On the asynchronous nature of the asynchronous π-calculus

Romain Beauxis, Catuscia Palamidessi, Frank D. Valencia

On the asynchronous nature of the asynchronous π-calculus*

Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia

INRIA Saclay and LIX, École Polytechnique

Abstract. We address the question of what kind of asynchronous communication is exactly modeled by the asynchronous π-calculus (π_a). To this purpose we define a calculus $\pi_{\mathfrak{B}}$ where channels are represented explicitly as special buffer processes. The base language for $\pi_{\mathfrak{B}}$ is the (synchronous) π-calculus, except that ordinary processes communicate only via buffers. Then we compare this calculus with π_a. It turns out that there is a strong correspondence between π_a and $\pi_{\mathfrak{B}}$ in the case that buffers are bags: we can indeed encode each π_a process into a strongly asynchronous bisimilar $\pi_{\mathfrak{B}}$ process, and each $\pi_{\mathfrak{B}}$ process into a weakly asynchronous bisimilar π_a process. In case the buffers are queues or stacks, on the contrary, the correspondence does not hold. We show indeed that it is not possible to translate a stack or a queue into a weakly asynchronous bisimilar π_a process. Actually, for stacks we show an even stronger result, namely that they cannot be encoded into weakly (asynchronous) bisimilar processes in a π-calculus without mixed choice.

1 Introduction

In the community of Concurrency Theory the asynchronous π-calculus (π_a) [14,5] is considered, as its name suggests, a formalism for *asynchronous communication*. The reason is that this calculus satisfies some basic properties which are associated to the abstract concept of asynchrony, like, for example, the fact that a send action is non-blocking and that two send actions on different channels can always be swapped (see, for instance, [24,22]).

In other communities, like Distributed Computing, however, the concept of asynchronous communication is much more concrete, and it is based on the assumption that the messages to be exchanged are placed in some communication means while they travel from the sender to the receiver. We will call such communication devices *buffers*. In general, it is also assumed that the action of placing a message in the buffer and the action of receiving the message from the buffer do not take place at the same time, i.e. the exchange is not instantaneous.

A frequent question that people then ask about the asynchronous π-calculus, is "In what sense is this a model of asynchronous communication"? Often they are puzzled by the communication rule of π_a, which is literally the same as the one of the (synchronous) π-calculus [18]:

* This work has been partially supported by the INRIA ARC project ProNoBiS and by the INRIA DREI Équipe Associée PRINTEMPS.

$$\frac{P \xrightarrow{\bar{x}y} P' \qquad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \tag{1}$$

where $\bar{x}y$ represents the action of sending a message y on channel x, and xy represents the action of receiving a message y from channel x. The rule suggests that these two actions take place simultaneously, in a handshaking fashion.

To our experience the most convincing explanation of the asynchrony of π_a is that, because of the lack of output prefix in π_a, in rule (1) P must be of the form $\bar{x}y \mid P'$, and the transition $P \xrightarrow{\bar{x}y} P'$ does not really represent the event of sending. Originally, P must come from a process of the form $C[\bar{x}y \mid R]$ for some prefix context $C[\ ]$ and some process R, and the "real" event of sending takes place at some arbitrary time between the moment $\bar{x}y$ gets at the top-level (i.e. when it is no more preceded by a prefix) and the event of receiving y, the latter being what the rule (1) really represents. In the interval between the two events, R evolves asynchronously into P'. Of course, at this point another question arises: "What happens to the message y in the meanwhile?" The best way to see it is that y is placed in some buffer labeled with x. But then the legitimate question follows on what kind of buffer this is, since it is well known that a distributed system can behave very differently depending on whether the channels are bags, queues, or stacks, for instance. In this paper we address precisely this latter question.

Our approach is to define a calculus π_T where buffers are represented explicitly, like it was done, for instance, in [3,9]. The symbol T stands for $\mathfrak{B}$, $\mathfrak{Q}$, and $\mathfrak{S}$, in the case of bags, queues, and stacks respectively. The actions of receiving from and sending to a given buffer are represented by input and output transitions, respectively. The object of the action is the name being transmitted and subject is the name (or type) of the source buffer if the action is an output, or the destination buffer if the action is an input. The base language for π_T is the (synchronous) π-calculus with guarded choice, except that processes communicate only via buffers. Then we compare this calculus with π_a. It turns out that there is a strong correspondence between π_a and $\pi_{\mathfrak{B}}$ (the case of bags). More precisely, if we interpret the π_a send process $\bar{x}y$ as a bag of type x which contains the single message y, then there is a strong asynchronous bisimulation between a π_a process P and its translation $[\![P]\!]$ in $\pi_{\mathfrak{B}}$ (notation $P \sim [\![P]\!]$). This result reflects the intuitive explanation of the asynchronous nature of π_a given above. On the other hand, we can encode (although in a more involved way) each $\pi_{\mathfrak{B}}$ process P into a π_a process $[\![P]\!]$[1], equivalent to P modulo weak asynchronous bisimilarity (notation $P \approx [\![P]\!]$).

We would like to point out that the the case of bags represents a feature of communication in distributed systems, namely the fact that the order in which messages are sent is not guaranteed to be preserved in the order of arrival. In the case of the π-calculus, it is sufficient to allow the messages to be made available

[1] We use the same notation $[\![\cdot]\!]$ to indicate both the encoding from π_a into $\pi_{\mathfrak{B}}$ and the one from $\pi_{\mathfrak{B}}$ into π_a. It will be clear from the context which is the intended one.

in any order to the receiver (which is exactly the property which characterize the bags as a data structure), because there are no primitives that are able to make a "snapshot" of the system, and in particular to detect the absence of a message. For languages which contain such a kind of primitive however (for instance Linda, with the `imp` construct) the abstraction from the order is not sufficient, and the faithful representation of distributed communication would require a more sophisticated model. A proposal for such model is the *unordered semantics* of [7]. In that paper the authors argue, convincingly, that the unordered semantics is the most "asynchronous" semantics and the "right" one for distributed systems.

In case the buffers are queues or stacks, on the contrary, the correspondence between π_T and π_a does not hold. We show indeed that there is no encoding of stacks or queues, represented as described above, into π_a modulo weak asynchronous bisimulation. By "encoding modulo $\mathcal{R}$" we mean an encoding that translates P into a process that is in relation $\mathcal{R}$ with P. Actually for stacks we prove a stronger result: they cannot be translated, modulo weak bisimilarity, even into π_{sc}, the fragment of the (synchronous) π-calculus where the mixed guarded choice operator is replaced by separate-choice, i.e. a choice construct that can contain either input guards, or output guards, but not both. In other words, the least we need to encode stacks is a mixed-choice construct where both input and output guards (aka prefixes) are present.

The above result does not mean, obviously, that queues and stacks cannot be simulated in π_a: we will indeed discuss a possible way to simulate them by encoding the send and receive actions on buffers into more complicated protocols. The meaning of our negative result is only that a queue (respectively a stack) and any translation of it in π_a (respectively in π_{sc}) cannot be related by a relation like weak (asynchronous) bisimilarity, which requires a strict correspondence between transitions.

1.1 Justifying the choice of the languages

The results presented in this paper would hold also if we had considered the asynchronous version of CCS [4] instead than the asynchronous π-calculus. The reasons why we have chosen the latter are the following. The asynchronous π-calculus was the first process calculus to represent asynchronous communication by using a send primitive with no continuation (*asynchronous send*), and in the concurrency community it has become paradigmatic of this particular approach to asynchrony. Moreover, the expressive power of the asynchronous π-calculus has been widely investigated, especially in relation to other asynchronous calculi, and in comparison with synchronous communication.

We have chosen the π-calculus with (mixed) guarded choice as a base language for π_T because in the π-calculus the main expressive gap between synchronous and asynchronous communication lies exactly in between mixed choice and separate choice [20,19]. In other words, the π-calculus with mixed choice cannot be encoded in the asynchronous π-calculus in any "reasonable" way, while the π-calculus with separate choice can. The choice of a synchronous language

as the basis for π_T is motivated by the fact that it allows a precise control of the communication mechanism: The processes communicate with each other via the buffers, but the interaction between a process and a buffer is synchronous. So the buffers are the only source of asynchrony in π_T, which makes the encoding from the asynchronous π-calculus into π_T more interesting. Furthermore this model of asynchronous communication is very close to the concrete implementation of distributed systems [21].

1.2 Justifying the criteria for the encodings

As we stated above, our main positive result is the correspondence between π_a and $\pi_{\mathfrak{B}}$, expressed in one direction by one encoding

$$[\![\cdot]\!] : \pi_a \to \pi_{\mathfrak{B}} \quad \text{with } P \sim [\![P]\!] \text{ for all } P \text{ in } \pi_a$$

and, in the other direction, by another encoding

$$[\![\cdot]\!] : \pi_{\mathfrak{B}} \to \pi_a \quad \text{with } P \approx [\![P]\!] \text{ for all } P \text{ in } \pi_{\mathfrak{B}}$$

We consider the above properties of the encodings as quite strong, and therefore supporting the claim of a strict correspondence between π_a and $\pi_{\mathfrak{B}}$. They imply for instance the condition of *operational correspondence*, which is one of the properties of a "good" encoding according to Gorla ([10,11]).

One may question why we did not rather prove the existence of a *fully abstract* encoding between π_a and $\pi_{\mathfrak{B}}$. We recall that, given a language $\mathcal{L}_1$ equipped with an equivalence relation $\sim_1$, and a language $\mathcal{L}_2$ equipped with an equivalence relation $\sim_2$, an encoding $[\![\cdot]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ is called fully abstract if and only if

$$\text{for every } P, Q \text{ in } \mathcal{L}_1, \quad P \sim_1 Q \;\Leftrightarrow\; [\![P]\!] \sim_2 [\![Q]\!] \text{ holds}$$

Full abstraction has been adopted sometimes in literature as a criterion for expressiveness. We do not endorse this approach: In our opinion, full abstraction can be useful to transfer the theory of a language to another language, but it is not a good criterion for expressiveness. The reason is that it can be, at the same time, both too strong and too weak a requirement. Let us explain why.

Too strong: Consider the asynchronous π-calculus π_a, equipped with its "natural" notion of equivalence, the weak asynchronous bisimilarity, and the π-calculus π, equipped with weak bisimilarity. Let $[\![\cdot]\!]$ be the standard encoding from π_a to π defined as

$$[\![\bar{x}y]\!] = \bar{x}y.0$$

and homomorphic on the other operators. Most people would agree that this is a pretty straightforward, natural encoding, showing that π is at least as powerful as π_a. Still, it does not satisfy the full abstraction criterion. In fact, there are weakly asynchronous bisimilar processes P, Q such that $[\![P]\!]$ and $[\![Q]\!]$ are not weakly bisimilar. Take, for example, $P = 0$ and $Q = x(y).\bar{x}y$.

Too weak: Consider an enumeration of Turing machines, $\{T_n\}_n$, and an enumeration of minimal finite automata $\{A_n\}_n$, with their standard language-equivalence $\equiv$. Consider the following encoding of Turing machines into (minimal) finite automata:

$$[\![T_m]\!] = [\![T_k]\!] \text{ if } k < m \text{ and } T_k \equiv T_m$$
$$[\![T_m]\!] = A_n \quad \text{otherwise}$$

where n is the minimum number such that A_n has not been used to encode any T_k with $k < m$. By definition, we have that $\forall m, n\ T_m \equiv T_n \Leftrightarrow [\![T_m]\!] \equiv [\![T_n]\!]$, but this certainly does not prove that finite automata are as powerful as Turing machines!

Note that the second encoding, from Turing machines to finite automata, is non-effective. This is fine for our purpose, which is simply to show that full abstraction alone, i.e. without extra conditions on the encoding, is not a very meaningful notion. Of course, it would be even more interesting to exhibit an *effective* and fully abstract encoding between some L_1 and L_2, while most people would agree that L_2 is strictly less powerful than L_1. But this is out of the scope of this paper, and we leave it as an open problem for the interested reader.

1.3 Plan of the paper

In the next section we recall some standard definitions. In Section 3, we introduce the notion of buffer and the different types of buffer we will consider. Then in Section 4, we define a π-calculus communicating through bags. In Section 5, we study the correspondence between the π-calculus with bags and the π_a-calculus. The main bisimilarity results are established there. In Section 6, we use the properties from Section 3 to prove the impossibility results for stacks and queues. Section 7 discusses related work. Finally, Section 8 concludes and outlines some directions of future research.

Acknowledgment

This work originates from a suggestion of John Mitchell and Andre Scedrov. They remarked that outsiders to Concurrency Theory do not see so clearly why the asynchronous π-calculus represents a model of asynchronous communication. So they recommended to give a justification in terms of a model (buffers) which looks probably more natural to a wide audience, as this could make the asynchronous π-calculus more widely appreciated.

We also wish to thank Roberto Gorrieri, who helped us with very insightful comments, and the anonymous reviewers.

2 Preliminaries

2.1 The asynchronous π-calculus: π_a

We assume a countable set of *names*, ranged over by $x, y, \ldots$, and for each name x, a *co-name* $\overline{x}$. In the asynchronous π-calculus, henceforth denoted as π_a,

$$(in)\ \frac{}{x(y).P \xrightarrow{xz} P[z/y]} \qquad (out)\ \frac{}{\overline{x}z \xrightarrow{\overline{x}z} 0}$$

$$(sync)\ \frac{P \xrightarrow{\overline{x}y} P',\ Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad (\nu)\ \frac{P \xrightarrow{\alpha} P',\ a \notin fn(\alpha)}{\nu a P \xrightarrow{\alpha} \nu a P'}$$

$$(open)\ \frac{P \xrightarrow{\overline{x}y} P',\ x \neq y}{\nu y P \xrightarrow{\overline{x}(y)} P'} \qquad (close)\ \frac{P \xrightarrow{\overline{x}(y)} P',\ Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} \nu y(P' \mid Q')}$$

$$(comp)\ \frac{P \xrightarrow{\alpha} P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad (bang)\ \frac{P \mid !P \xrightarrow{\alpha} R}{!P \xrightarrow{\alpha} R}$$

$$(cong)\ \frac{P \equiv P'\ P' \xrightarrow{\alpha} Q'\ Q' \equiv Q}{P \xrightarrow{\alpha} Q}$$

Table 1. Transition rules for the π_a-calculus

processes are given by the following syntax:

$$P, Q, \ldots\ :=\ 0 \,\Big|\, \overline{x}z \,\Big|\, x(y).P \,\Big|\, \nu x P \,\Big|\, P \mid Q \,\Big|\, !P$$

The 0 represents an empty (or terminated) process. Intuitively, an *output* $\overline{x}z$ represents a particle in an implicit medium tagged with a name x indicating that it can be received by an *input process* $x(y).P$ which behaves, upon receiving z, as $P[z/y]$, namely, the process where every free occurence of y is replaced by z. We assume that P is α-renamed before applying $[z/y]$ so to avoid name-capture. A substitution σ causes name-capture in P if it replaces a name y by a name z for which one or more free occurences of y in P are in the scope of a binder for z. Furthermore, $x(y).P$ binds y in P. The other binder is the *restriction* $\nu x P$ which declares a name x private to P. The *parallel composition* $P \mid Q$ means P and Q running in parallel. The *replication* $!P$ means $P \mid P \mid \ldots$, i.e. an unbounded number of copies of P.

We use the standard notations $bn(Q)$ for the *bound names* in Q, and $fn(Q)$ for the *free names* in Q, and write $\nu x_1 \ldots x_n P$ to denote $\nu x_1 \ldots \nu x_n P$.

The (early) transition semantics of π_a is given in terms of the relation $\xrightarrow{\alpha}$ in Table 1. The label α represents an action which can be of the form τ (silent), $\overline{x}z$ (free output), $\overline{x}(y)$ (bound output) or xz (free input). Transitions are quotiented by the *structural congruence* relation $\equiv$ below.

Definition 1. *The relation $\equiv$ is the smallest congruence over processes satisfying α-conversion and the commutative monoid laws for parallel composition with 0 as identity.*

3 Buffers

A buffer in this paper is basically a data structure that accepts messages and resends them later. We consider different types of buffers, depending on the policy used for outputting a previously received message. We focus on the following policies, that can be considered the most common:

- Bag, or unordered policy: any message previously received (and not yet sent) can be sent next.
- Queue, or FIFO policy: only the oldest message received (and not yet sent) can be sent next.
- Stack, or LIFO policy: only the last message received (and not yet sent) can be sent next.

Let us now formally define these three types of buffer. We need to keep the information about the order of reception to decide which message can be sent next. This will be achieved using a common core definition for all kinds of buffers.

We will use $M \in \mathcal{M}$ to denote a message that the buffers can accept.

Definition 2 (Buffer). *A buffer is a finite sequence of messages:*

$$B = M_1 * ... * M_k,\ k \geq 0,\ M_i \in \mathcal{M}\ \textit{(B is the empty sequence if } k = 0\textit{)}.$$

$*$ is a wild card symbol for the three types of buffers. Then, we will use the notation $M_1 \diamond ... \diamond M_k$ for a bag, $M_1 \triangleleft ... \triangleleft M_k$ for a queue, $M_1 \triangleright ... \triangleright M_k$ for a stack.

A reception on a buffer is the same for all kinds of policies:

Definition 3 (Reception on a buffer). *Let $B = M_1 * ... * M_k$. We write $B \xrightarrow{M} B'$ to represent the fact that B receives the message M, becoming $B' = M * B = M * M_1 * ... * M_k$.*

The emission of a message is different for the three types of buffers:

Definition 4 (Sending from a buffer). *Let $B = M_1 * ... * M_k$. We write $B \xrightarrow{\overline{M}} B'$ to represent the fact that B sends the message M, becoming B', where:*

- *If $* = \diamond$ (bag case) then $M = M_i$ for some $i \in \{1, ..., k\}$ and $B' = M_1 \diamond ... \diamond M_{i-1} \diamond M_{i+1} \diamond ... \diamond M_k$.*
- *If $* = \triangleleft$ (queue case) then $M = M_k$ and $B' = M_1 \triangleleft ... \triangleleft M_{k-1}$.*
- *If $* = \triangleright$ (stack case) then $M = M_1$ and $B' = M_2 \triangleright ... \triangleright M_k$.*

Finally, we introduce here the notion of *buffer's content* and *sendable items.*

Definition 5 (Buffer's content). *A buffer's content is the multiset of messages that the buffer has received and has not yet sent:*

$$C(M_1 * ... * M_k) = \{M_1, ..., M_k\}$$

Definition 6 (Buffer's sendable items). *A buffer's sendable items is the multiset of messages that can be sent immediately:*

$$S(M_1 \diamond M_2 \diamond \cdots \diamond M_k) = \{M_1, M_2, \ldots, M_k\}$$
$$S(M_1 \triangleleft M_2 \triangleleft \cdots \triangleleft M_k) = \{M_k\}$$
$$S(M_1 \triangleright M_2 \triangleright \cdots \triangleright M_k) = \{M_1\}$$

Note that $S(B)$ is empty iff $C(B)$ is empty. Furthermore, if B is a bag, then $C(B) = S(B)$.

Remark 1. If B is a buffer such that $B \xrightarrow{\overline{M_1}}$ and $B \xrightarrow{\overline{M_2}}$ with $M_1 \neq M_2$ then B must be a bag, i.e. B cannot be a stack or a queue.

4 A π-calculus with bags

In this section, we define a calculus for asynchronous communications obtained by enriching the synchronous π-calculus with bags, and forcing the communications to take place only between (standard) processes and bags.

We decree that the bag's messages are names. Each bag is able to send and receive on a single channel only, and we write B_x for a bag on the channel x. We use $\emptyset_x$ to denote an empty bag on channel x, and $\{y\}_x$ for the bag on channel x, containing a single message, y.

Definition 7. *The $\pi_{\mathfrak{B}}$-calculus is the set of processes defined by the grammar:*

$$P, Q ::= \textstyle\sum_{i \in I} \alpha_i.P_i \;\Big|\; P \mid Q \;\Big|\; \nu x\, P \;\Big|\; !P \;\Big|\; B_x$$

where B_x is a bag, I is a finite indexing set and each α_i can be of the form $x(y)$ or $\overline{x}z$. If $|I| = 0$ the sum can be written as 0 and if $|I| = 1$ the symbol "$\sum_{i \in I}$" can be omitted.

The early transition semantics is obtained by redefining the rules *in* and *out* in Table 1 and by adding the rules in_{bag} and out_{bag} for bag communication as defined in Table 3. Note that they are basically the rules for the (synchronous) π-calculus except that communication can take place only between (standard) processes and bags. In fact, the rule *out* guarantees that a process can only output to a bag. Furthermore the only rule that generates an output transition is out_{bag}, hence a process can only input, via *sync* and *close*, from a bag.

The structural equivalence $\equiv$ consists of the standard rules of Definition 1, plus scope extrusion, plus $P \equiv P|\emptyset_x$. This last rule allows any process to have access to a buffer even if the process itself is blocked by a binder. A typical example would be $P = \nu x(\overline{x}y.x(z).Q)$, which could not execute any action without this rule. Thanks to the rule, we have:

$$P \equiv \nu x(\overline{x}y.x(z).Q \mid \emptyset_x) \rightarrow \nu x(x(z).Q \mid \{y\}_x) \rightarrow \nu x(Q[y/z] \mid \emptyset_x)$$

$$
\begin{aligned}
P &\equiv Q \quad \text{if } P \text{ and } Q \text{ are } \alpha\text{-convertible} \\
P \mid Q &\equiv Q \mid P \\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R) \\
(\nu z\, P) \mid Q &\equiv \nu z\, (P \mid Q) \quad \text{if } z \notin fn(Q) \\
P &\equiv P \mid \emptyset_x \quad \text{for all possible } x
\end{aligned}
$$

Table 2. Structural congruence for the π-calculus with bags

Note that we could restrict the application of $P \equiv P|\emptyset_x$ to the case in which P is a pure process (not containing a bag already), i.e. a term of the asynchornous π-calculus). We do not impose this constraint here because it is not necessary, and also because we believe that allowing multiple bags on the same channel name and for the same process is a more natural representation of the concept of channel in distributed systems. Later in this paper, when dealing with stacks and queues, we will have to adopt this restriction in order to be consistent with the nature of stacks and queues.

A consequence of the rule $P \equiv P|\emptyset_x$ is that every process P is always *input-enabled*. This property is in line with other standard models of asynchronous communication, for example the Input/output automata (see, for instance, [15]), the input-buffered agents of Selinger [24] and the Honda-Tokoro original version of the asynchronous π-calculus [14].

The scope extrusion equivalence – $(\nu z\, P) \mid Q \equiv \nu z\, (P \mid Q)$ if $z \notin fn(Q)$ – has been added even though the *open* and *close* rules are present. This is to allow scope extrusion to apply also in some particular case where those rules would not help. A good example is $\nu x\, \overline{x}y$: only a buffer can make an output action, so this process would not be able to use the *open* rule.

The basic input and output transitions for bags given by in_{bag} and out_{bag} are defined in terms of receive and send transitions on buffers in Definition 3 and 4. The following remark follows trivially from the rules in Table 3.

Remark 2. Let B_x a bag process. Then $B_x \xrightarrow{y} B'_x$ iff $B_x \xrightarrow{xy} B'_x$. Similarly, $B_x \xrightarrow{\overline{y}} B'_x$ iff $B_x \xrightarrow{\overline{x}y} B'_x$.

The notions of *free names* and *bound names* for ordinary processes are defined as usual. For bags, we define them as follows. Recall that C gives the content of a buffer (see Definition 5).

Definition 8 (Bag's free and bound names). *Let B_x be a bag with content $C(B_x) = \{y_1, \ldots, y_k\}$. The free variables fn and the bound variables bn of B_x are defined as $fn(B_x) = \{x, y_1, \ldots, y_k\}$ and $bn(B_x) = \emptyset$.*

5 Relation between the asynchronous π-calculus and the π-calculus with bags

In this section, we study the relation between the π_a-calculus and the $\pi_{\mathfrak{B}}$-calculus. We first define the notions of asynchronous bisimilarities, along the

$$(in)\ \frac{\alpha_j = xy}{\sum_{i\in I}\alpha_i.P_i \overset{xz}{\to} P_j[z/y]} \qquad (out)\ \frac{B_x \overset{xy}{\longrightarrow} B'_x}{(\overline{x}y.P + \sum_{i\in I}\alpha_i.P_i) \mid B_x \overset{\tau}{\to} P \mid B'_x}$$

$$(in_{bag})\ \frac{B_x \overset{y}{\to} B'_x}{B_x \overset{xy}{\to} B'_x} \qquad (out_{bag})\ \frac{B_x \overset{\overline{y}}{\to} B'_x}{B_x \overset{\overline{x}y}{\to} B'_x}$$

$$(sync)\ \frac{P \overset{\overline{x}y}{\to} P',\ Q \overset{xy}{\to} Q'}{P \mid Q \overset{\tau}{\to} P' \mid Q'} \qquad (\nu)\ \frac{P \overset{\alpha}{\to} P',\ a \notin fn(\alpha)}{\nu aP \overset{\alpha}{\to} \nu aP'}$$

$$(open)\ \frac{P \overset{\overline{x}y}{\to} P',\ x \neq y}{\nu yP \overset{\overline{x}(y)}{\to} P'} \qquad (close)\ \frac{P \overset{\overline{x}(y)}{\to} P',\ Q \overset{xy}{\to} Q',\ y \notin fn(Q)}{P \mid Q \overset{\tau}{\to} \nu y(P' \mid Q')}$$

$$(comp)\ \frac{P \overset{\alpha}{\to} P',\ bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \overset{\alpha}{\to} P' \mid Q} \qquad (bang)\ \frac{P \mid !P \overset{\alpha}{\longrightarrow} R}{!P \overset{\alpha}{\longrightarrow} R}$$

$$(cong)\ \frac{P \equiv P'\ P' \overset{\alpha}{\to} Q'\ Q' \equiv Q}{P \overset{\alpha}{\to} Q}$$

Table 3. Transition rules for the π-calculus with bags

lines of [1] and [14] . They will constitute the formal basis for stating the correspondence.

In the following, we use the standard notation for weak transitions: $P \overset{\alpha}{\Longrightarrow} Q$ stands for $P \overset{\tau}{\longrightarrow}{}^* \overset{\alpha}{\longrightarrow} \overset{\tau}{\longrightarrow}{}^* Q$.

Definition 9.

Strong asynchronous bisimilarity *A symmetric relation $\mathcal{R}$ is a strong asynchronous bisimulation iff whenever $P\,\mathcal{R}\,Q$, then the following holds:*

- *If $P \overset{\alpha}{\longrightarrow} P'$ and α is not an input action, then: $Q \overset{\alpha}{\longrightarrow} Q'$ with: $P'\,\mathcal{R}\,Q'$*
- *If $P \overset{xy}{\longrightarrow} P'$ then*
  - *either $Q \overset{xy}{\longrightarrow} Q'$ with $P'\,\mathcal{R}\,Q'$,*
  - *or $P'\,\mathcal{R}\,(Q \mid \overline{x}y)$.*

We say that P and Q are strongly asynchronously bisimilar, written $P \sim Q$, iff there exists $\mathcal{R}$ such that: $P\,\mathcal{R}\,Q$.

Weak asynchronous bisimilarity *A symmetric relation $\mathcal{R}$ is a weak asynchronous bisimulation iff whenever $P\,\mathcal{R}\,Q$, then the following holds:*

- *If $P \overset{\alpha}{\longrightarrow} P'$ and α is not an input action, then: $Q \overset{\alpha}{\Longrightarrow} Q'$ with: $P'\,\mathcal{R}\,Q'$*
- *If $P \overset{xy}{\longrightarrow} P'$ then*
  - *either $Q \overset{xy}{\Longrightarrow} Q'$ with $P'\,\mathcal{R}\,Q'$,*
  - *or $P'\,\mathcal{R}\,(Q \mid \overline{x}y)$.*

We say that P and Q are weakly asynchronously bisimilar, written $P \approx Q$, iff there exists $\mathcal{R}$ such that: $P\,\mathcal{R}\,Q$.

Note that weak asynchronous bisimulation is weaker than weak bisimulation, and it is weaker than strong asynchronous bisimulation.

We will use the two notions of bisimulation introduced above to describe the properties of the encodings from π_a to $\pi_{\mathfrak{B}}$ and from $\pi_{\mathfrak{B}}$ to π_a, respectively. The notion of strong asynchronous bisimulation is almost the same, but not completely, as the one of [1]. The difference is that, in [1], when P performs an input action, Q can either perform a corresponding input action *or a τ step*. The reason for introducing the chance is essentially to get the correspondence stated in Theorem 1. We could have used weak asynchronous bisimulation instead, but we preferred to show how strong the correspondence is. As for the notion of weak asynchronous bisimulation, this is essentially the same as the one introduced by [14] (called *asynchronous bisimulation* in that paper). The formulation is different, since the labeled transition system of [14] is different from ours, however it is easy to show that the (weak) bisimulations induced by their system, as relations on process terms, coincide with our weak asynchronous bisimulations.

5.1 From π_a to $\pi_{\mathfrak{B}}$

We observe that there is a rather natural interpretation of the π_a-calculus into the $\pi_{\mathfrak{B}}$-calculus, formalized by an encoding defined as follows:

Definition 10. *Let $[\![\ ]\!] : \pi_a \longrightarrow \pi_{\mathfrak{B}}$ be defined homomorphically except for the send process, which is translated as $[\![\overline{x}y]\!] = \{y\}_x$.*

It is easy to see that there is an exact match between the transitions of P and the ones of $[\![P]\!]$, except that $\{y\}_x$ can perform input actions on x that the original process $\overline{x}y$ cannot do. This is exactly the kind of situation treated by the additional case in the definition of asynchronous bisimilarity (additional w.r.t. the classical definition of bisimilarity). Hence we have the following result:

Theorem 1. *Let $[\![\ ]\!] : \pi_a \longrightarrow \pi_{\mathfrak{B}}$ be the encoding in Definition 10. For every $P \in \pi_a$, $P \sim [\![P]\!]$.*

The encoding from $\pi_{\mathfrak{B}}$ into π_a is more complicated, but still we can give a rather faithful translation.

5.2 From $\pi_{\mathfrak{B}}$ to π_a

Our encoding of the $\pi_{\mathfrak{B}}$-calculus into the π_a-calculus is given below.

Definition 11. *The encoding $[\![\ ]\!] : \pi_{\mathfrak{B}} \longrightarrow \pi_a$ is defined as follows:*

$$\begin{aligned}
[\![\textstyle\sum_{i\in I} \alpha_i.P_i]\!] &= \nu(l,t,f)\ (\overline{l}t \mid \Pi_{i\in I}[\![\alpha_i.P_i]\!]_l) \\
[\![P \mid Q]\!] &= [\![P]\!] \mid [\![Q]\!] \\
[\![\nu v\ P]\!] &= \nu v\ [\![P]\!] \\
[\![!P]\!] &= ![\![P]\!] \\
[\![B_x]\!] &= \Pi_{y_i \in S(B_x)} \overline{x} y_i
\end{aligned}$$

where $[\![\ ]\!]_l$ *is given by*

$$[\![x(y).P]\!]_l = !x(y).l(\lambda).\Big[(\mathit{if}\ \ \lambda = t\ \ \mathit{then}\ \ \overline{l}f \mid [\![P]\!]\ \mathit{else}\ \overline{l}f \mid \overline{x}y)\Big]$$
$$[\![\overline{x}y.P]\!]_l = l(\lambda).\Big[(\mathit{if}\ \ \lambda = t\ \ \mathit{then}\ \ \overline{l}f \mid \overline{x}y \mid [\![P]\!]\ \mathit{else}\ \overline{l}f)\Big]$$

In this definition, we use a if-then-else construct in the form if $\lambda = t$ *then* P *else* Q *which is syntactic sugar for* $\overline{\lambda} \mid t.P \mid f.Q$*. This is correct within the scope of our definition because* λ *can only be* t *or* f*, and* λ*,* t *and* f *are private, and only one such construct can be active at a time.*

This encoding of the mixed choice is similar to the first encoding of input guarded choice defined in [19]. Note that the encoding of the input branch allows a non deterministic choice between going back to initial state, or following with $[\![P]\!]$. This is important to establish the bisimilarity result.

The soundness of the encoding depends crucially on the fact that in the $\pi_{\mathfrak{B}}$-calculus the output of a standard process is non-blocking.

Note that this encoding is not termination preserving. As in [19], this problem could be addressed by removing the backtracking possibility and using a coarser semantic (coupled bisimilarity). In this paper however we consider the stronger notion of weak asynchronous bisimilarity recalled above.

Theorem 2. *Let* $[\![\ ]\!] : \pi_{\mathfrak{B}} \longrightarrow \pi_a$ *be the encoding in Definition 11. Then, for every* $P \in \pi_{\mathfrak{B}}$*,* $P \approx [\![P]\!]$*.*

Proof. We give the proof only for the non-homomorphic cases of the encoding. The homomorphic ones follow trivially.

1. $[\![B_x]\!]$
2. $[\![\sum_{i\in I} \alpha_i.P_i]\!]$

We will show that the above encodings are weakly asynchronous bisimilar to their source processes. For (1), the statement follows from:

- $B_x \xrightarrow{xy} B'_x \Longrightarrow [\![B'_x]\!] = [\![B_x]\!] \mid \overline{x}y$
- $B_x \xrightarrow{\overline{x}y} B'_x \Longleftrightarrow [\![B_x]\!] \xrightarrow{\overline{x}y} [\![B'_x]\!]$

Let us now consider the case (2). For the sake of simplicity, we will outline the proof for a choice construct with only one input-guarded and one output-guarded branches, the proof for a choice with more than two branches can be easily generalized from this case. There are three kinds of possible transitions from this choice[2]:

1. $x(y).P + \overline{z}v.Q \xrightarrow{\tau} Q \mid \{v\}_z$
2. $x(y).P + \overline{z}v.Q \xrightarrow{xw} P[y/w]$
3. $x(y).P + \overline{z}v.Q \xrightarrow{xw} (x(y).P + \overline{z}v.Q) \mid \{w\}_x$

[2] In the third transition, the input could be on a channel different from x. The proof however proceeds in the same way.

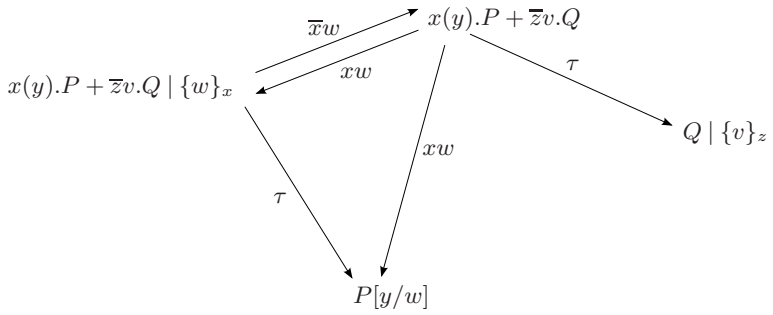

Fig. 1. Transitions of a $\pi_{\mathfrak{B}}$ sum.

These transitions are matched by the encoded process in the following way:

1. $[\![x(y).P + \overline{z}v.Q]\!] \xrightarrow{\tau} \nu l(\overline{z}v \mid [\![Q]\!] \mid \overline{l}f \mid [\![x(y).P]\!]_l)$
2. $[\![x(y).P + \overline{z}v.Q]\!] \xrightarrow{xw} \xrightarrow{\tau} \xrightarrow{\tau} \nu l([\![P[y/w]]\!] \mid \overline{l}f \mid [\![\overline{z}v.Q]\!]_l)$
3. $(x(y).P + \overline{z}v.Q) \mid \{w\}_x \approx [\![x(y).P + \overline{z}v.Q]\!] \mid \overline{x}w$

It is easy to see that $\nu l(\overline{l}f \mid [\![x(y).P]\!]_l)$ is weakly asynchronous bisimilar to 0, and $\nu l(\overline{l}f \mid [\![\overline{z}v.Q]\!]_l)$ is weakly asynchronous bisimilar to 0.

In the other direction, we have the above transitions plus the following one:

$$[\![x(y).P + \overline{z}v.Q]\!] \xrightarrow{xw} R$$

where $R = \nu l(\overline{l}t \mid l(x).((\text{if } x = true \text{ then } [\![P[y/w]]\!]_{l,x(w)} \text{ else } \overline{x}w) \mid \overline{l}f) \mid [\![\overline{z}v.Q]\!]_l)$. In this case, the choice is not yet committed: the value xw has been received, but the process can still choose to process $[\![\overline{z}v.Q]\!]_l$ and then release xw, or send xw and come back to its initial state, or receive the value on $x(y).P$. This is matched by the following transition from the original process:

$$x(y).P + \overline{z}v.Q \xrightarrow{xw} (x(y).P + \overline{z}v.Q) \mid \{w\}_x$$

Figures 1 and 2 show the transitions of a typical binary choice and its encoding and how they are related by weak asynchronous bisimilarity.

6 Negative results for other buffers

In this section, we show the impossibility of encoding other kinds of buffers (i.e. not bags) into the asynchronous π-calculus and into the π-calculus with separate choice. In particular, we show that a calculus with queues and stacks cannot be encoded into π_a modulo weak asynchronous bisimilarity. Then, we show a stronger result for stacks: a calculus with stacks cannot even be encoded, modulo weak asynchronous bisimilarity, in the π-calculus with separated-choice (which is a superset of π_a). Note that, since a weak bisimulation is a special

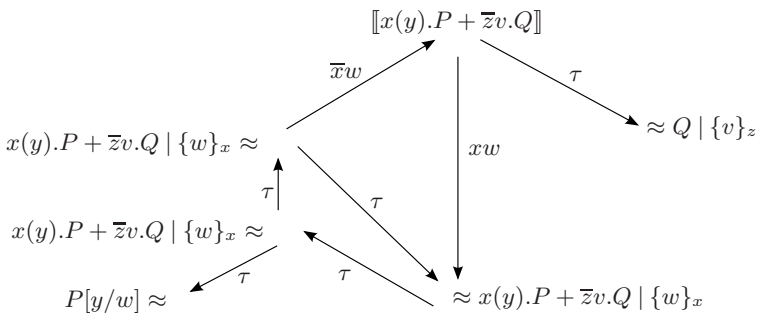

Fig. 2. Transitions of the π_a encoding of the $\pi_{\mathfrak{B}}$ sum in Figure 1.

case of weak asynchronous bisimulation, those results also hold modulo weak bisimilarity.

We stress the fact that these results strongly depend on the requirement that a term (in particular a stack or a queue) and its encoding be equivalent. We believe that it is possible to simulate stacks or queues in π_a. Our results only say that it cannot be done via an encoding that satisfies the requirement of translating a process into a weakly asynchronously bisimilar one.

We start by defining π-calculi with stacks and queues.

Definition 12. *The π-calculus with buffers of type T, written π_T, where T is either $\mathfrak{Q}$ (queues) or $\mathfrak{S}$ (stacks) is the set of processes defined by the grammar:*

$$P, Q ::= \textstyle\sum_{i \in I} \alpha_i.P_i \;\Big|\; P \mid Q \;\Big|\; \nu x\, P \;\Big|\; !P \;\Big|\; B_x$$

where B_x represents a buffer of type T.

The operational semantics of π_T is the same as the one defined in Section 4, except that the last congruence rule ($P \equiv P \mid \emptyset_x$) only applies when P is a pure π-calculus process (i.e. not already containing a buffer), in order to avoid behaviours that do not represent FIFO or LIFO strategies. Furthermore, the rules for bags (in_{bag} and out_{bag}) should be interpreted as rules for stacks (resp. queues) in the sense that the transitions in the premises should be those defined for stacks (resp. queues) in Definition 4.

6.1 Impossibility of encoding queues and stacks

In this section we show that it is not possible to find a valid encoding using the π_a-calculus for queues and stacks modulo weak asynchronous bisimilarity.

The result in this section depends critically on the following lemma, which is known in literature ([22], Lemma 5.3.2).

Lemma 1. *Let P be a process in the π_a-calculus and assume that $P \xrightarrow{\overline{x}y} \xrightarrow{\alpha} P'$. Then $P \xrightarrow{\alpha} \xrightarrow{\overline{x}y} \equiv P'$.*

Theorem 3. *Let $[\![\ ]\!]$ be an encoding from $\pi_{\mathfrak{Q}}$ into π_a (resp. from $\pi_{\mathfrak{S}}$ into π_a). Then there exists $P \in \pi_{\mathfrak{Q}}$ (resp. $P \in \pi_{\mathfrak{S}}$) such that $[\![P]\!] \not\approx P$.*

Proof. We prove the theorem by contradiction, for $P \in \pi_{\mathfrak{Q}}$. The case of $P \in \pi_{\mathfrak{S}}$ is analogous.

Let P be a queue B_x of the form $\cdots \triangleleft y \triangleleft z$ with $y \neq z$. Then we have:

$$B_x \xrightarrow{\overline{x}z} \xrightarrow{\overline{x}y} \tag{2}$$

Since we are assuming $[\![B_x]\!] \approx B_x$, we also have $[\![B_x]\!] \stackrel{\overline{x}z}{\Longrightarrow}\stackrel{\overline{x}y}{\Longrightarrow}$. By using Lemma 1 we obtain $[\![B_x]\!] \xrightarrow{\tau}{}^* \xrightarrow{\overline{x}z} \xrightarrow{\overline{x}y} \xrightarrow{\tau}{}^*$. Using Lemma 1 again we get: $[\![B_x]\!] \xrightarrow{\tau}{}^* \xrightarrow{\overline{x}y} \xrightarrow{\overline{x}z} \xrightarrow{\tau}{}^*$. Since $[\![B_x]\!] \approx B_x$ we have $B_x \stackrel{\overline{x}y}{\Longrightarrow}\stackrel{\overline{x}z}{\Longrightarrow}$, and, since a buffer in isolation does not give rise to τ steps, we also have

$$B_x \xrightarrow{\overline{x}y} \xrightarrow{\overline{x}z}$$

By the latter, and (2), and Remark 1, we have that B cannot be a queue.

Remark 3. We could give a stronger result, namely that for any encoding $[\![\ ]\!] : \pi_{\mathfrak{Q}} \to \pi_a$ (resp. $[\![\ ]\!] : \pi_{\mathfrak{S}} \to \pi_a$) and any queue (resp. any stack) B_x, $[\![B_x]\!] \not\approx B_x$. We leave the proof to the interested reader. The idea is that if B_x contains less than two elements, then we can always make input steps so to get a queue with two elements.

6.2 Impossibility of encoding stacks in the π-calculus without mixed-choice operator.

In this section we prove that stacks cannot be encoded in the language obtained by adding a separate-choice construct to π_a. We start by defining the π-calculus with separate choice.

The π-calculus with separate choice: π_{sc} This is a fragment of the synchronous π-calculus where mixed guarded choice is replaced by separate choice. The syntax is the following:

$$P, Q, \ldots := \sum_{i \in I} x_i(y_i).P_i \;\Big|\; \sum_{i \in I} \overline{x}_i z_i.P_i \;\Big|\; \nu x P \;\Big|\; P \mid Q \;\Big|\; !P$$

Here I is a set of indexes. Note that we have omitted the process 0 since it can be represented as the empty summation.

The definition of the transition semantics is the same as the one of the asynchronous π-calculus (Table 1), except for the rules *in* and *out*, that are replaced by the following ones:

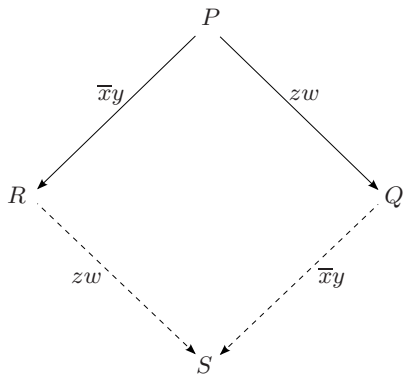

Fig. 3. Lemma of confluence

$$(in)\ \frac{}{\sum_{i\in I} x_i(y_i).P_i \overset{x_j(z_j)}{\rightarrow} P_j[z_j/y_j]} \qquad (out)\ \frac{}{\sum_{i\in I} \overline{x}_i z_i.P_i \overset{\overline{x}_j z_j}{\rightarrow} P_j}$$

The crucial property here is a sort of confluence that holds in the separate-choice π-calculus, as proved in [20](Lemma 4.1):

Lemma 2 (Confluence). *Let $P \in \pi_{sc}$. Assume that $P \xrightarrow{\overline{x}y} R$ and $P \xrightarrow{zw} Q$. Then there exists $S \in \pi_{sc}$ such that $Q \xrightarrow{\overline{x}y} S$ and $R \xrightarrow{zw} S$.*

Since we are working with weak asynchronous bisimilarity we need to consider the possible τ transitions. Therefore, we need the following extension of the above lemma.

Lemma 3 (Confluence with τ). *Let $P \in \pi_a$. Assume that $P \xrightarrow{\tau} R$ and $P \xrightarrow{\overline{x}y} Q$. Then, either*

1. *$P \xrightarrow{xz}$ for any z, or*
2. *there exists $S \in \pi_{sc}$ such that: $Q \xrightarrow{\tau} S$ and $R \xrightarrow{\overline{x}y} S$.*

Proof. We have to consider the possibility that the transition $P \xrightarrow{\tau} R$ is the result of a synchronization between $P_1 \xrightarrow{xy} Q_1$ and $P_2 \xrightarrow{\overline{x}y} Q_2$, where P_1 and P_2 are parallel subprocesses in P, and the latter transition is the one which induces $P \xrightarrow{\overline{x}y} Q$. If this is the case, then $P \xrightarrow{xz}$ for any z (note that x cannot be bound in P because $P \xrightarrow{\overline{x}y}$). On the other hand, if $P \xrightarrow{\tau} R$ does not involve the transition that induces $P \xrightarrow{\overline{x}y} Q$, then the proof is the same as for Lemma 2 (see [20], Lemma 4.1).

Theorem 4. *Let $[\![\ ]\!]$ be an encoding from π_{Θ} into π_{sc}. Then there exists $P \in \pi_{\Theta}$ such that $P \not\approx [\![P]\!]$.*

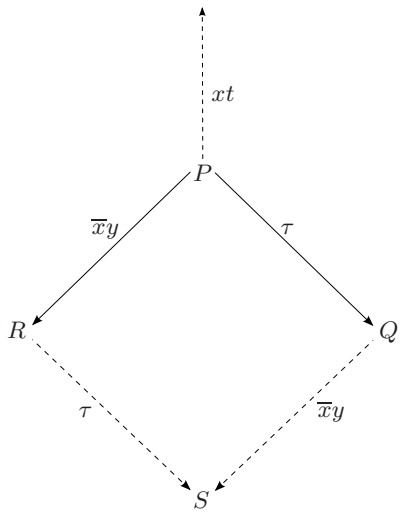

Fig. 4. Confluence with τ

Proof. Let P be a stack B_x of the form $y \triangleright \ldots$. Assume by contradiction that $B_x \approx [\![B_x]\!]$ (i.e. B_x is *weakly*asynchronously bisimilar to $[\![B_x]\!]$) . Then B_x must be weakly bisimilar to $[\![B_x]\!]$. In fact, if $B_x \xrightarrow{xz} B'_x \approx [\![B_x]\!]|\overline{x}z \approx B_x|\overline{x}z$, then we would have both $B_x \xrightarrow{\overline{x}y}$ and $B_x \xrightarrow{\overline{x}z}$, which by Remark 1 is not possible.

Let $z \neq y$. We have $B_x \xrightarrow{\overline{x}y} B'_x$ and $B_x \xrightarrow{xz}$. Since B_x is weakly bisimilar to $[\![B_x]\!]$, we have, for some P, $[\![B_x]\!] \xrightarrow{\tau}{}^* P \xrightarrow{\overline{x}y} \xrightarrow{\tau}{}^*$ and $P \xrightarrow{\tau}{}^* \xrightarrow{xz} \xrightarrow{\tau}{}^*$.

Let us assume that the number of τ steps before P inputs xz is not zero. That is to say, $P \xrightarrow{\tau} P' \xrightarrow{\tau}{}^* \xrightarrow{xz}$. From Lemma 3, we have that either $P \xrightarrow{xz}$ for any z, or $P' \xrightarrow{\overline{x}y}$. Then, by re-applying this reasoning to each sequence of τ transitions before the input of xz, we eventually get $P \xrightarrow{\overline{x}y}$ and $P \xrightarrow{xz}$. By applying Lemma 2 we have $P \xrightarrow{\overline{x}y} \xrightarrow{zz} P'$ and $P \xrightarrow{zz} \xrightarrow{\overline{x}y} P'$. From the fact that B_x and $[\![B_x]\!]$ are weakly bisimilar, we get $B_x \overset{\overline{x}y}{\Longrightarrow} \overset{xz}{\Longrightarrow}$ and $B_x \overset{xz}{\Longrightarrow} \overset{\overline{x}y}{\Longrightarrow}$. Finally, we observe that the last sequence is not possible, because after the input action xz a stack can only perform an output of the form $\overline{x}z$.

Figure 5 illustrates the fact that the encoded process must have a point where confluence occurs, which is used in this proof.

Remark 4. Also in this case we could give a stronger result, namely that for any encoding $[\![\ ]\!] : \pi_{\Theta} \to \pi_{sc}$ and any stack B_x, $[\![B_x]\!] \not\approx B_x$. Again the idea is that if B_x is not in the right form (i.e. it is empty), then we can make an input step so to get a stack with one element.

7 Related work

The first process calculi proposed in literature (CSP [6,13], CCS [16,17], ACP [2]) were all based on synchronous communication primitives. This is because

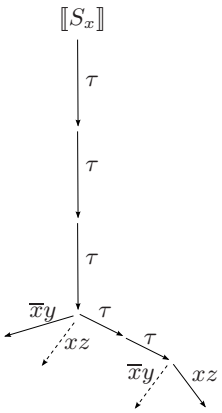

Fig. 5. Impossibility to encode a stack.

synchronous communication was considered somewhat more basic, while asynchronous communication was considered a derived concept that could be expressed using buffers (see, for instance, [13]). Some early proposals of calculi based purely on asynchronous communication were based on forcing the interaction between processes to be always mediated by buffers [3,9]. This is basically the same principle that we use in this paper for the definition of $\pi_{\mathfrak{B}}$.

At the beginning of the 90's, asynchronous communication became much more popular thanks to the diffusion of the Internet and the consequent increased interest for widely distributed systems. The elegant mechanism for asynchronous communication (the asynchronous send) proposed in the asynchronous π-calculus [14,5] was very successful, probably because of its simple and basic nature, in line with the tradition of process calculi. Thus it rapidly became the standard approach to asynchrony in the community of process calculi, and it was adopted, for instance, also in Mobile Ambients [8]. A communication primitive (*tell*) similar to the asynchronous send was also proposed, independently, within the community of Concurrent Constraint Programming [23].

We are not aware of any attempt to compare the two approaches to asynchrony (the one with explicit buffers and the one with the asynchronous send). However, our negative results concerning the non-encodability of stacks and queues in π_a use some properties of the asynchronous π-calculus that had been already presented in literature [20,22]. Similar properties were also investigated in [12,24] with the purpose of characterizing the nature of asynchronous communication.

An interesting study of various levels of asynchrony in communication for Linda-like languages has been carried out in [7]. In this paper, the authors investigate three different semantics of the output operation: the *instantaneous*, the *ordered*, and the *unordered* semantics. The first two essentially correspond to the semantics defined in this paper for π_a and $\pi_{\mathfrak{B}}$, respectively. The third one

corresponds to the semantics for $\pi_{\mathfrak{B}}$ with the additional possibility of temporal reordering of messages between their sending and their arrival. As argued in the introduction, the last two cases should coincide in languages which do not have the possibility of detecting the absence of a message, so these three calculi (π_a, $\pi_{\mathfrak{B}}$, and $\pi_{\mathfrak{B}}$ with unordered semantics) should be equivalent (up to weak asynchronous bisimilarity)[3].

8 Conclusion and future work

In this paper we have investigated the relation between the asynchronous π-calculus and a calculus $\pi_{\mathfrak{B}}$ where asynchronous communication is achieved via the explicit use of buffers. We have proved that there is a tight correspondence when the buffers are bags, namely we have exhibited encodings in both directions correct with respect to asynchronous bisimulation. For queues and stacks, on the contrary, we have proved an impossibility result, namely that they cannot be translated into asynchronously bisimilar processes belonging to the asynchronous π-calculus.

We aim at applying these results for modeling and verifying (using the tools developed for the asynchronous π-calculus) widely distributed systems with asynchronous and bag-like communication.

Another line of research is to develop variants of the asynchronous π-calculus in which communication is based on stack-like and queue-like disciplines, and investigate their theories. The motivation is to model and verify distributed systems with the corresponding kind of communication.

References

1. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.
2. J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1,3):109–137, 1984.
3. J. A. Bergstra, J. W. Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In A. W. R. S. D. Brookes and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, volume 197 of *LNCS*, pages 76–95, Pittsburgh, PA, July 1984. Springer.
4. M. Boreale, R. D. Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Information and Computation*, 172(2):139–164, 2002.
5. G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.
6. S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
7. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing three semantics for linda-like languages. *Theoretical Computer Science*, 240(1):49–90, 2000.

[3] This conjecture is due to Roberto Gorrieri.

8. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science (TCS)*, 240(1):177–213, 2000.
9. F. S. de Boer, J. W. Klop, and C. Palamidessi. Asynchronous communication in process algebra. In A. Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 137–147, Santa Cruz, CA, June 1992. IEEE Computer Society Press.
10. D. Gorla. On the relative expressive power of asynchronous communication primitives. In L. Aceto and A. Ingolfsdottir, editors, *Proc. of 9th Intern. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'06)*, volume 3921 of *LNCS*, pages 47–62. Springer, 2006.
11. D. Gorla. On the criteria for a 'good' encoding: a new approach to encodability and separation results. Technical report, Università di Roma "La Sapienza", 2007.
12. J. He, M. B. Josephs, and C. A. R. Hoare. A theory of synchrony and asynchrony. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods, Proc. of the IFIP WG 2.2/2.3, Working Conf. on Programming Concepts and Methods*, pages 459–478. North-Holland, 1990.
13. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
14. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.
15. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.
16. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, New York, NY, 1980.
17. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
18. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 & 41–77, 1992. A preliminary version appeared as Technical Reports ECF-LFCS-89-85 and -86, University of Edinburgh, 1989.
19. U. Nestmann. What is a 'good' encoding of guarded choice? *Journal of Information and Computation*, 156:287–319, 2000. An extended abstract appeared in the *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*.
20. C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. A short version of this paper appeared in POPL'97. `http://www.lix.polytechnique.fr/~catuscia/papers/pi_calc/mscs.pdf`.
21. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 1998.
22. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
23. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.
24. P. Selinger. First-order axioms for asynchrony. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR97: Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 376–390. Springer-Verlag, 1997.