness. However, a careful analysis of the algorithm in [24] 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 was replaced by a suspension command (obliging the scheduler to select another process) then the fairness assumption would not be necessary[8]. The same result was independently found by the authors of [32].

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 Table 5 $[\![x(y).P]\!]$ the acquisition of 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.

[8] We are referring here to the first algorithm of [24], the one which is deadlock-free and livelock-free, but not necessarily lockout-free. For lockout freedom the fairness hypothesis cannot be eliminated.

Another important ingredient of the correctness proof 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 [29] that this is a necessary condition for the algorithm of [24] to be livelock-free, even under the fairness hypothesis.

Definition 5.4 Let P be a π-calculus process. Let ζ be any adversary, and let ξ be an execution of $[\![P]\!]$ with respect to ζ. We define the set of graphs corresponding to the *interaction attempts* of ξ, $\mathit{InterAttempt}(\xi)$, inductively as follows.

Base $G_0 \in \mathit{InterAttempt}(\xi)$ is constructed as follows: Consider the prefix ξ_0 of ξ which ends at the first synchronization on a channel occurring in P if such synchronization happens, otherwise set $\xi_0 = \xi$. G_1 will contain all the edges between any two principal locks l and l' such that there is a process (corresponding to an input branch in P) for which both input actions on l and l' are enabled at some point of ξ_0. If $\xi_0 = \xi$ then the construction terminates, otherwise let ξ_0' be the rest of ξ (without ξ_0). We proceed with the inductive step as follows:

Inductive step $G_n \in \mathit{InterAttempt}(\xi)$ is constructed as follows: Consider the prefix ξ_n of ξ_{n-1}' which ends at the first synchronization on a channel occurring in P if such synchronization happens, otherwise set $\xi_n = \xi_{n-1}'$. G_n is defined as G_{n-1} minus the edge corresponding to the synchronization that has taken place at the end of ξ_{n-1}, plus all the edges between two any principal locks l and l' such that there a process (corresponding to an input branch in P) for which both input actions on l and l' are enabled at some point of ξ_n. If $\xi_n = \xi$ then the construction terminates, otherwise let ξ_n' be the rest of ξ_{n-1}' (without ξ_n), and continue with the inductive step.

Let us also recall the formal definition of cycle: Given an undirected graph, a cycle is a path that starts and ends at the same node. In the following, we will restrict ourselves to the case of simple cycles, which are defined as cycles not containing any proper subcycle. This restriction is not necessary, it is only convenient, to our opinion, to make the proofs more intuitive.

Lemma 5.5 *Let P be a π-calculus process. Let ζ be any adversary, and let ξ be an execution of $[\![P]\!]$ with respect to ζ. For any $G \in \mathit{InterAttempt}(\xi)$, we have that all cycles in the graph are disconnected, i.e. for any pair of different cycles there are no paths connecting one node of the first to one node of the second.*

Proof We reason by contradiction. Assume that it is possible to have a graph with two cycles connected by a path. Then we are in one of the three cases illustrated in Figure 7:

Configuration 1: There are at least two nodes which belong to both cycles.

Configuration 2: There is exactly one node which belong to both cycles.

Configuration 3: There are no nodes which belong to both cycles.

Clearly these three cases cover all possible situations. Note that in the first two cases the shortest path between the two cycles has length 0, while in the third case it has positive length.

The proof proceeds by considering the auxiliary locks h that the input branches must get in order to be active, i.e. to participate to the competition for the principal locks l. The crucial point is that initially there is exactly one h per node, and this

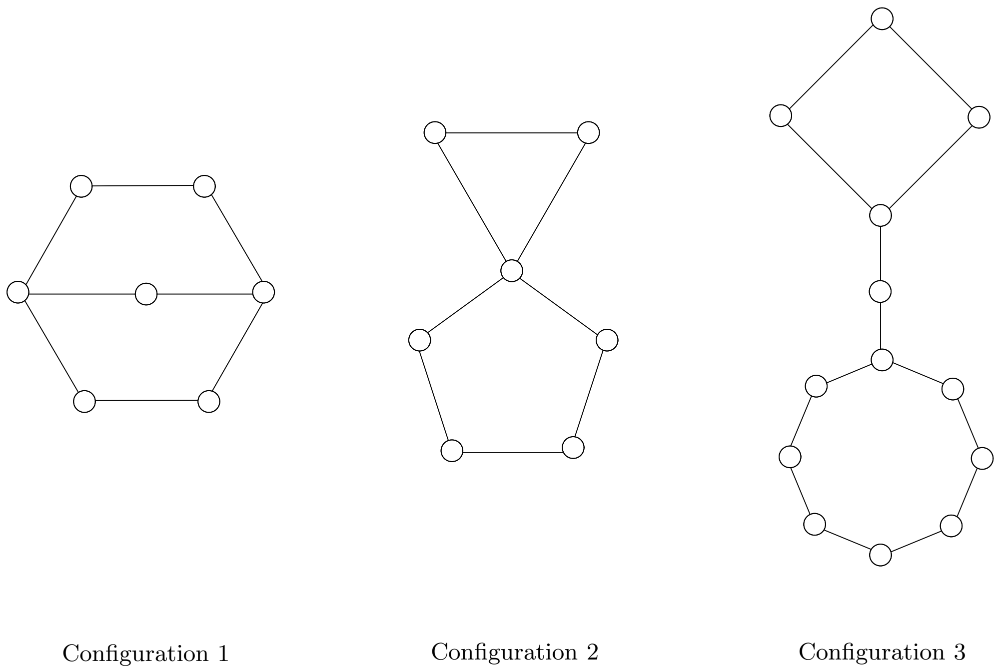


Figure 7: Cycles connected by a path: The three possible configurations

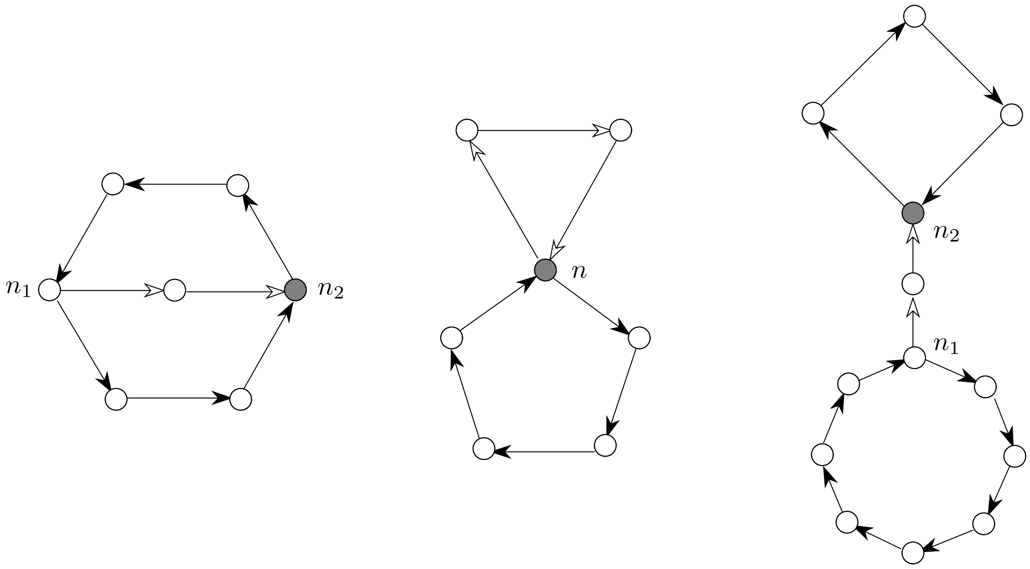
implies, intuitively, that we won't have enough active input branches to create one of the situations above.

In order to prove this intuition formally, it is convenient to consider an orientation on the edges: we stipulate that an edge goes from l to l' if the corresponding input branch has got the auxiliary lock from the node of l'. The fact that there is only one h per node implies that, in the directed graph, the in-degree of a node (i.e. the number of arcs coming into the node) is at most 1.

The rest of the proof consists in showing that the limitation of the in-degree to 1 is incompatible with the presence of cycles connected by a path. To show this incompatibility we proceed by case analysis on the three possible configurations of Figure 7. Our reasoning is illustrated in Figure 8. The gray nodes in the Figure 8 represent points where we find a contradiction to the assumption on the in-degree.

Let us start with Configuration 2, which is the simplest. Let n be the node in common. Consider the first cycle, represented by the black arrows (edges) in Figure 8. Clearly each node in this cycle, including n, must have exactly one incoming and one outcoming black arrow (if one node of the cycle had no incoming arrows, then another node should have two incoming arrows, which is impossible). Consider now the second cycle, represented by the white arrows. Also in this case we must have exactly one incoming and one outcoming white arrow for every node. Hence we conclude that the node n must have two incoming arrows, one for each cycle, which is impossible given that we had only one auxiliary lock h on n.

Consider now Configuration 1. Consider again the first cycle, represented by the black arrows. For the same reason as before, each node in this cycle must have exactly one incoming and one outcoming black arrow. Consider now the second cycle. Since the two cycles are distinct, there must be one or more edges in the second cycle which are not part of the first one. These are represented by white arrows in Figure 8. Since the two cycles have nodes in common, there must be



Configuration 1 Configuration 2 Configuration 3

Figure 8: Cycles connected by a path proved incompatible with the condition that the incoming degree of a node be at most 1

one node, n_1, adjacent to a white arrow. Furthermore, the white arrow must exit from n_1 (since n_1 has already an incoming black arrow). By visiting the nodes of the second cycle, which necessarily will proceed step by step in the same direction as the arrows, we must arrive to another node, n_2, in common with the first cycle (because the nodes in common are at least two). On n_2 the white arrow must be incoming, which gives us a contradiction because n_2 has already an incoming black arrow.

Finally, for Configuration 3, consider the two cycles (black arrows). Again, each node in them must have exactly one incoming arrow. Consider now the path (white arrows) and the two nodes, one for each cycle, which constitute its extremities. Denote them by n_1 and n_2. Like in previous case, the white arrow must be exiting n_1, but then, by following the path, which necessarily will go in the same direction of the arrows, we will arrive at n_2 with an incoming arrow. Contradiction. $\square$

We are now ready to state our main result. We begin by showing that, under proper schedulers, the translated processes reflect the may behavior of the original processes, provided that the observer are also translated (Theorem 5.1). To this end, we use the following two lemmata. In the sequel, given a π_{pa} process P, we denote by $\tilde{P}$ the π_a process obtained from P by removing all the probabilities from the choice constructs.

Lemma 5.6 *For every π_{pa} process P and observer O*

$$P \ \mathbf{may}_{\mathcal{P}} \ O \ \textit{iff} \ \tilde{P} \ \mathbf{may} \ \tilde{O}$$

Proof

only if) Assume $pb(sexec(\nu(P|O), \zeta)) > 0$ for some proper scheduler ζ. Then there exists an execution ξ of $\nu(P \mid O)$ under ζ such that ξ is successful, i.e. ξ is of the form

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

and $Q_i \xrightarrow[p]{\omega}$ for some i and p. By eliminating the probabilities from ξ, we obtain a successful interaction of $\tilde{P} \mid \tilde{O}$, namely:

$$\tilde{P} \mid \tilde{O} = \tilde{Q}_0 \xrightarrow{\tau} \tilde{Q}_1 \xrightarrow{\tau} \tilde{Q}_2 \xrightarrow{\tau} \dots$$

and $\tilde{Q}_i \xrightarrow{\omega}$.

if) Assume $\tilde{P}$ **may** $\tilde{O}$. Then there exists an interaction

$$\tilde{P} \mid \tilde{O} = \tilde{Q}_0 \xrightarrow{\tau} \tilde{Q}_1 \xrightarrow{\tau} \tilde{Q}_2 \xrightarrow{\tau} \dots$$

such that, for some i, $\tilde{Q}_i \xrightarrow{\omega}$. Therefore, for suitable probabilities p_0, p_1, p_2, ... we have an execution fragment ξ' of the form

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

such that, for some p, $Q_i \xrightarrow[p]{\omega}$. Furthermore, since ξ' is finite, we can define a proper scheduler ζ such that ξ' is an execution fragment of $\nu(P \mid O)$ under ζ, from which we derive that $C_{\xi'} \subseteq sexec(\nu(P|O), \zeta)$. Hence we have, by monotonicity of pb, that $pb(sexec(\nu(P|O), \zeta)) \geq pb(C_{\xi'})$. By definition, $pb(C_{\xi'}) = pb(\xi') = p_0 p_1 p_2 \dots p_i > 0$. $\square$

Next lemma proves that the may testing is preserved by the translation. From now on, we will assume that ω is the name of the channel on which the action denoting success is performed, i.e. we ignore the number of parameters. In other words, ω is not affected by the translation. This assumption allows us to use the same notion of success for the original and the translated process, thus simplifying the formulation of the correspondence. In the sequel, we use the symbol $\Longrightarrow$ to represent the reflexive and transitive closure of $\xrightarrow{\tau}$.

Lemma 5.7 *For every π process P and observer O*

P **may** O *iff* $\widetilde{[\![P]\!]}$ **may** $\widetilde{[\![O]\!]}$

Proof

only if) This part trivially follows from the observation that for every π processes Q and Q', if $Q \xrightarrow{\tau} Q'$, then $\widetilde{[\![Q]\!]} \Longrightarrow \widetilde{[\![Q']\!]}$, i.e. there are π_a processes $R_0, R_1, R_2, \dots, R_n$ such that

$$\widetilde{[\![Q]\!]} = R_0 \xrightarrow{\tau} R_1 \xrightarrow{\tau} R_2 \xrightarrow{\tau} \dots R_n = \widetilde{[\![Q']\!]}$$

The additional steps are necessary for performing the synchronization protocol. The processes $R_1, R_2, \dots$ represent the intermediate states during the execution of the protocol.

if) This part follows from the observation that for every π processes Q, if $\widetilde{[\![Q]\!]} \Longrightarrow R$, then there exists a π process Q' such that $R \Longrightarrow \widetilde{[\![Q']\!]}$ and $Q \Longrightarrow Q'$. Furthermore, if $R \xrightarrow{\omega}$, then $[\![Q']\!] \xrightarrow{\omega}$. Note that R may not correspond to the translation of any π process because there may be synchronization protocols which have been started but not yet completed in R, namely the branches which did not win the competition are still active. Eventually, by letting all of those branches get their lock, they will be able to see that the lock has value **f** (meaning that the competition has already been won by another branch) and disappear. At the end there will be still the garbage corresponding to the locks l and h, and a. These can be removed via the congruence rule, using the structural law for "collecting garbage". Thus we obtain a process which is a translation of a π process, namely $[\![Q']\!]$.

As for the the capability of R of performing an ω step, this is preserved in $[\![Q']\!]$ because by definition ω is not an internal name of the translation. $\square$

Theorem 5.1 (Correctness of the encoding with respect to may testing) *For every π process P and observer O*

$$P \ \mathbf{may}\ O \ \textit{iff}\ [\![P]\!] \ \mathbf{may}_{\mathcal{P}}\ [\![O]\!]$$

Proof From Lemma 5.7 we have that P **may** O iff $\widetilde{[\![P]\!]}$ **may** $\widetilde{[\![O]\!]}$. From Lemma 5.6 we have that $\widetilde{[\![P]\!]}$ **may** $\widetilde{[\![O]\!]}$ iff $[\![P]\!]$ $\mathbf{may}_{\mathcal{P}}$ $[\![O]\!]$. $\square$

We now prove the correctness of the embedding also with respect to must testing (Theorem 5.2). This part is more difficult, because the must version of Lemma 5.7 does not hold, due to the possibility of infinite loops generated by the synchronization protocol. We need to show that such loops have probability 0.

Theorem 5.2 (Correctness of the encoding with respect to must testing) *For every π process P, and every observer O*

$$P \ \mathbf{must}\ O \ \textit{iff}\ [\![P]\!] \ \mathbf{must}_{\mathcal{P}}\ [\![O]\!]$$

Proof

only if) Assume P **must** O. We have to show that $[\![P]\!]$ $\mathbf{must}_{\mathcal{P}}$ $[\![O]\!]$. Since $[\![P]\!] \mid [\![O]\!] = [\![P \mid O]\!]$, we need to show that for all adversaries $\zeta \in \mathcal{P}$, $pb(\{\xi \in exec(M_{[\![P|O]\!]}, \zeta) \mid succ(\xi)\}) = 1$. Given an interaction of $P \mid O$, we can mimic the same steps up to the point in which a synchronization involving some choice processes occurs. Suppose that $P_1, \ldots, P_n$ are the processes involved in the synchronization. For each pair P_i, P_j which can synchronize, we know that the interaction (in the original π process) will be successful. The risk is that $[\![P_1]\!], \ldots, [\![P_n]\!]$ will loop forever in the synchronization protocol. This will happen only if none of the processes $[\![P_1]\!], \ldots, [\![P_n]\!]$ will ever be able to acquire both the local and the remote lock. However, we can show that this situation has only probability 0. In fact, after the actions of the form $x(r, a, h, y)$ (see Table 5) have been executed (synchronized with their corresponding output actions), we are in the situation in which several parallel processes compete, pairwise, on the same locks. Consider the graph described in Definition 5.4. By Lemma 5.5, we know that each connected component contains at most one cycle. The proof then proceeds for each connected component as the proof of correctness of the dining philosophers without the fairness assumption ([32], see also [33] for an alternative proof–the result that the fairness hypothesis is not necessary was found independently.).

if) Assume by contradiction that there exists an interaction ξ between P and O of the form

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

such that, for all i, $Q_i \not\xrightarrow{\omega}$. It is easy to see that, using a scheduler that just selects, step by step, the pairs that constitute the interaction steps inξ, we obtain an interaction between $[\![P]\!]$ and $[\![O]\!]$ of the form

$$[\![P \mid O]\!] = [\![Q_0]\!] \underset{1}{\Longrightarrow} [\![Q_1]\!] \underset{1}{\Longrightarrow} [\![Q_2]\!] \underset{1}{\Longrightarrow} \ldots$$

where $\underset{1}{\Longrightarrow}$ stands for the transitive closure of $\xrightarrow[1]{\tau}$. Furthermore, for all i, $[\![Q_i]\!] \not\xrightarrow{\omega}$. This contradicts the hypothesis that $[\![P]\!]$ $\mathbf{must}_{\mathcal{P}}$ $[\![O]\!]$.

□

The above results refer to a notion of correctness which is specifically formulated for testing semantics. A more general notion of correctness, considered in several works about translations (like for instance [5]) is the following: two processes are semantically equivalent if and only if the encoded processes are. This property is often called *full abstraction*. In our case, as an immediate consequence of the above theorem,we obtain the if-part (soundness) of full abstraction:

Corollary 5.8 (Soundness) *For every π-calculus processes P and Q, if $[\![P]\!] \simeq_{\mathcal{P}} [\![Q]\!]$ then $P \simeq Q$.*

Proof Assume $[\![P]\!] \simeq_{\mathcal{P}} [\![Q]\!]$. Then, for every π_{pa} observer O, $[\![P]\!]$ **may** O iff $[\![Q]\!]$ **may** O, and $[\![P]\!]$ **must** O iff $[\![Q]\!]$ **must** O. In particular, this holds for $O = [\![O']\!]$, for any π process O'.

From Theorem 5.1 and Theorem 5.2 we deduce that, for every O', P **may** O' iff Q **may** O', and P **must** O' iff Q **must** O'. □

Note that the viceversa (completeness) 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 encoding on the translated processes.

We conclude this paper with a discussion of the properties of our encoding and other notions of encoding that have been proposed in literature.

6 Discussion

The notion of encoding is generally accepted as a basis for evaluating the (relative) expressiveness of languages: L_1 is no more expressive than L_2 if there is an encoding $[\![\cdot]\!]$ from L_1 to L_2 satisfying certain properties. There is no general agreement, however, on what a good set of properties should be.

One condition which is usually required is *compositionality*: it ensures that the translation will not require an entire reorganization of the whole program. Sometimes, in addition, we require homomorphism with respect to certain operators. The notion of *uniform* encoding that we use in this paper requires indeed homomorphism with respect to the parallel operator. This is to ensure that the encoding will preserve the degree of distribution and symmetry, with the motivations already illustrated in the introduction.

The other important requirement is, of course, some form of preservation of the semantic properties. The strongest condition one can consider, with respect to a given semantics $\mathcal{S}$, is the equivalence between the original terms and the encoded terms:

$$\forall P \in L_1 \;\; P \equiv_{\mathcal{S}} [\![P]\!] \tag{1}$$

where $P \equiv_{\mathcal{S}} Q$ means that P is semantically equivalent to Q with respect to $\mathcal{S}$. Of course this notion depends on how precise $\mathcal{S}$ is. In the introduction, for instance, we have discussed the relevance of $\mathcal{S}$ being sensitive to divergence, for certain domains of application.

Often however we the source language and the target language are of different nature and it does not make sense to use the same semantics (or we can't even define the same semantics). This is the case of the languages considered in this paper. The results in Theorems 5.1 and 5.2 are the best approximation of (1) one can obtain in the case of testing semantics.

Another approach is the so-called full abstraction, already mentioned in previous section:

$$\forall P, P' \in L_1 \;\; P \equiv_{\mathcal{S}_1} P' \;\text{ iff }\; [\![P]\!] \equiv_{\mathcal{S}_2} [\![P']\!] \tag{2}$$

were the if-part is called soundness and the only-if part is called completeness. If it is possible to define at least the same notion of observables $\mathcal{O}$ for L_1 and L_2, then, often, $\equiv_{\mathcal{S}_1}$ and $\equiv_{\mathcal{S}_2}$ are taken to be the observational congruences induced by $\mathcal{O}$ on L_1 and L_2 respectively. Note that, if $\equiv_{\mathcal{S}_1}=\equiv_{\mathcal{S}_2}=\equiv_{\mathcal{S}}$, then (1) implies (2).

Full abstraction is a convenient feature in that it allows one to use the encoding as a technique to prove equational properties in the original language: In fact, it reduces equivalence in L_1 to equivalence in L_2.

Another advantage of full abstraction is discussed in [34]: if L_1 can be mapped in L_2 via a fully abstract encoding, then the abstraction mechanisms of L_2 are at least as powerful as those in L_1, in the sense that if in L_1 two terms P and P' are made equivalent by a context $C[\,]$ then the encoding of $C[\,]$ in L_2 must make equivalent $[\![P]\!]$ and $[\![P']\!]$.

Our point of view on the matter is that, as a basis for the notion of expressiveness, soundness is a fundamental property which any "good" encoding should have. In fact, we don't want that two programs which produce different results become equivalent after the encoding. Completeness, on the contrary, in our opinion is not essential in general. More precisely as a condition for expressiveness it is, in certain cases, too strong. Let for example L_1 be a strict sublanguage of L_2, and let $\equiv_{\mathcal{S}_1}$, $\equiv_{\mathcal{S}_2}$ be the observational congruences induced on L_1, L_2 by a notion of observables common to the two languages. Assume that two terms P and P' are congruent in L_1, but not in L_2 because of the presence of a distinguishing context. Then the trivial encoding defined as the injection of L_1 into L_2 would not be a "good" encoding!

More in general, one should be allowed to introduce some "implementation details" in the translation from L_1 into L_2, but the requirement of completeness forbids this because in general the "implementation details" introduce semantic distinctions.

On the other hand, full abstraction alone is not enough, in our opinion, as foundation of a good notion of expressiveness. In fact, when we encode a program from L_1 to L_2, we want to be reassured that the translated program will produce the same output as the original one, or that at least we have an effective way to interpret (or decode) the output so to obtain the same result as the original program. But this condition a-priori is not guaranteed at all by the notion of full abstraction.

Of course the above criticisms concerns only the notion of full abstraction *in abstracto*. In specific cases there may be particular properties of the encoding and/or the semantic equivalences guaranteeing that the correspondence expressed by full abstraction is tight enough.

Acknowledgment

We would like to thank Dale Miller for his comments on preliminary versions of this paper, and the anonymous referees for their careful revision and extremely valuable help in correcting and improving the original submission.

References

[1] R. Milner, Communication and Concurrency, International Series in Computer Science, Prentice Hall, 1989.

[2] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I and II, Information and Computation 100 (1) (1992) 1–40 & 41–77, a preliminary version appeared as Technical Reports ECF-LFCS-89-85 and -86, University of Edinburgh, 1989.

[3] K. Honda, M. Tokoro, An object calculus for asynchronous communication, in: P. America (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 512 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 133–147.

[4] G. Boudol, Asynchrony and the π-calculus (note), Rapport de Recherche 1702, INRIA, Sophia-Antipolis (1992).
URL `http://www.inria.fr/RRRT/RR-1702.html`

[5] U. Nestmann, B. C. Pierce, Decoding choice encodings, Journal of Information and Computation 163 (2000) 1–59, an extended abstract appeared in the *Proceedings of CONCUR'96*, volume 1119 of *LNCS*.

[6] R. M. Amadio, I. Castellani, D. Sangiorgi, On bisimulations for the asynchronous π-calculus, Theoretical Computer Science 195 (2) (1998) 291–324, an extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[7] C. Palamidessi, Comparing the expressive power of the synchronous and the asynchronous pi-calculus, Mathematical Structures in Computer Science 13 (5) (2003) 685–719, a short version of this paper appeared in POPL'97.
URL `http://www.lix.polytechnique.fr/~catuscia/papers/pi_calc/mscs.pdf`

[8] U. Nestmann, What is a 'good' encoding of guarded choice?, Journal of Information and Computation 156 (2000) 287–319, an extended abstract appeared in the *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*.

[9] B. C. Pierce, D. N. Turner, Pict: A programming language based on the pi-calculus, in: G. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language and Interaction: Essays in Honour of Robin Milner, The MIT Press, 1998, pp. 455–494.

[10] D. Walker, Bisimulation and divergence, Information and Computation 85 (2) (1990) 202–241.

[11] O. M. Herescu, C. Palamidessi, Probabilistic asynchronous π-calculus, in: J. Tiuryn (Ed.), Proceedings of FOSSACS 2000 (Part of ETAPS 2000), Vol. 1784 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 146–160. URL `http://www.lix.polytechnique.fr/~catuscia/papers/Prob_asy_pi/fossacs.ps`

[12] R. Segala, N. Lynch, Probabilistic simulations for probabilistic processes, Nordic Journal of Computing 2 (2) (1995) 250–273, an extended abstract appeared in *Proceedings of CONCUR '94*, LNCS 836: 22–25.

[13] R. D. Nicola, M. C. B. Hennessy, Testing equivalences for processes, Theoretical Computer Science 34 (1-2) (1984) 83–133.

[14] M. Boreale, R. D. Nicola, Testing equivalence for mobile processes, Information and Computation 120 (2) (1995) 279–303.

[15] B. Jonsson, W. Yi, Compositional testing preorders for probabilistic processes, in: Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, San Diego, California, 1995, pp. 431–441.

[16] B. Jonsson, K. G. Larsen, W. Yi, Probabilistic extensions of process algebras, in: J. A. Bergstra, A. Ponse, S. A. Smolka (Eds.), Handbook of Process Algebras, Elsevier, 2001, Ch. 11, pp. 685–710.

[17] R. Segala, Testing probabilistic automata, in: U. Montanari, V. Sassone (Eds.), Proceedings of CONCUR '96: Concurrency Theory, 7th International Conference, Vol. 1119 of Lecture Notes in Computer Science, Springer-Verlag, Pisa, Italy, 1996, pp. 299–314.

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

[19] J. Parrow, P. Sjodin, Multiway synchronization verified with coupled simulation, in: R. Cleaveland (Ed.), CONCUR '92: 3rd International Conference on Concurrency Theory, Vol. 630, LNCS, Springer-Verlag, 1992, pp. 518–533.

[20] Y.-K. Tsay, R. L. Bagrodia, Fault-tolerant algorithms for fair interprocess synchronization, IEEE Transactions on Parallel and Distributed Systems 5 (7) (1994) 737–748.

[21] N. Francez, M. Rodeh, A distributed abstract data type implemented by a probabilistic communication scheme, in: Proc. 21st Ann. IEEE Symp. on Foundations of Computer Science, 1980, pp. 373–379.

[22] J. H. Reif, P. G. Spirakis, Real-time synchronization of interprocess communications, ACM Transactions on Programming Languages and Systems 6 (2) (1984) 215–238.

[23] Y.-J. Joung, S. A. Smolka, Strong interaction fairness via randomization, IEEE Transactions on Parallel and Distributed Systems 9 (2) (1998) 137–149.

[24] M. O. Rabin, D. Lehmann, On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem, in: A. W. Roscoe (Ed.), A Classical Mind: Essays in Honour of C.A.R. Hoare, Prentice Hall, 1994, Ch. 20, pp. 333–352, an extended abstract appeared in the *Proceedings of POPL'81*, pages 133-138.

[25] C. Palamidessi, O. M. Herescu, A randomized encoding of the π-calculus with mixed choice, in: Proceedings of the *2nd IFIP International Conference on Theoretical Computer Science*, 2002, pp. 537–549.
URL http://www.lix.polytechnique.fr/~catuscia/papers/prob_enc/ifiptcs02.ps

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

[27] R. Milner, J. Parrow, D. Walker, Modal logics for mobile processes, Theoretical Computer Science 114 (1) (1993) 149–171.

[28] M. Bernardo, R. Gorrieri, A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time, Theoretical Computer Science 202 (1–2) (1998) 1–54.

[29] O. M. Herescu, C. Palamidessi, On the generalized dining philosophers problem, in: Proceedings of the 20th ACM Symposium on Principles of Distributed Computing, 2001, pp. 81–89.
URL http://www.lix.polytechnique.fr/~catuscia/papers/Gen_Phil/podc.ps

[30] J. Baeten, W. Weijland, Procesalgebra, Vol. 18 of Cambridge tracts in theoretical computer science, Cambridge University Press, Cambridge, 1990.

[31] I. Phillips, CCS with priority guards, in: CONCUR: 12th International Conference on Concurrency Theory, Vol. 2154, LNCS, Springer-Verlag, 2001, pp. 305–320.

[32] M. Duflot, L. Fribourg, C. Picaronny, Randomized dining philosophers without fairness assumption, Distributed Computing 1 (17) (2004) 65–76.

[33] O. M. Herescu, The probabilistic asynchronous π-calculus, Ph.D. thesis, Department of Computer Science and Engineering, The Pennsylvania State University (Dec. 2002).
URL http://www.cse.psu.edu/~herescu/thesis.ps

[34] J. C. Mitchell, On abstraction and the expressive power of programming languages, Science of Computer Programming 21 (2) (1993) 141–163, selected papers of the Conference on Theoretical Aspects of Computer Software (TACS '91) (Sendai, 1991).