BRICS

Basic Research in Computer Science

BRICS RS-97-45 U. Nestmann: What Is a ‘Good’ Encoding of Guarded Choice?

What Is a ‘Good’ Encoding of Guarded Choice?

Uwe Nestmann

BRICS Report Series RS-97-45

ISSN 0909-0878 December 1997

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/97/45/`

What is a ‘Good’ Encoding of Guarded Choice?*

Uwe Nestmann[†]
BRICS[‡]Aalborg University, Denmark
uwe@cs.auc.dk

December 1997

Abstract

The π-calculus with synchronous output and mixed-guarded choices is strictly more expressive than the π-calculus with asynchronous output and no choice. As a corollary, Palamidessi recently proved that there is no fully compositional encoding from the former into the latter that preserves divergence-freedom and symmetries. This paper shows that there are nevertheless ‘good’ encodings between these calculi.

In detail, we present a series of encodings for languages with (1) input-guarded choice, (2) both input- and output-guarded choice, and (3) mixed-guarded choice, and investigate them with respect to compositionality and divergence-freedom. The first and second encoding satisfy all of the above criteria, but various ‘good’ candidates for the third encoding—inspired by an existing distributed implementation—invalidate one or the other criterion. While essentially confirming Palamidessi’s result, our study suggests that the combination of strong compositionality and divergence-freedom is too strong for more practical purposes.

[*] This is a revised and slightly extended version of a paper published in the Proceedings of EXPRESS’97 (5th International Workshop on Expressiveness in Concurrency), volume 7 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers.

[†] The work was mainly carried out while the author was supported by a post-doc fellowship from ERCIM (*European Research Consortium for Informatics and Mathematics*) and partially supported by the ESPRIT CONFER-2 WG-21836. The work was revised and completed under a grant from BRICS.

[‡] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

1 Introduction

The invention of the π-calculus [MPW92] by Milner, Parrow, and Walker, has triggered a wide range of encodings of other calculi into it, due to its well-developed semantic theory, but also because of the similarities between encodings and actual implementations by the use of name-passing. Soon the question arose, which operators would be responsible for this surprisingly expressive power of the original π-calculus. This paper contributes to the understanding of the role of choice operators for the expressiveness of the π-calculus.

A widely-used method for measuring the relative expressiveness of calculi is by (mutual) encodings. A calculus is considered more expressive than another, if it represents the target language of an encoding of the other calculus. The meaningfulness of such propositions rests on the (syntactic and semantic) properties that are preserved and/or reflected by the encoding. An example criterion for being a 'good' encoding is the popular notion of *full abstraction*: in the context of process calculi, it requires that the equivalence of terms is both preserved and reflected (cf. Sangiorgi [San93]). Of course, the choice of equivalence is crucial. Weak bisimulation equivalences and congruences have become prominent in this area, because they permit abstraction from internal steps that might be added by an encoding, and also because they provide handy proof techniques. Yet, weak bisimulation is not the only interesting equivalence relation; in particular, it is insensitive to divergence. Consequently, an encoding that uses infinite loops may nevertheless be fully abstract with respect to weak bisimulation; for stating that an encoding is divergence-free, we need additional arguments (or a different equivalence). If full abstraction can not be achieved for any known equivalences, then the mere preservation of states' properties like

- deadlock-freedom: it is possible to perform some transition
- livelock-freedom: it is always possible to escape infinite internal computations
- divergence-freedom: there are no infinite internal computations

may also used to argue that an encoding can be accepted as 'good'.

More traditional methods of measuring the expressiveness of models for concurrency are by checking the existence of solutions for certain well-known problems, e.g. algorithms for mutual exclusion [RL94], consensus [Ben83], and leader election [Bou88] in symmetric distributed systems, or else by checking their Turing power via the construction of random access machines [BGZ97]. Here, a model (possibly provided by a process calculus) is considered more expressive than another, if it provides solutions to more problems.

Many variations of the above-mentioned measures have been applied to study the expressiveness of a whole family of name-passing process calculi. Calculi with *asynchronous* name-passing like the ν-calculus [HT92] and the corresponding variant of the choice-free π-calculus [Bou92] have recently attracted particular interest, since they still have surprisingly expressive power. To study their expressiveness relative to the original π-calculus [MPW92], the existence of 'good' encodings of operators for synchronous output and guarded choice (we are not concerned with matching operators) is investigated. Figure 1 summarizes the respective results that are known from the literature, on which we comment in the following paragraphs. The subscripts a and s denote calculi with *a*synchronous and

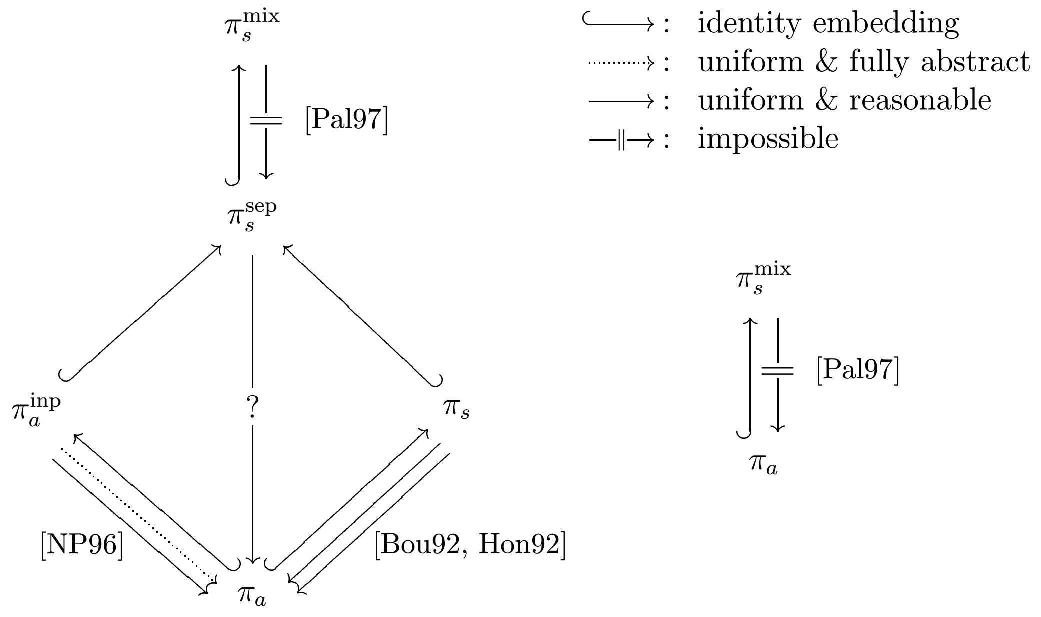


Figure 1: Encodings for choice and synchrony in the asynchronous π-calculus

*s*ynchronous output, respectively, whereas the superscripts inp, sep, and mix denote, which kind of guarded choice is contained in the language: input-guarded, separate (choices with either only input- or only output-guards), or mixed.

Synchronous output can be encoded by means of asynchronous output using explicit acknowledgement channels: Boudol [Bou92] provided an encoding from π_s into π_a and proved its correctness as *adequacy* (just the reflection part of full abstraction) with respect to Morris-style contextual congruence; Honda [Hon92] gave a more efficient (in terms of number of low-level steps needed for implementing one high-level step) encoding and showed correctness as adequacy with respect to some weak bisimulation, and as preservation of satisfaction for logical formulae via an embedding of a modal logic.

Choice operators play a crucial role in assessing the expressiveness of the original (synchronous) π-calculus and its asynchronous descendants, since they are usually present in the former, but not [HT92, Bou92] (or only restricted [ACS96]) in the latter. Nestmann and Pierce showed in [NP96] that at least input-guarded choice can be encoded into π_a and proven to be fully abstract with respect to weak bisimulation [HT92, ACS96] for an encoding with infinite loops, and fully abstract with respect to coupled simulation for a divergence-free encoding. However, Palamidessi proved that there is no *uniform* encoding from π_s^{mix} into π_a that preserves a *reasonable* semantics. In other words, it is impossible to encode mixed-guarded choice with only asynchronous name-passing, when imposing Palamidessi's criteria:

uniform means, according to Palamidessi [Pal97]: for all source terms P, P_1, and P_2,

$$\llbracket\,\sigma(P)\,\rrbracket \;=\; \sigma(\llbracket\,P\,\rrbracket) \tag{1}$$
$$\llbracket\,P_1 \,|\, P_2\,\rrbracket \;=\; \llbracket\,P_1\,\rrbracket \mid \llbracket\,P_2\,\rrbracket \tag{2}$$

where σ denotes an injective renaming function. While the first condition merely requires that the candidate encoding be compatible with the renaming of free channels, the second condition represents the requirement that an encoding of mixed-guarded choice should be 'truly distributed', in the sense that it is not allowed to have a mediating process M, as in

$$\llbracket\,P_1 \,|\, P_2\,\rrbracket \;=\; (\boldsymbol{\nu} x_1,\dots,x_n)\,(\,\llbracket\,P_1\,\rrbracket \mid M \mid \llbracket\,P_2\,\rrbracket\,) \tag{3}$$

which could monitor parallel activities via the internal names $x_1,\dots,x_n$.

reasonable means, according to Palamidessi [Pal97]: "*We call* reasonable *a semantics which distinguishes two processes P and Q whenever in some computation of P the actions on certain intended channels are different from those in any computation of Q.*" This includes sensitivity to divergence since an action on an intended channel in some computation of P is required to happen in *any* computation of Q, so infinite loops in computations of Q that do not mention the intended action are detected.

Palamidessi's impossibility theorem for encodings of mixed choice is a corollary of a formal separation result between π_s^{mix} and π_a (and also π_s^{sep}). Similar to previous work of Bougé within the setting of CSP [Bou88], it is based on the ability or inability of the calculi to express leader election algorithms in symmetric networks (here: of π-calculus processes). Such algorithms require the ability to break symmetries in communication graphs, like the atomic agreement of two processes between two values (e.g. the process_id of the leader). π_s^{mix} can break such symmetries, e.g. in the parallel composition of 'symmetric' choices

$$P \mid Q \;\overset{\mathrm{def}}{=}\; y_0![0].P_0 \;+\; y_1?[x].P_1 \;\Big|\; y_0?[x].Q_0 \;+\; y_1![1].Q_1 \tag{4}$$

where symmetry means that the program code of P and Q is identical under structural congruence and renaming of process_id's modulo 2, we end up with either of the asymmetric systems $P_0|Q_0\{{}^0\!/\!_x\}$ or $P_1\{{}^1\!/\!_x\}|Q_1$. In contrast, the above symmetric system could not be written in π_a since mixed-guarded choice is not a part of this language. Instead, corresponding systems with concurrently enabled input- and output-actions (see the diagram aside for a process which mimics the behavior of the above P) would behave under the regime of a confluence property. Here, since both P and the corresponding Q would behave confluently, the symmetry of $P \mid Q$ would be preserved under computation, i.e. no leader could be elected (an attempt for leader election in π_a would go on for an infinite amount of time, while leader election in π_s succeeds in finite time). Since encodings that satisfy Palamidessi's unifor-

$$\begin{tikzcd}
 & P \arrow[dl, "{y_0![0]}"'] \arrow[dr, "{y_1?[x]}"] & \\
P_0 \arrow[dr, "{y_1?[x]}"'] & & P_1 \arrow[dl, "{y_0![0]}"] \\
 & P' &
\end{tikzcd}$$

mity requirements preserve symmetry of networks, and since ‘reasonable’ semantics are divergence-sensitive, the impossibility result can be derived.

There has been remarkable interest in concurrent programming languages that include mixed choice for channel-based synchronous communication, as exemplified with Concurrent ML [Rep91] and Facile [TLK96]. Despite Palamidessi’s impossibility result, there also exist algorithms for the distributed implementation of such languages, and in particular of mixed choice, e.g. by Bernstein [Ber80], Buckley and Silberschatz [BS83] and Knabe [Kna93], which all have been proven—although rather informally—to be correct or, at least, to be deadlock-free.[1] The question arises how these practically satisfactory implementations relate to Palamidessi’s impossibility result. It is worthwhile to ask, whether the criteria of uniformity and reasonableness are adequate or, maybe, too strong in that the above implementations invalidate them.[2]

This paper sheds more light on the correctness of distributed implementations of choice by formally studying choice encodings (apart from uniformity) with respect to the preservation of deadlock-, divergence-, and livelock-freedom. These properties are tightly related to Palamidessi’s criteria, but they seem more precise than requiring to ‘preserve a reasonable semantics’.

Overview of the paper

First, we introduce the formal π-calculus framework for our study (§2). Then, quickly recalling the uniform encoding of input-guarded choice of [NP96], we extend it to a uniform encoding of output-guards in the context of separate choices (§3). For this case, we show how to prove important ‘reasonable’ properties like deadlock- and divergence-freedom. By the attempt to smoothly reuse this encoding for the case of choices with mixed guards (§4), we expose the inherent problems of deadlock by cyclic waiting and ‘incestuous’ self-communication. To overcome these problems, we propose various solutions, which, however, invalidate either uniformity or reasonableness. One successful possibility for an encoding of mixed choice is finally suggested by restricting the source and, at the same time, extending the target language. Finally (§5), we offer some possible interpretations of our work.

Throughout the paper, we emphasize the exposition of encodings, algorithms, and trade-offs instead of just presenting the formal proofs, which are rather straightforward in most cases. Those technicalities that are more interesting, because they require new theories or admit the application of recent techniques, are assembled in the Appendix.

[1] It has only recently (14 years after publication) turned out that the algorithm presented by Buckley and Silberschatz is not deadlock-free [KS97], although otherwise stated [BS83]. This emphasizes the need for more formal analysis of distributed implementations and, in particular, of guarded choice.

[2] Note also that all of the previous encodings in Figure 1 satisfy Palamidessi’s criteria, with one exception: the dotted arrow from π_a^{inp} to π_a indicates that one of the encodings studied in [NP96] is uniform and fully abstract, but not reasonable; this is due to infinite loops that were necessary to achieve full abstraction with respect to weak bisimulation, otherwise full abstraction could only be proved with respect to the weaker notion of coupled simulation.

2 Technical Preliminaries

We introduce various polyadic π-calculi [Mil93] as source and target languages. Let $\mathbf{N}$ be a countable set of *names*, and let $\tilde{x}$ denote a finite tuple $x_1, \cdots, x_n$ of names. The source languages $\mathbb{S}^\Sigma$ with $\Sigma \in \{\text{mix}, \text{sep}, \text{inp}\}$ are defined by:

$$
\begin{array}{lll}
P & ::= & N^\Sigma \quad \big| \quad P|P \quad \big| \quad (\boldsymbol{\nu} y)\, P \quad \big| \quad y?^*[\tilde{x}].P \\[1em]
N^{\text{mix}} & ::= & \sum_{i \in I} \pi_i.P_i \qquad\qquad \text{with} \qquad\qquad \pi \ ::= \ y![\tilde{z}] \quad \big| \quad y?[\tilde{x}] \\
N^{\text{sep}} & ::= & \sum_{i \in I} y_i?[\tilde{x}_i].P_i \quad \big| \quad \sum_{i \in I} y_i![\tilde{z}_i].P_i \\
N^{\text{inp}} & ::= & \sum_{i \in I} y_i?[\tilde{x}_i].P_i \quad \big| \quad y![\tilde{z}]
\end{array}
$$

where $x, y, z \in \mathbf{N}$, and I ranges over finite sets of indices i. The languages $\mathbb{S}^\Sigma$ are polyadic versions of π_s^{mix}, π_s^{sep}, and π_a^{inp}, respectively, of Figure 1.

The informal semantics of parallel composition and restriction is as usual. In choices, we use an *output guard* $y![\tilde{z}].P$ to denote the emission of names $\tilde{z}$ along channel y before behaving as P, and an *input guard* $y?[\tilde{x}].P$ to denote the reception of arbitrary names $\tilde{z}$ along channel y and afterwards behaving as $P\{^{\tilde{z}}/_{\tilde{x}}\}$, which denotes the simultaneous substitution of all free occurrences of names $\tilde{x}$ by the received names $\tilde{z}$, while silently performing α-conversion, wherever necessary. A *replicated input guard* $y?^*[\tilde{x}].P$ denotes a process that allows us to spawn off arbitrary instances of the form $P\{^{\tilde{z}}/_{\tilde{x}}\}$ in parallel by repeatedly receiving names $\tilde{z}$ along channel y. We use $N_1 + N_2$ to abbreviate binary choice (commutative and associative), and $\mathbf{0}$ to denote empty choice.

Operator precedence is, in decreasing order of binding strength: (1) substitution, (2) prefixing, restriction, replication, (3) choice, and (4) parallel composition. A term is *guarded* when it occurs as a subterm of some guard. In $y![\tilde{z}]$ and $y?[\tilde{x}]$, y is called *subject*, while $\tilde{x}$ and $\tilde{z}$ are called *objects*. Free and bound names $\mathit{fn}(P)$ and $\mathit{bn}(P)$ of a process P are defined as usual. Created names are assumed to be fresh, i.e. not occurring in any other term.

The target language $\mathbb{T}$ is then defined as replacing N^Σ with just asynchronous output, i.e. messages, and only single input-prefixes instead of choice

$$P \ ::= \ y![\tilde{z}] \quad \big| \quad y?[\tilde{x}].P \quad \big| \quad P|P \quad \big| \quad (\boldsymbol{\nu} y)\, P \quad \big| \quad y?^*[\tilde{x}].P$$

and, thus, is a polyadic version of π_a. Here, $\mathbf{0}$ can be defined by $(\boldsymbol{\nu} x)\,(x![])$.

For the sake of readability, we use primitive boolean names $\mathsf{t}, \mathsf{f} \in \mathbb{B}$ and conditional operators $\mathsf{test}\ y\ \mathsf{then}\ P\ \mathsf{else}\ Q$ for destructively reading and testing the current (boolean) value on channel y. The above conditional is an abbreviation of $y?[x]\ .\ \mathsf{if}\ x\ \mathsf{then}\ P\ \mathsf{else}\ Q$ with the usual meaning of if, which is only defined, if the name received for x is a boolean. For $\mathbb{T}$ with booleans, we require $x, y \in \mathbf{N}$, while $z \in \mathbf{V} := \mathbf{N} \cup \mathbb{B}$. Note that the boolean primitives can be cleanly encoded into the intended target language $\mathbb{T}$ (cf. [Nes96]).

$$
\begin{array}{llrl}
\mathbb{S}^{\mathrm{mix}}, \mathbb{S}^{\mathrm{sep}} : & & (\cdots + y?[\tilde{x}].P) \mid (y![\tilde{z}].Q + \cdots) & \rightarrow P\{^{\tilde{z}}\!/_{\tilde{x}}\} \mid Q \\
\mathbb{S}^{\mathrm{inp}} : & & (\cdots + y?[\tilde{x}].P) \mid y![\tilde{z}] & \rightarrow P\{^{\tilde{z}}\!/_{\tilde{x}}\} \\
\mathbb{S}^{\Sigma}, \mathbb{T} : & & y?^{*}[\tilde{x}].P \mid y![\tilde{z}] & \rightarrow P\{^{\tilde{z}}\!/_{\tilde{x}}\} \mid y?^{*}[\tilde{x}].P \\
\mathbb{T} : \mathit{if}\ \ \tilde{z} \subseteq \mathbf{N} & \mathit{then} & y?[\tilde{x}].P \mid y![\tilde{z}] & \rightarrow P\{^{\tilde{z}}\!/_{\tilde{x}}\} \\
\mathbb{T} : & & \mathsf{test}\ y\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid y![\mathsf{t}] & \rightarrow P \\
\mathbb{T} : & & \mathsf{test}\ y\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid y![\mathsf{f}] & \rightarrow Q \\
\mathit{if} \quad P \rightarrow P' & \mathit{then} & (\boldsymbol{\nu} x)\, P & \rightarrow (\boldsymbol{\nu} x)\, P' \\
\mathit{if} \quad P \rightarrow P' & \mathit{then} & Q \mid P & \rightarrow Q \mid P' \\
\mathit{if}\ P \equiv Q \rightarrow Q' \equiv P' & \mathit{then} & P & \rightarrow P' \\
\end{array}
$$

$$
\begin{array}{cc}
P \mid Q \equiv Q \mid P & (\boldsymbol{\nu} y)\, (\boldsymbol{\nu} x)\, P \equiv (\boldsymbol{\nu} x)\, (\boldsymbol{\nu} y)\, P \\
P \mid (Q|R) \equiv (P|Q) \mid R & (\boldsymbol{\nu} y)\, P \mid Q \equiv (\boldsymbol{\nu} y)\, (P \mid Q) \quad \mathit{if} \quad y \notin \mathsf{fn}(Q)
\end{array}
$$

Figure 2: Reduction relation & structural congruence

The formal semantics for the languages $\mathbb{S}^{\Sigma}$ and $\mathbb{T}$ is presented in Figure 2 as a reduction relation $\rightarrow$ (with reflexive-transitive closure $\Rightarrow$) on structural congruence classes (silently including α-conversion). The only difference among the languages is in the rules for communication, which arise from the different kinds of choices and receptors. With Milner [Mil93], we assume that all processes are well-typed according to the correct use of polyadic channels, i.e. matching senders and receivers always have the same expectation about the arity or the boolean type of transmitted values.

3 Implementing Separate Choice

Intuitively, branches in a guarded choice may be seen as individual, but concurrently available processes that have to synchronize each others progress by mutual exclusion. Reminiscent of distributed implementations, we should use parallel composition to express this concurrent activity of branches.

The encoding scheme in Figure 3 implements choice-states as boolean messages on private channels l, so-called *locks*: t means that no branch in the current choice has yet been chosen, f means the contrary (so the initial value must be t). Whenever (an encoding of) a branch wants to proceed, it must test its associated lock; it must also explicitly reset the lock after having tested it in order to enable competing branches to also test the choices' state. We use the scheme for several encodings. Instead of presenting them all at once, and studying their properties afterwards, we proceed stepwise, which allows us to emphasize their differences. Uniformity [Pal97] is guaranteed by the compositional encoding of parallel composition and restriction (see Appendix A).

$$\begin{array}{lcl}
[\![\, P_1 \mid P_2 \,]\!] & \overset{\text{def}}{=} & [\![\, P_1 \,]\!] \mid [\![\, P_2 \,]\!] \\
[\![\, (\boldsymbol{\nu} x)\, P \,]\!] & \overset{\text{def}}{=} & (\boldsymbol{\nu} x)\, [\![\, P \,]\!] \\
[\![\, \sum\limits_{i \in I} \pi_i . P_i \,]\!] & \overset{\text{def}}{=} & (\boldsymbol{\nu} l)\, (\; l![\mathsf{t}] \mid \prod\limits_{i \in I} [\![\, \pi_i . P_i \,]\!]_l \;)
\end{array}$$

Figure 3: Encoding scheme $\mathbb{S}^{\Sigma} \rightarrow \mathbb{T}$

3.1 Input-guarded choice

According to [NP96], input-guarded choice can be encoded as shown in Figure 3 and 4. The only non-trivial case is for input-guards: after receiving a value from the environment, the name l is used to test whether the current guard is allowed to proceed (by reading t from l), or whether it has to be aborted (by reading f from l) and obliged to resend the received value. The encoding obeys strong invariant properties on the use of locks:

- "On each lock, at most one message may ever be available at any time".
  This guarantee implements *locking*, which enables mutual exclusion.
- "Each reader of a lock eventually writes back to the lock".
  This obligation enables the correct abortion of non-chosen branches.

It is crucial for the correctness that send-requests that do not lead to communication—because of the receiver being aborted—are resent, i.e. possibly passed on to another receiver waiting on the same channel. Furthermore, abortion would not be handled correctly, were we not guaranteed that, once read, lock l eventually becomes available again with message f. This encoding preserves a 'reasonable' semantics since it is fully abstract with respect to coupled simulation, which implies deadlock-freedom, and it is also divergence-free. (In fact, a result stronger than full abstraction holds since it was shown that terms and their encodings are congruent themselves, thus cannot be distinguished by any context.)

$$\begin{array}{lcl}
[\![\, y![\tilde{z}] \,]\!] & \overset{\text{def}}{=} & y![\tilde{z}] \\
[\![\, y?[\tilde{x}].P \,]\!]_l & \overset{\text{def}}{=} & y?[\tilde{x}]\ .\ \mathsf{test}\ l\ \mathsf{then}\ (\; l![\mathsf{f}] \mid [\![\, P \,]\!] \;)\ \mathsf{else}\ (\; l![\mathsf{f}] \mid y![\tilde{x}] \;) \\
[\![\, y?^*[\tilde{x}].P \,]\!] & \overset{\text{def}}{=} & y?^*[\tilde{x}].[\![\, P \,]\!]
\end{array}$$

Figure 4: $\mathbb{S}^{\text{inp}} \rightarrow \mathbb{T}$

3.2 Output-guarded choice

If output is blocking, i.e. guarding some behavior that is only enabled if the output was successful, then synchronization is no longer local to the receiver's choice. The idea is (cf. Figure 5) that a sender asynchronously transmits its values $\tilde{z}$ together with a private acknowledgement channel a, which can be used just once by some matching receiver to signal either success or failure, i.e. either enabling the sender's continuation to proceed, or to abort it. Since output-guards are also branches in a choice whose state must be tested, the corresponding lock r is, in addition to $\tilde{z}$ and a, transmitted to some matching receiver that then performs the required choice-test.

Input-guards, revisited The encoding is more elaborate due to the increased information that is transmitted by send-requests. Firstly, there are now two locks that have to be tested in some order. In Figure 5, we chose to test the *local* lock l first, and only in the case of a positive outcome to test the *remote* lock r. (This particular order is useful in an actual distributed implementation, where remote communication is usually much more expensive than local communication.) Secondly, we have to use the acknowledgement channel correctly, which means that a positive acknowledgement may only be sent if both locks were tested positively. Thirdly, in the case that the test of sender's choice-lock was negative, we must not resend the send-request—instead, and only if the test of the receiver's choice-lock was positive, we have to restart the receiver process from the beginning by allowing it to try other send-requests. In Figure 5, this is implemented by recursively sending a trigger-signal to a replicated input process on b that represents the receiver-loop's entry point. In order to match this protocol of synchronous outputs, the encoding of input-guarded replication has to check the sender's lock, and based on its value either to *commit* and trigger a copy, or to *abort* the sender.

$$
\begin{array}{lcl}
\llbracket\, y![\tilde{z}].P \,\rrbracket_r & \overset{\mathrm{def}}{=} & (\boldsymbol{\nu} a)\,(\; y![r,a,\tilde{z}] \mid \mathsf{test}\ a\ \mathsf{then}\ \llbracket\, P \,\rrbracket\ \mathsf{else}\ \mathbf{0}\ ) \\
\llbracket\, y?[\tilde{x}].P \,\rrbracket_l & \overset{\mathrm{def}}{=} & (\boldsymbol{\nu} b)\,(b![] \mid\ b?^*[]. \\
 & & \qquad\qquad y?[r,a,\tilde{x}]. \\
 & & \qquad\qquad \mathsf{test}\ \ l \\
 & & \qquad\qquad \mathsf{then}\ \ \mathsf{test}\ \ r \\
 & & \qquad\qquad\qquad\quad \mathsf{then}\ \ l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket\, P \,\rrbracket \\
 & & \qquad\qquad\qquad\quad \mathsf{else}\ \ l![\mathsf{t}] \mid r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![] \\
 & & \qquad\qquad \mathsf{else}\ \ l![\mathsf{f}] \mid y![r,a,\tilde{x}]\ \ ) \\
\llbracket\, y?^*[\tilde{x}].P \,\rrbracket & \overset{\mathrm{def}}{=} & y?^*[r,a,\tilde{x}].\mathsf{test}\ r\ \mathsf{then}\ r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket\, P \,\rrbracket\ \mathsf{else}\ r![\mathsf{f}] \mid a![\mathsf{f}]
\end{array}
$$

Figure 5: $\mathbb{S}^{\mathrm{sep}} \rightarrow \mathbb{T}$

Evaluation An encoding is *deadlock/divergence-free*, if it does not *add* deadlocks/loops to the behavior of terms: a deadlock/loop that occurs in (some derivative of) an encoded term necessarily results from a deadlock/loop already occurring in (some derivative of) the original term. Note that divergence-freedom implies livelock-freedom.

To prove deadlock-freedom, we take advantage of type information for the channels that are added in the encoding. We refine channel types according to Kobayashi's classification [Kob97], which distinguishes between *reliable* and *un*reliable channels. The following three types of channels are reliable:

- *linear* channels, which are used just once (like our acknowledgement channels a),
- *replicated input* channels, whose input ends must not occur more than once, but whose output ends may be used arbitrarily often (like our restart channels b), and
- *mutex* channels, which need to obey the invariants (of our lock channels l) that we mentioned on page 8; also, a message must be available right after their creation.

Kobayashi also developed a typing system that provides a behavioral property for well-typed processes: every (immediate) deadlock can only be caused by unreliable channels. A subject reduction theorem extends the proposition to deadlocks that may ever occur in derivatives of well-typed processes.

As indicated above, every channel that is added by our choice encodings, is reliable. Since we can further show that every encoded term is well-typed with respect to Kobayashi's type system (when regarding every source-level channel as unreliable), we already get the desired proposition:

Proposition 3.2.1. $\mathbb{S}^{\text{sep}} \to \mathbb{T}$ *is deadlock-free.*

Proof. By type-checking. More details can be found in Appendix B. □

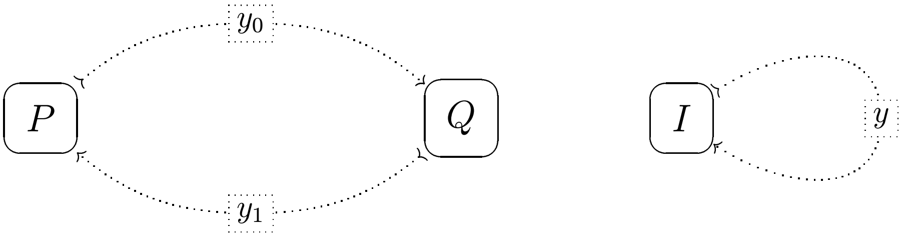
Proposition 3.2.2. $\mathbb{S}^{\text{sep}} \to \mathbb{T}$ *is divergence-free.*

Proof. The only possibility for an encoding to add an infinite loop would be in the translation of input-guards since it is only there that we use replication. In order to trigger a copy of this replication, three conditions must be met: (1) the receiver's lock must still contain t, (2) a matching send-request must be consumed from the environment, and (3) this sender's lock must contain f. However, in this situation, by 'looping back' the consumed message will not be given back to the system—in other words, the system's state is decreased. This cannot be done infinitely often unless an infinite number of matching send-requests is produced. This, in turn, is only possible by using replication, e.g. by $(\boldsymbol{\nu} x)\,(x![] \mid x?^*[].(x![] \mid \cdots y![\tilde{z}] \cdots))$, but then, due to the encoding of replicated input, this replication of messages must have been already present in the source language. [3] □

[3] Note that the presence of output-guarded replication in the source language would not invalidate this result, since in the encoding, this construct would be translated by mentioning a lock that always carries t due to the lack of competitors, thus invalidating condition (3) in the proof.

4 Implementing Mixed Choice

The naïve attempt is to simply reuse the encoding for separate choices of the previous section *as is* for encoding mixed choices. This seems sensible at first, because in both cases all input- and output-guards are branches in choices, so they should behave similar. However, we are faced with two sources of potential deadlock in the mixed setting: one is for the symmetric term $P \,|\, Q := y_0![0].P_0 + y_1?[x].P_1 \;\big|\; y_0?[x].Q_0 + y_1![1].Q_1$ of equation 4 in the Introduction, the other is for $I := y![z].P + y?[x].Q$. The deadlock situations may become clear from a spatial representation:



In process $[\![\, P \,|\, Q \,]\!]$, imagine the situation, where both receivers for y_0 and y_1 have input the matching request, and afterwards successfully tested their own choice-lock: here, both have to wait for their respective sender's choice-lock to become available again, but neither of them will do, so both receivers remain blocked forever. This symmetric cyclic-wait situation is very similar to the classical 'dining philosophers' problem [RL94], where several (in our case: two) processes compete for mutually exclusive access to forks (locks).

In process $[\![\, I \,]\!]$, the sender's request on y could be consumed by the competing receiver branch, which results in a deadlock situation, because the receiver would try to test the same lock twice, which is impossible.

Breaking the symmetry In Distributed Computing, one method to resolve cyclic dependencies among processes is by using time-outs or probabilistic algorithms for the attempt to acquire some lock. Then, however, we face the problem of infinite loops, such that randomized solutions are not 'reasonable' [Pal97], although it is known that solutions exist that guarantee progress with probability 1 [RL94]. If, in such cases, we assume fair execution schedulers, then divergence is not harmful anymore, as long as there is no danger for live-locks.

Another method, known from the distributed implementation of concurrent languages, is exploiting a total order among the threads in the system by, for example, always choosing the lock of the smaller thread first [Ber80, BS83, Kna93], when required to make a choice. Then, the above symmetric cyclic-wait situation is immediately prevented since both receivers choose the same thread, i.e. lock, as the first to interrogate. Note also that under a total order assumption symmetric networks according to [Pal97] do not exist.

In the following subsections, we adapt the methods of randomization (see §4.1) and total ordering of threads (see §4.2 and §4.3) to our case of encoding mixed choice into the asynchronous π-calculus, and we evaluate their properties.

4.1 A randomized solution

Randomization means removing determinism from an algorithm and adding randomly possible computation paths. In our case, instead of choosing a fixed order for testing the locks as in Figure 5, we might allow ourselves to test them nondeterministically in either order and allow first-phase locks to be given back (cf. [RL94]). Of course, in our target language we cannot trivially write down "*either receive from the second lock, or resend on the first lock*", because in order to do so, we would need a mixed choice construct. Note that we can neither use internal choice, because it would only delay potential deadlocks, which arise when the internal decision favors the branch "*waiting for the second lock*".

In Figure 6, we model a randomized solution based on the encoding in Figure 5 by only supplying a new clause for receivers.[4] We use a local state, implemented as a mutex channel s that carries a tag[5] (and a boolean value) that tells, whether none (tag N), the local (tag L) or the remote (tag R) lock are currently held by the receiver. The tag-information, initially N (w.l.o.g. with value f), is supplied by two processes, called lock-checkers, waiting at *lcl* and *rmt*, which try to get hold of the local lock l and remote lock r, respectively. After grabbing a lock, these processes need to read the current state: if the complementary

[4] A similar solution is used in the implementation of receivers in the join-calculus [FG96].

[5] We use this special syntax for the sake of readability; since we only need 3 different tags, we can easily simulate them by 2 boolean tags and use the corresponding if- and test-expressions for matching.

$$
\begin{array}{l}
\llbracket\, y?[\tilde{x}].P \,\rrbracket_l \;\overset{\mathrm{def}}{=}\; (\boldsymbol{\nu} b)\Big(\; b![] \mid b?^*[].y?[r,a,\tilde{x}] \;. \\
\quad (\boldsymbol{\nu} s, \mathit{lcl}, \mathit{rmt}, \mathit{rnd}, \mathit{bth})\;\big(\;
\begin{array}[t]{cl}
 & \mathit{lcl}?^*\,[]\,.\,l?[b_L]\;.\;s?[\mathit{tag},b]\;. \\
 & \qquad\qquad \mathsf{if}\ \mathit{tag}{=}\mathsf{R}\ \mathsf{then}\ \mathit{bth}![b_L,b] \mid s![\mathsf{N},\mathsf{f}]\ \mathsf{else}\ s![\mathsf{L},b_L] \mid \mathit{rnd}![] \\
\mid & \mathit{rmt}?^*\,[]\,.\,r?[b_R]\;.\;s?[\mathit{tag},b]\;. \\
 & \qquad\qquad \mathsf{if}\ \mathit{tag}{=}\mathsf{L}\ \mathsf{then}\ \mathit{bth}![b,b_R] \mid s![\mathsf{N},\mathsf{f}]\ \mathsf{else}\ s![\mathsf{R},b_R] \mid \mathit{rnd}![] \\
\mid & \mathit{rnd}?^*\,[]\,.\,s?[\mathit{tag},b]\;.\;(\,s![\mathsf{N},\mathsf{f}] \mid \mathsf{if}\ \mathit{tag}{=}\mathsf{L}\ \mathsf{then}\ l![b] \mid \mathit{lcl}![]\ \mathsf{else} \\
 & \qquad\qquad\qquad\qquad\qquad \mathsf{if}\ \mathit{tag}{=}\mathsf{R}\ \mathsf{then}\ r![b] \mid \mathit{rmt}![]\ \mathsf{else}\ \mathbf{0}\,) \\
\mid & \mathit{bth}?[b_L,b_R]\;.\;
\begin{array}[t]{llll}
\mathsf{if} & b_L \wedge b_R & \mathsf{then} & l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket\, P \,\rrbracket\ \mathsf{else} \\
\mathsf{if} & b_L & \mathsf{then} & l![\mathsf{t}] \mid r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![]\ \mathsf{else} \\
\mathsf{if} & \qquad\;\, b_R & \mathsf{then} & l![\mathsf{f}] \mid r![\mathsf{t}] \mid y![r,a,\tilde{x}] \\
 & & \mathsf{else} & l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{f}]
\end{array} \\
\mid & \mathit{lcl}![] \mid \mathit{rmt}![] \mid s![\mathsf{N},\mathsf{f}]\;\;\big)\Big)
\end{array}
\end{array}
$$

Figure 6: Randomized $\mathbb{S}^{\mathrm{sep}} \rightarrow \mathbb{T}$ for use as $\mathbb{S}^{\mathrm{mix}} \rightarrow \mathbb{T}$

lock is already held, then the two lock values are passed on to the analyzer process waiting at *bth* and the state s is initialized; otherwise, the state s is appropriately updated to announce success for getting the current lock and, in addition to this announcement, a randomizer process at *rnd* is started that competes with the lock-checkers for reading the state. If the randomizer succeeds in reading the state, it resets the state and resends the lock, while restarting the corresponding lock-checker. If both lock-checkers succeed reading the state without the randomizer interfering, then s is left with its initial value and is finally consumed by the active randomizer to terminate the system by resetting the state without restarting any of the lock-checkers and without restarting the randomizer itself. Note that after restarting the whole receiver at b in the case of local success ($b_L{=}\mathsf{t}$) and remote failure ($b_R{=}\mathsf{f}$), a new state will be created, when a new request on y arrives.

Evaluation As the encoding for separate choice, the randomized encoding for mixed choice in Figure 6 is uniform since restriction and parallelism are encoded purely compositionally. The randomized encoding is deadlock-free due to the ever present possibility of backing out, when a second-phase lock is not available: all receivers on the state channel—both lock-checkers and the randomizer—have equal priority, so in case of a potential dead-lock, the randomizer can help, since it is triggered, whenever a non-trivial state is set. (Note that, again, we only use channels of reliable type: b, *lcl*, *rmt*, and *rnd*, are replicated, *bth* is linear, and s is mutex. So we can apply Kobayashi's type system for the proof.)

However, the encoding is not divergence-free, since the randomizer introduces potentially infinite loops. Yet, under fair execution, divergence would be prevented with probability 1. Furthermore, the encoding is livelock-free—again due to the ever present possibility of backing out: whenever the randomizer is starting to loop by continuously trying to reset the state after one of the lock-checkers has proceeded, we know that there is always a second lock-checker ready and willing to interfere. The liveness of the lock-checker rests on the fact that lock messages are correctly used according to the obligations of mutex channels, which have the important property to become available again and again.

4.2 A 'bakery' algorithm

The π-calculus itself does not directly provide total ordering information as required for modelling the choice protocols as used in the distributed implementations mentioned in the introduction of §4. However, we may program a number server, which can be interrogated to dynamically provide unique global numbers when required, reminiscent of Lamport's *bakery algorithm* (cf. [Lam74]). Natural numbers as well as comparison operators can be easily encoded in the π-calculus [Mil93]. For convenience, we add them explicitly: let $\mathsf{if}\ n{<}m\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ be a comparison operator, where $m, n \in \mathbb{N}$ are integer names, and let now $z \in \mathbf{V} := \mathbf{N} \cup \mathbb{B} \cup \mathbb{N}$ in the grammar of $\mathbb{T}$.

A single globally accessible channel c suffices to implement a bakery algorithm for our purposes. However, this channel must not be accessible by external processes, which might possibly violate the numbering mechanism. Therefore, an encoding according to this programming scheme (see Figure 7) must appear as a two-level definition: an internal

$$
\begin{array}{lcl}
\llbracket P \rrbracket & \overset{\text{def}}{=} & (\boldsymbol{\nu} c)\,(\, c![42] \mid \llbracket P \rrbracket^c\,) \\
\llbracket P_1 \mid P_2 \rrbracket^c & \overset{\text{def}}{=} & \llbracket P_1 \rrbracket^c \mid \llbracket P_2 \rrbracket^c \\
\llbracket (\boldsymbol{\nu} x)\, P \rrbracket^c & \overset{\text{def}}{=} & (\boldsymbol{\nu} x)\, \llbracket P \rrbracket^c \\
\llbracket \sum\limits_{i\in I} \pi_i.P_i \rrbracket^c & \overset{\text{def}}{=} & c?[n].\big(\, c![n{+}1] \;\big|\; (\boldsymbol{\nu} l)\,(\, l![\mathsf{t}] \mid \prod\limits_{i\in I} \llbracket \pi_i.P_i \rrbracket^c_{n,l}\,)\,\big) \\
\llbracket y![\tilde z].P \rrbracket^c_{n,l} & \overset{\text{def}}{=} & (\boldsymbol{\nu} a)\,(\, y![n,l,a,\tilde z] \mid \mathsf{test}\ a\ \mathsf{then}\ \llbracket P \rrbracket^c\ \mathsf{else}\ \mathbf{0}\,) \\
\llbracket y?[\tilde x].P \rrbracket^c_{n,l} & \overset{\text{def}}{=} & (\boldsymbol{\nu} b)\,(\, b![] \mid b?^*[]. \\
 & & \qquad\qquad y?[m,r,a,\tilde x]. \\
 & & \qquad\qquad \mathsf{if}\quad n{=}m\ \mathsf{then}\ (\, y![m,r,a,\tilde x] \mid b![]\,)\ \mathsf{else} \\
 & & \qquad\qquad \mathsf{if}\quad n{<}m \\
 & & \qquad\qquad \mathsf{then}\ \mathsf{test}\ \ l \\
 & & \qquad\qquad\qquad \mathsf{then}\ \mathsf{test}\ \ r \\
 & & \qquad\qquad\qquad\qquad \mathsf{then}\ \ l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket P \rrbracket^c \\
 & & \qquad\qquad\qquad\qquad \mathsf{else}\ \ l![\mathsf{t}] \mid r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![] \\
 & & \qquad\qquad\qquad \mathsf{else}\ \ l![\mathsf{f}] \mid y![m,r,a,\tilde x] \\
 & & \qquad\qquad \mathsf{else}\ \ \mathsf{test}\ \ r \\
 & & \qquad\qquad\qquad \mathsf{then}\ \mathsf{test}\ \ l \\
 & & \qquad\qquad\qquad\qquad \mathsf{then}\ \ l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket P \rrbracket^c \\
 & & \qquad\qquad\qquad\qquad \mathsf{else}\ \ l![\mathsf{f}] \mid r![\mathsf{t}] \mid y![m,r,a,\tilde x] \\
 & & \qquad\qquad\qquad \mathsf{else}\ \ r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![]\ \ ) \\
\llbracket y?^*[\tilde x].P \rrbracket^c & \overset{\text{def}}{=} & y?^*[n,r,a,\tilde x].\mathsf{test}\ \ r\ \ \mathsf{then}\ \ r![\mathsf{f}] \mid a![\mathsf{t}] \mid \llbracket P \rrbracket^c\ \ \mathsf{else}\ \ r![\mathsf{f}] \mid a![\mathsf{f}]
\end{array}
$$

Figure 7: A 'bakery' solution for $\mathbb{S}^{\mathrm{mix}} \to \mathbb{T}$

compositional encoding (fully compositional according to [Pal97]) that is parameterized on the global channel, equipped with a top-level context that protects the global counting mechanism and restricts access to the translations of the original processes. At the top-level, c is initialized with some integer value and passed on as a parameter to the inner compositional encoding $\llbracket\ \rrbracket^c$. Essentially, c is only used, when a thread enters a choice point (our 'bakery'). There, it is *dynamically* equipped with a globally unique number n. Immediately incrementing the counter, this number is transmitted as an additional parameter of the threads' send requests and used later on in the protocol of the receivers. Figure 7 shows that the corresponding variant of the protocol for separate choice, now adapted to mixed choice using two different strands of actions based on the ordering of the locks.

Evaluation The encoding $[\![\ ]\!]$ (with top-level) is not uniform since $[\![P|Q]\!] \not\equiv [\![P]\!]|[\![Q]\!]$ (see also Appendix A), whereas the mere inner encoding $[\![\ ]\!]^c$ is uniform. The encoding is deadlock-free, since we (1) prevent cyclic waiting on locks by using a variant of the bakery algorithm, and (2) deal with 'incestuous' communication by checking equality $n{=}m$ of the request's id's, such that an unintended send-request is resent and the receiver's loop is restarted. Knabe's graph-based proof sketch [Kna93] for deadlock-freedom of his implementation could be adapted to the current setting. See Appendix B for a discussion on an extension of Kobayashi's typing system [Kob97] to cope with the encoding.

Unfortunately, the encoding is not quite divergence-free due to the way we avoid deadlocks in the case of 'incestuous' self-communication in the $n{=}m$ clause: a sender's request may be re-consumed again and again. Yet, the encoding is still livelock-free, since for every enabled matching competitor of an incestuous pair of branches it is always, i.e. again and again, possible to stop the self-communication.

4.3 A 'practical' solution

The main theme in this subsection is the approach of changing the source and target languages of the choice encodings to reflect some phenomena that occur in distributed implementations. It turns out that the source language can be restricted to shrink the number of programs that are difficult to cope with. On the other hand, we propose an extension of the target language that handles total ordering at an abstract level.

Dealing with 'incestuous' self-communication A quick solution for the above unwanted divergence defines the source language such that 'incestuous' self-communication in mixed choices is allowed, similar to the self-communication in the output prefixes of Milner's synchronous π-calculus, as observed by Bellin and Scott [BS94]. So, if we trigger the continuation processes in the $n{=}m$ clause of the receiver's protocol in the case that the local lock (which is then the same as the remote lock) can be successfully tested, as in

$$\mathsf{if}\ n{=}m\ \mathsf{then}\ (\ \mathsf{test}\ l\ \mathsf{then}\ l![\mathsf{f}] \mid a![\mathsf{t}] \mid [\![P]\!]\ \mathsf{else}\ l![\mathsf{f}] \mid a![\mathsf{f}]\ )\ \mathsf{else}\ \ldots$$

then we actually get a (still not uniform, but) deadlock- and divergence-free encoding.

In contrast, Knabe's implementation [Kna93] models a channel as a process that collects send and receive requests on queues. It only then considers two matching requests as candidates to enter the communication protocol, if they belong to different choices. Such implementations are not uniform (see Appendix A), but deadlock- and divergence-free.

Another practically motivated solution is due to the observation that, in distributed systems, it is often the case that receivers are *localized*, i.e. on each channel there is only one receiver waiting. This can be exploited for both for implementation and reasoning; see the work on the join-calculus [FG96, Ama97], where such forms of locality are guaranteed either syntactically or by a simple type system, and also the work on linear receptiveness [San97]. We also profit from a unique-receiver property: 'incest' can then be avoided without divergence by simply throwing away the critical send-request; no other receiver could be waiting for it. This is the approach taken in Figure 8 (see the line for $n{=}m$).

$$
\begin{array}{lcl}
[\![\, P_1 \mid P_2 \,]\!] & \stackrel{\text{def}}{=} & [\![\, P_1 \,]\!] \mid [\![\, P_2 \,]\!] \\
[\![\, (\boldsymbol{\nu} x)\, P \,]\!] & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} x)\, [\![\, P \,]\!] \\
[\![\, \sum_{i\in I} \pi_i.P_i \,]\!] & \stackrel{\text{def}}{=} & (\boldsymbol{\kappa} n)(\boldsymbol{\nu} l)\,(\; l![\mathsf{t}] \mid \prod_{i\in I} [\![\, \pi_i.P_i \,]\!]_{n,l} \;) \\
[\![\, y![\tilde z].P \,]\!]_{n,l} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} a)\,(\; y![n,l,a,\tilde z] \mid \mathsf{test}\ a\ \mathsf{then}\ [\![\, P \,]\!]\ \mathsf{else}\ \mathbf{0} \;) \\
[\![\, y?[\tilde x].P \,]\!]_{n,l} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} b)\,(\; b![] \mid b?^*[]. \\
 & & \qquad\qquad \begin{array}{llll}
\multicolumn{4}{l}{y?[m,r,a,\tilde x].} \\
\mathsf{if} & \multicolumn{3}{l}{n{=}m\ \mathsf{then}\ b![]\ \mathsf{else}} \\
\mathsf{if} & \multicolumn{3}{l}{n{<}m} \\
\mathsf{then} & \mathsf{test} & \multicolumn{2}{l}{l} \\
 & \mathsf{then} & \mathsf{test} & r \\
 & & \mathsf{then} & l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid [\![\, P \,]\!] \\
 & & \mathsf{else} & l![\mathsf{t}] \mid r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![] \\
 & \mathsf{else} & \multicolumn{2}{l}{l![\mathsf{f}] \mid y![m,r,a,\tilde x]} \\
\mathsf{else} & \mathsf{test} & \multicolumn{2}{l}{r} \\
 & \mathsf{then} & \mathsf{test} & l \\
 & & \mathsf{then} & l![\mathsf{f}] \mid r![\mathsf{f}] \mid a![\mathsf{t}] \mid [\![\, P \,]\!] \\
 & & \mathsf{else} & l![\mathsf{f}] \mid r![\mathsf{t}] \mid y![m,r,a,\tilde x] \\
 & \mathsf{else} & \multicolumn{2}{l}{r![\mathsf{f}] \mid a![\mathsf{f}] \mid b![] \;\;)}
\end{array} \\
[\![\, y?^*[\tilde x].P \,]\!] & \stackrel{\text{def}}{=} & y?^*[n,r,a,\tilde x].\mathsf{test}\ r\ \ \mathsf{then}\ \ r![\mathsf{f}] \mid a![\mathsf{t}] \mid [\![\, P \,]\!]^c\ \ \mathsf{else}\ \ r![\mathsf{f}] \mid a![\mathsf{f}]
\end{array}
$$

Figure 8: $\pi_s^{\mathrm{mix},1} \to \pi_a^{\kappa}$

Bakery primitive Here, we recall the idea of deriving a total order among threads (light-weight processes) from some distribution order among the nodes and processors and processes' creation ids for finer identification [Ber80, BS83, Kna93]. For the π-calculus, we choose an abstract view, that captures the idea syntactically. Let us assume an extended target language π_a^{κ} with a binding primitive $(\boldsymbol{\kappa} n)P$ for creating totally ordered identifiers n in process P, then our bakery algorithm can be programmed in a 'uniform' way without the need of a top-level: in the encoding of choice in Figure 8 (see line 3) the actual identity of n is not important—it only matters that every pair n, m of different identifiers is ordered.

With the 'bakery primitive' in the target (π_a^{κ}) and a unique-receiver property in the source ($\pi_s^{\mathrm{mix},1}$), we get the encoding in Figure 8. It is similar to Knabe's distributed implementation [Kna93], but replaces channel managers by assuming unique receivers.

Evaluation The encoding in Figure 8 gets rid of the problem of possible self-communication by exploiting uniqueness, and is deadlock-free—and also divergence-free—by exploiting the inherent total-order. Note that such encodings do not preserve the symmetry of networks, as necessary for Palamidessi's impossibility argumentation, because the target language is intrinsically asymmetric due to the totally ordering 'bakery' primitive. Since such languages allow for fully compositional encodings of mixed-guarded choice, while the standard (symmetric) asynchronous π-calculus only allows for 'semi-compositional' encodings, this can be interpreted as a separation between symmetric and asymmetric calculi.

5 Conclusion

The encodings presented in this paper should exhibit how to abstractly model distributed implementations of guarded choice within the asynchronous π-calculus. Prompted by Palamidessi's work [Pal97], we emphasized the problematic case of mixed choice by developing first the quite simpler encoding for separate choice. Whereas this case satisfies all of Palamidessi's required properties, the transition to encodings for mixed choice bears all of the awkwardnesses. Two sources of potential deadlock are identified: cyclic waiting on lock channels and 'incestuous' self-communication. In order to cope with them, we pointed out that either uniformity or divergence-freedom must be dropped, if we want to stay within the chosen framework, thus confirming Palamidessi's negative result. Furthermore, we motivated that slight changes to the framework would allow us to overcome the impossibility.

Since our encodings of mixed choice and the proposed variants in §4 can be seen as abstractions of practically 'good' distributed implementations [Kna93], one interpretation of our work might be an evaluation of whether Palamidessi's criteria are too strong for practical purposes. It was pointed out quite early [RL94] that probabilistic solutions—with divergence, but without livelock and with progress probability 1—might be practical, although they are not reasonable in [Pal97]. On the other hand, the standard way of implementing channel managers as autonomous threads [Kna93, LT95] contradicts the requirement of uniformity, if open systems are considered. As our work shows, relaxing uniformity by admitting a top-level context or relaxing reasonableness to admit some fair degree of well-behaved divergence turns many practically motivated encodings theoretically 'good'.

In the spirit of Bougé's informal notion of symmetry "*there is no priority or any other form of externally specified static partial ordering among processes*" [Bou88], we may note that none of the branches in choices is statically assigned priority over its competitors by the encoding in Figure 7—the symmetry is broken only dynamically by taking a totally-ordered ticket when entering the 'bakery'. In accordance with Rabin and Lehmann [RL94], we needed (only) one small piece of *global memory*: the protected message on channel c.

Acknowledgements

Cédric Fournet, Kohei Honda, Naoki Kobayashi, Catuscia Palamidessi, Benjamin Pierce, Davide Sangiorgi, Martin Steffen, and Nobuko Yoshida for many fruitful discussions.

A Channel managers are not uniform

Distributed implementations of channel-based communication usually employ so-called *channel manager* processes $\mathsf{CHAN}(y)$ for mediating between the activities of senders and receivers on channel y. Often, this is done by collecting send- and receive-requests in queues that are attached to channel managers; the synchronization protocol for a particular channel is then only started when a pair of complementary requests (from different choices) can be found in the respective queue [Kna93, LT95].[6]

In π-calculus encodings, the creation of channel managers would have to take place at the moment the corresponding channel name is created:

$$[\![\, (\boldsymbol{\nu} y)\, P \,]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} y) \left([\![\, P \,]\!] \mid \mathsf{CHAN}(y) \right)$$

However, free names in process terms would, in their encoding, have to be supplied explicitly with their managers at the top-level of encodings:

$$[\![\, P \,]\!] \stackrel{\text{def}}{=} [\![\, P \,]\!] \;\Big|\; \prod_{x \in \mathit{fn}(P)} \mathsf{CHAN}(x)$$

This, in turn, conflicts with Palamidessi's requirement of uniformity, since the equation $[\![\, P \,]\!] \mid [\![\, Q \,]\!] = [\![\, P \mid Q \,]\!]$ does not hold in general, because a free name shared by P and Q would be provided with *two* competing managers on the left side, but only one (as intended) on the right side. For this reason, e.g. encodings with 'centralized' channel managers are not uniform. Consequently, if we want to stick to 'uniformity', we either have to restrict ourselves to closed process terms with no observable behavior at all (i.e. no free names), or we have to leave the encoding of restriction as

$$[\![\, (\boldsymbol{\nu} y)\, P \,]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} y)\, [\![\, P \,]\!]$$

and distribute the functionality of channel managers, if possible, over all places where channels are used, as is exemplified in the encodings of §3 and §4.

B Type-checking partial deadlock-freedom

This section provides a quick overview of Kobayashi's type system for deadlock-freedom. Since it would be too space-consuming to present all the necessary formal definitions and theorems from [Kob97], we assume the more interested reader to have a copy of that paper at hand. We then provide the ingredients for carrying out the formal proof for an application of this type system to the encoding of separate choice as in Figure 5.

[6] In a distributed system, channel managers need to reside at some particular location, the choice of which may heavily influence the efficiency of computations: it decides whether individual communications are either local or remote. Consequently, a big advantage is the unique-receiver property (see §4.3 and [FG96]), because it allows the implementer to statically choose just the location of the unique receiver as the residence of the channel manager process.

B.1 Types for reliable channels

We sketch Kobayashi's [Kob97] non-recursive channel types $p^m[T_1, \ldots, T_n]^t$ with

- *polarity* $p \subseteq \{I, O\}$ known from Pierce and Sangiorgi [PS96] for denoting input and output capabilities (with abbreviations $| := \{\}$, $\uparrow := \{O\}$, $\downarrow := \{I\}$, and $\updownarrow := \{I, O\}$),
- *multiplicity* $m \in \{1, *, M, \omega\}$ for classifying channels,
- *arity* $n \in \mathbb{N}_0$ known from Milner's polyadic π-calculus [Mil93], and
- *time tag* $t \in \mathbf{T}_\star$, which we explain below.

The multiplicity of a channel constrains its usage according to capabilities and obligations: *linear* (1) channels must eventually be used once for input and output, after they become active; *replicated-input* ($*$) channels must be used in exactly one replicated input-prefix immediately after creation, but they can be used arbitrarily often for sending; *mutex* (M) channels have to be supplied with some value immediately after creation, and a reader of some mutex channel is obliged to eventually resend some value, thus, at any time, there is at most one message in the system; *unreliable* (ω) channels can be used arbitrarily with the exception of replicated input-prefixes. A channel is also called *reliable* if it is not unreliable; basically, every process that communicates on a reliable channel will eventually find its communication partner.

Kobayashi introduces a type system for a calculus very similar to our target language $\mathbb{T}$ with boolean primitives ($\mathbb{B}$ as base type, and if instead of test) and a *typed restriction* operator, which also introduces fresh time tags. (For some proof strategies, it also convenient to introduce additional time tags with input variables, which then have to match the tags of the communicated values.) Time tags t are used within the typing rules to express constraints on the order of using reliable channels (unreliable channels as well as replicated input channels always carry the tag $\star$ which is both smaller and greater than any other tag): basically, $s \prec t$ means that *obligations according to the type of a channel associated with* t *may (not must) be delayed until the completion of some communication on the channel associated with* s. Example obligations are, for example, that a linear channel needs to be find the complementary partner, once it has become active; with mutex channels, only the sender half represents an obligation: if the channel associated with tag t is a mutex channel, then the (re-) sending on channel may be blocked by some communication on the channel associated with tag s. Clearly, deadlocks may arise when the ordering among the channels indicates cyclic waiting, where an obligation might, by transitivity, be blocked by its own fulfillment. Thus, an ordering $\prec$ is indicating deadlock-freedom, if its transitive closure $\prec^+$ is a strict partial order on $\mathbf{T} \times \mathbf{T}$ (i.e. ignoring the $\star$).

Kobayashi then specifies *partial deadlock-freedom* of terms by means of his classification of channels: only the reliable channels are expected to be satisfy deadlock-freedom properties. For explanation, let us assume a process $P \not\rightarrow$ with no more reductions. Let us further regard P in normal form, i.e. $P \equiv (\boldsymbol{\nu}\tilde{w}) \prod N_j$ where N are either a message $y![\tilde{z}]$, a single input $y?[\tilde{x}].Q$, or a replicated input $y?^*[\tilde{x}].Q$. Whether N is interpreted as a deadlocked sub-process in P, depends on the type of its channel y:

- According to the intuition of replicated and mutex channels, *mutex messages* and *replicated receivers* are not counted as regards deadlock, because they are meant to be always (for mutex: always eventually) present after channel creation. On the other hand, *mutex receivers* and *replication triggers* indicate unwanted deadlock, since they are meant to always find their counterpart, so N should be neither of them.

- With respect to linear channels, N must be seen in context. If its channel y is restricted in P, then we regard N as causing deadlock, since it was never used after it has become active; if this y is free in P, then its complement might still be supplied from the outside, and then y is called *half-used*.

- If N's channel y is unreliable, then it is not interpreted as ill-behaved since we have not required it to behave reliably in any sense.

Let Γ be a list of typing assumptions for names, and $\prec$ a tag ordering, where $\prec^+$ is a strict partial order. A typing judgement $\Gamma, \prec \vdash P$ intuitively means

1. P uses only the capabilities specified in Γ,
2. P fulfills all the obligations specified by Γ, and
3. P obeys the ordering specified in $\prec$ in fulfilling the obligations.

There are two strategies for using Kobayashi's type system:

(A) One uses Kobayashi's two-phase type-checking algorithm: the first-phase is building up a type derivation tree witnessing the correct use of arities, polarities, and multiplicities, while collecting up constraints on time tags according to the applied rules; the second phase tries to solve the tag constraints and checks whether the transitive closure of the resulting tag ordering, if it exists, is a strict partial order.

(B) One supplies an appropriate candidate for a time tag ordering from the beginning and directly type-checks the terms under investigation in one single phase including the conditions on time tag required by the applied rules.

For well-typed processes, we have the following useful properties concerning partial deadlock-freedom [Kob97], which we present here in a weaker, but simplified version:

Theorem B.1.1 (Subject reduction).
If $\Gamma, \prec \vdash P$ *and* $P \to P'$*, then there are* $\Gamma', \prec'$ *with* $\Gamma', \prec' \vdash P'$*.*

Theorem B.1.2 (Immediate deadlock). *Suppose* $\Gamma, \prec \vdash P$ *and* $P \not\to$*, then pending communications in* P *are (1) on some unreliable channel, (2) on some linear channel that is half-used according to* Γ*, or (3) are either a mutex message or a replicated input.*

Together, the two theorems guarantee that during reduction of well-typed processes no partial deadlock, as specified by the classification of reliable channels, can ever occur.

B.2 Separate choice

In this subsection, we prove deadlock-freedom for the encoding of separate choice in Figure 5 via Kobayashi's type system in the following sense: since we use only reliable channels for encoding choice, and if we regard all high-level channels as *un*reliable, then the typability of translated terms implies that the encoding does not add deadlocks. This is true, because all of these reliable channels—especially the linear acknowledgement channels—are restricted, so only the cases (1) and (3) of Theorem B.1.2 apply. Since mutex messages and replicated inputs of case (3) can be regarded as garbage, the only remaining deadlocked subprocesses are on high-level channels. Since these have been translated in a $1-1$ fashion, we know that the deadlock must have been present already in the source term.

According to the above idea, we first assume a simple polyadic type system for the source language as a non-recursive structural variant of Milner [Mil93], extended with boolean-typed names. This system can be seen as Kobayashi's system by stripping off polarity, multiplicity, and time tags, or else by having only one polarity $\updownarrow$, one multiplicity ω, and one time tag $\star$, such that $[T] := \updownarrow^{\omega}[T]^{\star}$, where T is either the type boolean $\mathbb{B}$ or a finite tuple $\tilde{T}$. Let us assume that we have typing statements of the form $\{\tilde{y} : \tilde{T}\} =: \mathcal{T} \vdash P$ in $\mathbb{S}$, and that all source terms under investigation are well-typed, accordingly.

Next, we propose a tag ordering that we are going to apply for type-checking translated terms (i.e. we are following proof strategy B). It is intuitively derived from the encoding's algorithmic idea. Let $\mathbf{T} \supseteq \mathbf{T}_l \uplus \mathbf{T}_r \uplus \mathbf{T}_a$ distinguish three pairwise disjoint subsets of tags, where the indices of the tag sets indicate the channels they are going to be associated with.

$$\prec \quad \overset{\text{def}}{=} \quad (\mathbf{T}_l \times \mathbf{T}_a) \cup (\mathbf{T}_r \times \mathbf{T}_a) \cup (\mathbf{T}_r \times \mathbf{T}_l)$$

defines the tag ordering that corresponds to the use of mutex and linear channels in the encoding: a communication on the linear acknowledgement channels a may be delayed by both a communication a receiver-lock l $(\mathbf{T}_l \times \mathbf{T}_a)$ or on a sender-lock r $(\mathbf{T}_r \times \mathbf{T}_a)$, whereas the resending of the mutex message for the receiver-lock l may be blocked by a reception for the sender-lock r $(\mathbf{T}_r \times \mathbf{T}_l)$, but not the other way around. The latter blocking actually happens only if the receiver-lock carried $\mathtt{t}$, so only then it will not be resent until the sender-lock has also been checked. The following important fact holds by definition:

Lemma B.2.1. *$\prec^{+}$ is a strict partial order.*

With this ordering, we can now provide a *typed encoding* $[\![\ ]\!]^{\mathbf{T}}$ for separate choice, assuming that we only consider well-typed source terms. In Figure 9, each of the channels that are added by the encoding is given a reliable type and is associated the respective time tag, where $t_l \in \mathbf{T}_l$, $t_r \in \mathbf{T}_r$, and $t_a \in \mathbf{T}_a$. In contrast, every high-level channel is regarded as unreliable (note y's multiplicity ω), and its type is appropriately translated to carry additional information according to its use in encoded terms: a low-level send-request on a high-level channel y carries the input-end for some mutex channel $(\downarrow^{M}[\mathbb{B}^{\star}]^{t_r})$ and the output-end for some linear acknowledgement channel $(\uparrow^{1}[\mathbb{B}^{\star}]^{t_a})$ as additional parameters. With $[\![\,\mathbb{B}\,]\!]^{\mathbf{T}} := \mathbb{B}^{\star}$, let us also extend the typed encoding on types componentwise to type environments $\mathcal{T}$ of the source language by $[\![\,\emptyset\,]\!]^{\mathbf{T}} := \emptyset$ and $[\![\,\mathcal{T}, y : T\,]\!]^{\mathbf{T}} := [\![\,\mathcal{T}\,]\!]^{\mathbf{T}}, y : [\![\,T\,]\!]^{\mathbf{T}}$.

$$
\begin{array}{llll}
l \;:\; \updownarrow^{M}[\mathbb{B}^{\star}]^{t_l} & r \;:\; \updownarrow^{M}[\mathbb{B}^{\star}]^{t_r} & a \;:\; \updownarrow^{1}[\mathbb{B}^{\star}]^{t_a} & b \;:\; \updownarrow^{*}[\,]^{\star} \\
\multicolumn{4}{l}{y \;:\; \llbracket\, [\tilde{T}] \,\rrbracket^{\mathbf{T}} \;\overset{\text{def}}{=}\; \updownarrow^{\omega}[\downarrow^{M}[\mathbb{B}^{\star}]^{t_r}, \uparrow^{1}[\mathbb{B}^{\star}]^{t_a}, \llbracket\, \tilde{T} \,\rrbracket^{\mathbf{T}}]^{\star} \quad \mathit{in}\ \mathbb{T}, \qquad \mathit{if}\ y : [\tilde{T}]\ \mathit{in}\ \mathbb{S}}
\end{array}
$$

Figure 9: Types for $\mathbb{S}^{\text{sep}} \to \mathbb{T}$

On terms, the typed encoding $\llbracket\ \rrbracket^{\mathbf{T}}$ is then defined by simply adding the types proposed in Figure 9 for a, b, l and r, to the restriction occurrences in Figure 5, where each occurrence gets a fresh time tag. We further translate the typed restriction $(\boldsymbol{\nu} x : T)\, P$ for high-level channels in the source language into the enhanced typed restriction $(\boldsymbol{\nu} x : \llbracket\, T \,\rrbracket^{\mathbf{T}})\, \llbracket\, P \,\rrbracket^{\mathbf{T}}$ in the target language. For type-checking, we expand out test-expressions by using if-forms instead, and we also add time-tags in the case of the encoding of (replicated) inputs, so let us choose occurrences of tag variables $s_r \in \mathbf{T}_r$ and $s_a \in \mathbf{T}_a$ and attach them to the respective occurrences in the encoding of Figure 5. The third parameter of translations of high-level inputs $\tilde{x}$ is, according to Figure 9, implicitly tagged with $\star$.

Lemma B.2.2. $\big(\{{}^{[t_r,t_a]}\!/\!_{[s_r,s_a]}\}\big) \prec\ \subseteq\ \prec$

The order of the time tags t of the received channels r and a is always sufficient for the required order on the time tags s of the corresponding input variables. This lemma is needed, whenever we derive low-level communications on high-level channels y.

Lemma B.2.3. $\{t_r, t_a\} \not\prec\!\not\succ \{s_r, s_a\}$

Note that $t_r \prec s_l \prec s_a$ (and $s_r \prec s_l \prec t_a$), but that $t_r \not\prec s_a$ (and $s_r \not\prec t_a$), since $\prec$ is not transitive. This lemma is needed in the proof of IN-rules.

Since we interpret all high-level channels as unreliable, we face the problem, for the type-checking of typed translations, that rule T-IN of Kobayashi [Kob97] is not applicable, when trying to use it for a high-level replicated input. So, we need to assume an additional typing rule T-URIN (*un*reliable *r*eplicated *in*put), which is defined just like T-IN of Kobayashi [Kob97], but allows the conclusion for replicated input syntactically. This rule does not change the deadlock-properties of the well-typed processes, if we only allow its application for channels that we explicitly regard as *un*reliable. Finally, we get the main property, where $\prec\!\downarrow_{\mathbf{I}(\llbracket S \rrbracket^{\mathbf{T}})}$ denotes $\prec$ with all ordered pairs mentioning inner tags $\mathbf{I}(\llbracket\, S \,\rrbracket^{\mathbf{T}})$ of the translated term, i.e. those introduced as s_r and s_a above, removed.

Proposition B.2.4 (Preservation of typing).
Let $S \in \mathbb{S}^{\text{sep}}$*. If* $\mathcal{T} \vdash S$*, then* $\big(\llbracket\, \mathcal{T} \,\rrbracket^{\mathbf{T}} \cup \{\mathsf{t}, \mathsf{f} : \mathbb{B}^{\star}\}\big), \prec\!\downarrow_{\mathbf{I}(\llbracket S \rrbracket^{\mathbf{T}})} \vdash \llbracket\, S \,\rrbracket^{\mathbf{T}}$.

Proof. By induction on the structure of S. □

The intuition of well-typed processes is that deadlocks can only be caused by unreliable channels, so we also know that deadlocks can only be caused by high-level channels: every deadlock in some derivative originates from some deadlock present in the source language.

Corollary B.2.5. $\mathbb{S}^{\text{sep}} \to \mathbb{T}$ *is deadlock-free.*

B.3 Mixed choice

In analogy to the discussion on cyclic waiting (§4), the type-checker (strategy A) for deadlock-freedom fails, when reusing the encoding for separate choice in the case of mixed choice. In type-checking the example $[\![P|Q]\!]$, there is no ordering for the use of the two choice-locks that can be used consistently on both sides of the parallel composition (cf. proof sketch of Proposition B.2.4). Although Kobayashi's system is not complete, i.e. it rejects processes that are deadlock-free, in our case the rejection is correct as indicated in §3.

Similar to the approach in the previous section, we can provide a tag ordering for the randomized solution of Subsection 4.1: the pairs are given by $t_s{\prec}t_b{\prec}t_l, t_r, t_a$, where we again indicate the ordering among (sets of) tags by their associated channel names. The state channels s associated with tags t_s are never delayed since resending on s is always immediate, according to the encoding in Figure 6. With Kobayashi, we may say that well-typed terms never get into a deadlock, however they might fall into an infinite loop.

If we wanted to use Kobayashis's system to type-check the 'bakery' encoding for mixed choice in Figure 7, we would need to extend the system to deal with natural numbers. Then, we would need rules for checking consistency of the constraints on the time tag ordering on critical channels with the occurrences of the conditional operator: in contrast to Kobayashi's rules, we would need to allow the two strands of a conditional to be typed according two different time tag orderings. The study of the feasibility of such a system is left for future work.

C Full abstraction?

In the following subsections §C.1 and §C.2, we summarize a few useful observations that highlight the inherent problems of stating full abstraction properties for some choice encodings in this paper. In Subsection §C.3, we then state a rather restricted full abstraction result, which could be interpreted as: *'programming with choice' is implemented correctly.*

C.1 Separate choice

For the same reason that the encodings $\pi_s \to \pi_a$ (see Figure 1) are not fully abstract with respect to weak bisimulation [Hon92], also the encoding for the π-calculus with separate choice is not: the atomicity of guarding outputs is visibly broken into a send-request and an acknowledgement reception. Consequently, whereas the processes $y![].y![].\mathbf{0}$ and $y![].\mathbf{0} \mid y![].\mathbf{0}$ are weakly bisimilar, their translations are not. (In contrast, the breaking up of atomicity into two low-level steps in the encoding of input-guarded choice was not visible since the second step was always internally bound, thus not visible to the outside.)

Nevertheless, it is instructive to formalize some notion of bisimulation-like equivalence that is suitable for dealing with, i.e. equating, derivatives of intuitively equivalent encoded terms. The main problem is that after simulating the choice for branch k, as depicted in Figure 10, with a sequence of low-level steps (indicated as $[\![\pi_k]\!]$) there is some active non-chosen 'garbage' G_k (indicated above as the underlined remainder of the encoded choice

$$\begin{tikzcd}
\sum \pi_i.P_i \arrow[r, "\pi_k"] \arrow[d, mapsto] & P_k \arrow[d, mapsto] \\
\left\llbracket \sum \pi_i.P_i \right\rrbracket \arrow[r, Rightarrow, "\llbracket \pi_k \rrbracket"] & \llbracket P_k \rrbracket \mid \underline{\left(\!\!\left(\sum \pi_i.P_i \right)\!\!\right)_k} \overset{?}{\asymp} \llbracket P_k \rrbracket
\end{tikzcd}$$

Figure 10: Simulation of a choice term, committing to branch k

after choosing k) in the system that is running in parallel with the intended encoding $\llbracket P_k \rrbracket$ of the continuation of the chosen branch. For the encoding of input-guarded choice [NP96], we have found that an asynchronous observation principle [HT92, ACS96] yields the appropriate notion since it allows us to garbage-collect processes that do nothing else than resending (after a while) every message that they consume, so $G_k \approx \mathbf{0}$ holds in this case. With output-guards, however, non-chosen branches may still perform asynchronously visible outputs, which—according to their lock-information—are not valid, so G_k exhibits too much observable behavior in that case.

In order to 'bend' weak bisimulation for our application, we motivate a variant of barbed congruence [MS92] that captures a notion of observation that is more adequate for encodings of separate and mixed choice. As usual, it consists of reduction bisimulation, the formulation of an observation predicate and a closure under contexts. Note that every visible activity of a term P that is reachable via reduction from some term $\llbracket S \rrbracket$ is on some high-level channel, since the use of low-level channels l, r, a, b is restricted. Next, as in [NP96], we only rely on the observation of outputs. However, not every output on a high-level channel for P is 'valid' since its choice might already be resolved. Every receiver of such an output will eventually know the state f of the choice and, thus, discover this invalidity. For the above example G_k, no output at all is valid. By analysis of the encoding, we get that every possible output (of some derivative of some $\llbracket S \rrbracket$) is of the form

$$P \equiv (\boldsymbol{\nu} l, a, \tilde{x})\,(\; y![l, a, \tilde{z}] \mid \dot{P} \;) \xrightarrow{(\boldsymbol{\nu} l, a, \tilde{x} \cap \tilde{z})\; y![l, a, \tilde{z}]} (\boldsymbol{\nu} \tilde{x} - (\tilde{x} \cap \tilde{z}))\, \dot{P}$$

and, furthermore, for every such output we can always find a message on the mutex channel l that is mentioned in the output such that we can syntactically check by $\dot{P} \equiv l![\mathsf{b}] \mid \hat{P}$, whether the output was valid ($\mathsf{b} = \mathsf{t}$) or not ($\mathsf{b} = \mathsf{f}$). We define Σ-*barbs*

$$P{\downarrow}^{\Sigma}_{y} \quad \text{if} \quad P \equiv (\boldsymbol{\nu} l, a, \tilde{x})\,(\; y![l, a, \tilde{z}] \mid l![\mathsf{t}] \mid \hat{P} \;)$$

for some $\hat{P}$, and $P{\Downarrow}^{\Sigma}_{y}$ if $P \;\Rightarrow P'{\downarrow}^{\Sigma}_{y}$ for some P'. Since the internal names l, a become free after an observed output on some high-level channel, we need to require that a context behaves according to the protocol of the encoded terms. As in [VP96], we cannot expect correct behavior within alien contexts, so we enforce the requirements to Σ-*contexts* only, i.e. contexts occurring in $\llbracket\ \rrbracket$-encodings of $\mathbb{S}^{\mathrm{sep}}$-terms.

Definition C.1.1. *A relation $\mathcal{R}$ is a* Σ-barbed simulation*, if $(P, Q) \in \mathcal{R}$ implies*

- *If $P \rightarrow P'$, then there is $Q \Rightarrow Q'$ with $(P', Q') \in \mathcal{R}$.*
- *If $P{\downarrow}^{\Sigma}_{y}$, then $Q{\Downarrow}^{\Sigma}_{y}$.*

and vice versa. Two processes are Σ-barbed bisimilar*, written $P \stackrel{\bullet}{\bowtie} Q$, if there is some symmetric Σ-barbed simulation with $(P, Q) \in \mathcal{R}$; they are* Σ-barbed congruent*, written $P \bowtie Q$, if $C[P] \stackrel{\bullet}{\bowtie} C[Q]$ for all Σ-contexts C.*

It would be interesting to investigate, whether some form of typed observation could replace the somewhat delicate notion of Σ-barb and Σ-context. Basically, it should be captured by the notion of well-typed context composition whether some context respects an expected protocol and whether some output of some process is considered valid. However, the expected protocols for our choice encodings not only require that the pure typing aspects of Kobayashi's reliable channels are respected, but also that the boolean values on the lock channels are correctly handled by some reader of a lock in the context. More precisely, a reader of a lock not only must eventually send back *some* boolean value—in addition, it may never change the lock's value from f to t, but only from t to f, or leave it unchanged. This means that value-dependencies, although of a rather simple nature, would have to be included in the 'type' system. Work in that direction is not yet known to the author, but some extension of [Kob97, Yos96] seems worth pursuing.

As a first promising step towards full abstraction, we have:

Lemma C.1.2 (Garbage). $\left(\!\!\left(\sum \pi_i.P_i \right)\!\!\right)_k \;\bowtie\; \mathbf{0}$

Yet, we do *not* get full abstraction for the standard weak barbed congruence, denoted ≈, in $\mathbb{S}$ and Σ-barbed congruence ⋈ in $\mathbb{T}$, as

$$\text{for all } S_1, S_2 \in \mathbb{S} \;\; : \; S_1 \approx S_2 \text{ iff } [\![\, S_1 \,]\!] \bowtie [\![\, S_2 \,]\!],$$

does not hold because atomic commitments in source terms are implemented by gradual commitments in their translations. At the source-level, when a send-request is consumed by some receiver, this action also may resolve two choices atomically. At the target-level, the consumption of a high-level send-request merely means that some receiver has started its choice protocol; it is not yet decided, in general, that this send-request will win—other send-requests might join the competition on its way before the choice-lock is actually read. It is only clear that *some* request will resolve this choice and, unless that happens, the losing send-requests in this game will not be available to other receivers in other choices. So, the best full abstraction result we can expect would be for some Σ-barbed *coupled* congruence, i.e. a coupled pair of Σ-closed Σ-barbed simulations (see [NP96] for a very similar application of coupled simulation [PS92]).

C.2 Mixed choice

The definition of barbs of the previous subsection was sufficient, because in the encoding for separate choice, sender-locks are always checked in the second place: in contrast to receiver-locks, which might not be available for a couple of reductions, testing a sender's lock (always after having successfully tested the receiver's lock) immediately causes its re-set (Figure 5). If we changed the order of checking locks in $[\![\ ]\!] : \pi_s^{\text{sep}} \to \mathbb{T}$ to $\mathsf{test}\ r\ \mathsf{then}\ \mathsf{test}\ l\ \mathsf{then}\ \ldots$, there would be situations, where the required mutex message is not available.

For example, in $[\![\, u?[x].K + y?[x].P \mid y![z].Q + w![v].R \,]\!]$ let the sender and receiver on y have exchanged the send-request and the receiver checked the sender's lock r. In that situation, it cannot be observed directly, i.e. from the syntax of the (encoded) term, whether the possible output on w is valid, or not. It is valid, because its choice has not yet been resolved in favor of y (since the receivers' lock l has not yet been checked), but neither is the necessary lock r available (since it is currently held by the receiver on y), nor can a state be reached by reduction, where the lock r is available again, without committing to the communication on y and turning the senders' lock r to f; the only way to detect the validity of the output on w would be by supplying a message on u from the outside and observing that the communication on u could preempt the pending communication on y, thus resulting in resending the required lock r with state information t. An appropriate notion of barb may therefore be given by observing processes within 'saturating' contexts, but this remains to be investigated.

The same arguments as for the encoding of separate choice with reversed order of testing also hold for the encoding of mixed choice as of Figure 7, because sender-locks are not always the second lock to be tested, as would happen with the above example, when the order determines the sender-lock as smaller than the receiver-lock.

C.3 Restricted full abstraction

Let us explicitly introduce single prefixes into the source language and let us also distinguish the channels within the source language according to their syntactic usage as *single* (appear as subject in single prefixes) or *selectable* (appear as subjects in choice-branches). Let us restrict the source language such that communication on selectable channels is always restricted and let us forbid the passing of selectable channels as objects such that their scope is never extruded; then the only use of selectable channels is as subject of branches in choice expressions and as bound variable in restriction (never in object position of either input or output). Let us furthermore choose a target language with synchronous output, such that output on single channels may then be encoded trivially and, thus, it is no longer possible to observe a broken atomicity in that case.

With all these purely syntactic conditions and the additional clause for single output prefixes, we can simply rephrase the very same definitions for all uniform choice encodings in this paper, except that we need to give two different definitions for replicated inputs: one for single, and one for selectable channels. From the outside, source terms that internally might use selectable channels in choice expressions can then not be distinguished at all from their translations, because a context can never interfere on these selectable channels due to the above conditions. Formally, within this very restricted setting, we can get full abstraction results as a simple corollary of the indistinguishability

$$\text{for all } S \in \mathbb{S} : S \approx_s [\![S]\!]$$

where $\approx_s$ is even the standard synchronous weak bisimulation, i.e. with observable inputs. Intutively, we may say that 'programming with choice' can be implemented correctly.

References

[ACS96] R. M. Amadio, I. Castellani and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 147–162. Springer, 1996.

[Ama97] R. M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proceedings of COORDINATION '97*, 1997. Extended version as Rapport Interne LIM Marseille and Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[Ben83] M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *Proceedings of PODC '83*, pages 27–30. ACM, August 1983.

[Ber80] A. Bernstein. Output Guards and Nondeterminism in "Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, Apr. 1980.

[BGZ97] N. Busi, R. Gorrieri and G. Zavattaro. On The Turing-Equivalence of Linda Coordination Primitives. In C. Palamidessi and J. Parrow, editors, *Proceedings of EXPRESS '97*, volume 7 of *ENTCS*. Elesevier Science Publishers, 1997.

[Bou88] L. Bougé. On the Existence of Symmetric Algorithms to Find Leaders in Networks of Communicating Sequential Processes. *Acta Informatica*, 25(2):179–201, 1988.

[Bou92] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[BS83] G. Buckley and A. Silberschatz. An Effective Implementation for the Generalized Input-Output Construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, Apr. 1983.

[BS94] G. Bellin and P. Scott. On the π-Calculus and Linear Logic. *Theoretical Computer Science*, 135:11–65, 1994. Also published as LFCS report ECS-LFCS-92-232, LFCS, University of Edinburgh.

[FG96] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, Jan. 1996.

[Hon92] K. Honda. Notes on Soundness of a Mapping from π-calculus to ν-calculus. With comments added in October 1993, May 1992.

[HT92] K. Honda and M. Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz and P. Wegner, editors, *Object-Based Concurrent Computing 1991*, volume 612 of *LNCS*, pages 21–51. Springer, 1992.

[Kna93] F. Knabe. A Distributed Protocol for Channel-Based Communication with Choice. *Computers and Artificial Intelligence*, 12(5):475–490, 1993.

[Kob97] N. Kobayashi. A Partially Deadlock-Free Typed Process Calculus. In *Proceedings of LICS '97*, pages 128–139. Computer Society Press, July 1997. Full revised version from `http://web.yl.is.s.u-tokyo.ac.jp/members/koba/publications.html`, University of Tokyo.

[KS97] D. Kumar and A. Silberschatz. A Counter-Example to an Algorithm for the Generalized Input-Output Construct of CSP. *Information Processing Letters*, 61:287, 1997.

[Lam74] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Journal of the ACM*, 17(8):453–455, 1974.

[LT95] L. Leth and B. Thomsen. Some Facile Chemistry. *Formal Aspects of Computing*, 7(3):314–328, 1995. A Previous Version appeared as ECRC-Report ECRC-92-14.

[Mil93] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer and H. Schwichtenberg, editors, *Proceedings of Logic and Algebra of Specification, International NATO Summer School (Marktoberdorf, Germany, 1991)*. Springer, 1993.

[MPW92] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[Nes96] U. Nestmann. *On Determinacy and Nondeterminacy in Concurrent Programming.* PhD thesis, Universität Erlangen, November 1996. Arbeitsbericht IMMD-29(14).

[NP96] U. Nestmann and B. C. Pierce. Decoding Choice Encodings. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 179–194. Springer, 1996. Revised full version as report ERCIM-10/97-R051, European Research Consortium for Informatics and Mathematics, 1997.

[Pal97] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM, Jan. 1997.

[PS92] J. Parrow and P. Sjödin. Multiway Synchronization Verified with Coupled Simulation. In R. Cleaveland, editor, *Proceedings of CONCUR '92*, volume 630 of *LNCS*, pages 518–533. Springer, 1992.

[PS96] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[Rep91] J. Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of PLDI '91*, pages 293–259. ACM, June 1991. In *SIGPLAN Notices* 26(6).

[RL94] M. O. Rabin and D. Lehmann. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 20, pages 333–352. Prentice Hall, 1994. An extended abstract appeared in *Proceedings of POPL'81*, pages 133–138.

[San93] D. Sangiorgi. *Expressing Mobility in Process Algebras — First-Order and Higher-Order Paradigms.* PhD thesis, LFCS, University of Edinburgh, 1993. Report ECS-LFCS-93-266.

[San97] D. Sangiorgi. The Name Discipline of Uniform Receptiveness (Extended Abstract). In P. Degano, R. Gorrieri and A. Marchetti-Spaccamela, editors, *Proceedings of ICALP '97*, volume 1256 of *LNCS*, pages 303–313. Springer, 1997.

[TLK96] B. Thomsen, L. Leth and T.-M. Kuo. A Facile Tutorial. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 278–298. Springer, 1996.

[VP96] B. Victor and J. Parrow. Constraints as Processes. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 389–405. Springer, 1996.

[Yos96] N. Yoshida. Graph Types for Monadic Mobile Processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *LNCS*, pages 371–386. Springer, 1996.

Recent BRICS Report Series Publications

RS-97-45 Uwe Nestmann. *What Is a 'Good' Encoding of Guarded Choice?* December 1997. 28 pp. Revised and slightly extended version of a paper published in *5th International Workshop on Expressiveness in Concurrency,* EXPRESS '97 Proceedings, volume 7 of Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers.

RS-97-44 Gudmund Skovbjerg Frandsen. *On the Density of Normal Bases in Finite Field*. December 1997. 14 pp.

RS-97-43 Vincent Balat and Olivier Danvy. *Strong Normalization by Run-Time Code Generation*. December 1997.

RS-97-42 Ulrich Kohlenbach. *On the No-Counterexample Interpretation*. December 1997. 26 pp.

RS-97-41 Jon G. Riecke and Anders B. Sandholm. *A Relational Account of Call-by-Value Sequentiality*. December 1997. 24 pp. Appears in *Twelfth Annual IEEE Symposium on Logic in Computer Science,* LICS '97 Proceedings, pages 258–267.

RS-97-40 Harry Buhrman, Richard Cleve, and Wim van Dam. *Quantum Entanglement and Communication Complexity*. December 1997. 14 pp.

RS-97-39 Ian Stark. *Names, Equations, Relations: Practical Ways to Reason about 'new'*. December 1997. ii+33 pp. This supersedes the earlier BRICS Report RS-96-31. It also expands on the paper presented in Groote and Hindley, editors, *Typed Lambda Calculi and Applications: 3rd International Conference,* TLCA '97 Proceedings, LNCS 1210, 1997, pages 336–353.

RS-97-38 Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. *On the Distributed Complexity of Computing Maximal Matchings*. December 1997. 16 pp. To appear in *The Ninth Annual ACM-SIAM Symposium on Discrete Algorithms,* SODA '98.

RS-97-37 David A. Grable and Alessandro Panconesi. *Fast Distributed Algorithms for Brooks-Vizing Colourings (Extended Abstract)*. December 1997. 20 pp. To appear in *The Ninth Annual ACM-SIAM Symposium on Discrete Algorithms,* SODA '98.