Pict Correctness Revisited *

Philippe Bidinger[1] and Adriana Compagnoni[2]

[1] VERIMAG, Grenoble, France
[2] Stevens Institute of Technology, Hoboken, NJ, USA

Abstract. The Pict programming language is an implementation of the π-calculus in which executions of π-calculus terms are specified via an abstract machine. An important property of any concurrent programming language implementation is the fair execution of threads. After defining fairness for the π-calculus, we show that Pict abstract machine executions implement fair π-calculus executions. We also give new proofs of soundness and liveness for the Pict abstract machine.

1 Introduction

The π-calculus [14, 17] is a minimal language designed to capture and model key concepts of communicating concurrent systems in a formal setting. It emphasizes channel-based communication, dynamic channel creation and the ability to communicate channels as data. Pict [19, 16] is a high-level programming language purely based on π-calculus primitives, as well as to explore the applicability of theoretical work on type systems. Pict's runtime environment is based on a formal abstract machine specification, but little emphasis has been placed on its correctness.

The correctness of a programming language runtime is critical since, in order to be able to reason about programs, we need the guarantee that programs are executed according to their semantics. Correctness results of implementations usually relate executions of terms in a high-level language to its implementation in a low-level language. The low-level language can be an existing process calculus, or like in Pict, an abstract machine specification.

In recent years, many process calculi based on the π-calculus have been introduced to study the dynamics of existing or new paradigms of computation, such as distributed computing, global computing, or component-based programming. Much work has been done on the distributed implementation of these calculi [5, 7, 22, 20, 10, 8, 11, 12, 1]. On the other hand, since the definition of Pict, there has been no new insight for the local implementation of these calculi. Therefore, Pict is still a reference implementation of the π-calculus, and we think that proving its correctness is a first step toward more general proofs of correctness of implementations of these calculi.

The π-calculus is a concurrent language where concurrency is modeled using a non-deterministic reduction relation. The Pict Abstract Machine (PAM)

* This work was funded in part by the US Army under contract W15QKN-05-D-0011.

implements a particular scheduling strategy that corresponds to a subset of the possible executions in the π-calculus. It is therefore impossible to state an exact correspondence between π-calculus executions and PAM executions. Instead, we will prove the correctness of the abstract machine with three properties:

- A soundness property that states that PAM executions correspond to valid π-calculus executions.
- A liveness property that ensures that the abstract machine is not stuck when its state corresponds to a π-calculus term that can reduce.
- A fairness property that characterizes PAM executions among possible π-calculus executions.

These properties are fairly standard but have not been proven for Pict yet (see section 5 for details). The main contribution of this paper lies in the statement and proof of a fairness property for Pict. To our knowledge, no implementation of a process calculus has been proven fair so far, although fairness is conjectured in [21, 19]. Moreover, the technique we propose is general enough to be adapted to similar settings.

Informally, we say that an execution is *weakly fair* if a prefixed process able to communicate *continuously* will eventually do so. An execution is *strongly fair* if a prefixed process able to communicate *infinitely often* will eventually do so. Consider for instance the π-calculus term

$$x!(a) \mid *x?(z).x!(z) \mid y!(0) \mid y?(z).\mathbf{0}$$

where $*P$ represents replicated input. There are valid infinite executions in which the communication on y never takes place even though at any time this communication is possible. Similarly, in the term

$$x!(a) \mid *x?(z).y!(z) \mid *y?(z).x!(z) \mid *x?(z).x!(z)$$

there are infinite executions in which the communication on the last process never takes place, even though such a communication might happen infinitely often (but not continuously). The intuitive expectation of a programmer is that all processes running in parallel will be interleaved *fairly* and so such executions are considered unsatisfactory.

Stating a fairness property for the π-calculus is not immediate. The definition of the π-calculus makes it difficult to identify subprocesses within a process, and in particular, it is difficult to state properties about fair executions of these processes. When considering π-calculus processes, mainly two kinds of confusion can arise.

- Processes are identified up to renaming of bound names and lead to possible confusion of channels. For instance, we have

  $$\nu x.\nu y.x!() \mid y!() \mid R \rightarrow \nu x.\nu y.x!() \mid R'$$

  Because of possible renamings of x and y, we do not know which channel reacted.

– Confusion of structurally equivalent processes.

$$P = x!() \mid x?().x!() \mid *x?().x!()$$
$$P' = x!() \mid x?().x!() \mid x?().x!() \mid *x?().x!()$$

We have $P \equiv P'$ and $P' \to P$, and we do not know which receivers react with $x!()$.

A possible solution is to define an auxiliary calculus in which prefixes are annotated with *labels* in such a way that labels uniquely denote prefixes and that this property is invariant throughout reduction [4, 3]. A *live action* of a term is then defined as a couple of labels corresponding to prefixed processes that can react. An infinite labeled execution is strongly fair if there are no labels appearing in an infinity of live actions.

2 Fairness in the π-calculus

2.1 The π-calculus

We suppose given a set of names N ranged over by $x, y, \ldots$. We define the set of π-calculus processes P as follows:

$$P, Q, \ldots ::= \mathbf{0} \quad | \quad \pi.P \quad | \quad \nu x.P \quad | \quad P \mid P$$
$$\pi ::= x!(y) \quad | \quad x?(y) \quad | \quad *x?(y)$$

The π-calculus evaluation contexts are given by:

$$\mathbf{E} ::= \cdot \quad | \quad \nu x.\mathbf{E} \quad | \quad P \mid \mathbf{E} \quad | \quad \mathbf{E} \mid P$$

The operational semantics is defined as the smallest relation such that rules in Figure 2 hold. It makes use of a structural equivalence relation defined as the smallest equivalence relation satisfying the rules in Figure 1. As usual, $\mathtt{fn}(P)$ denotes the set of free names of process P, and $=_\alpha$ equates two processes that differ only by their bound names. We write $\mathbf{E}[P]$ for the context $\mathbf{E}$ in which the hole . has been substituted with P.

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \text{ S.Par.Assoc} \qquad P \mid Q \equiv Q \mid P \text{ S.Par.Com}$$

$$P \mid \mathbf{0} \equiv P \text{ S.Par.Nil} \qquad \nu x.\mathbf{0} \equiv \mathbf{0} \text{ S.Nu.Nil} \qquad \nu x.\nu y.P \equiv \nu y.\nu x.P \text{ S.Nu.Com}$$

$$\frac{x \notin \mathtt{fn}(Q)}{(\nu x.P) \mid Q \equiv \nu x.P \mid Q} \text{ S.Nu.Par} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ S.}\alpha \qquad \frac{P \equiv Q}{\mathbf{E}[P] \equiv \mathbf{E}[Q]} \text{ S.Ctx}$$

Fig. 1. Structural Equivalence

Without loss of generality, we restrict the usual replication operator to input processes. Rule R.Rep models communication with a replicated input process.

$$\frac{}{x!(y).P \mid x?(z).Q \to P \mid Q\{y/z\}}\ \text{R.Red}$$

$$\frac{}{x!(y).P \mid *x?(z).Q \to P \mid Q\{y/z\} \mid *x?(z).Q}\ \text{R.Rep} \qquad \frac{P \to Q}{\mathbf{E}[P] \to \mathbf{E}[Q]}\ \text{R.Ctx}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}\ \text{R.Str}$$

Fig. 2. Reduction Relation

2.2 A Labeled π-calculus

Informally, a fair execution of a process is an execution in which no subprocess is ready to participate in a communication infinitely often. To formalize this statement, we need to identify in a process the subprocesses that can participate in a communication, and keep track of their identities throughout reductions.

To do so, we follow [4, 3] and define a labeled version of the π-calculus in which prefixes are annotated with labels. A label has to identify a prefix uniquely in an entire execution of a process. In other words, not only do prefixes have distinct labels in a process, but when new prefixes are created, their labels are new with respect to all the labels in the past execution of the term. We then characterize the labels belonging to prefixes that can participate in a communication. Finally, we give the definition of fairness.

We denote by L a set of labels such that $\mathsf{L} \cap \mathsf{N} = \emptyset$. We use $\mathcal{P}^f(\mathsf{L})$ to denote the finite subsets of L. A labeled process is a pair made of a π-calculus process in which prefixes are labeled, and a finite set of labels. The set of labeled processes LP is generated by the grammar given below.

$$\begin{array}{l} C, D, \ldots ::= \overline{P}, \mathcal{L} \\ \overline{P}, \overline{Q}, \ldots ::= \mathbf{0} \ \mid\ \pi_l.\overline{P} \ \mid\ \nu x.\overline{P} \ \mid\ \overline{P} \mid \overline{P} \\ \qquad l \in \mathsf{L} \\ \qquad \mathcal{L} \in \mathcal{P}^f(\mathsf{L}) \end{array}$$

We also extend contexts with labels and we denote labeled contexts $\overline{\mathbf{E}}$.

We need several auxiliary functions. The function lab returns the set of all labels of a process or a context. The function unl erases all labeling information from a labeled process.

In order to ensure that labels occur uniquely in a process, we define a well-formation predicate wf as the smallest relation on LP such that rules in Figure 3 hold. We write $A \uplus B$ for $A \cup B$ when $A \cap B = \emptyset$. A labeled process C is said to be well-formed if we have $\mathrm{wf}(C)$. We denote by WFP the set of well-formed labeled processes.

$$\frac{}{\mathrm{wf}(\mathbf{0}, \mathcal{L})}\ \textsc{Wf.Nil} \qquad \frac{\mathrm{lab}(\overline{P}) \uplus \mathrm{lab}(\overline{P}') \subseteq \mathcal{L}}{\mathrm{wf}(\overline{P} \mid \overline{P}', \mathcal{L})}\ \textsc{Wf.Par} \qquad \frac{\mathrm{wf}(\overline{P}, \mathcal{L})}{\mathrm{wf}(\nu x.\overline{P}, \mathcal{L})}\ \textsc{Wf.New}$$

$$\frac{\mathrm{wf}(\overline{P}, \mathcal{L})}{\mathrm{wf}(\pi_l.\overline{P}, \mathcal{L} \uplus \{l\})}\ \textsc{Wf.Prefix}$$

Fig. 3. Well-Formed Labeled Process

The operational semantics is defined in the same way as for the π-calculus via a structural equivalence relation $\equiv$ and a reduction relation $\rightarrow$, both binary relations over LP. The structural equivalence is defined, as before, as the smallest equivalence relation that verifies rules in Figure 1 (where prefixes are labeled and equivalent processes have the same set of labels). The reduction relation is the smallest relation that verifies the rules in Figure 4. The main difference with the unlabeled reduction relation appears in the rule LR.Rep for replicated input in which fresh labels are generated.

$$\frac{}{x!(y)_l.\overline{P} \mid x?(z)_{l'}.\overline{Q}, \mathcal{L} \rightarrow \overline{P} \mid \overline{Q}\{y/z\}, \mathcal{L}}\ \textsc{LR.Red}$$

$$\frac{\alpha \text{ injective and } \mathcal{L}' = \mathcal{L} \uplus \alpha(\mathcal{L})}{x!(y)_l.\overline{P} \mid *x?(z)_{l'}.\overline{Q}, \mathcal{L} \rightarrow \overline{P} \mid \overline{Q}\{y/z\} \mid \alpha(*x?(z)_{l'}.\overline{Q}), \mathcal{L}'}\ \textsc{LR.Rep}$$

$$\frac{\overline{P}, \mathcal{L} \rightarrow \overline{P}', \mathcal{L}' \qquad \mathrm{lab}(\overline{\mathbf{E}}) \subseteq \mathcal{L}}{\overline{\mathbf{E}}[\overline{P}], \mathcal{L} \rightarrow \overline{\mathbf{E}}[\overline{P}'], \mathcal{L}'}\ \textsc{LR.Ctx} \qquad \frac{D \equiv D' \qquad D' \rightarrow C' \qquad C' \equiv C}{D \rightarrow C}\ \textsc{LR.Str}$$

Fig. 4. Labeled Reduction Relation

Labeling is stable under reduction and structural equivalence. Hence, in the following, we consider only well-formed processes.

Lemma 1 (Stability of Labeling).

(i) If $C \equiv D$ and $C \in \mathsf{WFP}$, then $D \in \mathsf{WFP}$.
(ii) If $C \rightarrow D$ and $C \in \mathsf{WFP}$, then $D \in \mathsf{WFP}$.

The following lemma shows that the labeling system has been designed so that no label can occur more than once in a labeled term, and once a label disappears, it does not reappear in the system.

Lemma 2 (Uniqueness of Labeling).

(i) If $C \in \mathsf{WFP}$ then no label l occurs more than once in C.
(ii) If $C \in \mathsf{WFP}$, $C \rightarrow^* C' \rightarrow^* C''$ and $l \in \mathrm{lab}(C) \cap \mathrm{lab}(C'')$, then $l \in \mathrm{lab}(C')$.

The labeled π-calculus is a conservative extension of the π-calculus. A labeled process has exactly the same reductions as the corresponding unlabeled process. Moreover, we can label any process into a well-formed labeled process

Proposition 1 (Operational Correspondence). *Let $P \in \mathsf{P}$ and $C \in \mathsf{WFP}$ such that $P = \mathrm{unl}(C)$. We have*

(i) $P \rightarrow P'$ implies $\exists C' \in \mathsf{LP}$ such that $C \rightarrow C'$ and $\mathrm{unl}(C') = P'$.
(ii) $C \rightarrow C'$ implies $P \rightarrow \mathrm{unl}(C')$.

Proposition 2 (Existence of a Labeling). *For all $P \in \mathsf{P}$ there exists $C \in \mathsf{WFP}$ such that $\mathrm{unl}(C) = P$.*

We now define the live actions of a labeled process. A live action is a pair of labels corresponding to prefixed processes that can immediately react.

Definition 1 (Live Actions). *The set of live actions of a labeled process $C = \overline{P}, \mathcal{L}$ is defined as*

$$\begin{aligned} LA(C) = \{\{l, l'\} / C \equiv \nu\widetilde{x}.y!(v)_l.\overline{P}_0 \mid y?(z)_{l'}.\overline{P}_1 \mid \overline{P}_2, \mathcal{L} \\ \texttt{or}\ C \equiv \nu\widetilde{x}.y!(v)_l.\overline{P}_0 \mid *y?(z)_{l'}.\overline{P}_1 \mid \overline{P}_2, \mathcal{L}\} \end{aligned}$$

We also define the set of labels belonging to a live action as

$$L(C) = \{l \in x / x \in LA(C)\}$$

The following lemma states a correspondence between live actions and reductions.

Lemma 3. *$C \rightarrow C'$ for some C' if and only if $LA(C) \neq \emptyset$.*

Definition 2 (Execution). *For an arbitrary relation $\rightarrow$, an execution is a sequence of terms $T_0, T_1, \ldots$, possibly infinite, such that $T_0 \rightarrow \ldots \rightarrow T_n \rightarrow \ldots$.*

We can now define a strong fairness property for the labeled calculus. An execution is fair if a prefix cannot potentially participate in a reduction infinitely often. According to this definition, we only need to consider infinite executions.

Definition 3 (Strong Fairness in the Labeled π-calculus). *An infinite execution $C_0 \rightarrow \ldots \rightarrow C_n \rightarrow \ldots$ is fair if for any strictly increasing sequence $(u_n)_{n \in \mathbb{N}}$, we have $\bigcap_{n \in \mathbb{N}} L(C_{u_n}) = \emptyset$.*

An execution in the π-calculus is fair if it corresponds to a fair execution in the labeled calculus.

Definition 4 (Strong Fairness in the π-calculus). *An infinite execution $P_0 \rightarrow \ldots \rightarrow P_n \rightarrow \ldots$ is fair if there is a fair execution $C_0 \rightarrow \ldots \rightarrow C_n \rightarrow \ldots$ such that $\forall i \in \mathbb{N}.\mathrm{unl}(C_i) = P_i$.*

3 Abstract Machine

3.1 Syntax and Operational Semantics

The syntax of the Pict abstract machine is given in Figure 5 and follows closely [19] [3]. A machine state, or PAM term, consists of a queue of π-calculus processes $\mathcal{P}$ (the runqueue), a heap $\mathcal{H}$ and a set of names $\mathcal{N}$. A heap is a function that maps channel names to processes queues. We denote by M the set of machine states. We often omit the set of names $\mathcal{N}$ in PAM terms when it is not important, in particular in reduction rules where it remains unchanged. We also write $\mathcal{P} :: \mathcal{Q}$ for the appending of $\mathcal{P}$ and $\mathcal{Q}$. The operational semantics is defined via two reduction

$$\begin{array}{rr} \mathcal{M} ::= \langle \mathcal{P}, \mathcal{H}, \mathcal{N} \rangle & \text{State} \\ \mathcal{P}, \mathcal{Q}, \ldots ::= [] \mid P :: \mathcal{P} & \text{Processes Queue} \\ \mathcal{H} ::= \{x \to \mathcal{P}_x\}_{x \in \mathsf{N}} & \text{Heap} \end{array}$$

Fig. 5. Syntax of PAM Terms

relations, defined as the smallest binary relations over machine states that satisfy the inference rules given in Figure 6. Intuitively, the relation $\rightsquigarrow$ corresponds to the implementation of $\equiv$, whereas $\mapsto$ implements the actual communication. An actual implementation of this abstract machine does not need to distinguish these relations and would implement $\to = \rightsquigarrow \uplus \mapsto$, but this distinction will help us to prove correctness properties. In rule AM.New, we suppose there is a function freshn : $\mathcal{P}^f(\mathsf{N}) \to \mathsf{N}$ such that freshn$(\mathcal{N}) \notin \mathcal{N}$. We also suppose that names generated by the freshn function never appear in the π-calculus processes in the PAM term (this could be enforced by defining a new syntactic category of names).

We refer the reader to [19, 16] for detailed explanations of these rules. We briefly summarize here the main ideas. An execution of the abstract machine starts with an empty heap (we denote it with $\mathcal{H}_{[]}$) that maps all names to empty queues of processes, and a runqueue containing the π-calculus process to be executed. Depending on the form of the process at the top of the runqueue, and the state of the heap, exactly one rule can apply. The execution stops when the runqueue is empty.

A nil process is discarded from the runqueue (rule AM.Nil). Parallel composition of processes is split into two processes that are split in the runqueue (rule AM.Par). Rule AM.New implements name restriction by generating new fresh names. When the first term of the runqueue is a prefixed process willing to communication on a name x, there are two possible cases. If there is no corresponding process in the heap, the process is pushed on the heap queue for x (rules

[3] In particular, this presentation makes use of synchronous communications.

AM.PushMessage, AM.PushReceiver, AM.PushRepReceiver). If there is a corresponding process in the heap queue (the first element), the communication is performed and the continuation of the receiver and sender are placed in the runqueue (rules AM.Com1, AM.RCom1, AM.Com2, AM.RCom2).

We can show that processes appearing in an association $x \to \mathcal{P}$ are of the form $\pi.P$, where all prefixes are either output on x, or input (replicated or not) on x. Moreover, this property is invariant by reduction. In the following, we only consider machine states of this form. We also suppose that $\mathcal{H}$ is finite. Moreover, we can notice that the relation $\to$ is deterministic. In particular, generated fresh names are fully determined by the function freshn in rule AM.New.

$$\frac{}{\langle \mathbf{0} :: \mathcal{Q}, \mathcal{H}\rangle \rightsquigarrow \langle \mathcal{Q}, \mathcal{H}\rangle}\ \text{AM.Nil} \qquad \frac{}{\langle (P \mid Q) :: \mathcal{Q}, \mathcal{H}\rangle \rightsquigarrow \langle P :: \mathcal{Q} :: Q, \mathcal{H}\rangle}\ \text{AM.Par}$$

$$\frac{z = \text{freshn}(\mathcal{N})}{\langle \nu x.P :: \mathcal{Q}, \mathcal{H}, \mathcal{N}\rangle \rightsquigarrow \langle P\{z/x\} :: \mathcal{Q}, \mathcal{H}, \mathcal{N} \uplus \{z\}\rangle}\ \text{AM.New}$$

$$\frac{\mathcal{P} = [] \vee \mathcal{P} = x?(z).Q :: \mathcal{P}' \vee \mathcal{P} = *x?(z).Q :: \mathcal{Q}}{\langle x?(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \rightsquigarrow \langle \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P} :: x?(y).P\}\rangle}\ \text{AM.PushReceiver}$$

$$\frac{\mathcal{P} = [] \vee \mathcal{P} = x!(z).Q :: \mathcal{P}'}{\langle x!(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \rightsquigarrow \langle \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P} :: x!(y).P\}\rangle}\ \text{AM.PushMessage}$$

$$\frac{\mathcal{P} = [] \vee \mathcal{P} = x?(z).Q :: \mathcal{P}' \vee \ \mathcal{P} = *x?(z).Q :: \mathcal{P}'}{\langle *x?(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \rightsquigarrow \langle \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P} :: *x?(y).P\}\rangle}\ \text{AM.PushRepReceiver}$$

$$\frac{\mathcal{P} = x!(z).Q :: \mathcal{P}'}{\langle x?(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \mapsto \langle P\{z/y\} :: \mathcal{Q} :: Q, \mathcal{H} \oplus \{x \to \mathcal{P}'\}\rangle}\ \text{AM.Com1}$$

$$\frac{\mathcal{P} = x!(z).Q :: \mathcal{P}'}{\langle *x?(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \mapsto \langle *x?(y).P :: \mathcal{Q} :: P\{z/y\} :: Q, \mathcal{H} \oplus \{x \to \mathcal{P}'\}\rangle}\ \text{AM.RCom1}$$

$$\frac{\mathcal{P} = x?(z).Q :: \mathcal{P}'}{\langle x!(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \mapsto \langle P :: \mathcal{Q} :: Q\{y/z\}, \mathcal{H} \oplus \{x \to \mathcal{P}'\}\rangle}\ \text{AM.Com2}$$

$$\frac{\mathcal{P} = *x?(z).Q :: \mathcal{P}'}{\langle x!(y).P :: \mathcal{Q}, \mathcal{H} \oplus \{x \to \mathcal{P}\}\rangle \mapsto \langle P :: \mathcal{Q} :: Q\{y/z\}, \mathcal{H} \oplus \{x \to \mathcal{P}' :: *x?(z).Q\}\rangle}\ \text{AM.RCom2}$$

Fig. 6. PAM Reduction Rules

3.2 Labeled Abstract Machine

We define a labeled version of the Pict abstract machine and essentially follow section 2. This auxiliary calculus is a technical tool, and it is only used for proving the correctness of the abstract machine. Its syntax is defined by adding labels to π-calculus processes appearing in PAM terms. We also extend PAM terms with a finite set of labels. We write LM for the set of labeled PAM terms, and we use $\overline{\mathcal{M}}$ and its variants to range over them.

$$\begin{array}{rr}
\overline{\mathcal{M}} ::= \langle \overline{\mathcal{P}}, \overline{\mathcal{H}}, \mathcal{N}, \mathcal{L} \rangle & \text{State} \\
\overline{\mathcal{P}}, \overline{\mathcal{Q}}, \ldots ::= [] \mid \overline{P} :: \overline{\mathcal{P}} & \text{Processes Queue} \\
\overline{\mathcal{H}} ::= \{x \to \overline{\mathcal{P}}_x\}_{x \in \mathsf{N}} & \text{Heap} \\
\mathcal{L} \in \mathcal{P}^f(\mathsf{L}) &
\end{array}$$

We define the set of well-formed PAM terms in Figure 7 and call it WFM. Reduction of labeled PAM terms is defined almost exactly as for the unlabeled calculus, apart from the rules AM.RCom1 and AM.RCom2. The functions lab and unl extend as expected on processes queues, heaps and machine states.

$$\frac{\mathrm{wf}(\overline{\mathcal{P}}) \qquad \mathrm{wf}(\overline{\mathcal{H}}) \qquad \mathrm{lab}(\overline{\mathcal{P}}) \uplus \mathrm{lab}(\overline{\mathcal{H}}) \subseteq \mathcal{L}}{\mathrm{wf}(\langle \overline{\mathcal{P}}, \overline{\mathcal{H}}, \mathcal{N}, \mathcal{L} \rangle)}\ \text{WF.State}$$

$$\frac{\mathrm{wf}(\overline{\mathcal{P}}) \qquad \mathrm{wf}(\overline{P}) \qquad \mathrm{lab}(\overline{P}) \cap \mathrm{lab}(\overline{\mathcal{P}}) = \emptyset}{\mathrm{wf}(\overline{P} :: \overline{\mathcal{P}})}\ \text{WF.ProcQueue}$$

$$\frac{\forall x, y \in \mathsf{N}. x \neq y \implies \mathrm{lab}(\overline{\mathcal{H}}(x)) \cap \mathrm{lab}(\overline{\mathcal{H}}(y)) = \emptyset \qquad \forall x \in \mathsf{N}. \mathrm{wf}(\overline{\mathcal{H}}(x))}{\mathrm{wf}(\overline{\mathcal{H}})}\ \text{WF.Heap}$$

Fig. 7. Well-Formed PAM Term

Lemma 4 (Stability of Labeling).

(i) If $\overline{\mathcal{M}} \equiv \overline{\mathcal{M}}'$ *and* $\overline{\mathcal{M}} \in \mathsf{WFM}$*, then* $\overline{\mathcal{M}}' \in \mathsf{WFM}$*.*
(ii) If $\overline{\mathcal{M}} \to \overline{\mathcal{M}}'$ *and* $\overline{\mathcal{M}} \in \mathsf{WFM}$*, then* $\overline{\mathcal{M}}' \in \mathsf{WFM}$*.*

Proposition 3 (Operational Correspondence). *Let* $\mathcal{M} \in \mathsf{M}$ *and* $\overline{\mathcal{M}} \in \mathsf{WFM}$ *such that* $\mathcal{M} = \mathrm{unl}(\overline{\mathcal{M}})$*. If* $\Rightarrow$ *denotes either* $\mapsto$ *or* $\rightsquigarrow$*, we have*

(i) $\mathcal{M} \Rightarrow \mathcal{M}'$ *implies there is* $\overline{\mathcal{M}}' \in \mathsf{LM}$ *such that* $\overline{\mathcal{M}} \Rightarrow \overline{\mathcal{M}}'$ *and* $\mathrm{unl}(\overline{\mathcal{M}}') = \mathcal{M}'$*.*
(ii) $\overline{\mathcal{M}} \Rightarrow \overline{\mathcal{M}}'$ *implies* $\mathcal{M} \Rightarrow \mathrm{unl}(\overline{\mathcal{M}}')$*.*

$$\frac{\mathcal{P} = x!(z)_{l'}.\overline{Q} :: \overline{\mathcal{P}}' \qquad \alpha \text{ injective and } \mathcal{L}' = \mathcal{L} \uplus \alpha(\mathcal{L})}{\begin{array}{c}\langle *x?(y)_l.\overline{P} :: \overline{\mathcal{Q}}, \overline{\mathcal{H}} \oplus \{x \to \overline{\mathcal{P}}\}, \mathcal{L}\rangle \mapsto \\ \langle \alpha(*x?(y)_l.\overline{P}) :: \overline{\mathcal{Q}} :: \overline{P}\{z/y\} :: \overline{Q}, \overline{\mathcal{H}} \oplus \{x \to \overline{\mathcal{P}}'\}, \mathcal{L}'\rangle\end{array}} \text{ AM.RCom1}$$

$$\frac{\mathcal{P} = *x?(z)_{l'}.\overline{Q} :: \overline{\mathcal{P}}' \qquad \alpha \text{ injective and } \mathcal{L}' = \mathcal{L} \uplus \alpha(\mathcal{L})}{\begin{array}{c}\langle x!(y)_l.\overline{P} :: \overline{\mathcal{Q}}, \mathcal{H} \oplus \{x \to \overline{\mathcal{P}}\}, \mathcal{L}\rangle \mapsto \\ \langle \overline{P} :: \overline{\mathcal{Q}} :: \overline{Q}\{y/z\}, \overline{\mathcal{H}} \oplus \{x \to \overline{\mathcal{P}}' :: \alpha(*x?(z)_{l'}.\overline{Q})\}, \mathcal{L}'\rangle\end{array}} \text{ AM.RCom2}$$

Fig. 8. Labeled PAM Reduction Rules

We now define the live actions of a labeled PAM term as the live actions of a corresponding π-calculus term. Intuitively, $\{l, l'\}$ is a live action whenever there are two matching prefixed processes somewhere in the PAM term that could *potentially* react.

Definition 5 (Live Actions). *The set of live actions of a labeled PAM term* $\overline{\mathcal{M}}$ *is defined as*

$$LA(\overline{\mathcal{M}}) = LA([\![\overline{\mathcal{M}}]\!]^r)$$

where $[\![.]\!]^r$ *is defined inductively on the structure of* $\overline{\mathcal{M}}$*.*

$$\begin{array}{c}[\![\langle \overline{\mathcal{P}}, \overline{\mathcal{H}}, \mathcal{N}, \mathcal{L}\rangle]\!]^r = \nu\mathcal{N}.[\![\overline{\mathcal{P}}]\!]^r \mid [\![\overline{\mathcal{H}}]\!]^r, \mathcal{L} \\ [\![\ [\,]\ ]\!]^r = \mathbf{0} \\ [\![\overline{P} :: \overline{\mathcal{P}}]\!]^r = \overline{P} \mid [\![\overline{\mathcal{P}}]\!]^r \\ [\![\{x \to \overline{\mathcal{P}}_x\}_{x \in \mathsf{N}}]\!]^r = |_{x \in \mathsf{N}}\ [\![\overline{\mathcal{P}}_x]\!]^r\end{array}$$

We also define the set of labels belonging to a live action as $L(\overline{\mathcal{M}}) = \{l \in x / x \in LA(\overline{\mathcal{M}})\}$*.*

Lemma 5. *If* $LA(\overline{\mathcal{M}}) \neq \emptyset$ *then* $\exists \overline{\mathcal{M}}'.\ \overline{\mathcal{M}} \rightsquigarrow^* \mapsto \overline{\mathcal{M}}'$*.*

The following theorem can be seen as a fairness property for the labeled abstract machine.

Theorem 1. *If* $\overline{\mathcal{M}}_0 \to \ldots \to \overline{\mathcal{M}}_n \to \ldots$ *is an infinite execution then for any strictly increasing sequence* $(u_n)_{n \in \mathbb{N}}$*, we have* $\bigcap_{n \in \mathbb{N}} L(\overline{\mathcal{M}}_{u_n}) = \emptyset$

The proof is technical but it relies on intuitive ideas. Informally, it follows from two key properties of the abstract machine reduction system:

- If a process $\pi_l.\overline{P}$ appears in an evaluation context in the runqueue, it will eventually reach the top of the runqueue.
- The heap queues are organized following a FIFO policy.

4 Correctness

From an operational point of view, the correctness of an abstract machine can be stated by relating abstract machine executions of a process P with π-calculus executions of the same process P executed by the abstract machine. The initial state of an abstract machine running P is $\langle P, \mathcal{H}_{[]} \rangle$, hence we introduce the following translation function.

Definition 6 (Translation from π-calculus to PAM).

$$[\![P]\!] = \langle P ::[], \mathcal{H}_{[]}, \emptyset \rangle \qquad\qquad [\![\overline{P}, \mathcal{L}]\!] = \langle \overline{P} ::[], \mathcal{H}_{[]}, \emptyset, \mathcal{L} \rangle$$

The first property we consider is the soundness of the abstract machine with respect to the calculus. Intuitively, this means that abstract machine executions correspond to valid π-calculus executions. If a machine state $\mathcal{M}$, corresponding to a process state P, reduces to a machine state $\mathcal{M}'$, then $\mathcal{M}'$ must correspond to a process state P' where P reduces to P'. One reduction in the π-calculus may be implemented by several reductions of the abstract machine. In order to model a one-to-one correspondence, we identify two kinds of reductions. Administrative reductions denoted by $\rightsquigarrow$ model structural equivalence. Communication reductions are denoted by $\mapsto$. We will establish a correspondence between the relations $\rightsquigarrow^* \mapsto$ over PAM terms and $\rightarrow$ over π-calculus terms. For that, we define a relation $\mathcal{M} \preceq P$ to mean that P corresponds to $\mathcal{M}$, read $\mathcal{M}$ implements P.

We still need to define the relation $\preceq$. It has to be convincing enough that it effectively relates equivalent process state and machine state. It should at least enjoy the following two properties:

– $[\![P]\!] \preceq P$ and – If $\mathcal{M} \rightsquigarrow \mathcal{M}'$ and $\mathcal{M} \preceq P$ then $\mathcal{M}' \preceq P$.

The first property follows the idea that the initial state of an abstract machine executing $[\![P]\!]$ is P. The second property follows the intuition that $\rightsquigarrow$ is a structural, or administrative, reduction and that abstract machine states still implement the same π-calculus process after such reductions. We define $\preceq$ as the smallest relation enjoying these two properties.

Definition 7. $\mathcal{M} \preceq P \iff [\![P]\!] \rightsquigarrow^* \mathcal{M}$.

The definition of $\preceq$ extends naturally to labeled processes.

Note that we do not have a notion of observables, although it would make the correspondence relation $\preceq$ more convincing. However, it should be straightforward to define an observation predicate on π-calculus processes and PAM terms (such as those in [1, 7, 11]) and show that $\preceq$ preserves the observables.

The following lemma relates the live actions of a labeled PAM term and a labeled process it implements.

Lemma 6. *If* $\overline{\mathcal{M}} \preceq C$ *then* $LA(\overline{\mathcal{M}}) = LA(C)$.

To prove the soundness property, we need a translation function from PAM terms to π-calculus processes. This function is very similar to $[\![.]\!]^r$. We do not give its full definition here but the following lemma states the properties needed for the proof of soundness.

Lemma 7. *There exists a function* $[\![.]\!]^{-1}$ *from* M *to* P *such that*

- $\mathcal{M} \preceq [\![\mathcal{M}]\!]^{-1}$
- $[\![[\![P]\!]]\!]^{-1} \equiv P$
- *if* $\mathcal{M} \rightsquigarrow \mathcal{M}'$ *then* $[\![\mathcal{M}]\!]^{-1} \equiv [\![\mathcal{M}']\!]^{-1}$
- *if* $\mathcal{M} \mapsto \mathcal{M}'$ *then* $[\![\mathcal{M}]\!]^{-1} \mapsto [\![\mathcal{M}']\!]^{-1}$.

Theorem 2 (Soundness). *If* $(\mathcal{M} \rightsquigarrow^* \mapsto \mathcal{M}' \wedge \mathcal{M} \preceq P)$ *then* $(\exists P'.P \to P' \wedge \mathcal{M}' \preceq P')$.

Proof. The theorem follows from $[\![P]\!] \rightsquigarrow^* \mapsto \mathcal{M} \implies (\exists P'.P \to P' \wedge \mathcal{M} \preceq P')$ which is a consequence of Lemma 7 with $P' = [\![\mathcal{M}]\!]^{-1}$.

This property is not sufficient to prove the correctness of the abstract machine. Other properties are needed to characterize which executions of the π-calculus are actually implemented. First, a liveness property ensures that a PAM term is never blocked when it corresponds to a π-calculus term that can reduce.

Theorem 3 (Liveness). *If* $P \to P' \wedge \mathcal{M} \preceq P$ *then* $\exists \mathcal{M}'.\mathcal{M} \rightsquigarrow^* \mapsto \mathcal{M}'$.

Proof. We first prove: $P \to P' \implies \exists \mathcal{M}.[\![P]\!] \rightsquigarrow^* \mapsto \mathcal{M}$. If $P \to P'$, we have $C \to C'$ with $\mathrm{unl}(C) = P$ and $\mathrm{unl}(C') = P'$, by propositions 2 and 1. Moreover, by Lemma 6, $LA([\![C]\!]) = LA(C)$ with $LA(C) \neq \emptyset$, by Lemma 3. We deduce $[\![C]\!] \rightsquigarrow^* \mapsto \overline{\mathcal{M}}'$ for some $\overline{\mathcal{M}}'$, by Lemma 5. We conclude, by Proposition 3, that $[\![P]\!] = \mathrm{unl}([\![C]\!]) \rightsquigarrow^* \mapsto \mathrm{unl}(\overline{\mathcal{M}}')$.

We know now that $[\![P]\!] \rightsquigarrow^* \mathcal{M}'' \mapsto \mathcal{M}'$ for some $\mathcal{M}''$ and $\mathcal{M}'$. Moreover we have $[\![P]\!] \rightsquigarrow^* \mathcal{M}$, by definition of $\mathcal{M} \preceq P$. Because $\to$ is deterministic, we conclude $\mathcal{M} \rightsquigarrow^* [\![P]\!] \rightsquigarrow^* \mathcal{M}'' \mapsto \mathcal{M}'$.

Finally, our main result is a fairness theorem.

Theorem 4 (Fairness). *If* $\mathcal{M}_0 \rightsquigarrow^* \mapsto \ldots \rightsquigarrow^* \mapsto \mathcal{M}_n \rightsquigarrow^* \mapsto \ldots$ *is an infinite execution then there exists a fair execution* $P_0 \to \ldots \to P_n \to \ldots$ *such that* $\mathcal{M}_i \preceq P_i$ *for all* i.

Proof. Let $\mathcal{M}_0 \rightsquigarrow^* \mapsto \ldots \rightsquigarrow^* \mapsto \mathcal{M}_n \rightsquigarrow^* \mapsto \ldots$ be an infinite execution. we have an execution $\overline{\mathcal{M}}_0 \rightsquigarrow^* \mapsto \ldots \rightsquigarrow^* \mapsto \overline{\mathcal{M}}_n \rightsquigarrow^* \mapsto \ldots$ such that for all i, $\mathrm{unl}(\overline{\mathcal{M}}_i) = \mathcal{M}_i$, by Proposition 3.

The soundness theorem (Theorem 2) extends to the labeled calculus and gives us an execution $C_0 \to \ldots \to C_n \to \ldots$ such that

$$\begin{array}{ccccccc}
\overline{\mathcal{M}}_0 & \xrightarrow[\rightsquigarrow^* \mapsto]{} & \overline{\mathcal{M}}_1 & \xrightarrow[\rightsquigarrow^* \mapsto]{} & \overline{\mathcal{M}}_2 & \xrightarrow[\rightsquigarrow^* \mapsto]{} & \cdots \\
\preceq \vdots & & \preceq \vdots & & \preceq \vdots & & \\
C_0 & \xrightarrow[\to]{} & C_1 & \xrightarrow[\to]{} & C_2 & \xrightarrow[\to]{} & \cdots
\end{array}$$

From Lemma 6, we have $LA(C_i) = LA(\overline{\mathcal{M}}_i)$ for all i. Then we deduce from Theorem 1 that the execution $C_0 \to \ldots C_n \to \ldots$ is fair. By erasing the labels in both executions, we deduce the result.

5 Related Work

Comparison with Pict Correctness results in [19] include a soundness and a liveness property based on the translation function $[\![.]\!]^r$ from PAM terms to π-calculus terms given in Definition 5:

(i) $\mathcal{M} \to \mathcal{M}' \implies [\![\mathcal{M}]\!]^r \equiv [\![\mathcal{M}']\!]^r \vee [\![\mathcal{M}]\!]^r \to [\![\mathcal{M}']\!]^r$
(ii) $P \to P' \implies \exists \mathcal{M}.\ [\![P]\!] \to \mathcal{M}$

However, these properties are not sufficient for proving soundness or liveness. The first property means that we can build a π-calculus reduction from a PAM reduction, but does not prove that PAM reductions implement π-calculus reductions. A property relating $\mathcal{M}$ and $[\![[\![\mathcal{M}]\!]^r]\!]$, such as our Lemma 7 is missing. The second property tells us that if P reduces to P', there is a PAM reduction $[\![P]\!] \to M$. However, the property cannot be applied on more than the first step of execution, as we do not know if there is P'' such that $P \to P''$ and $[\![P'']\!] = \mathcal{M}$.

In [18], the Pict abstract machine is proven correct using a notion of testing, and a realistic model of the interactions between the abstract machine and its environment. However, they do not consider fairness issues.

Fairness Fairness has been defined using labels in CCS [3] and in the π-calculus [4, 2]. We essentially followed the same idea but our presentation is simpler as we annotate labeled terms with a set of labels that allow us to generate fresh labels in the replication rules, without relying on a structured labeling language.

In [13], fairness is defined for the π-calculus by considering *normal* reductions where α-equivalence is restricted and tags similar to labels are used to distinguish processes. Fresh tags are generated using the π-calculus name restriction operator.

Correctness of Abstract Machine There have been several recent papers devoted to the formal description of implementations of process calculi based on the π-calculus or the Ambient calculus. In addition to Pict, one can notably cite the Jocaml distributed implementation of the Join calculus [6, 5], the Join calculus implementation of Mobile Ambients [7], Nomadic Pict [22, 20], the abstract machine for the M-calculus [10], the Fusion Machine [8], the PAN and GCPAN abstract machines for Safe Ambients [11, 12], the CAM abstract machine for Channel Ambients [15] and the abstract machine for the Kell calculus [1]. Most of these works [7, 22, 20, 10, 8, 11, 12, 15, 1] deal with distributed implementations of calculi, rather than local implementation of concurrent processes like in Pict. They are defined by a translation to a low-level calculus or abstract machine. Their correctness is proven in terms of bisimilary that does not apply to our setting, since Pict implementation makes deterministic choice and PAM reductions do not match all π-calculus reductions. Implementations that consider scheduling of processes are given in [15, 10]. In [15], a soundness result is given similar to the one given in Pict. In [10], scheduling of processes is done as in Pict using FIFO lists, but no proof of correctness is given.

6 Conclusion

In this paper, we first defined strong fairness in the π-calculus. We then proved that Pict abstract machine executions are sound with respect to π-calculus executions and that they enjoy fairness and liveness properties. These correctness results for Pict are new and in particular, fairness has not been proven for any implementation of process calculi based on the π-calculus. We believe that these techniques are simple and general enough to be adapted to other calculi.

Very little work has been done on the scheduling of processes in the π-calculus or its variants. For future research, we will investigate alternative scheduling strategies. In particular, we would like to extend Pict and its implementation with priority constraints. Processes could be prioritized in order to allocate more processor time to more important processes. In Pict, even though executions are strongly fair, in a term $P \mid Q$, P can monopolize the processor usage by spawning new subprocesses much faster than Q. One can imagine annotated processes like in $P_h \mid Q_l$ where the annotations are taken into account by the scheduler. Such a scheme would fit naturally in a calculus with hierarchical localities such as [1]. For instance, a term of the form $a[b[P] \mid c[Q]]$ can be interpreted as two (possibly untrusted) agents b and c executed by a site a. The parent site a should be able to control the processor usage of the agents it is executing.

Most correctness results of the implementations of process calculi with localities concern their distributed implementation, but do not deal with the correctness of their local implementation, *i.e.* the scheduling of processes. On the other hand, Pict defines a local implementation. It would be interesting to consider correctness results combining these two approaches. We are currently investigating the proof of a refined abstract machine based on [1].

Acknowledgments We are grateful to Healfdene Goguen, Benjamin Pierce, Alan Schmitt and Jean-Bernard Stefani, as well as the anonymous reviewers, for their comments on earlier drafts. We thank Pablo Garralda, whose PhD thesis work inspired us to use labeled processes to study fairness [9].

References

1. Philippe Bidinger, Alan Schmitt, and Jean-Bernard Stefani. An abstract machine for the Kell calculus. In *7th IFIP International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS)*, pages 43–58, Athens, Greece, June 2005.
2. Diletta Cacciagrano, Flavio Corradini, and Catuscia Palamidessi. Fairpi. In *Proceedings of EXPRESS'06*, ENTCS, 2006.
3. Gerardo Costa and Colin Stirling. Weak and strong fairness in CCS. In Michal Chytil and Václav Koubek, editors, *MFCS*, volume 176 of *Lecture Notes in Computer Science*, pages 245–254. Springer, 1984.
4. F. Corradini D.R. Cacciagrano. Fairness in the pi-calculus. Technical report, Dipartimenti di Informatica, Università di L'Aquila, TR 005/2004, 2004.
5. Fabrice Le Fessant. *JoCaml: Conception et Implantation d'un Langage à Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2001.

6. C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*, pages 406–421. Springer Verlag, 1996.
7. C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*, pages 348–364. Springer, 2000.
8. Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine. In L. Brim, P. Jancar, and M. Kretinsky, editors, *CONCUR 2002*, volume 2421 of *LNCS*, pages 418–433. Springer-Verlag, 2002.
9. Pablo Garralda. *Boxed Ambients for Global Computing*. PhD thesis, Stevens Institute of Technology, New Jersey, USA, 2007.
10. F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.
11. Paola Giannini, Davide Sangiorgi, and Andrea Valente. Safe ambients: abstract machine and distributed implementation. *Sci. Comput. Program.*, 59(3):209–249, 2006.
12. Daniel Hirschkoff, Damien Pous, and Davide Sangiorgi. A correct abstract machine for safe ambients. In *COORDINATION*, pages 17–32, 2005.
13. Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.
14. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1):1–78, 1992.
15. A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of ESOP 2004*, LNCS. Springer-Verlag, April 2004.
16. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
17. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
18. Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR 97: Concurrency Theory (Warsaw). LNCS 1243*, pages 391–405, 1997.
19. D. Turner. The polymorphic π-calculus: Theory and implementation. Technical report, University of Edinburgh, GB, 1996.
20. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, pages 116–127, 2001.
21. Lucian Wischik. *Explicit Fusions: Theory and Implementation*. PhD thesis, Computer Laboratory, University of Cambridge, 2001.
22. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, pages 42–52, 2000.