Linear Forwarders

Philippa Gardner [a], Cosimo Laneve [b,*], Lucian Wischik [c]

[a] *Imperial College, London, United Kingdom*
[b] *University of Bologna, Italy*
[c] *Microsoft, Redmond, USA*

Abstract

A *linear forwarder* is a process that receives one message on a channel and sends it on a different channel. We use linear forwarders to provide a distributed implementation of Milner's asynchronous pi calculus. Such a distributed implementation is known to be difficult due to *input capability*, where a received name is used as the subject of a subsequent input. This allows the dynamic creation of large input processes in the wrong place, thus requiring comparatively large code migrations in order to avoid consensus problems. Linear forwarders constitute a small atom of *input capability* that is easy to move.

We show that the full input capability can be simply encoded using linear forwarders. We also design a distributed machine, demonstrating the ease with which we can implement the pi calculus using linear forwarders. We also show that linear forwarders allow for a simple encoding of distributed choice and have "clean" behaviour in the presence of failures.

Key words: Pi-calculus, distributed implementation, input capability, linear forwarders.

1 Introduction

Distributed interaction has become a necessary part of modern programming languages. The asynchronous pi calculus, Aπ for short, is widely regarded as a foundation for such languages. In this calculus, a program (or process) has

* Corresponding author.
Email addresses: pg@doc.ic.ac.uk (Philippa Gardner), laneve@cs.unibo.it (Cosimo Laneve), lu@wischik.com (Lucian Wischik).

a collection of channels, and it executes through interaction over these channels. A natural, well-studied distributed interpretation is to let each channel belong to a single location: for instance, one location for the channels x, y, z and another for u, v, w. Interaction on channel x only occurs at the location assigned to x: an *input resource* $x(u).P$ resides at location x, waits to receive formal parameter u and then continues with P; an *output* $\overline{x}\,v$ knows to go to the location x in order to find the matching input resource. Outputs are small and so have the freedom to move between locations, whereas inputs have possibly large continuations and so their movement is restricted.

This restricted input movement is subtle, since Aπ has a behaviour called *input capability*, which is the ability to receive a channel name and subsequently accept input on it. Consider the example $x(u).u(v).Q$. This program is located at (the location of) x, but upon reaction with $\overline{x}\,w$ it produces the continuation $w(v).Q\{^w/_u\}$. This continuation is in the wrong place, since it is still at x whereas it should be at w. A key challenge with distributing Aπ is to find sensible ways of restricting this input capability so that inputs only reside at their correct locations.

The join calculus [13], the local pi calculus [25] and the $\pi_{1\ell}$ calculus [1] are examples of *local* calculi that simply disallow input capability: that is, in a term $x(u).P$, the P may not contain any input on channel u. The join calculus achieves localisation with an elegant syntactic constraint in which the same language construct is used both to declare a channel and define its input behaviour. The local pi calculus achieves it with a well-formedness constraint on processes, and the $\pi_{1\ell}$ calculus with a type system. Whilst these calculi and resulting implementations [10] are elegant and highly successful, we believe the argument for *always* removing input capability is not convincing.

An initial reason for removing input capability, used for example in the development of the join calculus and its associated implementation, was just that it was unimplementable. Researchers then argued that input capability was also unnecessary, since much can be done without it and Aπ can be encoded in the join calculus. This is a compelling yet incomplete reasoning. There remains the open problem of whether it is in fact possible to provide a distributed implementation of input capability, and whether such an implementation is useful. In this paper, we show that such a distributed implementation is theoretically possible. The `PiDuce` project [7] is a substantial prototype for assessing whether such an implementation is useful.

To help introduce our ideas, we first recall the encoding of input capability into the join calculus. A key point of the encoding is to split channels into two parts: the *input part* and the *output part*. The input $x(u).P$ is encoded as (we write it in Aπ syntax rather than using the join calculus syntax) $(x')(\overline{x_i}\,x' \mid !x'(u_i, u_o).Q)$, which sends an *input proxy* x' to the input part x_i of x. When

an output on x is available – encoded as a message on the output part x_o of x – the proxy x' will be activated and the translated continuation Q of P be triggered. To obtain a tight correspondence with Aπ – a full abstraction result – the join calculus encodings must be wrapped by terms protecting the access to the channels (enforcing the protocol splitting names into input and output parts). Therefore, whilst this is an interesting expressivity result, it cannot be regarded as a sensible implementation of Aπ; such an implementation should not split channels and should not require the use of wrappers.

We show that a simple and direct distributed implementation of Aπ is possible, contrary to folklore. Our idea is to introduce a limited form of input capability, the *linear forwarder*, using which it is possible to encode input capability in a straightforward way. A linear forwarder $x{-\!\circ}y$ is a process that allows just one x to be turned into a y. It plays a similar role of an input proxy in the join calculus, except that it really does behave like an input and so does not require a firewall. It is like an input in that it reacts with an output on x, by forwarding (renaming it) to y. It is an *input capability* in that, in the term $u(x).P$ for example, a forwarder $x{-\!\circ}y$ is allowed in P. Using this limited form of input capability, we provide a simple encoding of general input capability, prove a simple full-abstraction result, and design a distributed abstract machine showing that it is indeed possible to give a simple distributed implementation in Aπ.

One interpretation of $x{-\!\circ}y$ is just as the pi process $x(u).\overline{y}\,u$. In fact, a linear forwarder may be viewed as an input proxy in a point-to-point network such as the Internet. However, we choose to use forwarders as first-class operators in order to abstract away from a particular style of implementation. Other networks might provide different implementations of linear forwarders. In a broadcast network for example, the forwarder $x{-\!\circ}y$ might be located at y: when it hears an offer of $\overline{x}\,\widetilde{u}$ being broadcast, the machine at y can take up the offer.

Our results demonstrate that it is possible to implement Aπ on a distributed machine. An essential next step is study practical applications of our implementation of Aπ, in particular fully contrasting our implementation with the indirect implementation given by the join machine. We are actively applying our ideas to Web services, where channel update is essential: for example, when query patterns must be combined, as argued in [34,16,23], or when existing services must be orchestrated in order to provide new services [21]. In particular, this orchestration of services is currently being studied in University of Bologna as part of the extensive `PiDuce` project [7]; see conclusions for more details. We also note that in `BizTalk`, a language for Web services by Microsoft [40], input capability is offered when run over a reliable message service (MSMQ), but not otherwise. This paper provides a formal treatment of one possible implementation of `BizTalk` (the implementation details of `BizTalk`

have not been published). A more precise account of our work follows.

We begin by introducing the *linear forwarder calculus*, AL$\ell\pi$ for short, which is like Aπ except that input capability is restricted to linear forwarders. We show that AL$\ell\pi$ has a direct distributed implementation by describing an abstract machine and provide a simple encoding of Aπ in AL$\ell\pi$. To illustrate the encoding, consider the Aπ term $x(u).u(v).Q$. We encode it as

$$[\![x(u).u(v).Q]\!] \;=\; x(u).(u')(u{-\!\circ}u' \mid u'(v).[\![Q]\!])$$

where the input $u(v)$ has been turned into a local input $u'(v)$ at the same location as x, and the forwarder allows one output on u to interact with u' instead. Thus, for example, the process $\overline{x}\,w \mid \overline{w}\,y \mid [\![x(u).u(v).Q]\!]$ manifests the following behaviour:

$$\begin{array}{lll} \overline{x}\,w \mid \overline{w}\,y \mid x(u).(u')(u{-\!\circ}u' \mid u'(v).[\![Q]\!]) & & (1)\\ \quad\rightarrow\; \overline{w}\,y \mid (u')(w{-\!\circ}u' \mid u'(v).[\![Q]\!]) & \textit{interact on } x & \\ \quad\rightarrow\; (u')(\overline{u}\,'y \mid u'(v).[\![Q]\!]) & \textit{forward } \overline{w}\,y \textit{ to } \overline{u}\,'y & \\ \quad\rightarrow\; (u')[\![Q]\!]\{y/v\} & \textit{interact on } u' & \end{array}$$

In the encoding there are exactly as many forwarders $u{-\!\circ}u'$ as there are available inputs on u'. We remark that the forwarders being linear is crucial; if there were unlimited forwarders for just one input, then any further u forwarded to u' would become inert. Because of linearity, we have a full abstraction result for the encoding using barbed congruence.

Linear forwarders also give an appealing encoding of *distributed choice*. The process $x(u).P + y(v).Q$ allows either a reaction on x, or one on y, but not both. If x and y are at the same location, the choice is easy to implement. If they are remote, we *linearly forward* x and y to co-located channels x' and y' where a local choice can be made:

$$\begin{array}{l}[\![x(u).P + y(v).Q]\!] \;=\\ \qquad (x'y')\Big(\, x{-\!\circ}x' \mid y{-\!\circ}y' \mid x'(u).([\![P]\!]|y'{-\!\circ}y) + y'(v).([\![Q]\!]|x'{-\!\circ}x)\,\Big)\end{array}$$

Linearity is again crucial. If the x' branch were taken, then the other forwarder $y{-\!\circ}y'$ can be completely undone by releasing a reverse linear forwarder $y'{-\!\circ}y$ (or vice versa). With choice, the full abstraction result is weaker, since the correspondence is mediated by coupled simulation [30] rather than barbed congruence. This weaker result is unsurprising, as coupled simulation is also used in the result of Nestmann and Pierce on encoding input-guarded choice in the choice-free Aπ [29].

We design a distributed abstract machine for AL$\ell\pi$, called the *linear forwarder machine*. It is a channel-based machine which is distributed in the sense that

the channels are partitioned between locations. All processes must initially be in the "right" place for a machine to be well-formed. This means that every process inputing on free channels, let us say x, are located at the machine of x. Interaction is always local, occuring at the nominated locations of the channels. A property of our machine is that, during interaction, either processes remain in the correct place, or they are ouputs and linear forwarders which are small processes with the freedom to move. We show that the linear forwarder machine is a natural implementation of the linear forwarder calculus, proving a full-abstraction result up to barbed congruence. This result, together with our encoding of Aπ into the linear forwarder calculus, shows that it is possible to give a fully-distributed implementation of Aπ.

To examine the robustness of the linear forwarder machine, we analyze two failure models usually studied for distributed machines: (1) when failures are due to *message losses* and (2) when failures are due to *crashes of locations*. In (1), since outputs and linear forwarders are the only processes that move over the network, the failure model only accounts for their losses. The effect of one lost output or forwarder is that one input process may deadlock. Our formal account of message failure in the linear forwarder machine amounts to adding two rules that allow the outputs and linear forwarders to disappear. We show that this fallible linear forwarder calculus, FA$\ell\pi$ for short, can be simulated via the corresponding machine with failures. This is a weaker result than one using coupled similation. Coupled simulation cannot be established in this case because, in the machine, local messages can never be lost. On the contrary, in FA$\ell\pi$, missing the location information, every message may be lost. For (2), we give a formal account of location crashes in the linear forwarder machine, where processes running on a location are rebooted to some safe initial state. Such rebooted states are assumed to be the empty location for simplicity. We establish similar results to (1).

Related works. Forwarders have already been studied in detail, but for very different reasons to the work presented here. Much work centres around the private pi calculus [35] – a variant of the pi calculus in which only private names may be emitted, as in $(w)\overline{x}\,w$. Boreale uses forwarders to encode the emission of free names [6]; the reaction $x(u).Q \mid \overline{x}\,w$ does not perform the substitution $\{^w/_u\}$, but instead encodes it as a persistent (non-linear) forwarder from u to w. The forwarders must be persistent since names may occur many times in a term. Because of persistence, the number of forwarders increases during an execution of a program. The same technique is used by Merro and Sangiorgi [25] in proofs about the local pi calculus. Both were inspired by Honda's *equators* [19,24], which are bidirectional forwarders.

In our work, we use linear forwarders in a very different way, to move messages between locations rather than for encoding substitutions. Our forwarders must be linear, and hence their number decreases during execution. In a translated

pi process, if a linear forwarder moves an output message to another location, there will be an input waiting to interact with it. In contrast, this property would not hold if we used persistent forwarders, hence messages would be lost and we would not be able to prove our full-abstraction result. Our proofs are similar in structure to those of Boreale, but are much simpler due to linearity. We have only seen one other use of linear forwarders, in Kobayashi et al.'s work on using linear forwarders to simulate substitutions when channels are used linearly [20]. This work follows Boreale's agenda, rather than the ideas presented here.

The linear forwarder machine evolved from our previous work on the *fusion machine* [14], which had a similar mechanism for distributing processes. The fusion machine is based on the explicit fusion calculus [17] that uses explicit fusions $x = y$ rather than linear forwarders $x \multimap y$. Fusions yield equivalence classes of names, without giving information about the direction to which outputs should be forwarded – the representatives of the classes. In the fusion machine, this issue was solved using spanning trees. While our solution was a good first attempt, it is in fact a poor implementation strategy since it is not robust to failures of the network. A break of the spanning tree of fused names can result in many messages being lost. This limitation provided our initial motivation for exploring the work presented here.

Other distributed abstract machines for $\text{A}\pi$ in the literature include Facile [18], the Jocaml prototype [10], Distributed pi calculus [2], Nomadic Pict [38], the Ambient Calculus [8], the Channel Ambient Machine [31], and the Klaim Machine [11]. Facile uses two classes of distributed entities: (co-)located processes which execute, and channel-managers which mediate interaction. This requires a hand-shake discipline for communication between the locations of the inputs and outputs, and the different locations of the channel managers. Jocaml simplifies the Facile approach by combining input processes with channel-managers, and hence avoiding the hand-shake. However, it uses a quite different form of interaction, which does not relate that directly to pi calculus communication. This is illustrated by the encoding of the pi calculus, which has a strong correctness result but only because it is mediated by firewall processes. In addition, Jocaml forces a coarser granularity, in that every channel must be co-located with at least one other. Like Jocaml, our linear forwarder machine combines processes with channel-managers. Unlike Jocaml, our machine has finer granularity and uses the same form of interaction as the pi calculus. The other machines mentioned are based on a completely different approach to distribution, which adds explicit location constructs to the pi calculus and uses agent migrations for remote interactions. In summary, the linear forwarder machine describes a direct distributed implementation of the asynchronous pi calculus, which is correct in a stronger way than the other proposed implementations.

Summary of paper. Section 2 describes the linear forwarder calculus, and its reference semantics – barbed congruence. Section 3 gives the encoding of the pi calculus (i.e. the encoding of input capability) and proves the encoding correct. Section 4 gives a distributed abstract machine for implementing the linear forwarder calculus and proves the implementation correct. Section 5 extends the pi calculus with choice and the linear forwarder calculus with local choice, and proves a correctness result for the encoding of choice into local choice. Section 6 discusses the extension of the calculi and the abstract machines with failures, and provides correctness results. Section 7 analyzes the issue of loading program codes into the machine. We conclude in Section 8.

This paper extends our conference paper [15] in several ways: we give proofs in full, extend the work to include distributed choice, and analyze failures.

2 The linear forwarder calculus

We assume an infinite set of *names* ranged over by $u, v, x, y, \ldots$. Names represent communication channels; these names can also be transmitted during communication. Write $\widetilde{x}$ for a (possibly empty) finite sequence $x_1 \cdots x_n$ of names. *Name substitutions* $\{\widetilde{y}/\widetilde{x}\}$ are as usual.

Definition 1 (The calculus AL$\ell\pi$) *The (asynchronous)* local linear forwarder calculus*, which we abbreviate* AL$\ell\pi$ [1]*, is the calculus whose terms P are given by*

$$P \ ::= \ \mathbf{0} \ \Big| \ \overline{x}\,\widetilde{u} \ \Big| \ x(\widetilde{u}).P \ \Big| \ (x)P \ \Big| \ P|P \ \Big| \ !P \ \Big| \ x{-\!\circ}y$$

and which satisfy the no-input-capability *constraint: in $x(\widetilde{u}).P$, the P has no free occurrence of $u \in \widetilde{u}$ as subject of $u(\widetilde{v}).Q$.*

Free and bound names are standard: x is bound in $(x)P$ and $\widetilde{u}$ is bound in $x(\widetilde{u}).P$; names are free when they are not bound. Write $\mathtt{fn}(P)$ *for the free names of P.*

The operators in AL$\ell\pi$ are all standard apart from the linear forwarder $x{-\!\circ}y$. This allows one output on x to be transformed into one on y, through Definition 3 below. The meaning of the other operators is standard: the term $\mathbf{0}$ is inert; $\overline{x}\,\widetilde{u}$ is a command to send data $\widetilde{u}$ over channel x; $x(\widetilde{u}).P$ receives formal arguments $\widetilde{u}$ and then continues as P; restriction $(x)P$ limits the scope of x to P; parallel composition $P|Q$ allows two terms to run concurrently and

[1] The acronym AL$\ell\pi$ follows naming conventions in [37]: A stands for "asynchronous", L for "local" and ℓ for "linear-forwarders".

interact; and repetition $!P$ behaves like infinitely many copies of P. Write $(x_1 \cdots x_n)P$ for $(x_1) \cdots (x_n)P$.

Following Milner's presentation [27], we first define a structural congruence which equates all agents that have essentially the same structure and which we will never wish to distinguish. We then use structural congruence when giving the operational semantics.

Definition 2 *Structural congruence $\equiv$ is the smallest equivalence relation which satisfies the following axioms and is closed with respect to contexts and alpha-renaming:*

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad !P \equiv P|!P$$
$$(x)\mathbf{0} \equiv \mathbf{0} \qquad (x)(y)P \equiv (y)(x)P \qquad (x)(P|Q) \equiv P \mid (x)Q \quad \textit{if } x \notin \mathtt{fn}(P)$$

Definition 3 *The reduction relation $\rightarrow$ is the least relation satisfying the following rules and closed under $\equiv$, $(x)\_$ and $\_ \mid \_$:*

$$x(\widetilde{u}).P \mid \overline{x}\,\widetilde{v} \;\rightarrow\; P\{\widetilde{v}/\widetilde{u}\} \qquad\qquad \overline{x}\,\widetilde{u} \mid x \multimap y \;\rightarrow\; \overline{y}\,\widetilde{u}$$

Notice that the no-input-capability constraint of AL$\ell\pi$ is preserved by structural congruence and reaction.

For behavioural equivalence in AL$\ell\pi$ we use the standard notion of *barbed bisimulation* [28]. According to this notion, two agents are considered equivalent if their reductions match and they are indistinguishable under global observations:

Definition 4 *The name x is a* barb *of P, written $P \downarrow x$, when*

$$\overline{x}\,\widetilde{u} \downarrow x \qquad\qquad (y)P \downarrow x, \quad \textit{if } P \downarrow x \textit{ and } x \neq y$$
$$!P \downarrow x, \quad \textit{if } P \downarrow x \qquad\qquad P|Q \downarrow x, \quad \textit{if } P \downarrow x \textit{ or } Q \downarrow x$$

Write $\Rightarrow$ for $\rightarrow^*$ and $\Downarrow$ for $\Rightarrow\downarrow$.

Barbed bisimulation $\dot{\approx}$ *is the largest symmetric relation such that whenever $P \dot{\approx} Q$ then (1) $P \downarrow u$ implies $Q \Downarrow u$, and (2) $P \rightarrow P'$ implies $Q \Rightarrow Q'$ and $P' \dot{\approx} Q'$.*

Let $C[\,]$ be the set of contexts of AL$\ell\pi$ *generated by*

$$C[\,] ::= [\cdot] \;\Big|\; x(\widetilde{u}).C[\,] \;\Big|\; (x)C[\,] \;\Big|\; P \mid C[\,] \;\Big|\; C[\,] \mid P \;\Big|\; !C[\,]$$

The barbed congruence *is the largest symmetric relation $\cong_a^{Lc}$ such that whenever $P \cong_a^{Lc} Q$ then, for all contexts $C[\,]$ such that $C[P]$ and $C[Q]$ are* AL$\ell\pi$ *terms, $C[P] \dot{\approx} C[Q]$.*

As examples of barbed congruent terms in AL$\ell\pi$, we recall a couple of laws.

Proposition 5 *(Merro,Sangiorgi [24])*

(1) If P is a term in AL$\ell\pi$ *and u does not occur free in P as subject of an input then $P\{u'/u\} \cong_a^{Lc} (u)(!u{\multimap}u' \mid P)$.*
(2) $(u)\overline{x}\,u \cong_a^{Lc} (u)(\overline{x}\,u \mid u(v).\mathbf{0})$.

We discuss the first law. The rationale behind it is that, in an asynchronous calculus with a semantics that is unsensible to internal moves, input consumptions cannot be tested. Additionally, since u does not occurs as subject of inputs in P, emitting on u' by $P\{u'/u\}$ is the same as outputting on u, then letting the message be consumed by the (persistent linear) forwarder $!u{\multimap}u'$ and transformed into a message on u'.

3 The asynchronous pi calculus and its encoding

In this section we present the asynchronous pi caluclus Aπ and encode it into AL$\ell\pi$.

Definition 6 (Pi calculus) *The asynchronous pi calculus* Aπ *has the same terms P as for* AL$\ell\pi$ *(Definition 1), but with no linear forwarders and no input-capability condition [26]. Structural congruence $\equiv$ and reduction $\rightarrow$ are defined as in Definitions 2 and 3, but in this case the reduction relation $\rightarrow$ only has the rule $x(\widetilde{u}).P \mid \overline{x}\,\widetilde{v} \rightarrow P\{\widetilde{v}/\widetilde{u}\}$. Barbs $P \downarrow u$, barbed bisimulation $\dot{\approx}$, and barbed congruence $\cong_a^{\pi c}$ are as in Definition 4, replacing* AL$\ell\pi$ *with* Aπ.

Note that if P and Q are valid terms in AL$\ell\pi$ and Aπ, then $P \dot{\approx} Q$ in AL$\ell\pi$ if and only if $P \dot{\approx} Q$ in Aπ. This justifies our use of the same symbol $\dot{\approx}$ for both. Note also that, for congruence $\cong_a^{Lc}$ in AL$\ell\pi$, the contexts are a subset of those that are used for $\cong_a^{\pi c}$ in Aπ. We will see, however, that AL$\ell\pi$ contexts and Aπ contexts are equally discriminating (Theorem 18).

The following law generalizes Proposition 5.1 to Aπ and, therefore it also holds in AL$\ell\pi$.

Proposition 7 *(Honda, Yoshida [19])* $P\{u'/u\} \cong_a^{\pi c} (u)(!u'{\multimap}u \mid !u{\multimap}u' \mid P)$.

We use the law of Proposition 7 in the *context lemma* below. This lemma usually states that contexts gain no additional discriminating power through restriction and input-prefixing [24,37], namely $P \cong_a^{\pi c} Q$ if and only if $R \mid P\sigma \dot{\approx} R \mid Q\sigma$, for every R and renaming σ. However, in asynchronous calculi a simpler statement without renamings is equally strong. (We suspect that this is a standard result, but have not been able to find a reference for it.)

Lemma 8 *$R \mid P \mathrel{\dot{\approx}} R \mid Q$, for every R, if and only if, for every $C[\ ]$, $C[P] \mathrel{\dot{\approx}} C[Q]$.*

PROOF. The reverse direction is clear. For the forward direction, we negate the consequent: i.e. we suppose there exists $C[\ ]$ such that $C[P] \mathrel{\dot{\not\approx}} C[Q]$. By the Context Lemma in [37] there exist R' and σ such that $R' \mid P\sigma \mathrel{\dot{\not\approx}} R' \mid Q\sigma$. By Proposition 7, there exist R'' and $\widetilde{x}$ such that $(\widetilde{x})(R'' \mid P) \cong_a^{Lc} P\sigma$ and $(\widetilde{x})(R'' \mid Q) \cong_a^{Lc} Q\sigma$. Henceforth it must be the case that $R' \mid (\widetilde{x})(R'' \mid P) \mathrel{\dot{\not\approx}} R' \mid (\widetilde{x})(R'' \mid Q)$. Without loss of generality, let $\widetilde{x} \cap \mathtt{fn}(R') = \emptyset$. This reduces the proof to $(\widetilde{x})(R' \mid R'' \mid P) \mathrel{\dot{\not\approx}} (\widetilde{x})(R' \mid R'' \mid Q)$. Hence $R' \mid R'' \mid P \mathrel{\dot{\not\approx}} R' \mid R'' \mid Q$.

Definition 9 (Encoding pi) *The encoding $[\![\cdot]\!]$ maps terms from* Aπ *to* AL$\ell\pi$ *by $[\![P]\!] = [\![P]\!]_\emptyset$, where $[\![P]\!]_{\widetilde{u}}$ for set of names $\widetilde{u}$ is an auxiliary function defined inductively by:*

$$
\begin{array}{rcl}
[\![x(\widetilde{y}).P]\!]_{\widetilde{u}} & = & \begin{cases} x(\widetilde{y}).[\![P]\!]_{\widetilde{uy}} & \text{if } x \notin \widetilde{u} \\ (x')(x{\multimap}x' \mid x'(\widetilde{y}).[\![P]\!]_{\widetilde{uy}}) & \text{if } x \in \widetilde{u},\ x' \notin \mathtt{fn}(P) \cup \{x, \widetilde{u}, \widetilde{y}\} \end{cases} \\
[\![\mathbf{0}]\!]_{\widetilde{u}} & = & \mathbf{0} \\
[\![\overline{x}\,\widetilde{y}]\!]_{\widetilde{u}} & = & \overline{x}\,\widetilde{y} \\
[\![(x)P]\!]_{\widetilde{u}} & = & (x)([\![P]\!]_{\widetilde{u}}) \\
[\![P|Q]\!]_{\widetilde{u}} & = & [\![P]\!]_{\widetilde{u}} \mid [\![Q]\!]_{\widetilde{u}} \\
[\![!P]\!]_{\widetilde{u}} & = & ![\![P]\!]_{\widetilde{u}}
\end{array}
$$

In the input and restriction cases, we assume that the bound names do not clash with $\widetilde{u}$. We write $\widetilde{u}\widetilde{v}$ for set union and $\widetilde{u} \setminus \widetilde{v}$ for set difference.

It is worth remarking that we are abusing notation slightly. For input processes, $\widetilde{u}$ denotes a sequence of names. In the subscript $[\![P]\!]_{\widetilde{u}}$, $\widetilde{u}$ denotes a *set* of names.

To understand the encoding, note that the subscript contains all names that have been received in input. By the no-input-capability constraint (Definition 1), they cannot therefore be used as the subject of subsequent input. To achieve this, the encoding uses "primed" names to denote local copies of them. So the encoding of $x(u).u(y).P$ will consume a message on x and result in the processes $u'(y).[\![P]\!]_{uy}$, where u' is a new channel, and $u{\multimap}u'$ forwarding one message from u to u'. Meanwhile, any output use of u is left unchanged, since it will be forwarded if appropriate using a linear forwarder.

To illustrate the connection between the reactions of a term and its translation, we consider the Aπ reduction $\overline{x}\,v \mid x(u).P \;\rightarrow\; P\{^v/_u\}$. By translating we

obtain:

$$\begin{array}{rcl} [\![\,\overline{x}\,v \mid x(u).P\,]\!]_x & = & \overline{x}\,v \mid (x')(x{\multimap}x' \mid x'(u).[\![P]\!]_{xu}) \\ & \to & (x')(\overline{x}\,'v \mid x'(u).[\![P]\!]_{xu}) \\ & \to & (x')([\![P]\!]_{xu}\{^v/_u\}) \\ & \equiv & [\![P]\!]_{xu}\{^v/_u\} \end{array}$$

Note that the final state of the translated term is subscripted on x and u, not just on x. In addition, the translated term ends up with some garbage that was not present in the original (the restriction $(x')(\cdots)$). Because of this garbage, it is not in general true that $Q \to Q'$ implies $[\![Q]\!] \to^* [\![Q']\!]$; instead we must work up to some behavioural congruence.

Linearity is crucial in the translation. For instance, consider a nonlinear translation where forwarders are replicated:

$$(\!|x(u).P|\!)_x \;=\; (x')(!x{\multimap}x' \mid x'(u).(\!|P|\!)_{xu})$$

Then consider the example

$$\begin{array}{l} (\!|x(u).P \mid x(u).Q \mid \overline{x}\,v \mid \overline{x}\,w|\!)_x \\ \quad = \;\; (x')(!x{\multimap}x' \mid x'(u).(\!|P|\!)_{xu}) \mid (x'')(!x{\multimap}x'' \mid x''(u).(\!|Q|\!)_{xu}) \mid \overline{x}\,v \mid \overline{x}\,w \\ \quad \Rightarrow \;\; (x')(!x{\multimap}x' \mid (\!|P|\!)_{xu}\{^v/_u\}) \mid (x'')(!x{\multimap}x'' \mid x''(u).(\!|Q|\!)_{xu}) \mid \overline{x}\,w \\ \quad \to \;\; (x')(!x{\multimap}x' \mid (\!|P|\!)_{xu} \mid \overline{x'}\,w) \mid (x'')(!x{\multimap}x'' \mid x''(u).(\!|Q|\!)_{xu}) \end{array}$$

Here, both outputs were forwarded to the local name x', even though the resource $x'(u).(\!|P|\!)_{xu}$ had already been consumed by the first one. This precludes the second one from reacting with $(x'')(!x{\multimap}x'' \mid x''(u).(\!|Q|\!)_{xu})$, a reaction that would have been possible in the original Aπ term. Linearity prevents the possibility of such dead ends.

Remark 10 *A simpler encoding is also possible, which does not use subscripts and always applies the $x \in \widetilde{u}$ case. We chose the current one for the following appealing property: if P has no free input on x then $[\![P]\!]_{\widetilde{z}x} = [\![P]\!]_{\widetilde{z}}$. As an immediate consequence, if a term P in Aπ satisfies the no-input-capability constraint, then the encoding $[\![\cdot]\!]$ leaves it unchanged.*

3.1 The correctness of the encoding

The correctness of the encoding $[\![\cdot]\!]$, that $P \cong_a^{\pi c} Q$ if and only if $[\![P]\!] \cong_a^{Lc} [\![Q]\!]$, is not straightforward to prove because of the garbage left by the encoded term. To show that it is indeed garbage, we must prove that $[\![P]\!]_u$ and $[\![P]\!]_{ux}$ are congruent. But the barbed semantics offers too weak an induction hypothesis for this proof. A standard alternative technique (used for instance by Boreale [6])

is to use the barbed semantics as the primary definition, but to switch to a labelled transition system and labelled bisimulation in the proofs. Labelled bisimualtion is defined coinductively, which makes it easier to use in proofs, and is stronger than $\cong_a^{Lc}$. It therefore provides a useful proof technique for establishing results about $\cong_a^{Lc}$.

To help our argument, we introduce a further calculus, called the asynchronous, *non-local* linear forwarder calculus A$\ell\pi$ in Definition 11.

Definition 11 *The calculus* A$\ell\pi$ *has terms* P *and contexts* $C[\ ]$ *as in* AL$\ell\pi$ *(Definitions 1 and 4), but without the no-input-capability constraint. Structural congruence* $\equiv$ *reduction* $\rightarrow$*, and barbs are as in* AL$\ell\pi$ *(Definitions 2, 3, and 4).*

The calculus A$\ell\pi$ has both AL$\ell\pi$ and Aπ as subcalculi. It is AL$\ell\pi$ without the no-input-capability constraint or, equivalently, Aπ with linear forwarders. We first define the labelled transition system for A$\ell\pi$, and therefore for AL$\ell\pi$ and Aπ. To this aim, let μ range over input labels $x(\widetilde{u})$, bound output labels $(\widetilde{z})\overline{x}\,\widetilde{u}$ where $\widetilde{z} \subseteq \widetilde{u}$, and the label τ. Let also $\mathtt{fn}(x(\widetilde{u})) = \{x\}$, $\mathtt{fn}((\widetilde{z})\overline{x}\,\widetilde{u}) = \{x, \widetilde{u}\} \setminus \widetilde{z}$, $\mathtt{bn}(x(\widetilde{u})) = \{\widetilde{u}\}$, $\mathtt{bn}((\widetilde{z})\overline{x}\,\widetilde{u}) = \{\widetilde{z}\}$, and $\mathtt{fn}(\tau) = \mathtt{bn}(\tau) = \emptyset$.

Definition 12 *The transition relation* $P \xrightarrow{\mu} Q$*, for terms* P *and* Q *in* A$\ell\pi$*, is defined inductively on the structure of* P*:*

$$x(\widetilde{u}).P \xrightarrow{x(\widetilde{u})} P \qquad \overline{x}\,\widetilde{u} \xrightarrow{\overline{x}\widetilde{u}} \mathbf{0} \qquad x{\multimap}y \xrightarrow{x(\widetilde{u})} \overline{y}\,\widetilde{u}$$

$$\frac{P \xrightarrow{\mu} Q \quad y \notin \mathtt{fn}(\mu)}{(y)P \xrightarrow{\mu} (y)Q} \qquad \frac{P \xrightarrow{(\widetilde{z})\overline{x}\widetilde{u}} Q \quad y \neq x,\ y \in \widetilde{u}\setminus\widetilde{z}}{(y)P \xrightarrow{(y\widetilde{z})\overline{x}\widetilde{u}} Q} \qquad \frac{P \mid !P \xrightarrow{\mu} Q}{!P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathtt{bn}(\mu) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \frac{P \xrightarrow{(\widetilde{z})\overline{x}\widetilde{v}} P' \quad Q \xrightarrow{x(\widetilde{u})} Q' \quad \widetilde{z} \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} (\widetilde{z})(P' \mid Q'\{\widetilde{v}/\widetilde{u}\})}$$

The transitions of $P|Q$ *have mirror cases, which we have omitted. We implicitly identify terms up to alpha-renaming* $\equiv_\alpha$*: that is, if* $P \equiv_\alpha P'$ *and* $P' \xrightarrow{\mu} P''$ *then* $P \xrightarrow{\mu} P''$*. Write* $\stackrel{\tau}{\Longrightarrow}$ *for* $\xrightarrow{\tau}{}^*$*, and* $\stackrel{\mu}{\Longrightarrow}$ *for* $\stackrel{\tau}{\Longrightarrow}\xrightarrow{\mu}\stackrel{\tau}{\Longrightarrow}$ *when* $\mu \neq \tau$*.*

The reduction and barb relations in Definition 11 and the transition relation in Definition 12 are related by the proposition below.

Proposition 13 *(1)* $P \rightarrow P'$ *if and only if* $P \xrightarrow{\tau} \equiv P'$*;*
(2) $P \downarrow x$ *if and only if* $P \xrightarrow{\overline{x}\widetilde{u}} P'$*, for some* $\widetilde{u}$*.*

The calculus A$\ell\pi$ is equipped with an observational semantics called *asynchronous bisimulation* [4]. The next definition also introduces the asynchronous simulation, to be used in Section 5.

Definition 14 *An* asynchronous simulation *is a binary relation* $\mathcal{R}$ *between* A$\ell\pi$ *processes such that* $P\,\mathcal{R}\,Q$ *implies:*

(1) if $P \xrightarrow{\tau} P'$, *then* $Q \stackrel{\tau}{\Longrightarrow} Q'$ *and* $P' \mathcal{R} Q'$;
(2) if $P \xrightarrow{(\widetilde{z})\overline{x}\widetilde{u}} P'$ *and* $\widetilde{z} \cap \mathtt{fn}(Q) = \emptyset$, *then* $Q \stackrel{(\widetilde{z})\overline{x}\widetilde{u}}{\Longrightarrow} Q'$ *and* $P' \mathcal{R} Q'$;
(3) if $P \xrightarrow{x(\widetilde{u})} P'$ *and* $\widetilde{u} \cap \mathtt{fn}(Q) = \emptyset$ *then*
  (a) either $Q \stackrel{x(\widetilde{u})}{\Longrightarrow} Q'$, *and* $P' \mathcal{R} Q'$;
  (b) or $Q \stackrel{\tau}{\Longrightarrow} Q'$, *and* $P' \mathcal{R} (Q' \mid \overline{x}\,\widetilde{u})$.

A relation $\mathcal{R}$ *is an* asynchronous bisimulation *when both* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are asynchronous simulations.* Asynchronous bisimilarity, *written* $\approx_a$ *is the largest asynchronous bisimulation.*

Some standard results, from asynchronous bisimilarity for Aπ (see [37], Chapter 5), extend naturally to A$\ell\pi$:

Theorem 15 *(1)* $\approx_a$ *is an equivalence relation;*
(2) $\equiv \subseteq \approx_a$;
(3) $\approx_a$ *is a congruence.*

Clearly $\approx_a$ is a barbed bisimulation; therefore $\approx_a \subseteq \dot{\approx}$. Because it is a congruence, it is closed under all contexts in A$\ell\pi$, then *a fortiori* it is also closed under Aπ contexts and AL$\ell\pi$ contexts, and so

$$\approx_a \subset \cong_a^{\pi c} \qquad \text{and} \qquad \approx_a \subset \cong_a^{Lc}. \tag{2}$$

These key equations justify our choice to perform our proofs in $\approx_a$ rather than $\cong_a^{\pi c}$ or $\cong_a^{Lc}$. We recall that asynchronous bisimilarity, although useful as a proof technique, is stronger than desirable for general use. For instance, by Proposition 5.1, $\overline{x}\,u \cong_a^{Lc} (v)(\overline{x}\,v \mid !(v \multimap u))$. However this equality is false in $\approx_a$. In particular, the term on the right hand side may undergo $\xrightarrow{(v)\overline{x}\,(v)}$, but the term on the left cannot.

The following laws are key to the correctness proof.

Lemma 16 *For all* P *in* A$\ell\pi$ *and* $y \notin \mathtt{fn}(P)$,

(1) $x \multimap z \approx_a x(\widetilde{u}).\overline{z}\,\widetilde{u}$.
(2) $P \approx_a (y)(\overline{y}\,\widetilde{u} \mid y(\widetilde{u}).P)$.
(3) $x(\widetilde{u}).P \approx_a (y)(x \multimap y \mid y(\widetilde{u}).P)$.

PROOF. For item (i), consider the relation $\mathcal{S} = \{(x \multimap z, x(\widetilde{u}).\overline{z}\,\widetilde{u})\} \cup \approx_a$. It is easy to show that $\mathcal{S}$ is contained in $\approx_a$.

For item (ii), this is an instance of a well-known law in Aπ: $P\{\widetilde{v}/\widetilde{u}\} \approx_a (y)(\overline{y}\,\widetilde{v} \mid y(\widetilde{u}).P)$.

Item (iii) is omitted because it is similar to item 2.

The following corollary shows the operational correspondence of the encoding: any transition of P is matched by one in $[\![P]\!]_{\widetilde{u}}$, and vice versa up to asynchronous bisimilarity.

Corollary 17 (Operational correspondence) *$P \approx_a [\![P]\!]_{\widetilde{u}}$, for all $\widetilde{u}$.*

PROOF. This follows directly from Lemma 16.3 and Definition 9. We spell out the details. Consider the smallest relation $\mathcal{S}$ containing the identity and closed under the following.

- If $P \mathcal{S} Q$ and $Q \approx_a R$ then $P \mathcal{S} R$.
- If $P \mathcal{S} Q$ then also $x(\widetilde{u}).P \mathcal{S} x(\widetilde{u}).Q$, $(x)P \mathcal{S} (x)Q$, $!P \mathcal{S} !Q$, and $P\sigma \mathcal{S} Q\sigma$ for all substitutions σ.
- If $P \mathcal{S} Q$ and $P' \mathcal{S} Q'$ then $P \mid P' \mathcal{S} Q \mid Q'$.

The proof that $\mathcal{S}$ is a $\approx_a$-bisimulation is standard. By Lemma 16.3 we derive $\{(P, [\![P]\!]_{\widetilde{u}})\} \subseteq \mathcal{S}$, thus concluding the proof.

We have all the preliminaries in place for demonstrating the correctness of the encoding.

Theorem 18 (Correctness) *For P and Q in $\mathrm{A}\pi$,*

(1) $P \dot{\approx} Q$ if and only if $[\![P]\!] \dot{\approx} [\![Q]\!]$;
(2) $P \cong_a^{\pi c} Q$ if and only if $[\![P]\!] \cong_a^{Lc} [\![Q]\!]$.

PROOF. Item 1 is a simple consequence of Corollary 17, since $\approx_a \subset \dot{\approx}$. For item 2 we use Lemma 8 and its analogue in $\mathrm{A}\pi$; namely, for P, Q in $\mathrm{A}\pi$ then

$$\forall R : R|P \dot{\approx} R|Q \Leftrightarrow P \cong_a^{\pi c} Q. \tag{3}$$

To prove item 2, start in the reverse direction and assume the converse that $P \ncong_a^{\pi c} Q$. By Equation 3 there exists a term R in $\mathrm{A}\pi$ such that $R|P \dot{\not\approx} R|Q$. The followings results hold:

$$\begin{array}{ccc} R \mid P & \dot{\not\approx} & R \mid Q \\ \approx_a & & \approx_a \\ [\![R]\!] \mid [\![P]\!] & \dot{\not\approx} & [\![R]\!] \mid [\![Q]\!] \end{array} \qquad \begin{array}{l} \text{in } \mathrm{A}\pi \\ \text{by Corollary 17} \\ \text{in } \mathrm{AL}\ell\pi \end{array}$$

Hence $[\![P]\!] \ncong_a^{Lc} [\![Q]\!]$ in $\mathrm{AL}\ell\pi$.

We now prove the forwards direction. Define a translation $\overset{\frown}{\cdot}$ from $\mathrm{A}\ell\pi$ to

Aπ as follows:

$$\begin{aligned}
\widehat{u{\multimap}v} &= u(\widetilde{x}).\overline{v}\,\widetilde{x} \\
\widehat{u(\widetilde{x}).P} &= u(\widetilde{x}).\,\widehat{P} \\
\widehat{\mathbf{0}} &= \mathbf{0} \\
\widehat{\overline{u}\,\widetilde{x}} &= \overline{u}\,\widetilde{x} \\
\widehat{(x)P} &= (x)\,\widehat{P} \\
\widehat{P|Q} &= \widehat{P} \mid \widehat{Q} \\
\widehat{!P} &= !\,\widehat{P}
\end{aligned}$$

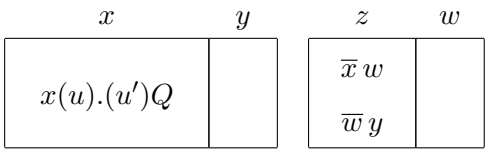
By Lemma 16.1, for any R in A$\ell\pi$, we have $R \approx_a \widehat{R}$. Assume $[\![P]\!] \not\approx_a^{Lc} [\![Q]\!]$ in AL$\ell\pi$. Then by Lemma 8 there exists R in AL$\ell\pi$ such that $R \mid [\![P]\!] \;\dot{\not\approx}\; R \mid [\![Q]\!]$. By Corollary 17, $P \approx_a [\![P]\!]$. Therefore by congruence and transitivity of $\approx_a$ we have $R \mid [\![P]\!] \approx_a \widehat{R} \mid P$ and $R \mid [\![Q]\!] \approx_a \widehat{R} \mid Q$. Additionally, since $\approx_a \subseteq \dot{\approx}$ (see the discussion after Theorem 15) then $\widehat{R} \mid P \;\dot{\not\approx}\; \widehat{R} \mid Q$. Hence $P \not\approx_a^{\pi c} Q$.

4 The linear forwarder machine

In this section we develop a distributed machine for AL$\ell\pi$, called the *linear forwarder machine*, which is suitable for a point-to-point network such as the Internet. We start with a set of locations identified by IP numbers. Each location contains several *channel-managers* identified by port numbers, which run in parallel. Each channel-manager contains an unordered set of program-fragments; some fragments may be blocked, waiting to receive on that channel; others may be waiting to be deployed to the right channel manager; others may be waiting to be interpreted locally. Let the names $u, v, x, y, \ldots$ range over IP:port pairs, so that each name identifies a particular channel-manager at a particular location.

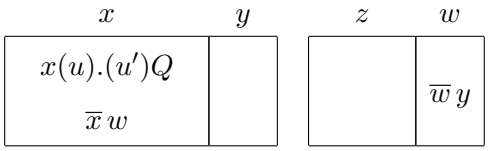
We first give a diagrammatic overview of the machine. Then we provide a formal syntax and semantics, give a translation from AL$\ell\pi$ to the machine, and prove a full abstraction result with respect to barbed congruence.

The following diagram shows a machine consisting of two locations, one with channel managers x, y and the other with z, w.



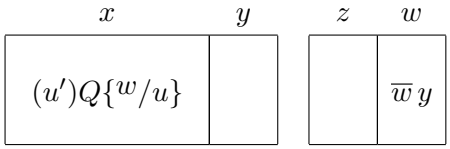
This machine corresponds to the AL$\ell\pi$ term $\overline{x}\,w \mid \overline{w}\,y \mid x(u).(u')Q$. Let $Q = u{\multimap}u' \mid u'(v).R$, so that the above term also corresponds to the example in the Introduction, which illustrates the encoding of input capability into linear forwarders.

The input fragment at x is currently blocked, waiting for input. The output fragments at z will be sent asynchronously to their intended destinations:



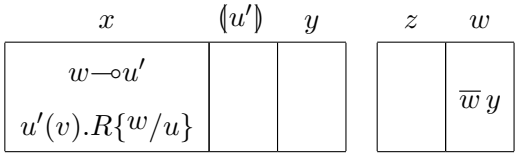
d.out

Now the channel-manager x has an available input and output, and so reacts them together. This gives rise to the substitution $\{^w/_u\}$:



react

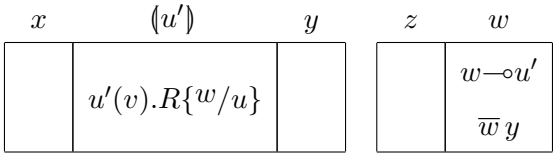
Next the channel-manager x executes the (u') command, by creating a new co-located channel manager. In order to evidence that u' is a channel manager local to the current machine, we wrap u' into the parentheses $(\!|\cdot|\!)$. Recall that $Q\{^w/_u\} = w{\multimap}u' \mid u'(v).R\{^w/_u\}$.



new

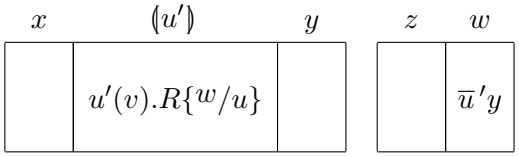
Next the linear forwarder $w{\multimap}u'$ is sent to its intended destination w. This is easy to implement because the linear forwarder is a small data structure carrying only two channel names. Additionally, the input $u'(z).R\{^w/_u\}$ is sent

to u'. This is also easy to implement because x and u' are co-located:



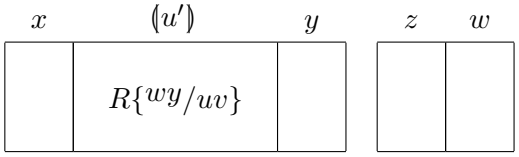
d.fwd

The channel-manager at w now contains a linear forwarder and an output, and so allows them to react:



fwd

Finally, this output at w is forwarded to channel u' where it can react



A crucial machine step is the deployment of the input $u'(v).R\{^w/_u\}$ from x to u'. This deployment is only allowed when x and u' are co-located. Otherwise, it would be too unwieldy to move the potentially-large program across the network. We must ensure that programs are never deployed at runtime between channels that are not co-located. We enforce this with well-formedness properties. First, we require that, in a machine $x[P]$, every free input $z(v).Q$ in P is such that the channels x and z are co-located. Second, we require that P satisfies the no-input-capability constraint. Together, these constraints are sufficient to avoid the deployment problem.

4.2 The formal account

Definition 19 (Linear forwarder machine) *The* linear forwarder machines *M are given by the following grammar, where P ranges over terms in* AL$\ell\pi$ *(Definition 1).*

$$M \quad ::= \quad \mathbf{0} \;\Big|\; \mathring{x}[P] \;\Big|\; (\!|x|\!)\,[P] \;\Big|\; M, M$$

The presentation here is similar to that given for the fusion machine [14]. The *basic channel-manager* $\mathring{x}[P]$ denotes a channel-manager at channel x containing a *body* P. The *local channel-manager* $(\!|x|\!)\,[P]$ denotes a channel-manager where the name x is not visible outside the machine. For example,

if $M = (\!|x|\!)\,[P]$, M' then x cannot be accessed from the environment without being extruded in advance. We write $\texttt{lchan}(M)$ for the names of only the *local channel-managers*; we write $\texttt{chan}(M)$ to denote the set of names of all channel-managers, local and not, in the machine. We also write x to denote $\mathring{x}$ or $(\!|x|\!)$.

In this presentation, we have used arbitrary replication $!P$ for simplicity. In a real machine, we would instead use guarded replication $!x(\widetilde{u}).P$ [36], as it is more amenable to implementation.

Assume a *co-location* equivalence relation L on channels. Write $x@y$ to mean that $(x, y) \in L$ with the intended meaning that the two channels are at the same location. It is always possible to create a fresh channel at an existing location: to this end we assume that each equivalence class in L is infinitely large. In the machine calculus, we generally assume L rather than writing it explicitly. Machines meet the following *well-formed conditions*:

Localization. All code is in the right place, and does not need to be moved at runtime. Formally, for every channel machine $x[P]$, then every free input $z(\widetilde{u}).Q$ in P satisfies $x@z$. Moreover, P satisfies the no-input-capability constraint.

Single-definition. There is exactly one channel-manager per channel. Formally, a machine $x_1[P_1], \cdots, x_n[P_n]$ is singly-defined when $i \neq j$ implies $x_i \neq x_j$.

For example, the machines $x[y().P], y[\overline{y}\,]$ with $x \not@ y$ and $x[P], x[Q]$ are ill-formed. Note that the localization constraint for machines is stricter than that used for AL$\ell\pi$ or the $L\pi$ of [37]. This is to be expected, since these calculi do not have explicit locations. Well-formedness is preserved by structural congruence and reactions defined below.

Definition 20 *The* structural congruence $\equiv$ *is the smallest equivalence relation satisfying the following axioms and closed with respect to alpha-equivalence of local channel managers:*

$$M, \mathbf{0} \equiv M \qquad M_1, M_2 \equiv M_2, M_1 \qquad M_1, (M_2, M_3) \equiv (M_1, M_2), M_3$$
$$P \equiv Q \quad \textit{implies} \quad \mathring{x}[P] \equiv \mathring{x}[Q] \ \textit{and}\ (\!|x|\!)\,[P] \equiv (\!|x|\!)\,[Q]$$

The reduction step $\rightarrow$ *and the* heating step $\rightharpoonup$ *are the smallest relations satisfying the rules below, and closed with respect to structural congruence.*

$$x[\overline{x}\,\widetilde{v} \mid x(\widetilde{u}).P \mid R] \;\rightarrow\; x[P\{\widetilde{v}/\widetilde{u}\} \mid R] \qquad (\textit{react})$$
$$x[x{\multimap}z \mid \overline{x}\,\widetilde{v} \mid R] \;\rightarrow\; x[\overline{z}\,\widetilde{v} \mid R] \qquad (\textit{fwd})$$

$$x[(y)P \mid R] \;\rightharpoonup\; x[P\{^z/_y\}|R],\ (\!|z|\!)\,[\mathbf{0}],\quad \textit{if } z \notin \{x\} \cup \mathtt{fn}(P|R),\ z@x \qquad (\textit{new})$$

$$x[z{\multimap}y \mid P],\ z[Q] \;\rightharpoonup\; x[P],\ z[z{\multimap}y \mid Q], \qquad (\textit{d.fwd})$$

$$x[\overline{z}\,\widetilde{v} \mid P],\ z[Q] \;\rightharpoonup\; x[P],\ z[\overline{z}\,\widetilde{v} \mid Q], \qquad (\textit{d.out})$$

$$x[z(\widetilde{u}).P \mid Q],\ z[R] \;\rightharpoonup\; x[Q],\ z[z(\widetilde{u}).P \mid R],\quad \textit{if } x@z \qquad (\textit{d.in})$$

$$\frac{M \to M',\quad \mathtt{chan}(M') \cap \mathtt{chan}(N) = \emptyset}{M, N \to M', N} \qquad \frac{M \rightharpoonup M',\quad \mathtt{chan}(M') \cap \mathtt{chan}(N) = \emptyset}{M, N \rightharpoonup M', N}$$

We write $M \Rightarrow M'$ *if* $M\ (\to \cup \rightharpoonup)^*\ M'$.

Alpha-equivalence of local channel managers means that, if $M = (\!|x|\!)\,[P]$, M' and $z \notin \mathtt{chan}(M)$, then $M \equiv (\!|z|\!)\,[P\{^z/_x\}]$, $M'\{^z/_x\}$. We observe that, in rules *d.fwd*, *d.out*, and *d.in*, $x \neq z$ by well-formedness.

The rationale of splitting steps in reductions and heatings is as follows: reduction steps closely reflect the reductions in AL$\ell\pi$; heating steps model movements of codes between locations of the distributed machine. We draw attention to steps (*new*) and (*d.in*). The step (*new*) picks a fresh channel-name z and this channel is deemed to be at the location where the command was executed. The step (*d.in*) will only move an input from one channel x to another channel z, if the two channels are co-located; hence, there is no "real" movement of input processes. In other words, the migration of input processes from a location to a *different* one is disallowed in the linear forwarder machine. Actually, such machines are banned by well-formedness. Note that well-formedness is a sufficient but not a *necessary* condition for avoiding such migrations. For instance, the *localization* constraint says that nested free inputs must be co-located. However, if x and u are not co-located then $x[(z)(z(\,).u(\,))]$ violates the localization property, but it is inert and would show no problem at run-time.

Definition 21 *A name* x *is a* barb *in a machine* M*, written* $M \downarrow x$*, when*

$$\mathring{z}[P] \downarrow x \qquad \textit{if } P \downarrow x$$

$$(\!|z|\!)\,[P] \downarrow x \qquad \textit{if } P \downarrow x \textit{ and } x \neq z$$

$$M, N \downarrow x \qquad \textit{if } x \notin \mathtt{lchan}(M, N) \textit{ and } (M \downarrow x \textit{ or } N \downarrow x)$$

We write $M \Downarrow x$*, when* $M \Rightarrow M'$ *and* $M' \downarrow x$.

For example

$$\mathring{x}[\overline{x}\,u] \downarrow x \qquad y[\overline{x}\,u] \downarrow x \qquad (\!|x|\!)\,[\overline{x}\,u] \not\downarrow x \qquad (\!|x|\!)\,[\mathbf{0}],\ y[\overline{x}\,u] \not\downarrow x.$$

Definition 22 Barbed bisimulation $\dot{\approx}$ *is defined as in Definition 4, where processes are replaced by machines.* Machine equivalence $M \cong_a M'$ *holds*

when, for all machines N, then $M, N \stackrel{\bullet}{\approx} M', N$ assuming that M, N and M', N are well-formed.

For example $\mathring{x}[\overline{x}\,u] \not\cong_a \mathring{x}[\overline{x}\,v]$ because of the context machine $\mathring{z}[x{\multimap}z \mid z(w).\overline{w}\,]$ or because the machine $\mathring{z}[x(w).\overline{w}\,]$ where $x@z$. This is an interesting point: in machine contexts, we may always add inputs on remote names either by using linear forwarders or by using co-located names.

The linear forwarder machine is a faithful implementation of AL$\ell\pi$, as shown by the the translation and results below.

Definition 23 (Translation) *Let* $\mathtt{calc}(M) = (\widetilde{z})(\widehat{M})$ *where* $\widetilde{z} = \mathtt{lchan}(M)$ *and*

$$\widehat{\mathbf{0}} = \mathbf{0} \qquad \widehat{\mathring{x}[P]} = P \qquad \widehat{(\!(x)\!)\,[P]} = P \qquad \widehat{M, N} = \widehat{M} \mid \widehat{N}.$$

Before demonstrating the correctness of the translation, we give notations for machines and state some preliminary results.

Notation 24 $(\widetilde{x})M$**,** *with* $\widetilde{x} \subseteq \mathtt{chan}(M)$*, denotes the machine M where every channel in $\widetilde{x}$ has been turned local;*

$M \mid\mid N$**,** *with* $\mathtt{lchan}(M) = \mathtt{lchan}(N) = \emptyset$*, denotes the machine defined as follows.*

$$M \mid\mid N \stackrel{def}{=} \begin{cases} N & \text{if } M = \mathbf{0} \\ \mathring{x}[P|Q], (M' \mid\mid (N', N'')) & \text{if } M = \mathring{x}[P], M',\ N = N', \mathring{x}[Q], N'' \\ \mathring{x}[P], (M' \mid\mid N) & \text{if } M = \mathring{x}[P], M',\ x \notin \mathtt{chan}(N) \end{cases}$$

$M \mid N$**,** *where* $M = (\widetilde{x})M'$*,* $N = (\widetilde{y})N'$*,* $\mathtt{lchan}(M') = \mathtt{lchan}(N') = \emptyset$*, and* $\widetilde{x} \cap \mathtt{chan}(N') = \emptyset = \widetilde{y} \cap \mathtt{chan}(M')$*, denotes the machine* $(\widetilde{x}\widetilde{y})(M \mid\mid N)$*.*

It is easy to verify that $(M \mid M') \mid M'' = M \mid (M' \mid M'')$. Therefore, in the following, we compose machines in parallel without caring about parentheses. The proposition below collects some properties that follow directly from definitions.

Proposition 25 *(1) If* $M \to M'$ *and* $x \notin \mathtt{lchan}(M)$ *then* $(x)(M \mid x[\mathbf{0}]) \to (x)(M' \mid x[\mathbf{0}])$*. Similarly, if* $M \rightharpoonup M'$ *and* $x \notin \mathtt{lchan}(M) \cup \mathtt{lchan}(M')$ *then* $(x)(M \mid x[\mathbf{0}]) \rightharpoonup (x)(M' \mid x[\mathbf{0}])$*.*

(2) If $M \to M'$ *and* $M \mid N$ *is defined then* $M \mid N \to M' \mid N$*. Similarly, if* $M \rightharpoonup M'$ *and both* $M \mid N$ *and* $M' \mid N$ *are defined then* $M \mid N \rightharpoonup M' \mid N$*.*

Note that Proposition 25 has additional constraints for heating steps. These constraints guarantee the absence of clashes of names created by steps in the context.

Labelled transitions in the calculus and heating steps in the machine are related as detailed by the following lemma. Let $\mathbf{0}_{x_1 \cdots x_n}$ denote the machine

$x_1[\mathbf{0}], \cdots, x_n[\mathbf{0}]$.

Lemma 26 *(1) If* $\mathtt{calc}(M) \xrightarrow{x(\widetilde{u})} P$ *then:*

- *either* $M \rightharpoonup^* (\widetilde{y})(x[x(\widetilde{u}).Q] \mid M')$ *and* $P \equiv \mathtt{calc}((\widetilde{y})(x[Q] \mid M'))$,
- *or* $M \rightharpoonup^* (\widetilde{y})(x[x \multimap z] \mid M')$ *and* $P \equiv \mathtt{calc}((\widetilde{y})(x[\overline{z}\,\widetilde{u}] \mid M'))$

according to whether the process that moves in $\mathtt{calc}(M)$ *is an input or a linear forwarder;*

(2) if $\mathtt{calc}(M) \xrightarrow{(\widetilde{z})\overline{x}\,\widetilde{v}} P$ *then* $M \rightharpoonup^* (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M')$ *with* $\widetilde{z} \cap \widetilde{y} = \emptyset$ *and* $P \equiv \mathtt{calc}((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M'))$.

PROOF. The first item is not difficult to prove. We demonstrate the second item by induction on the depth of the derivation of $\mathtt{calc}(M) \xrightarrow{(\widetilde{z})\overline{x}\,\widetilde{v}} P$. The basic case is when $\mathtt{calc}(M) = \overline{x}\,\widetilde{v}$. Therefore, by definition of `calc`, there is x' such that $M = x'[\overline{x}\,\widetilde{v}]$. The statement follows immediately.

The inductive steps regard the rules for parallel, scope, and replication. For parallel we have $\mathtt{calc}(M) = Q \mid R$ with $Q \xrightarrow{(\widetilde{z})\overline{x}\,\widetilde{v}} P$ and $\widetilde{z} \notin \mathtt{fn}(R)$. In this case $\mathtt{lchan}(M) = \emptyset$, therefore, by definitions of `calc` and the operation $\mid$ on machines, there are M' and M'' such that $M = M' \mid M''$, $\mathtt{calc}(M') = Q$, and $\mathtt{calc}(M'') = R$. By inductive hypothesis, $M' \rightharpoonup^* (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1')$ and $P \equiv \mathtt{calc}((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1'))$. Then, by Proposition 25.2, $M \rightharpoonup^* (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid M''$ and $P \mid R = \mathtt{calc}((\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid M'')$ follows by definition of `calc`. For scope we have $\mathtt{calc}(M) = (u)Q$ and $Q \xrightarrow{(\widetilde{z})\overline{x}\,\widetilde{v}} P$. There are two cases: (a) when $u \notin \widetilde{v} \setminus \widetilde{z}x$, and (b) when $u \in \widetilde{v} \setminus \widetilde{z}x$. We discuss subcase (b), the subcase (a) is simpler. By definition of `calc`, either (b1) $M = (\!|u|\!)\,[Q'] \mid M'$ or (b2) $M = x[(u)Q]$. We discuss (b2). In this case, $M \rightharpoonup (u)(x'[Q] \mid u[\mathbf{0}])$ with a *(new)*. By inductive hypothesis $x'[Q] \rightharpoonup^* (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1')$ and $P \equiv \mathtt{calc}((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1'))$. Without loss of generality, let $u \notin \widetilde{y}$. By Proposition 25, $M \rightharpoonup^* (u)(\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1' \mid u[\mathbf{0}]) = (u\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{u\widetilde{z}} \mid M_1')$. By definition of `calc` and structural congruence: $\mathtt{calc}((\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{u\widetilde{z}} \mid M_1')) \equiv \mathtt{calc}((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1')) \equiv P$. The case when the last rule is replication may be reduced to the parallel case.

The theorem about the correctness of the encoding `calc` follows.

Theorem 27 (Machine correctness) *(1)* $M \equiv M'$ *implies* $\mathtt{calc}(M) \equiv \mathtt{calc}(M')$, $M \rightharpoonup M'$ *implies* $\mathtt{calc}(M) \equiv \mathtt{calc}(M')$, *and* $M \to M'$ *implies* $\mathtt{calc}(M) \to \mathtt{calc}(M')$.

(2) $\mathtt{calc}(M) \to P$ *implies there exists* N *such that* $M \Rightarrow N$ *and* $P \equiv \mathtt{calc}(N)$.

(3) $M \Downarrow x$ *if and only if* $\mathtt{calc}(M) \Downarrow x$.

(4) $M \mathrel{\dot{\approx}} N$ *if and only if* $\mathtt{calc}(M) \mathrel{\dot{\approx}} \mathtt{calc}(M')$.

(5) $M \cong_a M'$ *if and only if* $\mathtt{calc}(M) \cong_a^{Lc} \mathtt{calc}(M')$.

PROOF. Item 1 is obvious.

For Item 2, the reduction $\mathtt{calc}(M) \to P$ might have been deduced using structural congruence, which is difficult to match with structural congruence in the machine. Instead, we analyse the reduction via labelled transitions because, by Proposition 13, $\mathtt{calc}(M) \to P$ implies $\mathtt{calc}(M) \xrightarrow{\tau} P' \equiv P$, for some P'. Therefore it is enough to prove that $\mathtt{calc}(M) \xrightarrow{\tau} P'$. The argument is by induction on the proof of this transition. The basic case is when the last rule is a communication rule:

$$\frac{R \xrightarrow{(\widetilde{z})\overline{x}\,\widetilde{v}} R' \quad Q \xrightarrow{x(\widetilde{u})} Q' \quad \widetilde{z} \cap \mathtt{fn}(Q) = \emptyset}{\mathtt{calc}(M) = R \mid Q \xrightarrow{\tau} (\widetilde{z})(R' \mid Q'\{\widetilde{v}/\widetilde{u}\}) = P'}$$

We discuss the case when $Q \xrightarrow{x(\widetilde{u})} Q'$ is due to an input process; the case when the transition is due to a linear forwarder is similar. By Proposition 25, let M', M'' be such that $M = M' \mid M''$ and $\mathtt{calc}(M') = R$ and $\mathtt{calc}(M'') = Q$. By Lemma 26, $M' \rightharpoonup^* (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1')$ and $M'' \rightharpoonup^* (\widetilde{y'})(x[x(\widetilde{u}).Q''] \mid M_1'')$. Therefore, by assuming that names $\widetilde{u}$ and $\widetilde{z}$ do not clash with any other name:

$$\begin{aligned}
M \rightharpoonup^* & (\widetilde{z}\widetilde{y})(x'[\overline{x}\,\widetilde{v}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid (\widetilde{y'})(x[x(\widetilde{u}).Q''] \mid M_1'') && \text{by Proposition 25}\\
\rightharpoonup & (\widetilde{z})((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid (\widetilde{y'})(x[\overline{x}\,\widetilde{v} \mid x(\widetilde{u}).Q''] \mid M_1'')) && \text{by } \textit{(d.out)}\\
\rightarrow & (\widetilde{z})((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid (\widetilde{y'})(x[Q''\{\widetilde{v}/\widetilde{u}\}] \mid M_1'')) && \text{by } \textit{(react)}\\
= & (\widetilde{z})((\widetilde{y})(x'[\mathbf{0}] \mid \mathbf{0}_{\widetilde{z}} \mid M_1') \mid (\widetilde{y'})(x[Q''] \mid M_1'')\{\widetilde{v}/\widetilde{u}\})
\end{aligned}$$

The last machine, say N, is equal to $(\widetilde{z})(N' \mid N''\{\widetilde{v}/\widetilde{u}\})$ and, by Lemma 26, $\mathtt{calc}(N) \equiv (\widetilde{z})(R' \mid Q'\{\widetilde{v}/\widetilde{u}\})$. This concludes the proof of the basic case.

For the inductive step, the rules of interest are the parallel composition, the scope, and the replication. The first two follow by Proposition 25. If the last rule is

$$\frac{P \mid !P \xrightarrow{\tau} P'}{!P \xrightarrow{\tau} P'}$$

and $\mathtt{calc}(M) = !P$, then we observe that there is $M' \equiv M$ such that $\mathtt{calc}(M') = P \mid !P$. Therefore we reduce to reason on M', thus concluding the proof.

The result about barbs (Item 3) follows directly from Items 1 and 2.

The bisimulation result (Item 4) follows by Items 1 , 2, and 3.

For full abstraction (Item 5), by Lemma 8, we may restrict the definition of $\cong_a^{Lc}$ to parallel contexts. The reverse direction of item 5 is straightforward, because machine contexts are partial parallel compositions while calculus contexts allow arbitrary parallel composition. In the forward direction we must show that the partial is as discriminating as the arbitrary. Suppose there ex-

ists a AL$\ell\pi$ context R such that $R|\,\mathtt{calc}(M) \dot{\not\approx} R|\,\mathtt{calc}(M')$. Let $\widetilde{z}$ be the set of free names in R, and suppose without loss of generality that $\widetilde{z}$ does not clash with $\mathtt{lchan}(M)$ or $\mathtt{lchan}(M')$. Now define $E_R = u[[\![R]\!]_{\widetilde{z}}]$ for some fresh u. Clearly E_R, M and E_R, M' are well-formed. We may derive the following relations:

$$\begin{array}{rcll}
R \mid \mathtt{calc}(M) & \dot{\not\approx} & R \mid \mathtt{calc}(M') & \text{by assumption} \\
\cong_a^{Lc} & & \cong_a^{Lc} & \text{by Corollary 17} \\
[\![R]\!]_{\widetilde{z}} \mid \mathtt{calc}(M) & \dot{\not\approx} & [\![R]\!]_{\widetilde{z}} \mid \mathtt{calc}(M') & \\
\equiv & & \equiv & \text{since there are no clashes} \\
(\mathtt{lchan}(M))([\![R]\!]_{\widetilde{z}} \mid \widehat{M}) & \dot{\not\approx} & (\mathtt{lchan}(M'))([\![R]\!]_{\widetilde{z}} \mid \widehat{M'}) &
\end{array}$$

Hence, by Item 4, $E_R, M \dot{\not\approx} E_R, M'$.

We observe that from Theorems 18 and 27 one cannot derive a correspondence between a Aπ process and its implementation (i.e. $M \cong_a^{Lc} \mathtt{calc}(M)$, or replacing $\cong_a^{Lc}$ with $\cong_a$). The reason is because the two formalisms are different. However our results relate the formalisms in a very strong way. Practically, their strength means that a program can be debugged purely at source-level rather than at machine-level. This means that the linear forwarder machine is a natural implementation of Aπ, in contrast to other ones, such as Join. Of course, one might establish similar results with weaker semantics, such as *coupled simulation* [32], thus obtaining weaker relationships between source processes and their implementations.

5 The encoding of distributed choice

In this section we study another standard operator in Aπ, the *input-guarded choice*, written $x(\widetilde{u}).P + y(\widetilde{v}).Q$. This process can either react on x and so discard the y branch, or *vice versa*. It is well known that choice is problematic to implement in a distributed setting because the two channel managers x and y might be remote. In this case, the choice becomes a problem of global consensus.

Nestmann and Pierce [29] have encoded the input-guarded choice in the choice-free Aπ. Their encoding used booleans `t` and `f` and the conditional statement `if` b `then` P' `else` P''. For example, the process $x(\widetilde{u}).P + y(\widetilde{v}).Q$ is translated

into:

$$(\ell)\Big(\overline{\ell}\,\mathtt{t} \;\mid x(u).\ell(w).(\overline{\ell}\,\mathtt{f} \mid \mathtt{if}\ w\ \mathtt{then}\ P\ \mathtt{else}\ \overline{x}\,u) \\ \mid y(v).\ell(w).(\overline{\ell}\,\mathtt{f} \mid \mathtt{if}\ w\ \mathtt{then}\ Q\ \mathtt{else}\ \overline{y}\,v)\Big)$$

They demonstrated for such encoding a full abstraction result weaker than Theorem 27; the correspondence uses coupled simulation [30] rather than barbed congruence. (Nestmann and Pierce also gave a similar encoding without using booleans and conditionals.)

We design a different encoding of input-guarded choice using linear forwarders. Our encoding is amenable to a distributed implementation because it reduces the choice between remote channels to a choice between co-located channels. We demonstrate a full-abstraction result similar to Nestmann and Pierce's result [29]. We begin by defining the extension of AL$\ell\pi$ with input-guarded choice.

Definition 28 (Calculi with choice) *The calculi* AL$\ell\pi$ *(Definition 1),* A$\ell\pi$ *(Definition 11), and* Aπ *(Definition 6) are augmented with* choice *as follows* [2] *:*

$$P \;::=\; \ldots \;\;\Big|\;\; x(\widetilde{u}).P + y(\widetilde{v}).Q$$

along with the structural congruence law $P{+}Q \equiv Q{+}P$ *and the reduction rule*

$$\overline{x}\,\widetilde{w} \mid x(\widetilde{u}).P + y(\widetilde{v}).Q \;\;\rightarrow\;\; P\{\widetilde{w}/\widetilde{u}\}$$

The AL$\ell\pi$ *calculus with choice has an extended no-input-capability constraint that, in* $x(\widetilde{u}).P$ *occurring alone or as part of a choice, the* P *has no free occurrence of* $v \in \widetilde{u}$ *as the subject of an input.*

Labelled transition semantics of AL$\ell\pi$ *are as in Definition 12 but with two additional rules:*

$$x(\widetilde{u}).P{+}y(\widetilde{v}).Q \xrightarrow{x(\widetilde{u})} P \qquad\qquad x(\widetilde{u}).P{+}y(\widetilde{v}).Q \xrightarrow{y(\widetilde{v})} Q$$

We extend the function $[\![\cdot]\!]$ of Definition 9 mapping Aπ processes to AL$\ell\pi$ processes to accout for choice, and we again demonstrate its correctness.

Definition 29 (Encoding choice) *The encoding* $[\![\cdot]\!]$ *is as in Definition 9 but*

[2] A multi-way choice operator $\sum u_i(\widetilde{x}_i).P_i$ is often studied, instead of the two-way choice used here. Multi-way choice is used theoretically because it allows an axiomatisation of bisimulation congruence in the replication-less calculus. We stick to the two-way choice for simplicity.

with the additional rule that $[\![x(\widetilde{u}).P + y(\widetilde{v}).Q]\!]_{\widetilde{z}} =$

$$\begin{array}{ll}
x(\widetilde{u}).[\![P]\!]_{\widetilde{uz}} + y(\widetilde{v}).[\![Q]\!]_{\widetilde{vz}} & \text{if } x \notin \widetilde{z},\ y \notin \widetilde{z} \\
(y')(y{\multimap}y' \mid x(\widetilde{u}).(y'{\multimap}y|[\![P]\!]_{\widetilde{uz}}) + y'(\widetilde{v}).[\![Q]\!]_{\widetilde{vz}}) & \text{if } x \notin \widetilde{z},\ y \in \widetilde{z} \\
(x')(x{\multimap}x' \mid x'(\widetilde{u}).[\![P]\!]_{\widetilde{uz}} + y(\widetilde{v}).(x'{\multimap}x|[\![Q]\!]_{\widetilde{vz}})) & \text{if } x \in \widetilde{z},\ y \notin \widetilde{z} \\
(x'y')(x{\multimap}x' \mid y{\multimap}y' \mid x'(\widetilde{u}).(y'{\multimap}y|[\![P]\!]_{\widetilde{uz}}) + y'(\widetilde{v}).(x'{\multimap}x|[\![Q]\!])_{\widetilde{vz}}) & \text{if } x \in \widetilde{z},\ y \in \widetilde{z}
\end{array}$$

where x' *and* y' *are fresh names and* $\widetilde{u}, \widetilde{v}$ *are disjoint from* $x, y, \widetilde{z}$.

This encoding is similar to Definition 9, but with the addition of *undo* agents $y'{\multimap}y$ and $x'{\multimap}x$. Their function is to undo the forwarder that is not taken, using $(x')(x{\multimap}x' \mid x'{\multimap}x) \approx_a \mathbf{0}$ [3].

The correctness of the encoding of distributed choice cannot be established in the same way as for the choice-free fragment. In particular, a basic property of Definition 9 – the Corollary 17 that $P \approx_a [\![P]\!]_{\widetilde{u}}$ – fails in this case because bisimulation is sensitive to the branching structure of processes (*cf. gradual commitment* in [29]). For example $x(u).P + y(v).Q \not\approx_a [\![x(u).P + y(v).Q]\!]_{xy}$ because $[\![x(u).P + y(v).Q]\!]_{xy} \xrightarrow{x(u)} (x'y')(\overline{x'}\,u \mid y{\multimap}y' \mid x'(u).(y'{\multimap}y|[\![P]\!]_{xyu}) + y'(v).(x'{\multimap}x|[\![Q]\!])_{xyv})$ can be mimicked by $x(u).P + y(v).Q \xrightarrow{x(u)} P$, but the processes P and $(x'y')(\overline{x'}\,u \mid y{\multimap}y' \mid x'(u).(y'{\multimap}y|[\![P]\!]_{xyu}) + y'(v).(x'{\multimap}x|[\![Q]\!])_{xyv})$ are not bisimilar because the latter has not yet committed to the choice. The asynchronous bisimulation might match this last process with $\overline{x}\,u \mid x(u).P + y(v).Q$. But this match also fails because $\overline{x}\,u \mid x(u).P + y(v).Q \xrightarrow{x(u')} \overline{x}\,u \mid P\{u'/u\}$, while $(x'y')(\overline{x'}\,u \mid y{\multimap}y' \mid x'(u).(y'{\multimap}y|[\![P]\!]_{xyu}) + y'(v).(x'{\multimap}x|[\![Q]\!])_{xyv})$ cannot mimick this transition.

To solve a similar problem, Nestmann and Pierce used a coarser semantics – the *coupled simulation* [29]. We adapt this notion to our case by using asynchronous simulation as defined in Definition 14.

Definition 30 *A* coupled simulation *is a pair* $(\mathcal{R}, \mathcal{S})$ *of relations such that* $\mathcal{R}$ *and* $\mathcal{S}^{-1}$ *are asynchronous simulations and*

(1) $P\,\mathcal{R}\,Q$ *implies there is* Q' *such that* $Q \stackrel{\tau}{\Longrightarrow} Q'$ *and* $P\,\mathcal{S}\,Q'$*;*
(2) $P\,\mathcal{S}\,Q$ *implies there is* P' *such that* $P \stackrel{\tau}{\Longrightarrow} P'$ *and* $P'\,\mathcal{R}\,Q$*.*

Coupled similarity*, written* $\rightleftarrows_a$*, is the largest coupled simulation.*

Coupled similarity retains useful properties that we recall from [29].

Theorem 31 *(1)* $\rightleftarrows_a$ *is an equivalence relation;*
(2) $\rightleftarrows_a$ *is a congruence in* A$\ell\pi$ *(and therefore in* Aπ *and in* AL$\ell\pi$*);*
(3) $\approx_a \subseteq \rightleftarrows_a$*.*

[3] Remark 10 also applies for this extension.

Next we may establish the analogous statement to Corollary 17, this time for coupled simulation.

Lemma 32 (Operational correspondence for choice) *$P \rightleftarrows_a [\![P]\!]_{\widetilde{u}}$, for all $\widetilde{u}$.*

PROOF. To simplify the proof, we introduce some notation. Let $[\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'} \stackrel{\text{def}}{=} x'(\widetilde{v}).(y'{\multimap}y \mid [\![P]\!]_{\widetilde{uv}}) + y'(\widetilde{w}).(x'{\multimap}x \mid [\![Q]\!]_{\widetilde{uw}})$ and let $[\![P,Q]\!]_{\widetilde{u}}^{x,x',y} \stackrel{\text{def}}{=} x'(\widetilde{v}).[\![P]\!]_{\widetilde{uv}} + y(\widetilde{w}).(x'{\multimap}x \mid [\![Q]\!]_{\widetilde{uw}})$, where the names x', y', $\widetilde{v}$, and $\widetilde{w}$ are disjoint from the names in $\widetilde{u}$.

Consider the smallest pair of relations $(\mathcal{R},\mathcal{S})$ such that both $\mathcal{R}$ and $\mathcal{S}$ contain structural congruence and are closed under the following conditions:

- $$\begin{array}{rcl} P & \mathcal{R} & [\![P]\!]_{\widetilde{u}} \\ P\{\widetilde{v'}/\widetilde{v}\} & \mathcal{R} & (x',y')(\overline{x'}\,\widetilde{v'} \mid y{\multimap}y' \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'}) \\ \overline{y}\,\widetilde{w'} \mid P\{\widetilde{v'}/\widetilde{v}\} & \mathcal{R} & (x',y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'}) \\ P\{\widetilde{v'}/\widetilde{v}\} & \mathcal{R} & (x')(\overline{x'}\,\widetilde{v'} \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y}) \end{array}$$

  where x', y' are fresh. Additionally, if $P\,\mathcal{R}\,Q$, $P'\,\mathcal{R}\,Q'$, and $x' \notin \mathtt{fn}(Q)$ then

  $$P\;\mathcal{R}\;(x')(x{\multimap}x' \mid x'{\multimap}x \mid Q) \quad \text{and} \quad P|P'\;\mathcal{R}\;Q|Q' \quad \text{and} \quad (x)P\;\mathcal{R}\;(x)Q$$

- $$\begin{array}{rcl} P & \mathcal{S} & [\![P]\!]_{\widetilde{u}} \\ \overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q & \mathcal{S} & (x',y')(\overline{x'}\,\widetilde{v'} \mid y{\multimap}y' \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'}) \\ \overline{x}\,\widetilde{v'} \mid \overline{y}\,\widetilde{w'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q & \mathcal{S} & (x',y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'}) \\ \overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q & \mathcal{S} & (x')(\overline{x'}\,\widetilde{v'} \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y}) \end{array}$$

  where x', y' are fresh. Additionally, if $P\,\mathcal{S}\,Q$, $P'\,\mathcal{S}\,Q'$, and $x' \notin \mathtt{fn}(Q)$ then

  $$P\;\mathcal{S}\;(x')(x{\multimap}x' \mid x'{\multimap}x \mid Q) \quad \text{and} \quad P|P'\;\mathcal{S}\;Q|Q' \quad \text{and} \quad (x)P\;\mathcal{S}\;(x)Q$$

It is worth observing that $\mathcal{R}$ also pairs the processes

– $Q\{\widetilde{v'}/\widetilde{v}\}$ and $(x',y')(\overline{y'}\,\widetilde{v'} \mid x{\multimap}x' \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'})$;
– $\overline{x}\,\widetilde{w'} \mid Q\{\widetilde{v'}/\widetilde{v}\}$ and $(x',y')(\overline{y'}\,\widetilde{v'} \mid \overline{x'}\,\widetilde{w'} \mid [\![P,Q]\!]_{\widetilde{u}}^{x,x',y,y'})$

because it includes structural congruence. Similarly for $\mathcal{S}$.

The proof that $\mathcal{R}$ and $\mathcal{S}^{-1}$ are asynchronous simulations is entirely standard. We therefore focus on the proof that $(\mathcal{R}, \mathcal{S})$ is a coupled simulation. Since $\mathcal{R}$ and $\mathcal{S}$ are inductively defined, the argument is by induction on the number of rules used for deriving $P\mathcal{R}Q$ and $P\mathcal{S}Q$.

- relation $\mathcal{R}$:

(1) P is $\mathcal{R}$-paired to Q since $P \equiv Q$. This follows by $\equiv \subseteq \rightleftarrows_a$ by using Theorem 15.2 and Theorem 31.3.

(2) P is $\mathcal{R}$-paired to $[\![P]\!]_{\widetilde{u}}$. Immediate because these two processes are also $\mathcal{S}$-paired.

(3) $P\{\widetilde{v'}/\widetilde{v}\}$ is $\mathcal{R}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid y \multimap y' \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{uv}})$. There is a transition $(x', y')(\overline{x'}\,\widetilde{v'} \mid y \multimap y' \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{uv}}) \xrightarrow{\tau} (x', y')(y \multimap y' \mid y' \multimap y \mid [\![P]\!]_{\widetilde{uv}}\{\widetilde{v'}/\widetilde{v}\}) \equiv (y')(y \multimap y' \mid y' \multimap y \mid [\![P\{\widetilde{v'}/\widetilde{v}\}]\!]_{\widetilde{uv'}})$ since $x' \notin \mathtt{fn}([\![P\{\widetilde{v'}/\widetilde{v}\}]\!]_{\widetilde{uv'}})$ and $\widetilde{v}$ are disjoint from $\widetilde{u}$. This last process is $\mathcal{S}$-paired with $P\{\widetilde{v'}/\widetilde{v}\}$.

(4) $\overline{y}\,\widetilde{w'} \mid P\{\widetilde{v'}/\widetilde{v}\}$ is $\mathcal{R}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}})$. There is a derivation $(x', y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}}) \xrightarrow{\tau} (x', y')(\overline{y'}\,\widetilde{w'} \mid [\![P]\!]_{\widetilde{uv}}\{\widetilde{v'}/\widetilde{v}\} \mid y' \multimap y) \xrightarrow{\tau} (x', y')(\overline{y}\,\widetilde{w'} \mid [\![P]\!]_{\widetilde{uv}}\{\widetilde{v'}/\widetilde{v}\}) \equiv \overline{y}\,\widetilde{w'} \mid [\![P\{\widetilde{v'}/\widetilde{v}\}]\!]_{\widetilde{uv'}}$. This last process is $\mathcal{S}$-paired with $\overline{y}\,\widetilde{w'} \mid P\{\widetilde{v'}/\widetilde{v}\}$ because of parallel closure.

(5) $P\{\widetilde{v'}/\widetilde{v}\}$ is $\mathcal{R}$-paired with $(x')(\overline{x'}\,\widetilde{v'} \mid [\![P, Q]\!]^{x,x',y}_{\widetilde{u}})$. Similar to 3.

(6) The inductive cases (addition of a forward and backward linear forwarder, parallel closure, and restriction closure) follow by the inductive hypotheses.

- relation $\mathcal{S}$:

(1) P is $\mathcal{S}$-paired to Q because $P \equiv Q$. As for $\mathcal{R}$, this case follows since $\equiv \subseteq \rightleftarrows_a$.

(2) P is $\mathcal{S}$-paired to $[\![P]\!]_{\widetilde{u}}$. Immediate because these two processes are also $\mathcal{R}$-paired.

(3) $\overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q$ is $\mathcal{S}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid y \multimap y' \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}})$. The coupling condition follows by $\overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q \xrightarrow{\tau} P\{\widetilde{v'}/\widetilde{v}\}$ and the process $P\{\widetilde{v'}/\widetilde{v}\}$ is $\mathcal{R}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid y \multimap y' \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}})$.

(4) $\overline{x}\,\widetilde{v'} \mid \overline{y}\,\widetilde{w'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q$ is $\mathcal{S}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}})$. The coupling condition follows in two ways: either (a) $\overline{x}\,\widetilde{v'} \mid \overline{y}\,\widetilde{w'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q \xrightarrow{\tau} \overline{y}\,\widetilde{w'} \mid P\{\widetilde{v'}/\widetilde{v}\}$ or (b) $\overline{x}\,\widetilde{v'} \mid \overline{y}\,\widetilde{w'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q \xrightarrow{\tau} \overline{x}\,\widetilde{v'} \mid Q\{\widetilde{w'}/\widetilde{w}\}$. The final processes are both $\mathcal{R}$-paired with $(x', y')(\overline{x'}\,\widetilde{v'} \mid \overline{y'}\,\widetilde{w'} \mid [\![P, Q]\!]^{x,x',y,y'}_{\widetilde{u}})$.

(5) $\overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q$ is $\mathcal{S}$-paired with $(x')(\overline{x'}\,\widetilde{v'} \mid [\![P, Q]\!]^{x,x',y}_{\widetilde{u}})$. There is a transition $\overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P + y(\widetilde{w}).Q \xrightarrow{\tau} P\{\widetilde{v'}/\widetilde{v}\}$ and this last process is $\mathcal{R}$-paired with $(x')(\overline{x'}\,\widetilde{v'} \mid [\![P, Q]\!]^{x,x',y}_{\widetilde{u}})$.

(6) The inductive cases are immediate consequences of the inductive hypotheses.

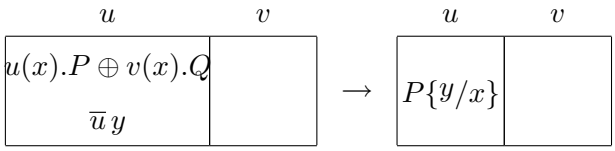
Our full abstraction result for the encoding of choice is an immediate consequence of Lemma 32 and the transitivity of $\rightleftarrows_a$.

Theorem 33 *For P and Q in* Aπ*,*

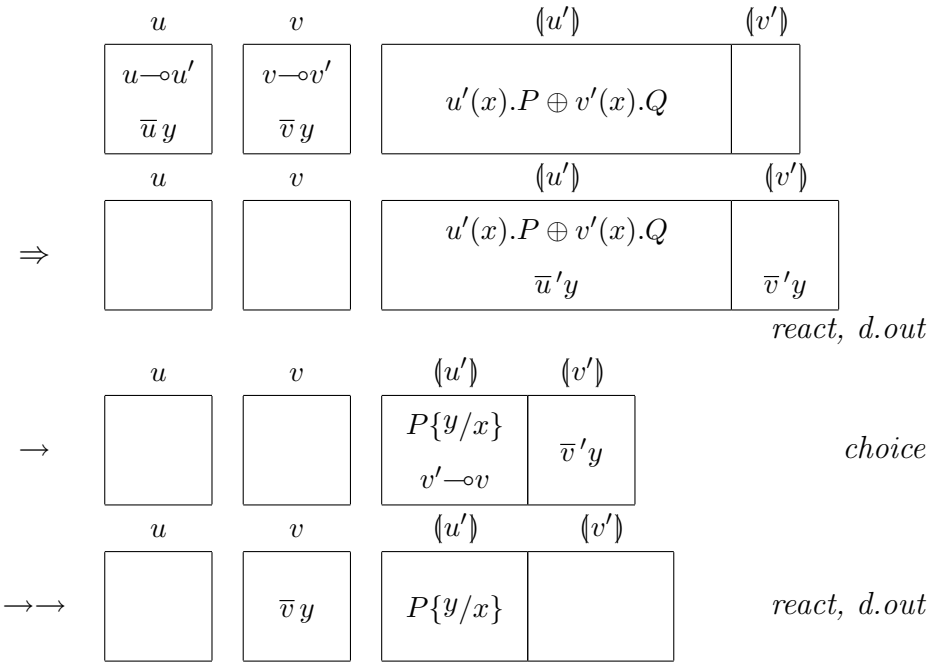
(1) $P \rightleftarrows_a [\![P]\!]$.
(2) $P \rightleftarrows_a Q$ if and only if $[\![P]\!] \rightleftarrows_a [\![Q]\!]$.

5.1 Choice in the machine

Our machine will only implement local choice, $x(\widetilde{u}).P + y(\widetilde{v}).Q$ where x and y are co-located. The low-level implementation is easy: a single thread can manage both x and y and so it chooses one branch atomically; or two threads could manage x and y but the choice is protected by a mutex. The machine reduction rule is



The following diagrams show the execution of $[\![u(x).P + v(x).Q]\!]$ on the machine:



We now give a formal treatment of choice in the machine.

Definition 34 *The machine with choice is as before (Definition 19), but where terms P now come from the* AL$\ell\pi$ *calculus with choice (Definition 28).*

The machine with choice has the same locality constraint as before: in $x[P]$ and $(\!|x|\!)\,[P]$, then every free input $y(\widetilde{v}).Q$ in P satisfies $x@y$. But this constraint now also applies to inputs that are parts of summations.

The machine dynamics are as before (Definition 20), plus two additional reaction rules:

$$\begin{array}{rcl} x[y(\widetilde{u}).P{+}z(\widetilde{v}).R \mid S],\ y[\overline{y}\,\widetilde{w} \mid T] & \rightarrow & x[S],\ y[P\{\widetilde{w}/\widetilde{u}\} \mid T] \quad \textit{if } x@y \quad (\textit{choice-1}) \\ x[x(\widetilde{u}).P{+}z(\widetilde{v}).R \mid \overline{x}\,\widetilde{w} \mid S] & \rightarrow & x[P\{\widetilde{w}/\widetilde{u}\} \mid S] \qquad\qquad\qquad\quad (\textit{choice-2}) \end{array}$$

Rule (*choice-1*) reduces a machine even if the subject y of the chosen branch is different from the channel manager x performing the reduction. In this case we constrain the two channels to be co-located. This seems at odds with (*d-in*) that allowed migration of input processes on co-located machines by means of an heating step. We prefer the reduction rule in order to avoid divergency of heating steps. The reduction (*choice-2*) is similar to the choice rule in the calculus.

We remark that all the machine operations preserve locality. Now we have to prove again the bisimulation and full abstraction results, analogous to those in Theorem 27. The new work is in re-proving the deduction from $\mathtt{calc}(M) \rightarrow P'$ to $M \Rightarrow M'$ and $P' \equiv \mathtt{calc}(M')$. If the calculus reaction involves (choice-1), we rely on the well-formedness of M to satisfy the (choice-1) side condition that $x@y$.

Proposition 35 *(1) $M \stackrel{\bullet}{\approx} M'$ if and only if $\mathtt{calc}(M) \stackrel{\bullet}{\approx} \mathtt{calc}(M')$.*
(2) $M \cong_a M'$ if and only if $\mathtt{calc}(M) \cong_a^{Lc} \mathtt{calc}(M')$.

PROOF. The proofs are substantially the same to those given earlier (from Proposition 25 to Theorem 27). The only difference is in the analogous result to Lemma 26.1 which, as well as the two cases given previously, requires the additional case:

- $M \multimap^* (\widetilde{y})(M',\ z[x(\widetilde{u}).Q{+}y(\widetilde{v}).Q' \mid R],\ M'')$, $z@x$ and $P \equiv \mathtt{calc}((\widetilde{y})(M',\ x[Q \mid R],\ M''))$.

Apart from this difference, the proof carries over unmodified from Theorem 27.

6 Failures

In this section we study two extensions of the linear forwarder machine to incorporate failure information. The first extension considers failures that are due to message losses or to software exceptions that deviate the normal flow of

execution. The timescale of such failures is milliseconds. The second extension considers failures due to crashes of locations. This model, whose timescale is between minutes and hours, is similar to those discussed in [13,33,3]. For each model of failure, we establish correctness results between the fallible versions of the pi calculus, of the linear forwarder calculus and the machine.

6.1 Message loss failures

When failures such as message losses occur there are a number of problems (for instance, consensus) that cannot be solved in an asynchronous framework as AL$\ell\pi$. In order to avoid these limitations, a calculus should ultimately use timeout-based failure detectors. At the low-level (eg. TCP and UDP programming), these will have the form $x(\widetilde{u})^{\mathtt{t}}.P?Q$ and meaning: "if a message arrives on channel x within $\mathtt{t}$ milliseconds then continue with P; but if nothing arrives within a given timeout then execute Q". It is beyond the scope of the current contribution to deal with the *compensations* Q and with timed issues (the reader is referred to [5,22] for details). We therefore fix $\mathtt{t} = \infty$ and $Q = \mathbf{0}$ in what follows, and write $\alpha.P?\mathbf{0}$ as just $\alpha.P$.

Definition 36 *The calculus* FAπ *is as in Definitions 6 and 12, but with the additional reactions:*

$$\overline{x}\,\widetilde{u} \xrightarrow{\tau} \mathbf{0} \qquad\qquad x(\widetilde{u}).P \xrightarrow{\tau} \mathbf{0}$$

The calculus FA$\ell\pi$ *is as in Definitions 11 and 12, but with these additional reactions* [4] *.*

$$x{\multimap}y \xrightarrow{\tau} \mathbf{0} \qquad\qquad \overline{x}\,\widetilde{u} \xrightarrow{\tau} \mathbf{0}$$

The fallible local linear forwarder machine is as in Definition 20 but with these additional reaction steps:

$$\begin{array}{rcll} x[z{\multimap}y \mid P] & \rightarrow & x[P] & \text{if } x \not@ z \qquad (f.fwd) \\ x[\overline{z}\,\widetilde{v} \mid P] & \rightarrow & x[P] & \text{if } x \not@ z \qquad (f.out) \end{array}$$

Note that, in the fallible machine, only forwarders and outputs can be deployed to remote locations. That is why we have added failure of forwarders *(f.fwd)* and failure of outputs *(f.out)*, but no rule for failure of input such as $x[z(\widetilde{y}).Q \mid P] \xrightarrow{\tau} x[P]$. Similarly, in the fallible version of A$\ell\pi$, failures only concern linear forwarders and outputs, which are the unique objects that are intended

[4] For simplicity, in this section and the next one, we discuss failures in the choice-free fragment of Aπ.

to move. However, *every* forwarder and output may fail in FA$\ell\pi$ since the calculus does not specify any location information.

As regards the relationship between FAπ and the fallible machine, consider the FAπ term $P = \overline{x}\,u \mid x(w).Q$. When translated into FA$\ell\pi$, the term $[\![P]\!]_x$ admits two reactions, one of which is

$$\overline{x}\,u \mid (x')(x \multimap x' | x'(w).[\![Q]\!]_{xw}) \quad \xrightarrow{\tau} \quad (x')(\overline{x}\,'u | x'(w).[\![Q]\!]_{xw}).$$

At this point, $[\![P]\!]_x$ has committed to consuming the $\overline{x}\,u$, but it remains in an intermediate state where it is open either to succeed and yield $[\![Q]\!]_{wx}\{^u/_w\}$, or to fail and yield a process that is $\approx_a$-equivalent to $\mathbf{0}$. Meanwhile, the original term P in FAπ admits no such intermediate state. This is an instance of a phenomenon similar to the one discussed in Section 5, namely that of an implementation having intermediate states even though they are not observable. As before, the behavioural equivalence we use is coupled simulation.

Theorem 37 *For P and Q in* FAπ*,*

(1) $P \rightleftarrows_a [\![P]\!]$.
(2) $P \rightleftarrows_a Q$ if and only if $[\![P]\!] \rightleftarrows_a [\![Q]\!]$.

PROOF. The proof is similar to the one of Theorem 33. We therefore focus on the proof of $P \rightleftarrows_a [\![P]\!]_{\widetilde{u}}$, for all $\widetilde{u}$, by providing a coupled simulation.

Let $(\mathcal{R}, \mathcal{S})$ be the smallest pair such that both $\mathcal{R}$ and $\mathcal{S}$ contain $\approx_a$ and are closed under the following $(x' \notin \mathtt{fn}(P))$.

- For every $\widetilde{u}$:

$$\begin{array}{rcl} P & \mathcal{R} & [\![P]\!]_{\widetilde{u}} \\ P\{\widetilde{v'}/\widetilde{v}\} & \mathcal{R} & (x')(\overline{x'}\,\widetilde{v'} \mid x'(\widetilde{v}).[\![P]\!]_{\widetilde{uv}}) \end{array}$$

  Additionally, if $P\,\mathcal{R}\,Q$ and $P'\,\mathcal{R}\,Q'$ then both $P \mid P' \;\mathcal{R}\; Q \mid Q'$ and $(x)P\,\mathcal{R}\,(x)Q$.
- For every $\widetilde{u}$:

$$\begin{array}{rcl} P & \mathcal{S} & [\![P]\!]_{\widetilde{u}} \\ \overline{x}\,\widetilde{v'} \mid x(\widetilde{v}).P & \mathcal{S} & (x')(\overline{x'}\,\widetilde{v'} \mid x'(\widetilde{v}).[\![P]\!]_{\widetilde{uv}}) \end{array}$$

  Additionally, if $P\,\mathcal{S}\,Q$ and $P'\,\mathcal{S}\,Q'$ then both $P \mid P' \;\mathcal{S}\; Q \mid Q'$ and $(x)P\,\mathcal{S}\,(x)Q$.

The proof that $(\mathcal{R}, \mathcal{S})$ is a coupled simulation is omitted because similar to that of Lemma 32.

Regarding fallible machines and their associated processes, we do not have an analogous result to Theorem 27, not even for coupled simulation. This is

because M manifests less failures than $\texttt{calc}(M)$ since the latter misses the co-location information, thus making every $x{-\!\circ}y$ and $\overline{x}\,\widetilde{u}$ fail. For example, if $y@x$, $(\!|x|\!)\,[\overline{x}\ \mid x(\ ).\overline{y}\,], \mathring{y}[\ ]$ never fails and always emits y, while $\texttt{calc}((\!|x|\!)\,[\overline{x}\ \mid x(\ ).\overline{y}\,], \mathring{y}[\ ])$ may fail without emitting anything.

It turns out that the relationship one may establish between a M and $\texttt{calc}(M)$ is a so-called *barbed simulation*, defined as in Definitions 4 and 22 without the constraint of symmetry. Let $P \leq Q$ if Q simulates P. The proof of the following statement is omitted because it is straightforward.

Proposition 38 $M \leq \texttt{calc}(M)$.

Remark 39 *It is worth remarking that Proposition 38 may be strenghtened because the machine implementation does not introduce deadlocks. Deadlock-avoidance has been demonstrated for the implementation in Pict and in join calculus of pi calculus [39,12].*

6.2 Location crash failures

The second model we analyze considers locations that may crash and be rebooted into some safe initial state. This style of failure is studied in detail in [3,33]. As in these papers, we assume that the rebooted state is the empty location and that channel managers located at crashed locations cannot be used anymore. The usual failure detector has the form $\texttt{ping}^{\texttt{t}}\ x.\,P?Q$ with the meaning: "if the location x is active (not crashed) then continue with P; but if nothing arrives within a given timeout $\texttt{t}$ then execute Q". For the sake of simplicity we we will not explicitly use any $\texttt{ping}$ command. Instead, in $\texttt{ping}^{\texttt{t}}\ x.\,P?Q$ we let $\texttt{t}$ be infinite, $Q = \mathbf{0}$ and shorten $\texttt{ping}^{\texttt{t}}\ x.\,P?\mathbf{0}$ into $\texttt{ping}\ x.\,P$. Additionally, the process $\texttt{ping}\ x.\,P$ is encoded by $(z)(\overline{\texttt{ping}_x}\,z \mid z.P)$, where $\texttt{ping}_x$ is a channel co-located with x whose behaviour is $!\texttt{ping}_x(z).\overline{z}\,$.

The machine in Definition 20 is extended with terms representing *crashed channel managers*:

$$M \quad ::= \qquad \cdots \qquad \mid \quad x\langle\mathbf{0}\rangle$$

The machine $x\langle\mathbf{0}\rangle$ represents a location that cannot perform any action. This means that every other machine co-located with x is crashed as well. The following semantics enforces this constraint. We write $\{x_1 \cdots x_n\}\langle\mathbf{0}\rangle$ for the (crashed) machine $x_1\langle\mathbf{0}\rangle, \cdots, x_n\langle\mathbf{0}\rangle$. We use function $\texttt{fchan}(M)$ to collect the names of the failed channel managers in M and extend the function $\texttt{chan}(M)$ to include both failed and not-failed channel managers; we have $\texttt{fchan}(M) \subseteq \texttt{chan}(M)$.

Definition 40 *The calculus* CFA$\ell\pi$ *is as in Definitions 11 and 12, but with*

these additional reactions.

$$x{\multimap}y \xrightarrow{\tau} \mathbf{0} \qquad \overline{x}\,\widetilde{u} \xrightarrow{\tau} \mathbf{0} \qquad x(\widetilde{u}).P \xrightarrow{\tau} \mathbf{0}$$

The machine with crash-failures *is as in Definition 20 but with the additional reaction steps:*

$$\frac{\textit{for every } x, y \in \mathtt{chan}(M)\,.\; x@y}{M \to \mathtt{chan}(M)\langle\mathbf{0}\rangle} \qquad (\textit{crash})$$

$$\frac{M \to M', \quad \textit{for every } x \in \mathtt{fchan}(M'), y \in \mathtt{chan}(N).\; x\not@y}{M, N \to M', N}$$

With respect to FA$\ell\pi$, the calculus CFA$\ell\pi$ also admits that an input process may fail. This is because the corresponding channel manager may unexpectedly crash: see the rule (*crash*) of the machine with crash-failures. Rule (*crash*) constraints co-located channel managers to crash simultaneously. In fact, the inference rule has a premise verifying that failed channel managers in M' are not co-located with machines in the environment.

We extend $\mathtt{calc}(\cdot)$ as follows:

$$\mathtt{calc}(x\langle\mathbf{0}\rangle) \;=\; \mathbf{0}$$

By definition of CFA$\ell\pi$, the statements corresponding to Theorem 37 also hold for this calculus. As regards the correspondence between the machine with crash failures and CFA$\ell\pi$, we observe that, in contrast to the previous failure model, M manifests *more* failures than $\mathtt{calc}(M)$ because the co-location information of the formers causes failures of a group of processes at once. For instance, take $M = x[x(u).P, x(v).Q]$ then $\mathtt{calc}(M) \xrightarrow{\tau} x(v).Q$ but there is no M' yielded from M such that $\mathtt{calc}(M') = x(v).Q$. However $x(v).Q \xrightarrow{\tau} \mathbf{0}$ and $M \to x\langle\mathbf{0}\rangle$ with $\mathtt{calc}(x\langle\mathbf{0}\rangle) = \mathbf{0}$. Henceforth, Proposition 38 may be also established for location crash failures.

7 Loadings of program codes

Due to the localisation constraint in AL$\ell\pi$, loading a calculus program onto a machine is not straightforward. Let us briefly discuss this topic. A given program typically can be loaded in several ways: for instance, $\overline{x}\,u$ may be loaded onto a machine either at x or at u (or indeed at any existing channel manager). If all the channels are co-located – said otherwise, the machine is a multiprocessor – then every loading of a AL$\ell\pi$ program is satisfactory and they are all equivalent (by rule (*d.in*)). However, when the machine is distributed,

there are AL$\ell\pi$ programs that *cannot* be loaded directly onto it. This is the case when the program has nested free inputs but they are not co-located. Following Remark 10 it is straightforward to use the encoding $[\![\cdot]\!]$ to avoid such nested free inputs, so making every calculus term loadable. For instance, if $\widetilde{z} = \mathtt{fn}(P)$, then $[\![P]\!]_{\widetilde{z}}$ is loadable.

Alternatively, one may achieve the same result by replacing *on-the-fly* the input on a remote name with an input on a new co-located name and a linear forwarder:

$$x[z(\widetilde{u}).P \mid Q],\ z[R] \ \rightarrow\ x[Q],\ (\!(x')\!)\,[x'(\widetilde{u}).P],\ z[z{\multimap}x' \mid R] \qquad \textit{if}\ \ x' \textit{ fresh, } x@x',\ x{\not@}z \qquad (d.din)$$

We actually use the on-the-fly solution in practice in `PiDuce` [9] since it avoids the need for the encoding. The theory developed in this paper sustains the correctness of this solution.

8 Conclusions

In this paper we have implemented the asynchronous pi calculus, and in particular its problematic feature of input capability on a distributed machine. We have shown that a basic form of input capability, the linear forwarders, is sufficient to express the general form. We have demonstrated this by introducing a calculus with linear forwarders, providing a correct encoding of the asynchronous pi calculus in the linear forwarder calculus, and presenting a distributed abstract machine with a correct implementation of the linear forwarder calculus. Distributed choice and failures have also been studied.

Laneve now has a large project in Bologna which provides a distributed imlementation of `PiDuce` [9], a language that combines pi calculus processes and `XML` datatypes, whose implementation relies on linear forwarders. This machine, as in the Join Calculus Machine, groups processes at their channels (or locations). In contrast with the Join Calculus Machine that does not support messages carrying channels with input capability, the `PiDuce` Machine supports input capability using linear forwarders as illustrated in this paper. The ease of implementation of linear forwarders is demonstrated in the `PiDuce` Machine: the programs managing channels and linear forwarders are implemented by a code of one thousand lines in `C#`.

It still remains for us to fully assess the practical significance of linear forwarders. One promising application is the so-called orchestration and choreography languages in Web services (`WS-BPEL`, `WSCDL`, `BizTalk`, etc.). These

languages use input capability to allow services to change dynamically the behaviour of other services, or to design new services out of existing ones. For example the service

$$w(x, y, z).x(u).y(v).\overline{z}\, uv$$

takes three services addresses (URI) x, y, and z, then catches one message to x and to y, and finally packages the two messages and send them to z. The theory of these simple patterns have been recently studied in [21]. It tuns out that, combining linear forwarders and join-patterns, it is possible to describe several workflow patterns in Web services. The next aim is to extend these ideas to develop an expressive calculus for Web services, where every primitive is easily amenable to a distributed implementation.

Acknowledgments. We thank the anonymous referees for providing relevant comments on the submitted draft. In particular, one of them pointed out a significant flaw in a previous version of Lemma 32.

References

[1] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Métayer, editors, *COORDINATION 1997*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer-Verlag, 1997.

[2] Roberto Amadio, Gérard Boudol, and Cédric Lhoussaine. The receptive distributed pi-calculus (extended abstract). In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *FSTTCS 1999*, volume 1738 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999.

[3] Roberto Amadio and Sanjiva Prasad. Localities and failures (extended abstract). In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer-Verlag, 1994.

[4] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[5] Martin Berger. Basic theory of reduction congruence for two timed asynchronous π-calculi. In *CONCUR '04: Proceedings of the 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2004.

[6] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.

[7] A. Brown, C. Laneve, and L.G. Meredith. `PiDuce`: A process calculus with native `xml` datatypes. In *Proceedings of 2nd International Workshop on Web Services and Formal Methods*, volume 3670 of *LNCS*, pages 18–34. Springer-Verlag, 2005.

[8] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[9] Samuele Carpineti, Cosimo Laneve, and Paolo Milazzo. The `bopi` machine: a distributed machine for experimenting web services technologies. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, pages 202–211. IEEE Computer Society Press, 2005.

[10] Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, pages 22–29. IEEE Computer Society Press, October, 1999.

[11] Rocco De Nicola, Gian Luigi Ferrari, and Pugliese Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.

[12] Cédric Fournet. *The Join-Calculus: A Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Paris, France, 1998.

[13] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL 1996*, pages 372–385. ACM, ACM Press, 1996.

[14] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine. In M. Kretinsky A. Kucera L. Brim, P. Jancar, editor, *CONCUR 2002*, volume 2421 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2002.

[15] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear forwarders. In D. Lugiez R. Amadio, editor, *CONCUR 2002*, volume 2761 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 2003.

[16] Philippa Gardner and Sergio Maffeis. Modelling dynamic Web data. In Georg Lausen and Dan Suciu, editors, *Proc. of DBPL'03*, volume 2921 of *LNCS*, pages 130–146, September 2003.

[17] Philippa Gardner and Lucian Wischik. Explicit fusions. In Mogens Nielsen and Branislav Rovan, editors, *MFCS 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2000.

[18] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[19] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[20] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[21] Cosimo Laneve and Luca Padovani. Smooth orchestrators. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS 2006)*, volume 3921 of *LNCS*, pages 32–46. Springer-Verlag, 2006.

[22] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FOSSACS 2005: Proceedings of Foundations of Software Science and Computation Structure*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer Verlag, 2005.

[23] Sergio Maffeis. Dynamic web data: A process-algebraic approach. PhD thesis, Imperial College London, 2006.

[24] Massimo Merro. On equators in asynchronous name-passing calculi without matching. In *EXPRESS 1999*, volume 27 of *Electronic Notes in Theoretical Computer Science*, pages 57–70. Elsevier Science Publishers, 1999.

[25] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.

[26] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[27] Robin Milner. The polyadic pi calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification, NATO 1991*, volume 94 of *Series F*. Springer-Verlag, 1993.

[28] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *ICALP 1992*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[29] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Information and Computation*, 163(1):1–59, 2000.

[30] Joachim Parrow and Peter Sjödin. Multiway synchronization verified with coupled simulation. In W. R. Cleaveland, editor, *CONCUR 1992*, volume 630 of *Lecture Notes in Computer Science*, pages 518–533. Springer-Verlag, 1992.

[31] A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of the European Symposium on Programming (ESOP 2004)*, LNCS. Springer-Verlag, April 2004.

[32] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[33] James Riely and Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226:693–735, 2001.

[34] Arnaud Sahuguet and Val Tannen. Resource Sharing Through Query Process Migration. University of Pennsylvania Technical Report MS-CIS-01-10, 2001.

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

[36] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[37] Davide Sangiorgi and David Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[38] Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location independence for mobile agents. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *ICCL 1998*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer-Verlag, 1999.

[39] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[40] URL. Microsoft biztalk server. `www.microsoft.com/biztalk`.