Math. Struct. in Comp. Science (2018), *vol.* 28, *pp.* 202–240. © Cambridge University Press 2016
doi:10.1017/S0960129516000256 First published online 22 September 2016

On the expressiveness of π-calculus for encoding mobile ambients

LINDA BRODO

Dipartimento di Scienze Politiche, Scienze della Comunicazione e Ingegneria dell'Informazione, Università degli Studi di Sassari
viale Mancini, 5 - 07100 - Sassari, Italia
Email: brodo@uniss.it

Received 18 February 2014; revised 18 June 2016

We investigate the expressiveness of two classical distributed paradigms by defining the first encoding of the pure mobile ambient calculus into the synchronous π-calculus. Our encoding, whose correctness has been proved by relying on the notion of *operational correspondence*, shows how the hierarchical ambient structure can be reformulated within a flat channel interconnection amongst independent processes, without centralised control. To easily handle the computation for simulating a capability, we introduce the notions of *simulating trace* (representing the computation that a π-calculus process has to execute to mimic a capability) and of *aborting trace* (representing the computation that a π-calculus process executes when the simulation of a capability cannot succeed). Thus, the encoding may introduce loops, but, as it will be shown, the number of steps of any trace, therefore of any aborting trace, is limited, and the number of states of the transition system of the encoding processes still remains finite. In particular, an aborting trace makes a sort of backtracking, leaving the involved sub-processes in the same starting configurations. We also discuss two run-time support methods to make these loops harmless at execution time. Our work defines a relatively simple, direct, and precise translation that reproduces the ambient structure by means of channel links, and keeps track of the dissolving of an ambient.

1. Introduction

The comparison of different languages is a crucial topic in the area of formal languages, see for example, Sangiorgi (1996), Palamidessi (2003), Busi et al. (2009). It is a tool to characterise the expressiveness power of different languages, whereas all languages have the same computational power. This kind of investigation plays an important role in modelling case studies which come from different areas and at different abstract levels. Here, we compare two classical, yet very different paradigms for mobility: the Mobile Ambient (MA) calculus and the π-calculus. MAs (Cardelli and Gordon 2000) have been introduced to model distributed systems where complete computing environments (i.e. programs being executed) may change their location. In contrast, the π-calculus (Milner et al. 1992) allows only names or pieces of code, in its higher order version, to be sent along communication channels. The two languages use different paradigms for distributed computing, although both of them are Turing-equivalent.

MAs are characterised by the ambient construct, $n[\dots]$, that defines computing environments. Actions on ambients, called capabilities, can destroy an ambient (*open n*), or

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

make an ambient enter a second one (*in n*), or make an ambient leave the ambient within which it lies (*out n*).

In spite of the fact that the two languages are very well known and very well studied, we believe that our work can still be of interest as it shows how an ambient construct embodies two different functionalities: the delimitation of a location, and the bearer of an identity. This study is also at the base of the work in Bodei et al. (2013) that introduces a multiparty interaction process algebra, the `link`-calculus.

We define an encoding that allows π-processes to mimic the execution of a capability (corresponding to a MA transition) by performing a number of π-calculus transitions. These series of transitions are required to guarantee that all the conditions for a correct capability simulation are satisfied. Since we do not have a one-to-one operational correspondence between the MA and the π-calculus transitions, we collect all the π processes that have a corresponding MA process in a set, called Aπ. We will show that given a MA process P, with $Q \in A\pi$ its π-calculus encoding, then if there exists P' such that $P \rightarrow P'$, it follows that there exist $Q' \in A\pi$ and $n > 0$ processes $Q_1, \ldots, Q_n \notin$ Aπ such that $Q \rightarrow Q_1 \cdots \rightarrow Q_n \rightarrow Q'$, where Q' is the encoding of P'.

To prove the correctness of our encoding, we relay on the notion of *operational correspondence*: (1) we prove that whenever the MA process executes a capability, then its π-calculus encoding can execute a series of transitions that mimic the capability; (2) we prove that whenever a translating π-calculus process performs a transition, this is part of a computation trying to simulate a capability. When the capability simulation does not succeed, our encoding introduces loops, still keeping the number of states introduced finite, thus we get this somewhat ‘weak’ soundness.

In order to prove the main properties of our encoding, we introduce the notions of *simulating traces* and *aborting traces*. A simulating trace corresponds to a series of transitions simulating a capability execution. When the conditions required for the execution of a capability do not hold, we have an aborting trace that leaves no side-effects of its execution. Thus, the final state of a simulating trace records the effects of a capability execution, whilst the final state of an aborting trace is congruent to the initial one, in the hypothesis that only transitions related to that single trace have been fired. Summarising, the main points of our encoding are:

— the divergence introduced is, from a practical point of view, harmless, as we will see in Subsection 4.5, that we can easily discriminate between the execution of a simulating trace and the execution of an aborting trace (which is the cause of our divergence);
— there is not a process which has a central control of the execution of the encoding process, as each sub-process produced by the translation function only reflects the activities of the corresponding MA sub-process;
— it is compositional with respect to the parallel and the non-deterministic operators, up to minor preliminary settings.

The structure of our paper is as follows. Section 2 introduces the MA calculus, Section 3 presents the π-calculus. In Section 4, we formally define our encoding and its properties, together with some examples to help the intuition. In particular, in Subsection 4.5, we discuss the *harmless* character of the divergence introduced by our encoding. In

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Section 5, we present our conclusions. Appendix A shows the code of the auxiliary encoding functions. Appendix B contains the proofs of Proposition 4.3, Proposition 4.4, Theorem 4.1, and Theorem 4.2.

Related work. To show the expressiveness of the ambients, in Cardelli and Gordon (2000), an encoding of the π-calculus into the pure MAs is presented; Levi and Sangiorgi (2003) proposes an encoding of the π-calculus into Safe Ambients, and Zimmer (2003) defines an encoding of the π-calculus into the pure Safe Ambients.

Vice versa, there are some implementations of MAs in other formalisms for distributed computations.

In Cenciarelli et al. (2005), Gadducci and Monreale (2010), the target language consists of a graphical formalism, thus the encoding follows a completely different mechanism from the one we adopt. The only similarity with our work is that each ambient construct generates a sub-process separated from the processes generated by the ambient content. In our work, these sub-processes (encoding the content of an ambient and of the ambient itself) are linked by special channels that we will call coordinates.

The work in Fournet et al. (2000) presents an encoding of MAs into the Join calculus, a calculus that provides constructs for defining local, possibly nested environments, equipped with local rules. This kind of constructs allows the encoding of the ambient nesting structure by means of upper and down links between nested environments. The execution of a capability is then simulated by the changing of the links between environments and thus by changing the nested structure of the environments. Our target calculus does not have constructs for environments, thus our encoding is based on a different notion of links (coordinates) from children to parent ambients.

To the best of our knowledge, the only encoding of MAs into the π-calculus is proposed in Ciobanu and Zakharov (2007). Here, the basic idea is that each ambient, and its content, is encoded in a π-calculus subprocess; the simulation of capabilities is ruled by a distinguished π-calculus process that randomly chooses two sub-processes, checks the necessary conditions for a capability to be executed and, if they hold, the two sub-processes reproduce the execution of a capability. Our encoding does not rely on a process acting as a central capability execution controller.

In Brodo et al. (2003), we have proposed an encoding of the pure MA calculus into a version of the π-calculus where the application of the congruence rules is strictly controlled: Each inference rule of the operational semantics is equipped with side conditions which force the application of the congruence to reproduce the effects of the capabilities executions. Here, the main result was then to prove that the MA transition system (TS) is a proper subset of the π-calculus one.

The work in Phillips and Vigliotti (2008) shows that the MAs without communication, restriction, and open capability can solve the leader election problem. As a consequence, the MAs cannot be encoded into the π-calculus with separated choice (i.e. input and output cannot be mixed in the same choice), and without divergence. In fact, in our encoding, we can introduce loops and we do not use the separated choice. We rather use the ‘mixed’ choice.

The work in Gorla (2010) gives a set of requirements for an encoding from MAs to π-calculus. It requires an encoding to be compositional (the encoding of a compound term

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

must be expressed in terms of the encoding of its components), to have an operational correspondence (namely, that the computations of a source term must correspond to the computations of the encoded term, and vice versa), to be name invariant (the encoding should reflect all the name substitutions carried out in the source term), to be divergence reflecting (the encoding should avoid introducing infinite computations), and to be success sensitive (the encoding should reflect the behaviour with respect to a success). One of the main results in Gorla (2010) states that there is not an encoding of MAs into the asynchronous π-calculus which satisfies the abovementioned requirements. Our work does not contradict such a result, as we may introduce divergence. Our encoding rather represents a step in the direction of defining a translation as accurately as possible.

As we will see, our encoding function has a compositional style up to some preliminary syntactical settings. It may introduce loops, that we avoid by applying a run time support that allows to check if the structural conditions for executing the simulation of a capability hold. We prove the correctness of our encoding by relying on the notion of operational correspondence.

This means that our encoding processes can mimic all the capabilities that the source processes can perform, and only them. As we will see later, in Section 4, we encode the free ambient names as restricted names (that we will call *structural names*) over which we impose not to apply α-conversion. With this restriction, we can detect the presence of original ambient names, hence we could say that our encoding is success sensitive to the presence of the ambient names, up to syntactical restrictions.

2. The mobile ambients

We briefly recall the syntax and the semantics of the pure MAs, i.e. the version without communication primitives and variables.

Definition 2.1. Let n range over a numerable set of names $\mathcal{N}$. The set of MA processes $\mathcal{P}_{MA}$ (with metavariables $P, P', \dots$) and the set of capabilities $\mathcal{C}ap$ (with metavariable M) are defined below:

$$P ::= \mathbf{0} \mid M.P \mid (\nu n)P \mid P|P' \mid !P \mid n[P]$$
$$M ::= in\ n \mid out\ n \mid open\ n$$

Intuitively, the null process $\mathbf{0}$ does nothing. X is the process variable. The process $M.P$ executes the capability M and then behaves as P; $(\nu\ n)P$ defines P to be the scope of the name n; $P \mid P'$ may alternatively behave as P or as P' and the two sub-processes may also interact; $!P$ creates new parallel copies of the process P; $n[P]$ denotes the ambient n containing process P. The capability $in\ n$ allows an ambient enter ambient n; $out\ n$ allows an ambient exit ambient n; $open\ n$ destroys ambient n.

The semantics of the MAs is given by the rules in Table 1, and by the smallest congruence relation $\equiv$ satisfying rules in Table 2, up to α-congruence to identify processes that differ in the choice of bound names. The relation $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. We adopt the classical notion of free and bound names of process P, denoted as $fn(P)$ and $bn(P)$, respectively.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 1. *Semantic rules for mobile ambients.*

$n[in\ m.P\,\vert P']\vert m[R] \to m[n[P\,\vert P']\vert R]$	*In*	$m[n[out\ m.P\,\vert P']\vert R] \to n[P\,\vert P']\vert m[R]$	*Out*
$open\ n.P\,\vert n[P'] \to P\,\vert P'$	*Open*	$P \to P' \Rightarrow (\nu\ n)P \to (\nu\ n)P'$	*Res*
$P \to P' \Rightarrow n[P] \to n[P']$	*Amb*	$P \to P' \Rightarrow P\vert P'' \to P\vert P''$	*Par*
$P' \equiv P, P \to R, R \equiv R' \Rightarrow P' \to R'$	$\equiv$		

Table 2. *Structural congruence rules for mobile ambients.*

$P \equiv P$	$P \equiv P' \Rightarrow P' \equiv P$	$P \equiv P', P' \equiv R \Rightarrow P \equiv R$
$P \mid P' \equiv P' \mid P$	$(\nu\, n)(\nu\, m)P \equiv (\nu\, m)(\nu\, n)P$	$P \equiv P' \Rightarrow P\vert R \equiv P'\vert R$
$(\nu\, n)\mathbf{0} \equiv \mathbf{0}$	$(\nu\, n)(P \mid Q) \equiv P \mid (\nu\, n)Q,\ if\ n \notin fn(P)$	$P \equiv P \Rightarrow (\nu\, n)P \equiv (\nu\, n)P'$
$P \mid \mathbf{0} \equiv P$	$(\nu\, n)(m[P]) \equiv m[(\nu\, n)P],\ if\ n \neq m$	$P \equiv P' \Rightarrow n[P] \equiv n[P']$
$(P \mid P') \mid R \equiv P \mid (P' \mid R)$	$!P \equiv P \mid !P$, up to α-conversion	$P \equiv P' \Rightarrow M.P \equiv M.P'$

3. The π-calculus

We consider the polyadic version of the π-calculus (π for short).

Definition 3.1. Let $\mathcal{N}$ be a numerable set of names ranged over by $a, b, \ldots, z$, which will function as all communication channels, variables, and data values. The set of processes $\mathcal{P}_\pi$ (with metavariables $Q, Q', \ldots$) and the set of prefixes $\mathcal{A}$ (with metavariable π) are defined below:

$$\begin{array}{lll} Q & ::= & \mathbf{0} \mid \pi.Q \mid (\nu a)Q \mid Q|Q' \mid Q + Q' \mid [x = a]Q \mid [x \neq a]Q \mid A(\tilde{n}) \\ \pi & ::= & a(\tilde{x}) \mid \overline{a}\,(\tilde{b}), \end{array}$$

where $\tilde{b}$ and $\tilde{x}$ stand for tuples of variables and tuples of names pairwise different.

Roughly, the null process $\mathbf{0}$ cannot perform any action. The process $\pi.Q$ executes action π and then behaves as Q. The input $a(\tilde{x})$ allows an ordered tuple of names $\tilde{b}$, that has been received along channel a, to replace the tuple of placeholders $\tilde{x}$ following the same order; the output $\overline{a}\,(\tilde{b})$ allows the ordered tuple of names $\tilde{b}$ to be sent along channel a. Process $(\nu a)Q$ behaves as Q, where name a is local. We write $(\nu\tilde{a})$ for the sequence $(\nu a_1)\ldots(\nu a_n)$, where $a = a_1 \ldots a_n$. Process $Q|Q'$ independently executes Q and Q', and the two processes may also communicate. Process $Q + Q'$ non-deterministically behaves as Q or as Q'. The match process $[x = a]Q$ behaves as Q only if placeholder x will be substituted by name a, whereas $[x \neq a]Q$ behaves as Q only if placeholder x will be substituted by a different name from a. Each agent identifier A has a unique defining equation of the form $A(\tilde{x}) = Q$, where $\tilde{x} = fn(Q)$, and the names in $\tilde{x}$ are pairwise distinct. We would like to remark that the translating function, see Definition 4.1, adopts meaningful names for the process identifiers such as $Amb(\ldots)$, $In(\ldots)$, $Out(\ldots)$, $Open(\ldots)$, and $Opened(\ldots)$ to help the understanding of the target code. In other words, this choice makes more clear the presence of ambients and capabilities, and of dissolved ambients.

The semantics of the π-calculus is defined by the reduction rules in Table 3, and by the smallest congruence relation satisfying rules in Table 4, up to α-equivalence, that

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 3. *Reduction semantics for π-calculus.*

$P \to P'$	$\Rightarrow$	$P\|Q \to P'\|Q$	(R-PAR)
$P \to P'$	$\Rightarrow$	$(\nu\ n)P \to (\nu\ n)P'$	(R-RES)
$P \equiv Q, Q \to Q', Q' \equiv P'$	$\Rightarrow$	$P \to P'$	(R-STRUCT)
$(a(\tilde{x}).Q + Q') \,\|\, (\overline{a}(\tilde{b}).R + R') \to Q\{\tilde{b}/\tilde{x}\}\|R$			(R-COM)

Table 4. *Congruence rules for π-calculus.*

$(Q_1\|Q_2)\|Q_3 \equiv Q_1\|(Q_2\|Q_3)$	$Q_1\|Q_2 \equiv Q_2\|Q_1$	$Q\|\mathbf{0} \equiv Q$
$(Q_1 + Q_2) + Q_3 \equiv Q_1 + (Q_2 + Q_3)$	$Q_1 + Q_2 \equiv Q_2 + Q_1$	$Q + \mathbf{0} \equiv Q$
$Q(\tilde{a}) \equiv R[\tilde{a}/\tilde{x}]$, if $Q(\tilde{x}) = R$	$[x \neq y]Q \equiv \mathbf{0}$, if $x \neq y$	$[x = x]Q \equiv Q$
$(\nu\ n)(\nu\ m)Q \equiv (\nu\ m)(\nu\ n)Q$	$(\nu\ n)Q \equiv Q$, if $n \notin fn(Q)$	$(\nu\ n)\mathbf{0} \equiv \mathbf{0}$
$(\nu\ n)(Q_1\|Q_2) \equiv (\nu\ n)Q_1\|Q_2$, if $n \notin fn(Q_2)$	$\dfrac{Q_1 \equiv Q_2, Q_2 \equiv Q_3}{Q_1 \equiv Q_3}$	
$(\nu\ n)(Q_1 + Q_2) \equiv (\nu\ n)Q_1 + Q_2$, if $n \notin fn(Q_2)$		$Q \equiv Q$
$(\nu\ n)[u = v]\ P \equiv [u = v]\ (\nu\ n)P$, if $n \neq u$ and $n \neq v$		
$(\nu\ n)[u \neq v]\ P \equiv [u \neq v]\ (\nu\ n)P$, if $n \neq u$ and $n \neq v$	$\dfrac{Q_2 \equiv Q_1}{Q_1 \equiv Q_2}$	

identifies processes that differ in the choice of bound names. We would like to remark that, see Parrow (2001), the choice of a reduction semantics is based on the facts that the processes generated by our encoding function, see Definition 4.1, have no τ prefixes, all the sums are guarded and the match and mismatch constructs are guarded or can be evaluated, i.e. all the variables have been substituted with data values when the match and mismatch are evaluated.

The rule (R-PAR) says that if P reduces to P', then $P|Q$ can reduce to $P'|Q$. The rule (R-RES) allows a reduction to take place under a name restriction. The rule (R-STRUCT) allows to apply congruence rules, see Table 4, to derive transitions between congruent processes. Finally, rule (R-COM) allows an interaction on a channel a to take place between two processes put in parallel: one offering the input prefix and the other one offering the output prefix. If $\tilde{b}$ and $\tilde{x}$ are empty tuples, then a synchronisation takes place, otherwise there is a data communication: Each variable in the ordered tuple $\tilde{x}$ in Q' is substituted by the corresponding names in the tuple $\tilde{b}$, following the same order.

The transition relation $\to^*$ is the reflexive and transitive closure of the possible empty sequence of the transition relation $\to$.

Functions $fn()$, $bn()$, define the notions of free names and bound names of prefixes, respectively, and are defined as follows:

Kind	π	$fn(\pi)$	$bn(\pi)$
Output	$\overline{a}(\tilde{y})$	$\{a, \tilde{y}\}$	$\varnothing$
Input	$a(\tilde{y})$	$\{a\}$	$\{\tilde{y}\}$

The function $n()$, that collects all the names in a prefix, is defined as $n(\pi) = fn(\pi) \cup bn(\pi)$. The function $fn()$, $bn()$, and $n()$ are extended in the standard way to processes.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

4. The encoding

We aim to define a direct encoding from the MAs to the π-calculus, i.e. we would like to have the following property:
Let P be a MA process and $Q \equiv \mathcal{T}(P)$ its π-calculus encoding, see Definition 4.1, then if $P \to P'$, then $\exists\, Q'$ such that $Q \to^* Q'$ and $Q' \equiv \mathcal{T}(P')$.

To get this result, we have to face the problem that in the source language (MA) the effect of the execution of an open capability causes the fact that ambient boundaries dissolve; whereas in the π-calculus encoding, the mimicking of an open capability has the effect to make the involved process $Amb(\dots)$ evolve in a process $Opened(\dots)$. Hence, there is no more a syntactical correspondence between the source and the target language. To bypass the problem, we could introduce an equivalence relation for abstracting away from the presence of the forwarders, i.e. the $Opened(\dots)$ processes, as proposed in Brodo (2011). We discard this cumbersome solution as it also leads to a slightly weaker result. Instead, we adopt the solution in Bodei et al. (2013) that offers a simple theoretical presentation, and allows us to get a stronger result; thus, we introduce the ambients with brackets, where each bracket keeps trace of the execution of an open capability.

4.1. *Ambients within brackets*

We just extend the syntax of MA with the possibility of enclosing a process P within a pair of brackets:

$$P \;::=\; \cdots \;\mid\; (\!|P|\!)$$

making the presence of brackets inessential w.r.t the behaviour of the process. In order to do this, we introduce the additional structural congruence axioms:

$$(\!|(\nu\, n)P|\!) \equiv (\nu\, n)(\!|P|\!) \qquad\qquad P \equiv Q \Rightarrow (\!|P|\!) \equiv (\!|Q|\!)$$

Finally, we define the notion of *passive context* $\mathbb{C}$, to adjust the basic reduction rules to deal with the presence of an arbitrary number of balanced brackets.

$$\mathbb{C}, \mathbb{D}, \mathbb{E} \;::=\; \bullet \;\mid\; (\!|\mathbb{C}|\!) \;\mid\; \mathbb{C}|P \;\mid\; P|\mathbb{C}$$

and write $\mathbb{C}(P)$ to denote the process obtained by replacing the hole $\bullet$ in $\mathbb{C}$ with P. Thus, we add the suitable reduction rules, and we use the symbol $\to_p$ to denote the reductions between MA processes equipped with passive contexts:

$$\frac{}{\mathbb{D}(n[\,\mathbb{C}(in\, m P)\,]) \mid \mathbb{E}(m[\,R\,]) \to_p \mathbb{D}(\mathbf{0}) \mid \mathbb{E}(m[\,n[\,\mathbb{C}(P)\,] \mid R\,])}\ (\text{In})$$

$$\frac{}{m[\,\mathbb{D}(n[\,\mathbb{C}(out\, m P)\,])\,] \to_p n[\,\mathbb{C}(P)\,] \mid m[\,\mathbb{D}(\mathbf{0})\,]}\ (\text{Out})$$

$$\frac{}{\mathbb{C}(open\, n P) \mid \mathbb{D}(n[\,Q\,]) \to_p \mathbb{C}(P) \mid \mathbb{D}((\!|Q|\!))}\ (\text{Open}) \qquad \frac{P \to_p Q}{(\!|P|\!) \to_p (\!|Q|\!)}\ (\text{Brac})$$

4.2. *The encoding function*

The translation will assign a unique name (introduced under a restriction operator) to each ambient, even to each pair of brackets $(\!|\dots|\!)$. This helps to easily denote the position of

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 5. *The graphical representation of the encoding.*

$A = n[\underline{in\,n}.P_1 \mid m[out\,n.P_2]] \mid n[open\,m.P_3] \mid n[P_4]$ top $n[]^{(a,top)}$ $n[]^{(b,top)}$ $n[]^{(c,top)}$ $in\,n.P_1^{(a,top)}$ $m[]^{(d,a)}$ $open\,m.P_3^{(b,top)}$ $P_4^{(c,top)}$ $out\,n.P_2^{(d,a)}$ (1)	$B = n[n[P_1 \mid m[\underline{out\,n}.P_2]] \mid open\,m.P_3] \mid n[P_4]$ top $n[]^{(b,top)}$ $n[]^{(c,top)}$ $n[]^{(a,b)}$ $open\,m.P_3^{(b,top)}$ $P_4^{(c,top)}$ $P_1^{(a,b)}$ $m[]^{(d,a)}$ $out\,n.P_2^{(d,a)}$ (2)
$C = n[n[P_1] \mid m[P_2] \mid \underline{open\,m}.P_3] \mid n[P_4]$ top $n[]^{(b,top)}$ $n[]^{(c,top)}$ $n[]^{(a,b)}$ $m[]^{(d,b)}$ $open\,m.P_3^{(b,top)}$ $P_4^{(c,top)}$ $P_1^{(a,b)}$ $P_2^{(d,b)}$ (3)	$D = n[n[P_1] \mid P_2 \mid P_3] \mid n[P_4]$ top $n[]^{(b,top)}$ $n[]^{(c,top)}$ $n[]^{(a,b)}$ $P_2^{(b,top)}$ $P_3^{(b,top)}$ $P_4^{(c,top)}$ $P_1^{(a,b)}$ (4)

an ambient within the ambient hierarchy. For example, in the configuration $m[\ldots \mid n[\ldots]]$, we assign the name a (for ambient) to the ambient n and the name p (for parent) to the ambient m. The pair (a, p), associated to ambient n, says that it is univocally identified by a, and it lies within the ambient m, univocally identified by p. The pair (a, p) stands for the coordinates of the ambient: The first component records the information *who I am*, the second one records the information *where I am*. Our translating function will also introduce a special name, *top*, to identify the topmost ambient that contains all the other ones; and in $m[\ldots \mid n[\ldots]]$, the ambient m will have (p, top) as coordinates.

To give a graphical idea of our encoding, let us consider process $A \in \mathcal{P}_\pi$ in Table 5 (1), where we also depict the tree structures of the nested ambients: we assign the unique names a, b, c, d to the four ambients in process A, so that their complete coordinates result (a, top), (b, top), (c, top), (d, a), respectively. Note that all the sub-processes lying within an ambient share the same coordinates. We then let process A evolve to B by executing the $in\,n$ capability, see Table 5 (2), where the coordinates of the components involved in the capability execution have been modified consequently. When the capability $out\,n$ fires, B evolves to C and correspondingly, the coordinates of our encoding change as in Table 5 (3). Process C evolves to D by executing the *open m* capability that dissolves ambient m, and the coordinates change as in Table 5 (4).

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 6. *Definitions of $In(a, inn)$, $Out(a, outn)$, and $Open(a, openn)$.*

$In(a, inn)$	$=$	$\mathbf{Start(a, inn)}$ @ $\mathbf{Link1_{in}(a, pc, a_x, p_x, inn)}$ @ $\mathbf{Link2_{in}(a, p_{x1}, pc, a_x, p_x, pc1, inn)}$ @ $\mathbf{Link3_{in}(a, p_{x1}, pc, a_x, p_x, pc1, pc2, inn)}$@ $\mathbf{SimulateIn(a, p, pc, a_x, p_x, pc1, pc3, a_{x3}, p_{x3})}$
$Out(a, outn)$	$=$	$\mathbf{Start(a, outn)}$ @ $\mathbf{Link1_{out}(a, pc, a_x, p_x, outn)}$ @ $\mathbf{Link2_{out}(a, p_{x1}, pc, a_x, p_x, pc1, pc1, outn)}$ @ $\mathbf{SimulateOut(a, p_{x1}, pc, pc1, a_{x3}, p_{x3}, outn)}$
$Open(a, openn)$	$=$	$\mathbf{Start(a, openn)}$ @ $\mathbf{Link1_{open}(a, pc, a_x, p_x, openn)}$ @ $\mathbf{Link3_{open}(a, p_{x1}, pc, a_x, p_x, pc1, pc1, openn)}$ @ $\mathbf{SimulateOpen(a, p_{x1}, pc, pc1)}$
$Start(a, ch_{cap})$	$=$	$(\nu\ pc)\overline{ch_{cap}}(pc).pc(a_x, p_x).0$
$SimulateIn(a, p, pc, a_x, p_x, pc1, pc3, a_{x3}, p_{x3})$	$=$	$\overline{pc}(a_x, p_x).\overline{pc1}(a, a_x).\overline{pc3}(a_{x3}, p_{x3}).0$
$SimulateOut(a, p, pc, pc1, a_{x3}, p_{x3})$	$=$	$\overline{pc}(a_{x3}, p_{x3}).\overline{pc1}(a, p_{x3}).0$
$SimulateOpen(a, p, pc, pc1)$	$=$	$\overline{pc}(a, p).\overline{pc1}(a, p).0$

Definition 4.1. Given $P \in \mathcal{P}_{\mathcal{MA}}$, the encoding function $\mathcal{T} : \mathcal{P}_{MA} \rightarrow \mathcal{P}_\pi$ is defined by means of an auxiliary function $\mathcal{T}_{ma}$: $\mathcal{T}(P) \triangleq (\nu\, \tilde{m})\mathcal{T}_{ma}(P, top)|Amb(top)$, where $\tilde{m} = \{m \mid m \in fn(\mathcal{T}_{ma}(P, top)) \wedge m \in \{in_x, out_x, open_x \mid x \in fn(P)\}\} \cup \{top\}$. The function $\mathcal{T}_{ma} : \mathcal{P}_{MA} \times \mathcal{N} \rightarrow \mathcal{P}_\pi$ is defined as follows:

1.	$\mathcal{T}_{ma}(in\ n.P, a)$	$\triangleq$	$In(a, inn)@\mathcal{T}_{ma}(P, a)$
2.	$\mathcal{T}_{ma}(out\ n.P, a)$	$\triangleq$	$Out(a, outn)@\mathcal{T}_{ma}(P, a)$
3.	$\mathcal{T}_{ma}(open\ n.P, a)$	$\triangleq$	$Open(a, openn)@\mathcal{T}_{ma}(P, a)$
4.	$\mathcal{T}_{ma}((\nu\ n)P, a)$	$\triangleq$	$\mathcal{T}_{ma}(P[n_\nu/n], a)$
5.	$\mathcal{T}_{ma}(\mathbf{0}, a)$	$\triangleq$	$\mathbf{0}$
6.	$\mathcal{T}_{ma}(P \mid P', a)$	$\triangleq$	$\mathcal{T}_{ma}(P, a) \mid \mathcal{T}_{ma}(P', a)$
7.	$\mathcal{T}_{ma}(!P, a)$	$\triangleq$	$Ide(\tilde{n})$, *where* $Ide(\tilde{n}) = Ide(\tilde{n})|\mathcal{T}_{ma}(P, a) \wedge$ $\tilde{n} = fn(P)\ \wedge\ Ide$ *is a new name*
8.	$\mathcal{T}_{ma}(n[P], a)$	$\triangleq$	$(\nu\ b)\ (Amb(b, a, \tilde{n}) \mid \mathcal{T}_{ma}(P, b))$, *where* $\tilde{n} = \{inn, outn, openn\}$
9.	$\mathcal{T}_{ma}((\!|P|\!), a)$	$\triangleq$	$(\nu\ b)\ (Opened(b, a) \mid \mathcal{T}_{ma}(P, b))$

The definitions of the process constants $Amb(b, a, \tilde{n})$, $Amb(top)$, $In(a, inn)$, $Out(a, outn)$, and $Open(a, openn)$ are in Table 6 and in Table 7. The definition of the operator @ : $\mathcal{P}_\pi \times \mathcal{P}_\pi \rightarrow \mathcal{P}_\pi$ is in Table 8.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 7. *Definitions of* $Amb(a, p, \tilde{n})$, *and* $Opened(a, a')$. *For the sake of readability, the parameter lists of the process constants are deliberately not complete.*

$Amb(a, p, \tilde{n})$	$=$	$inn(pc_x).\overline{pc_x}(a, p).Busy(a, p, pc_x, in, \tilde{n})$
		$+$
		$outn(pc_x).\overline{pc_x}(a, p).Busy(a, p, pc_x, out, \tilde{n})$
		$+$
		$openn(pc_x).\overline{pc_x}(a, p).Busy(a, p, pc_x, open, \tilde{n})$
		$+$
		$a(pc_x).\overline{pc_x}(a, p).Busy(a, p, pc_x, nocap, \tilde{n})$
		$+$
		$a(pc_x, pc'_x).\overline{pc_x}(nomatch, nomatch).Amb(a, p, \tilde{n})$
$Opened(a, a')$	$=$	$a(pc_x).\overline{pc_x}(a', a').Opened(a, a')$
		$+$
		$a(pc_x, pc'_x).\overline{pc_x}(\tilde{a'}).Opened(a, a')$
$Busy(a, p, pc, cap, \tilde{n})$	$=$	$pc(a_{new}, f_{new}).([a_{new} = release]Amb(a, p, \tilde{n})$
		$+$
		$[a_{new} \neq release]Simulation(\tilde{a}_{new}, p_{new}, cap, \tilde{n}))$
		$+$
		$a(pc_x).\overline{pc_x}(busy, busy).Busy(a, p, pc, cap, \tilde{n})$
		$+$
		$a(pc_x, pc'_x).\overline{pc_x}(a, p).Busy(a, p, pc, cap, \tilde{n})$
$Simulation(\tilde{a}, p, cap, \tilde{n})$	$=$	$[cap = open]Opened(a, a_{new})$
		$+$
		$[cap \neq open]Amb(a, p, \tilde{n})$

Table 8. *Definition of the* @ *operator.*

$\mathbf{0}@Q$	$=$	Q	$R \mid R'@Q$	$=$	$R@Q \mid R'@Q$	$[a = x]R@Q$	$=$	$[a = x](R@Q)$
$\pi.R@Q$	$=$	$\pi.(R@Q)$	$R + R'@Q$	$=$	$R@Q + R'@Q$	$[a \neq x]R@Q$	$=$	$[a \neq x](R@Q)$
$(\nu\tilde{a})R@Q$	$=$	$(\nu\tilde{a})(R@Q)$	$Ide(\tilde{n})@Q$	$=$	$R@Q$, if $Ide(\tilde{n}) \equiv R$			

A detailed explanation for the treatment of names by our encoding function is provided. For each capability in the MA source process, the Rules 1, 2, 3, in Definition 4.1, introduce a name composed by the capability name (in, our, open) and by the ambient name which is the argument of the capability. This kind of names are only used as communication channels, i.e. they will not be sent as data, and we distinguish them as *structural names*. Together with the special name *top*, the structural names are closed under restriction by the function $\mathcal{T}(\dots)$ (in Definition 4.1) to force synchronisation. We impose that the *structural names* will not be α-converted. This restriction will be required later, when we discuss about compositionality.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Rule 4, of function $\mathcal{T}_{ma}(\dots)$ in Definition 4.1, deals with the encoding of restriction. With *original restricted names*, we identify those names that were already restricted in the MA source processes and that we keep separated from the others by adding the special symbol v as subscript. For example, consider the MA process $(\nu\, n)n[]$, then its encoding will be

$$(\nu\; top, a, inn_v, outn_v, openn_v)(Amb(top)|Amb(a, top, \tilde{n}).$$

To clarify the difference between the *structural names* and the *original restricted names*, we consider the MA process $n[]|(\nu n)n[]$, and its translation where the free name n has been translated with the three names $inn, outn, openn$, and the restricted name n has been translated as $inn_v, outn_v, open_v$:

$$(\nu\; inn, outn, openn)(\nu\; inn_v, outn_v, open_v)(\nu\; a, b, top)(Amb(top)|Amb(a, top, \tilde{n})|Amb(b, top, \tilde{n_v})).$$

The third kind of restricted names (the *generic names*) is defined by the auxiliary function $\mathcal{T}_{ma}(\dots)$, and it includes all the restricted names that are neither *structural names* nor *original restricted names*. Also, the encoding function introduces a set of free names: *in*, *out*, *open*, *nocap*, *nomatch*, *busy*, *release*; these names have been introduced to perform some technical checks on the status of the computation; more details on free names will be given in the following.
Some more comments are required for Rule 7 in Definition 4.1. We remark that we can guarantee the uniqueness for each process identifier introduced to translate the recursion operator $!P$ of the source language. We can use a third argument for the function $\mathcal{T}_{ma}(\dots)$ to keep track of the syntactical position with respect to the parallel operators. In fact, we can assume that during the encoding phase no structural congruence rule can be applied. For example, MA process $!P|!P$ will be translated as $(\nu\; \widetilde{names})(Amb(top)|Ide_{P0}(\tilde{n})|Ide_{P1}(\tilde{m}))$, where $Ide_{P0}(\tilde{n}) = Ide_{P0}(\tilde{n})|P$ and $Ide_{P1}(\tilde{m}) = Ide_{P1}(\tilde{m})|P$, for some set of names $\tilde{n}$ and $\tilde{m}$, with $\{\widetilde{names}\} \subseteq \{\tilde{n}\} \cup \{\tilde{m}\} \cup \{top\}$.

For the sake of readability, we prefer not to show the complete list of parameters of process identifiers where we only specify the coordinates and the channel names directly derived from the encoding: The complete parameter list can be easily derived from the process identifier definitions. For example, the complete parameter list of process $In(\dots)$ should include the free names *busy* and *release*: $In(a, inn, busy, release)$ (the same for $Out(\dots)$, $Open(\dots)$); the complete parameter list for process $Amb(top)$ is $Amb(top, busy, release)$; the complete parameter list of processes $Amb(\dots)$ is
$Amb(a, p, \tilde{n}, in, out, open, nocap, nomatch, release, busy)$. All the names that we added in the above parameter lists are the free names introduced by our encoding function. In particular, the names *busy* and *release* are used by $Busy(\dots)$, in Table 7, and by $Release1(\dots)$, $Release2(\dots)$, $Release3(\dots)$, in Table A5, respectively, to send the message that the trace will abort. The names *in*, *out*, and *open* are used by $Amb(\dots)$, in Table 7, to discriminate between the three types of capability. The name *nocap* is used by process $Amb(\dots)$, in Table 7, when it is involved in a trace which is not involved in a capability. The name *nomatch* is sent by $Amb(\dots)$ to inform the other sub-processes involved in the trace that the structural conditions are not satisfied.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

As mentioned before, hereafter, we assume that no α-renaming is applied on the restricted set of *structural names*. This assumption allows to show that our encoding is compositional with respect to the parallel and the non-deterministic operators, up to preliminarily settings. In fact, if we have $Q_1 = \mathcal{T}(P_1)$, $Q_2 = \mathcal{T}(P_2)$, and $Q_3 = \mathcal{T}(P_1|P_2)$, where $Q_1 = (\nu\tilde{n})(Amb(top)|\mathcal{T}_{ma}(P_1, top))$, $Q_2 = (\nu\tilde{m})(Amb(top)|\mathcal{T}_{ma}(P_2, top))$, and $Q_3 = (\nu\tilde{\omega})(Amb(top)|\mathcal{T}_{ma}(P_1|P_2, top))$, we can obtain the direct encoding of $P_1|P_2$ by setting $\mathcal{T}(P_1|P_2) = (\nu\tilde{z})(Amb(top)|\mathcal{T}_{ma}(P_1, top)|\mathcal{T}_{ma}(P_2, top))$, where $\tilde{z} = \tilde{n} \cup \tilde{m} = \tilde{w}$. Please note that no α-renaming has been applied on the set of names $\{\tilde{n}\} \cup \tilde{m} \cup \{top\}$.
For example, consider $P = a[in\ b]|b[]$, then $Q = \mathcal{T}(P) = (\nu\ ina, outa, opena, inb, outb, openb, top)\ (Amb(top)|(\nu\ c)(Amb(c, top, \tilde{a})|In(c, inb))|(\nu\ d)Amb(d, top, \tilde{b}))$,
$Q_1 = \mathcal{T}(a[in\ b]) = (\nu\ ina, outa, opena, top)(Amb(top)|(\nu\ c)(Amb(c, top, \tilde{a})|In(c, inb)))$, and $Q_2 = \mathcal{T}(b[]) = (\nu\ inb, outb, openb, top)(Amb(top)|(\nu\ d)Amb(d, top, \tilde{b}))$.
Following the above preliminary syntactic settings, we can easily obtain the encoding of P starting from Q_1 and Q_2.

The @ operator takes $Q_1, Q_2 \in \mathcal{P}_\pi$ as arguments, and returns a process that behaves as Q_1 and when it stops, behaves as Q_2, in other words @ substitutes all the occurrences of $\mathbf{0}$ in Q_1 with Q_2.

In MA processes, the ambient names carry out two different roles: denoting delimited spaces, the ambients, and allowing capabilities to act on them. Our encoding keeps the two roles separated: the original ambient name allows interactions to happen and we introduce a new name to univocally identify each ambient: in Definition 4.1, the Rule 8 introduces the unique name b for the identifier $Amb(b, a, \tilde{n})$, and the original name n becomes the suffix for the channel names $inn, outn, openn$. The construction of the channel names in Rules 1, 2, 3, in Definition 4.1, follows the same idea. Rule 9 introduces a $(\nu b)Opened(b, a)$ sub-process, corresponding to the MA $(\!|\ldots|\!)$ construct. As mentioned before, the introduction of passive contexts in MA is due to the fact that our encoding generates an $Opened(\ldots)$ process every time an open capability is simulated (see code in Table 7, the subprocess $Simulation(a, p, cap, \tilde{n})$, case $[cap = open]$); instead, in classical MAs, when P executes an open capability, $P \to P'$, in P' an ambient construct is just dissolved. The problem arises as the π-calculus process $Opened(\ldots)$ has not a syntactical correspondence in the classical MA source processes. With passive contexts, the execution of an open capability in MAs exactly produces a passive context which is encoded as an $Opened(\ldots)$ process. Please note that the first argument of $Opened(b, a)$ is the fresh name associated to the ambient that has been dissolved, see Definition 4.1 Rule 8. The process $Opened(a, a')$ (see the definition in Table 7) acts as a forwarder from the dissolved process $Amb(a, \ldots)$ to the one, $Amb(a', \ldots)$, that substitutes it (i.e. its previous parent ambient).

Relying on the notion of the *operational correspondence*, we will prove the correctness of our encoding: A derived π process can simulate a capability execution in a finite number of steps, and that all the steps that a derived π process may execute belong either to the simulation of some MA capability (that we will call *simulating trace*) or to a series of transitions returning to an existing state (that we will call *aborting trace*).

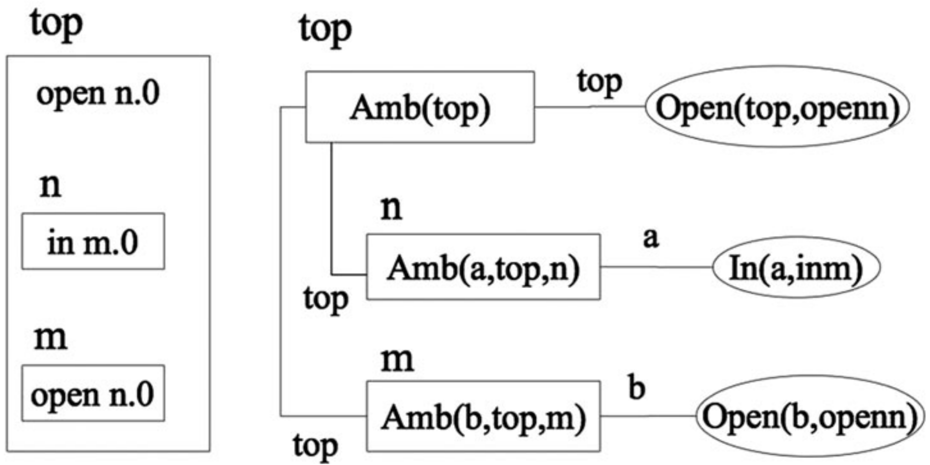
Example 4.1. Given the MA process $P = open\ n.\mathbf{0}|n[in\ m.\mathbf{0}]|m[open\ n.\mathbf{0}]$, it can execute one of the two open capabilities. We assign unique identifiers to ambients: a to ambient n, and

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

b to m. The translation works as follows: (1) for each ambient n, we generate a process $Amb(\dots)$; (2) for each sequential MA process, we generate a process identifier encoding all the capabilities, linked by the sequential operator dot. Applying our encoding, we get

$$\begin{aligned} \mathcal{T}(P) \equiv Q \;=\; & (\nu\ top, inn, openn, outn, inm, outm, openm) \\ & Amb(top)|Open(top, openn) && \{\text{ambient } top \text{ and top level capability}\} \\ | \;\; & (\nu\ a)(Amb(a, top, \tilde{n})|In(a, inm)) && \{\text{ambient n and its content}\} \\ | \;\; & (\nu\ b)(Amb(b, top, \tilde{m}) \mid Open(b, openn)) && \{\text{ambient m and its content}\} \end{aligned}$$

Graphically,



In the above picture, the left part of the graph, representing the MA process, is quite intuitive: The ambients n and m are at the topmost position in the nested ambient structure; on the right part of the graph, the π-calculus processes are represented as a rectangle when there is a translation of a MA ambient, and as an oval when there is a translation of a capability. Please note how the π-calculus processes are connected by (unique) channel names and thus reflecting the original nested ambients structure.

As it is shown in Table 6, the simulation of a capability begins with the $Start()$ routine that contacts the process $Amb(\dots)$ over which the capability acts. Then, processes $LinkN_{cap}(\dots)$ (with $N \in \{1, 2, 3\}$, and cap $\in \{in, out, open\}$) contact the other processes $Amb(\dots)$ involved in the capability simulation and check if the actual coordinates are compatible with the capability simulation (see processes $CheckN_{cap}(\dots)$, with $N \in \{1, 2, 3\}$, and cap $\in \{in, out, open\}$, in Tables A2–A4, A6 in Appendix A). If this is not the case, recovery routines (see processes $ReleaseN$, with $N \in \{1, 2, 3, 4\}$ in Table A5, in Appendix A), restore the initial situation. Instead, if the coordinates of the processes $Amb(\dots)$ involved verify the required structural conditions, then the computation ends with the final steps of the simulation in $SimulateIn(\dots)$, or $SimulateOut(\dots)$, or $SimulateOpen(\dots)$ (see process definitions in Table 6).

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

It is important to remark that our encoding does not rely on a process acting as a central capability execution controller, in particular, the process $Amb(top)$ only acts as the encoding of the topmost *parent ambient*, that does not have a syntactical correspondence in the MA processes. In fact, our encoding is based on a network of channels where each encoding of an ambient knows its unique name and the unique name of the encoding of its parent ambient. For the top level ambients in the MA processes, there is not such a *parent* ambient, and we force the encoding to add this further process $Amb(top)$: $\mathcal{T}(P) = (\nu\, \tilde{n})(Amb(top)|\mathcal{T}_{ma}(P, top))$. The behaviour of the process $Amb(top)$ can be checked in Table A6 in Appendix A, and it is possible to verify that it is a short version of the code of $Amb(\dots)$ processes, as $Amb(top)$ can not be the target of a capability (on channels *intop*, *outtop*, and *opentop*) and hence it is not equipped to act as the target ambient of a capability.

4.3. *Traces*

To simplify the proofs of our properties, we introduce the notion of traces, which we will denote with ξ.

A trace can be a *simulating trace*, when a capability is simulated (Definition 4.2), or can be an *aborting trace*, when no capability is simulated (Definition 4.3), i.e. a sort of back-tracking is executed and the final state of an *aborting trace* is congruent to its initial one.

Definition 4.2 (simulating trace). Let Q, Q' be two π-calculus processes, and let P, P' be two MA processes such that $P \rightarrow_p P'$. Let $\xi = Q \rightarrow Q_1 \rightarrow \cdots \rightarrow Q_t \rightarrow Q'$, with $t > 0$, be a computation, then we say that ξ is a simulating trace of Q if t is the minimum number such that $\mathcal{T}(P) = Q$, $\mathcal{T}(P') = Q'$, and $\nexists \hat{P} \in \mathcal{P}_{MA}$ s. t. $\mathcal{T}(\hat{P}) = Q_i\ \forall i \in \{1, \dots, t\}$.

Definition 4.3 (aborting trace). Let Q be a π-calculus process, and let P be a MA process, and let $\xi = Q \rightarrow Q_1 \rightarrow \cdots \rightarrow Q_t \rightarrow Q$, with $t > 0$, be a computation. We say that ξ is a aborting trace of Q if t is the minimum number such that $\mathcal{T}(P) = Q$, and $\nexists \hat{P} \in \mathcal{P}_{MA}$ s. t. $\mathcal{T}(\hat{P}) = Q_i\ \forall i \in \{1, \dots, t\}$.

We will write $Q \rightarrow^*_\xi Q'$ to denote the execution of all and only the transitions corresponding to trace ξ, and with $Q \rightarrow_\xi Q'$, we refer to a generic transition of the trace ξ.
We identify a strict subset of the π-calculus processes, that we denote as A_π, composed by processes that are a parallel composition of: $Amb(\dots)$, $In(\dots)$, $Out(\dots)$, $Open(\dots)$, $Amb(top)$, and $Opened(\dots)$.

Formally,

$$\begin{aligned} A_\pi \;=\; & \{Amb(top), Amb(a, p, \tilde{n}), In(b, in_x), Out(c, out_x), Open(d, open_x), Opened(e, f)\ | \\ & a, p, b, c, e, f \in \mathcal{N} \cup \{top\} \wedge x \in \mathcal{N} \wedge \tilde{n} \in \{(in_x, out_x, open_x) | x \in \mathcal{N}\}\} \end{aligned}$$

The set $A\pi$ identifies a super-set of the π-calculus processes encoding the MA processes. Of course, the above definition could be refined in a way that we could prove the following: $Q \in A\pi$ iff $\exists P \in \mathcal{P}_{MA}$ such that $\mathcal{T}(P) = Q$. This result requires a cumbersome definition that will be not applied in the rest of the work instead we will use the result in the following Proposition 4.1.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Proposition 4.1. Let P be a MA process, then $\mathcal{T}(P) \in A\pi$.

Proof. The proof directly derives from the definition of the encoding function $\mathcal{T}()$ in Definition 4.1. The most interesting cases are:

— rules 1, 2, and 3: Each capability is encoded by a process of the form $In(\dots)$, $Out(\dots)$, and $Open(\dots)$, respectively, and the first parameter of each of the previous processes is the actual name identifying the process encoding the ambient that contains the capability;
— rule 8: A new name, say b, is introduced to be associated to the new ambient and it is recorded, together with the previous name identifying the parent ambient, say a, in the process $Amb(b, a, \dots)$;
— rule 9: A *bracket* $(\!|\,|\!)$, (recall the MAs with brackets in Section 4.1), recording that an ambient has been dissolved, is encoded by a forwarder process $Opened(b, a)$ that just redirects the requests on channel b to channel a. □

The processes directly generated by our encoding are in A_π, whereas their derivatives may not be in $A\pi$, but, as we will show in Theorem 4.1, if $\mathcal{T}(P) \equiv Q \in A\pi$, and $Q \rightarrow Q'$, then there exist $Q'' \in A\pi$ such that $Q' \rightarrow^* Q''$.

To help this intuition, Figure 1 shows the relationship between the MA TS and the π-calculus TS. In the figure, the MA states $S1$, $S2$, $S3$, $S4$, $S5$, $S6$ have a direct correspondence in the π-calculus TS: $\mathcal{T}(Si) = Si'$, with $i \in [1, 6]$. Please note that where the MA needs only one transition to move from one state to another by executing a capability, the π-calculus needs a number of transitions, i.e. a simulating trace. Not all the intermediate states of the simulating traces are shown. In the π-calculus TS, there are also other kinds of computation paths, where the single transitions of two (or more) different traces are interleaved, shown in grey. Along these paths, none of the intermediate states have a direct correspondence with the MA states (we have depicted this situation with grey clouds). In the π-calculus, TS an aborting trace ξ is also depicted, starting and ending in state $S5'$.
Now, we can precisely identify the first transition of a trace, as stated by the following proposition.

Proposition 4.2. The first transition of a trace ξ is always generated by a communication between a sub-process of the form: $In(\dots)$, $Out(\dots)$, $Open(\dots)$, and a process of the form $Amb(\dots)$.

Proof. The proof directly derives from Definition 4.2, and Definition 4.3. A trace is defined as a computation of a process $Q = \mathcal{T}(P)$, with $P \in \mathcal{P}_{MA}$, and, by Proposition 4.1, we have $Q \in A_\pi$. Thus, by definition of the set A_π, the only possible interaction for a process in A_π is between a sub-process of the form: $In(\dots)$, $Out(\dots)$, $Open(\dots)$, and a process of the form $Amb(\dots)$, on a channel of the form in_x, out_x, $open_x$, respectively, for some x. □

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

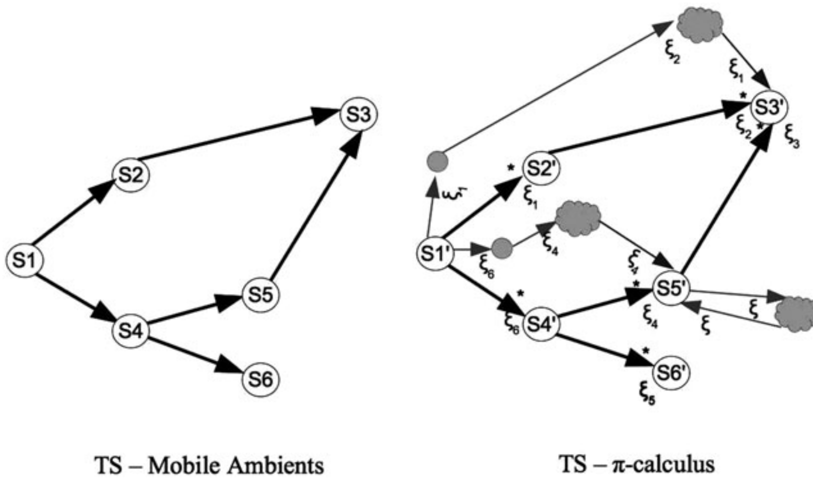


Fig. 1. A graphical comparison between the MA TS and the π-calculus TS, where $Si' = \mathcal{T}(Si)$, for $i \in [1, 6]$.

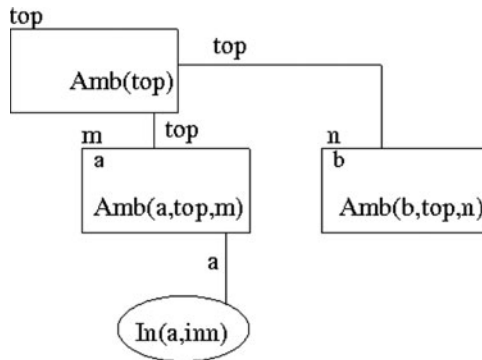


Fig. 2. The graphical representation of the encoding of the process P.

In the following Example 4.2 we will show a complete execution of a simulating trace, and to help the intuition, we graphically outline the temporal creation of new channel names used to link the sub-processes involved in the trace in order to complete the capability simulation.

Example 4.2. We consider the simple MA process $P = m[in\ n]|n[]$, where the ambient m has the capability to enter the ambient n. The translation of the process P is $\mathcal{T}(P) = (\nu\ top, a, b, \tilde{m}, \tilde{n})(Amb(top)|Amb(a, top, \tilde{m})|In(a, inn)|Amb(b, top, \tilde{n}))$. To simplify the presentation, we will refer to π-calculus processes encoding ambients and capabilities as *ambients* and *capability*, respectively.

In Figure 2, we give a graphical representation of $\mathcal{T}(P)$, and we use the notation already adopted in Example 4.1: A square for a π-calculus process encoding an ambient, and an oval for a π-calculus process encoding a capability. In this simulation, four sub-processes are involved: the one encoding the 'in' capability ($In(a, inn)$), and the three ones encoding

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

the ambients involved (the ambient that moves, $Amb(a, top, \tilde{m})$, the receiving ambient, $Amb(b, top, \tilde{n})$, and the ambient $Amb(top)$ containing the previous two). Please note that in this case the two ambients, $Amb(a, top, \tilde{m})$ and $Amb(b, top, \tilde{n})$, have a common parent ambient $Amb(top)$; in general, we can have two different parent ambients, for each one of the two involved processes of type $Amb(\dots)$, and this is a cause for a trace to evolve in an aborting trace.

In Table 9, we show the trace execution performed by the process $\mathcal{T}(P)$. To simplify the presentation, we use the restriction operator $(\nu\ \tilde{ch_s})$, where $\tilde{ch_s}$ stands for one or more of the private channel names introduced along the trace computation: *pc*, *pc*1, *pc*2, and *p*3. In Figure 3, we outline the creation of the previous four private channel names used to link all the ambients which make up the environment where the capability acts. For the sake of legibility, here and later on in the paper, we will label each transition with the input and the output prefixes taking part in it, whereas, according to the semantics in Table 3, there should be no labels.

To simplify the execution, in our example, no $Opened(\dots)$ sub-processes are involved, thus the trace execution needs the minimum number of transitions, 11, to complete the *in* simulating trace.

— In the first transition, from state 1 to state 2, a private channel *pc* is created by $In(a, inn)$ and sent to process $Amb(b, top, \tilde{n})$, see picture (a) in Figure 3.
— In the second transition, from state 2 to state 3, process $Amb(b, top, \tilde{n})$ sends back to $In(a, inn)$ its coordinates along the private channel *pc*.
— In the third transition, from state 3 to state 4, a private channel *pc*1 is created by $In(a, inn)$ and sent to process $Amb(a, top, \tilde{m})$, see picture (b) in Figure 3.
— In the fourth transition, from state 4 to state 5, process $Amb(a, top, \tilde{m})$ sends back to $In(a, inn)$ its coordinates along the private channel *pc*1.
— In the fifth transition, from state 5 to state 6, some checks on the received coordinates are performed by $In(a, inn)$. If all the checks succeed, $In(a, inn)$ creates a private channel *pc*2 and sends it to $Amb(top)$, see picture (c) in Figure 3.
— In the sixth transition, from state 6 to state 7, the process $Amb(top)$ sends back to $In(a, inn)$ its coordinates along the private channel *pc*2.
— In the seventh transition, from state 7 to state 8, a private channel name *pc*3 is created by $In(a, inn)$ and sent to process $Amb(top)$, see picture (d) in Figure 3.
— In the eighth transition, from state 8 to state 9, process $In(a, inn)$ performs some checks to verify that $Amb(a, top, \tilde{m})$ and $Amb(b, top, \tilde{n})$ have a common parent ambient (otherwise the simulation trace would turn out to be an aborting trace), and the parent ambient, that is in this case process $Amb(top)$, sends its coordinates along the private channel *pc*3, see picture (e) in Figure 3.
— In the ninth transition, from state 9 to state 10, process $In(a, inn)$ sends the new coordinates (that in this case are useless) to $Amb(b, top, \tilde{n})$ along the private channel *pc*, and closes the communication with $Amb(b, top, \tilde{n})$, see picture (f) in Figure 3.
— In the tenth transition, from state 10 to state 11, process $In(a, inn)$ sends the new coordinates to $Amb(a, top, \tilde{m})$ along the private channel *pc*1, and closes the communication with $Amb(a, top, \tilde{m})$, see picture (g) in Figure 3.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table 9. *A complete execution of a in simulating trace.*

1. $(\nu\ \tilde{ch_s})(Amb(top) \mid Amb(a, top, \tilde{m}) \mid \mathbf{In(a, inn)} \mid \mathbf{Amb(b, top, \tilde{n})})$
$\rightarrow\ (\nu\ pc)\langle\overline{inn}(pc), inn(pc_x)\rangle$

2. $(\nu\ \tilde{ch_s})(Amb(top) \mid Amb(a, top, \tilde{m}) \mid \mathbf{pc(a_x, p_x)}.Link1_{in}(a, pc, a_x, pc_x, inn)@Link2_{in}(\dots)@$
$Link3_{in}(\dots)@SimulateIn(\dots) \mid \overline{\mathbf{pc}}\mathbf{(b, top)}.Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle\overline{pc}(b, top), pc(a_x, p_x)\rangle$

3. $(\nu\ \tilde{ch_s})(Amb(top) \mid \mathbf{Amb(a, top, \tilde{m})} \mid \mathbf{Link1_{in}(a, pc, b, top, inn)}@Link2_{in}(\dots)@$
$Link3_{in}(\dots)@SimulateIn(\dots) \mid Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ (\nu\ pc1)\langle\overline{a}(pc1), a(pc_x)\rangle$

4. $(\nu\ \tilde{ch_s})(Amb(top) \mid \overline{\mathbf{pc1}}\mathbf{(a, top)}.Busy(a, top, pc1, nocap, \tilde{m}) \mid$
$\mathbf{pc1(a_{x1}, p_{x1})}.[a_{x1} = busy]Release1(a, p_{x1}, pc, inn)@Inn(a, inn)$
$+[a_{x1} \neq busy]Check1_{in}(a, p_{x1}, pc, b, top, pc1, a_{x1}, p_{x1}, inn)@Link2_{in}(\dots)@$
$Link3_{in}(\dots)@SimulateIn(\dots) \mid$
$Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle\overline{pc1}(a, top), pc1(a_{x1}, p_{x1})\rangle$

5. $(\nu\ \tilde{ch_s})(\mathbf{Amb(top)} \mid Busy(a, top, pc1, nocap, \tilde{m}) \mid$
$\mathbf{Link2_{in}(a, top, pc, b, top, pc1, inn)}@Link3_{in}(\dots)@SimulateIn(\dots) \mid Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle(\nu\ pc2)\overline{top}(pc2), top(pc_x)\rangle$

6. $(\nu\ \tilde{ch_s})(\overline{\mathbf{pc2}}\mathbf{(top, top)}.Busy(top, pc2) \mid Busy(a, top, pc1, nocap, \tilde{m}) \mid$
$\mathbf{pc2(a_{x2}, p_{x2})}.[a_{x2} = busy]Release2(a, top, pc, pc1, inn)@Inn(a, inn)$
$+[a_{x2} \neq busy]Check2_{in}(a, top, pc, b, top, pc1, pc2, a_{x2}, p_{x2}, inn)@$
$Link3_{in}(\dots)@SimulateIn(\dots) \mid$
$Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle\overline{pc2}(top, top), pc2(a_{x2}, p_{x2})\rangle$

7. $(\nu\ \tilde{ch_s})(\mathbf{Busy(top, pc2)} \mid Busy(a, top, nocap, \tilde{m}) \mid \mathbf{Link3_{in}(a, top, pc, b, top, pc1, pc2, inn)}@SimulateIn(\dots) \mid$
$Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle\overline{top}(pc3, pc2), top(pc_x, p'_x)\rangle$

8. $(\nu\ \tilde{ch_s})(\overline{\mathbf{pc3}}\mathbf{(top, top)}.Busy(top, pc2) \mid Busy(a, top, pc1, nocap, \tilde{m}) \mid$
$\mathbf{pc3(a_{x3}, p_{x3})}.([a_{x3} = busy]Release3(a, top, pc, pc1, pc2, inn)@Inn(a, inn)$
$+[a_{x3} \neq busy]Check3_{in}(a, top, pc, b, top, pc1, pc2, pc3, a_{x3}, p_{x3}, inn)@$
$SimulateIn(\dots)) \mid$
$Busy(b, top, pc, in, \tilde{n}))$
$\rightarrow\ \langle(\overline{pc3}(top, top), pc3(a_{x3}, p_{x3})\rangle$

9. $(\nu\ \tilde{ch_s})(Busy(top, pc2) \mid Busy(a, top, pc1, nocap, \tilde{m}) \mid \mathbf{SimulateIn(a, top, pc, b, top, pc1, pc2, top, top)} \mid \mathbf{Busy(b, top, pc, in, \tilde{n})})$
$\rightarrow\ \langle\overline{pc}(b, top), pc(a_{new}, p_{new})\rangle$

10. $(\nu\ \tilde{ch_s})(Busy(top, pc2) \mid \mathbf{Busy(a, top, pc1, nocap, \tilde{m})} \mid \overline{\mathbf{pc1}}\mathbf{(a, b)}.\overline{pc2}(top, top).\mathbf{0} \mid Amb(b, top, \tilde{n}))$
$\rightarrow\ \langle\overline{pc1}(a, b), pc1(a_{new}, p_{new})\rangle$

11. $(\nu\ \tilde{ch_s})(\mathbf{Busy(top, pc2)} \mid Amb(a, b, \tilde{m}) \mid \overline{\mathbf{pc2}}\mathbf{(top, top)}.\mathbf{0} \mid Amb(b, top, \tilde{n}))$
$\rightarrow\ \langle\overline{pc2}(top, top), pc2(a_{new}, p_{new})\rangle$

12. $(\nu\ \tilde{ch_s})(Amb(top) \mid Amb(a, b, \tilde{m}) \mid \mathbf{0} \mid Amb(b, top, \tilde{n}))$

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

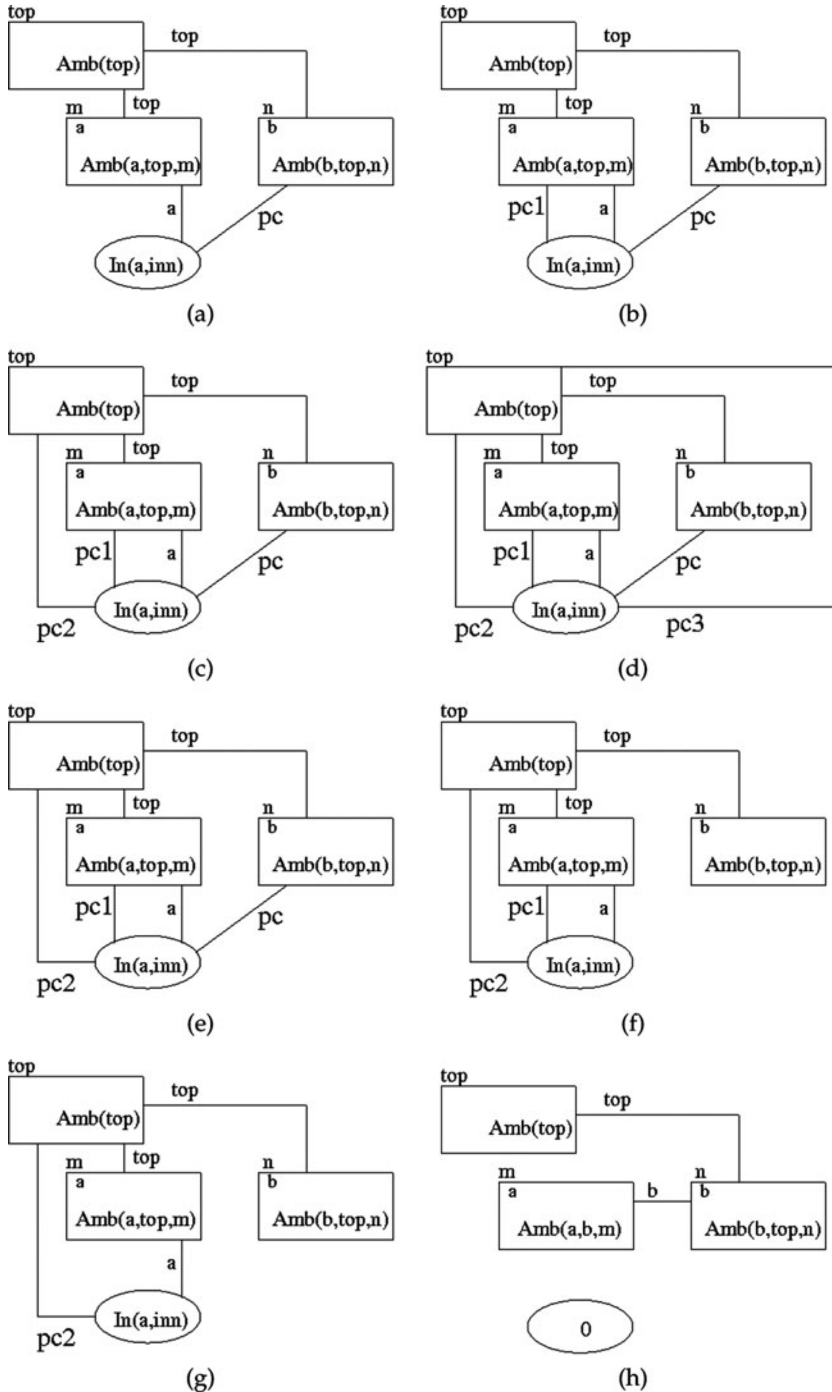


Fig. 3. The creation and the use of private channels during an 'in' capability simulation.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

— In the eleventh transition, from state 11 to state 12, process $In(a, inn)$ sends the new coordinates (that in this case are useless) to $Amb(top)$along the private channel $pc2$, and closes the communication with $Amb(top)$, see picture (h) in Figure 3.

Please note that, in the previous example, in all the trace transitions, the process $In(a, inn)$ is always involved. As we will see later, we can then detect all the transitions related to a specific trace, by only considering the process encoding the capability involved. The next proposition proves two different facts:

i. When more traces interleave their transitions, no deadlock occurs and the computation of each single trace is not altered, i.e. each trace can either successfully terminate or, if the computation cannot proceed, aborts itself (making backtracking). The last case happens when either the required structural conditions are not present, or the needed processes are involved in the computation of other traces.
ii. Whenever a derivative of process $Q \in A\pi$ executes a transition, then that transition can only be part of a trace.

The proofs of the following Proposition 4.3, and Proposition 4.4 are in the Appendix B.

Proposition 4.3 (no trace interference).
Let $P \in \mathcal{P}_{MA}$ be a MA process and let $\mathcal{T}(P) \equiv Q$ be its encoding. If $Q \to^* Q'$, then $\exists$ $Q'' \in A\pi$ such that $Q' \to^* Q''$ and $\exists$ $n \geqslant 1$ traces $\xi_1, \ldots, \xi_n$ such that $Q \to^*_{\xi_1} \cdots \to^*_{\xi_n} Q''$.

Previously, we have proved that all the traces start with an interaction between processes of the form $In(\ldots)$, or $Out(\ldots)$, or $Open(\ldots)$, and processes of the form $Amb(\ldots)$. In the following Proposition 4.4, we prove some further trace properties that allows one to precisely detect the execution of a trace: how to distinguish the transition of a trace from the transition of another one, and how to detect the last transition of a trace.

Proposition 4.4 (trace properties). Let P be a MA process, and let ξ be a trace executed by the process $Q = \mathcal{T}(P)$, then

1. the sub-process of the form $In(\ldots)$, $Out(\ldots)$, $Open(\ldots)$, involved in the first transition of a trace ξ, will be involved in all the transitions of ξ. We will call this sub-process the sub-process characterising the trace ξ;
2. in the last transition of a trace ξ: $Q \to_\xi Q'$, the sub-process C in Q, characterising the trace, in Q' will evolve into a composition of sub-processes $In(\ldots)$, $Out(\ldots)$, $Open(\ldots)$, $Amb(\ldots)$, $Opened(\ldots)$, and $\mathbf{0}$.
3. the number of the transitions of a trace is limited.

4.4. *Operational correspondence*

We rely on the notion of the operational correspondence to prove the correctness of our encoding. To this end, the following Theorem 4.1 proves that whenever a MA process executes a capability, then its π-calculus encoding can execute a series of transitions that mimic the capability; Theorem 4.2 proves that whenever a translating π-calculus process performs a transition, this is part of a computation trying to simulate a capability. The proofs are in the Appendix B.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Theorem 4.1 (operational correspondence 1). Let P be a MA process, and let $Q \equiv \mathcal{T}(P)$ be its encoding. If $Q \rightarrow^* Q'$, then $\exists\ P' \in \mathcal{P}_{MA}$, and $Q'' \in A_\pi$ such that $P \rightarrow_p^* P'$, $Q' \rightarrow^* Q''$, and $Q'' \equiv \mathcal{T}(P')$.

In the following, we will use the classical notion for contexts: $\mathcal{C}\{.\}$ is any process with a *hole* that will be filled by process P when we write $\mathcal{C}\{P\}$. We write $\mathcal{C}\{.\}$ both for MA and π processes. It will be clear from the context whether we refer to a MA or a π process.

Theorem 4.2 (operational correspondence 2).
Let P be a MA process, if $P \rightarrow_p^* P'$, then there exists a π-calculus process Q' such that $Q = \mathcal{T}(P) \rightarrow^* Q'$ and $\mathcal{T}(P') \equiv Q'$.

4.5. *Run time supports to avoid loops*

In this section, we will discuss how the divergence introduced by our encoding is, in a practical sense, harmless, as it can easily be detected and set apart.

We will consider two alternative methods for avoiding aborting traces. The first method consists of a dynamic check over the process coordinates. Then, we will consider a second alternative method based on a lookahead technique applied on a finite set of transition steps. The first method (dynamic check) is more efficient from a computational viewpoint, whilst the second method (lookahead) exploits the properties of traces which we have shown in this paper.

Checking process coordinates.
We can avoid aborting traces by dynamically inspecting the syntax of the processes that will be involved in the computation of a trace. This check can be fulfiled by inspecting the parameters of those processes involved in the execution of a trace. Please remember that the first transition of a trace is always performed by one process of the form $In(\dots)$, or $Out(\dots)$, or $Open(\dots)$, and a process of the form $Amb(\dots)$, see Proposition 4.2. Also, we have shown that along the trace more than one process of the form $Amb(\dots)$ and zero or more processes of the form $Opened(\dots)$ can be involved. In any case, by checking the coordinates of the processes involved in the first transition of a trace, before executing it, it is possible to exactly detect all the processes that will be involved in the trace and check their coordinates. This allows to understand if the trace would be an aborting or a simulating one before executing.

As an example, let us consider the following process, where for simplicity we omit the restriction of the names:

$In(a, inn)|Amb(a, c, \tilde{m})|Amb(b, c, \tilde{n})|Amb(c, top, \tilde{s})|Amb(d, b, \tilde{n})$. This process reflects the following hierarchical structure (we use the ambient notation for an easier insight) where each ambient is decorated with the unique names assigned by the encoding:

$s[\ m[in\ n]^a \mid n[\ n[\ ]^d]^b]^c$.

Now, we can inspect the processes parameters to verify that the interaction between processes $In(a, inn)$ and $Amb(b, c, \tilde{n})$ will be the first transition of a trace that respects the structural conditions for simulating a *in* capability (hence it will be a simulating trace). In fact, the ambient a and b share the same parent ambient c.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Differently, the interaction between processes $In(a, inn)$ and $Amb(d, b, \tilde{n})$ would generate a trace that does not fulfil the structural conditions required (and hence it would be an aborting trace). In fact, the ambients d and a have two distinct parent ambients: b and s, respectively. This kind of checks can be performed at execution time, by using a run-time support which implements the analysis that we have described.

In the above example, no $Opened(\dots)$ processes have been considered. However, it is always possible to verify the nature of a trace, i.e. simulating or aborting, even when $Opened(\dots)$ processes are involved. Of course, the presence of $Opened(\dots)$ processes causes an increased computation cost due to the further controls required to follow the chains of *expired* names formed by the $Opened(\dots)$ processes.

We summarise all the possible situations in Table A1: The first two columns contain the two processes that interact to initiate the trace; the third column contains a second $Amb(\dots)$ process that is involved later in the trace; the fourth column contains the $Opened(\dots)$ chain possibly involved in the trace; the fifth column says if the structural conditions are satisfied (simulating trace) or not (aborting trace); the sixth column says if the trace is a simulating or an aborting one (depending on the satisfaction of the structural conditions in the previous column). Let us now explain the interpretation of the first four rows of the table. In the first row, the two involved $Amb(\dots)$ processes share the same parent ambient, thus the structural condition are satisfied. In the second row, the process $In(a_1, inn)$ keeps a not updated name of the ambient where is lying. In this case, if a chain of $Opened(\dots)$ processes connecting the name a_1 to the name a_2 exists, then the structural conditions are still verified. In the fourth row, two chains of $Opened(\dots)$ processes connecting the names b_1 and b_2 to the same name do not exist, thus the trace is aborting. The interpretation for the other rows is similar.

The formal proof for the results in Table A1 can be done by considering all the possible kinds of traces, and by showing all the possible structural conditions that can be found during the execution of a trace. Thus, the proof would be similar to the proof of Proposition 4.3.

Lookahead technique.

The results in Propositions 4.2 and 4.4 allow to prove that it is always possible to check the execution of the encoding π-calculus processes: In each state of the computation, we can determine how many and which traces are running. In particular, by Proposition 4.4 point 1, each trace is characterised by a sub-process of the form $In(\dots)$, $Out(\dots)$, $Open(\dots)$; this sub-process participates in each transition of the trace. Proposition 4.4, point 3 states that the number of transitions in each trace is limited, in fact, the number of transitions of each trace depends on the capability simulated, on the type of release routine that is possibly executed, and on the number of $Opened(\dots)$ processes involved. Please recall the Example 4.2 where we show the complete execution of a trace simulating a *in* capability. In the following, we use the notation $\rightarrow^k$, with $k \geqslant 0$, as a shorthand for k transition steps. More formally, in case of a simulating trace, we can claim that if we have two MA processes P, P' such that $P \not\equiv P'$, and $\mathcal{T}(P) \rightarrow^k \mathcal{T}(P')$, where $k = 11 + 2N$, or $k = 8 + 2N$, with N is the number of $Opened(\dots)$ processes involved in the computation, then $P \rightarrow_p P'$. This property can be proved by exploit Proposition 4.4 and Theorem 4.1, here we also need

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

to count the number of transition steps for each sub-case. In the case of an aborting trace, we need to introduce the set TS(Aπ) to denote the TS generated by π-calculus processes in Aπ. Please recall that we have defined the set Aπ as an over-approximation of the set of processes generated by our encoding. Now, we can identify the aborting traces, i.e. those generating divergence, in order to prune them from the TS(Aπ): If we have a MA process P such that $\mathcal{T}(P) \rightarrow Q \rightarrow^{k-1} \mathcal{T}(P)$, where $k = 12+2N$, or $k = 11+2N$, or $k = 8+2N$, or $k = 5+2N$, where N is the number of $Opened(\dots)$ processes involved in the computation, then we can eliminate the loop by defining $\mathrm{TS}(\mathrm{A}\pi) = \mathrm{TS}(\mathrm{A}\pi) \setminus \mathcal{T}(P) \rightarrow Q$.

As before, the previous property can be proved by exploit Proposition 4.4 and Theorem 4.1, here we also need to count the number of transition steps. It is important to remark that it is possible to perform the above checks along the computation of the π-calculus processes, and hence it suggests an effective run-time method to make the divergence of our encoding harmless. In fact, as soon as the run-time support detects a loop, the first transition of that loop can be dropped from the actual state of the TS. More sophisticated checks could be implemented: When the run-time support detects an aborting trace, then the interaction between the two initial involved processes can be labelled as forbidden until some changes have occurred in the *ambient nested structure* regarding the two above processes.

5. Conclusion and future work

We have proposed an encoding of the pure MAs into the polyadic π-calculus with match and mismatch. We have encoded each ambient with a process identifier which shares a private channel with the π-processes encoding its content. Each π-process encoding an ambient also keeps trace of its position within the original ambient hierarchy by sharing a second private name with its parent ambient. The two names keep the two basic notions of an ambient separated: *Who I am*, and *Where I am*. The simulation of a capability execution involves at least three processes: One encoding the capability, one encoding the ambient over which the capability acts, and one encoding the ambient where the capability is lying. If all the conditions required for the execution hold, i.e. if all the involved sub-processes share the right channel names reflecting the correct hierarchical ambient structure, the simulation ends successfully, otherwise the simulation aborts and no side effects are produced. In the last case, our encoding introduces loops.

Our proposal represents an attempt to define an encoding of MA in the π-calculus as precisely as possible. The main limitation is due to the possible divergence it introduces. Nevertheless, we have also discussed how to check the execution of the traces in order to make divergence detectable. We are confident that our technique could be applied for translating other membrane calculi to π-calculus, such as Brane calculi (Cardelli 2005) and P Systems (Pǎun 2000).

We have also introduced the mechanism of forwarding processes, i.e. every time a open capability is simulated and an ambient name is disabled, we substitute the old process encoding the ambient $(Amb(a,b,\tilde{n}))$, with a process $Opened(a,b)$ that keeps trace of the fact that all the requests on channels a must be redirected to channels b.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

This mechanism has also been used for the formulation of an open multiparty interaction calculus, the `link`-calculus (Bodei et al. 2013), whose expressiveness has been proved by the definition of an easy and direct encoding of the MAs.

The forwarding mechanism is also a way to keep track of the history of the dissolution of ambients, and we are interested in applying it in a biological context where the code could keep track of the moment a membrane disappears. The encoding technique we have applied suggests that locality, which is an important aspect of real distributed systems, also of biological systems, can be modelled as a third component in each interaction. See, for example, a preliminary work in this direction (Bodei et al. 2014).

Acknowledgements

We would like to thank Roberto Bruni and Pierpaolo Degano for their hints and valuable ideas, and the anonymous referees for their helpful comments and suggestions.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Appendix A. Auxiliary Encoding Functions

Table A1. *All the possible situations for process coordinates to verify or not structural conditions.*

Starts the trace	Starts the trace	Involved in the trace	Opened(...) processes chain	Structure	Kind
$In(a, inn)$	$Amb(c, b, \tilde{n})$	$Amb(a, b, \tilde{m})$		Satisfied	Sim
$In(a_1, inn)$	$Amb(c, b, \tilde{n})$	$Amb(a_2, b, \tilde{m})$	$a_1 \neq a_2$, $\exists\ n \geqslant 0$ s. t. $Opened(a_1, p_1)\ldots Opened(p_n, a_2)$	Satisfied	Sim
$In(a, inn)$	$Amb(c, b_1, \tilde{n})$	$Amb(a, b_2, \tilde{m})$	$b_1 \neq b_2$, $\exists\ n, m \geqslant 0$ s. t. $Opened(b_1, p_1)\ldots Opened(p_n, p_j)$ $Opened(b_2, p'_1)\ldots Opened(p'_m, p_i)$ with $p_j = p_i$	Satisfied	Sim
$In(a, inn)$	$Amb(c, b_1, \tilde{n})$	$Amb(a, b_2, \tilde{m})$	$b_1 \neq b_2$, $\nexists\ n, m \geqslant 0$ s. t. $Opened(b_1, p_1)\ldots Opened(p_n, p_j)$ $Opened(b_2, p'_1)\ldots Opened(p'_m, p_i)$ with $p_j = p_i$	Not satisfied	Ab
$Out(a, outn)$	$Amb(b, c, \tilde{n})$	$Amb(a, b, \tilde{m})$		Satisfied	Sim
$Out(a_1, outn)$	$Amb(b, c, \tilde{n})$	$Amb(a_2, b, \tilde{m})$	$a_1 \neq a_2$, $\exists\ n \geqslant 0$ s.t. $Opened(a_1, p_1)\ldots Opened(p_n, a_2)$	Satisfied	Sim
$Out(a, outn)$	$Amb(b_2, c, \tilde{n})$	$Amb(a, b_1, \tilde{m})$	$b_1 \neq b_2$, $\exists n \geqslant 0$ s.t. $Opened(b_1, p_1)\ldots Opened(p_n, b_2)$	Satisfied	Sim
$Out(a, outn)$	$Amb(b_2, c, \tilde{n})$	$Amb(a_2, b_1, \tilde{m})$	$b_1 \neq b_2$, $\nexists\ n \geqslant 0$ s.t. $Opened(b_1, p_1)\ldots Opened(p_n, b_2)$	Not satisfied	Ab
$Open(a, openn)$	$Amb(b, a, \tilde{n})$	$Amb(a, c, \tilde{m})$		Satisfied	Sim
$Open(a_1, openn)$	$Amb(b, a, \tilde{n})$	$Amb(a_2, c, \tilde{m})$	$a_1 \neq a_2$, $\exists\ n \geqslant 0$ s. t. $Opened(a_1, p_1)\ldots Opened(p_n, a_2)$	Satisfied	Sim
$Open(a_2, openn)$	$Amb(b, a_1, \tilde{n})$	$Amb(a_2, c, \tilde{m})$	$a_1 \neq a_2$, $\exists\ n \geqslant 0$ s. t. $Opened(a_1, p_1)\ldots Opened(p_n, a_2)$	Satisfied	Sim
$Open(a_2, openn)$	$Amb(b, a_1, \tilde{n})$	$Amb(a_2, c, \tilde{m})$	$a_1 \neq a_2$, $\nexists\ n \geqslant 0$ s. t. $Opened(a_1, p_1)\ldots Opened(p_n, a_2)$	Not satisfied	Ab

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table A2. *Definitions of* $Link1_{in}(\dots)$*,* $Link2_{in}(\dots)$*, and* $Link3_{in}(\dots)$.

$$
\left.\begin{array}{lcl}
Link1_{in}(a, pc, a_x, p_x, inn) & = & (\nu\, pc1)\overline{a}(pc1).pc1(a_{x1}, p_{x1}). \\
 & & \quad ([a_{x1} = busy]Release1(a, p_{x1}, pc, inn)@In(a, inn) \\
 & & \quad + \\
 & & \quad [a_{x1} \neq busy]\; Check1_{in}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, inn) \\
 & & \quad) \\
Check1_{in}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, inn) & = & [a = a_{x1}]0 \;+\; [a \neq a_{x1}]Link1_{in}(\mathbf{a_{x1}}, pc, a_x, p_x, inn)
\end{array}\right\}
$$

linking the $Amb(a, p_{x1}, \tilde{s})$ where the capability lay

$$
\left.\begin{array}{lcl}
Link2_{in}(a, p_{x1}, pc, a_x, p_x, pc1, inn) & = & (\nu\, pc2)\overline{p_{x1}}(pc2).pc2(a_{x2}, p_{x2}). \\
 & & \quad ([a_{x2} = busy]Release2(a, p_{x1}, pc, pc1, inn)@In(a, inn) \\
 & & \quad + \\
 & & \quad [a_{x2} \neq busy]\; Check2_{in}(a, p_{x1}, pc, a_x, p_x, pc1, pc2, a_{x2}, p_{x2}, inn) \\
Check2_{in}(a, p_{x1}, pc, a_x, p_x, pc1, pc2, a_{x2}, p_{x2}, inn) & = & [p_{x1} = a_{x2}]0 \;+\; [p_{x1} \neq a_{x2}]\; Link2_{in}(a, \mathbf{a_{x2}}, pc, a_x, p_x, pc1, inn)
\end{array}\right\}
$$

linking the $Amb(p_{x1}, p_{x2}, \tilde{s})$ containing the 2 Amb() processes involved in the in capability execution

$$
\left.\begin{array}{lcl}
Link3_{in}(a, a_{x2}, pc, a_x, p_x, pc1, pc2, inn) & = & (\nu\, pc3)\overline{p_x}(pc3, pc2).pc3(a_{x3}, p_{x3}). \\
 & & \quad ([a_{x3} = busy]Release3(a, a_{x2}, pc, pc1, pc2, inn)@In(a, inn) \\
 & & \quad + \\
 & & \quad [a_{x3} \neq busy]\; Check3_{in}(a, a_{x2}, pc, a_x, p_x, pc1, pc2, pc3, a_{x3}, p_{x3}, inn) \\
Check3_{in}(a, a_{x2}, pc, a_x, p_x, pc1, pc2, pc3, a_{x3}, f_{x3}, inn) & = & [a_{x3} = p_x]\; Verify_{in}(a, a_{x2}, pc, a_x, p_x, pc1, pc2, pc3, a_{x3}, p_{x3}, inn) \\
 & & \quad + \\
 & & [a_{x3} \neq p_x]Link3_{in}(a, a_{x2}, pc, a_x, \mathbf{a_{x3}}, pc1, pc2, inn)
\end{array}\right\}
$$

second time linking the $Amb(p_x, p_{x3}, \tilde{s})$ containing the 2 Amb() involved in the in capability execution

$$
\left.Verify_{in}(a, a_{x2}, pc, a_x, p_x, pc1, pc2, pc3, a_{x3}, p_{x3}, inn) \;=\; [\mathbf{a_{x2}} = \mathbf{a_{x3}}]\,0 \;+\; [a_{x2} \neq a_{x3}]Release4(a, a_{x2}, pc, pc1, pc2, pc3, inn)\right\}
$$

checking the structural conditions for the in capability to be simulated

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table A3. *Definitions of* $Link1_{out}(\ldots)$*, and* $Link2_{out}(\ldots)$*.*

$$
\begin{aligned}
Link1_{out}(a, pc, a_x, p_x, outn) \;&=\; (\nu\, pc1)\overline{a}(pc1).pc1(a_{x1}, p_{x1}).\\
&\quad ([a_{x1} = busy]Release1(a, p_{x1}, pc, outn)@Out(a, outn)\\
&\quad +\\
&\quad [a_{x1} \neq busy]\; Check1_{out}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, outn)\\
&\quad)\\
Check1_{out}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, outn) \;&=\; [a = a_{x1}]0 \;+\; [a \neq a_{x1}]Link1_{out}(\mathbf{a_{x1}}, pc, a_x, p_x, outn)
\end{aligned}
$$

linking the $Amb(a, p_{x1}, \tilde{s})$ where the capability lay

$$
\begin{aligned}
Link2_{out}(a, p_{x1}, pc, a_x, p_x, pc1, outn) \;&=\; (\nu\, pc2)\overline{p_{x1}}(pc2, pc2).pc2(a_{x2}, p_{x2}).\\
&\quad ([a_{x2} = busy]Release2(a, p_{x1}, pc, pc1, outn)@Out(a, outn)\\
&\quad +\\
&\quad [a_{x2} \neq busy]\; Check2_{out}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x2}, p_{x2}, outn)\\
Check2(a, p_{x1}, pc, a_x, p_x, pc1, a_{x2}, p_{x2}) \;&=\; [a_{x2} = p_{x1}]\; Verify_{out}(a, p_{x1}, pc, a_x, p_x, pc1, pc2, outn)\\
&\quad +\\
&\quad [a_{x2} \neq p_{x1}]Link2_{out}(a, \mathbf{a_{x2}}, pc, a_x, p_x, pc1, outn)
\end{aligned}
$$

linking the $Amb(p_{x1}, p_{x2}, \tilde{s})$ for updating the coordinates

$$
Verify_{out}(a, p_{x_1}, pc, a_x, p_x, pc1, pc2, cap) \;=\; [\mathbf{p_{x_1}} = \mathbf{a_x}]\,0 \;+\; [p_{x_1} \neq p_x]Release3(a, p_{x_1}, pc, pc1, pc2, outn)@Out(a, outn)
$$

checking the structural conditions for the out capability to be simulated

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table A4. *Definitions of* $Link1_{open}(\ldots)$*, and* $Link3_{open}(\ldots)$.

$$
\left.
\begin{aligned}
Link1_{open}(a, pc, a_x, p_x, openn) \;=\;& (\nu\, pc1)\overline{a}(pc1).pc1(a_{x1}, p_{x1}).\\
& \quad ([a_{x1} = busy]Release1(a, p_{x1}, pc, openn)@Open(a, openn)\\
& \quad +\\
& \quad [a_{x1} \neq busy]\; Check1_{open}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, openn)\\
& \quad)\\
Check1_{open}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x1}, p_{x1}, open) \;=\;& [a = a_{x1}]0 \;+\; [a \neq a_{x1}]Link1_{open}(\mathbf{a_{x1}}, pc, a_x, p_x, openn)
\end{aligned}
\right\}
$$

linking the $Amb(a, p_{x1}, \tilde{s})$ where the capability lay

$$
\left.
\begin{aligned}
Link3_{open}(a, p_{x1}, pc, a_x, p_x, pc1, openn) \;=\;& (\nu\, pc2)\overline{p_x}(pc2, pc2).pc2(a_{x2}, p_{x2}).\\
& \quad ([a_{x2} = busy]Release2(a, p_{x1}, pc, pc1, openn)@Open(a, openn)\\
& \quad +\\
& \quad [a_{x2} \neq busy]\; Check3_{open}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x2}, p_{x2}, pc2, openn)\\
Check3_{open}(a, p_{x1}, pc, a_x, p_x, pc1, a_{x2}, p_{x2}, pc2, openn) \;=\;& [p_x = a_{x2}]Verify_{open}(a, p_{x1}, pc, a_x, p_x, pc1, pc2)\\
& \; +\\
& [p_x \neq a_{x2}]\; Link3_{open}(a, p_{x1}, pc, a_x, \mathbf{a_{x2}}, pc1, openn)
\end{aligned}
\right\}
$$

linking the $Amb(p_x, p_{x2}, \tilde{s})$ containing the process Amb() where the open capability acts on

$$
\left.
Verify_{open}(a, p_{x1}, pc, a_x, p_x, pc1, pc2) \;=\; [\mathbf{a} = \mathbf{p_x}]\,0 \;+\; [a \neq p_x]Release3(a, p_{x1}, pc, pc1, pc2, openn)@Open(a, openn)
\right\}
$$

checking the structural conditions for the open capability to be executed

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Table A5. *Definitions of* $Release1(\dots)$, $Release2(\dots)$, $Release3(\dots)$.

$$Release1(a,p,pc,cap) = \overline{pc}(release,release).0$$
$$Release2(a,p,pc,pc1,cap) = \overline{pc}(release,release).\overline{pc1}(release,release).0$$
$$Release3(a,p,pc,pc1,pc2,cap) = \overline{pc}(release,release).\overline{pc1}(release,release).\overline{pc2}(release,release).0$$
$$Release4(a,p,pc,pc1,pc2,pc3,cap) = \overline{pc}(release,release).\overline{pc1}(release,release).\overline{pc2}(release,release).\overline{pc3}(release,release)0$$

Table A6. *Definition of* $Amb(top)$.

$$Amb(top) = top(pc_x).\overline{pc_x}(top,top).Busy(top,pc_x) + top(pc_x,pc'_x).\overline{pc_x}(nomatch,nomatch).Amb(top)$$

$$Busy(top,pc) = pc(a_{new},p_{new}).Amb(top) + top(pc_x).\overline{pc_x}(busy,busy).Busy(top,pc) + top(pc_x,pc'_x).[pc = pc'_x]\overline{pc}(top,top).Busy(top,pc) + [pc \neq pc'_x]\overline{pc'_x}(nomatch,nomatch).Busy(top,pc)$$

Appendix B. Proofs

Despite we adopted a reduction semantics, in the proofs of this section, we use transition labels to help the understanding of each π-calculus interaction.

Proposition 4.3 (no trace interference).
Let $P \in \mathcal{P}_{MA}$ be a MA process and let $\mathcal{T}(P) \equiv Q$ be its encoding. If $Q \rightarrow^* Q'$, then $\exists$ $Q'' \in A\pi$ such that $Q' \rightarrow^* Q''$ and $\exists$ $n \geqslant 1$ traces $\xi_1,\dots,\xi_n$ such that $Q \rightarrow^*_{\xi_1} \cdots \rightarrow^*_{\xi_n} Q''$.

Proof. The proof is by induction on the number n of traces involved in the computation.

base step n=0

By hypothesis, Q executes no traces; thus, we may also say that Q executes no transitions. In fact, if $Q \rightarrow Q'$ for some Q', by Proposition 4.2, it would be the first move of a trace. The thesis follows.

base step n=1

By Proposition 4.2, any transition that Q can execute can only be the first transition of a trace, either a simulating or an aborting one. As there is only one trace, there is no interference between different traces.
Then, the proof proceeds by cases on the type of the process $Open(\dots)$, or $Out(\dots)$, or $In(\dots)$ involved in the first transition of Q.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

case in

The proof proceeds by executing a trace computation simulating the in capability, which we will denote as ξ. The computation is similar to the one in Example 4.2. Please note that in the example we did not consider the causes that make a trace computation abort. We will show in detail the causes of a trace to abort in the n+1-th step, thus we skip the computation.

Let Q' be a derivate of Q reached by only executing transitions of ξ. It is important to remark that there are two ways for Q' to evolve: continuing executing the trace ξ, or starting a new one. Thus, we can conclude that by executing only one trace ξ, Q can only perform transitions of trace ξ.

Cases out and open are similar.

inductive step n + 1

Let $Q \rightarrow^{*} Q'$ be a series of transitions related to n complete or incomplete traces; by inductive hypothesis, there is no interference between them.

Also, by inductive hypothesis, the next transition of the process Q' is either a transition of one of the previous n traces or it is an interaction between processes $Amb(\ldots)$ and $Open(\ldots)$ (or $Out(\ldots)$, or $In(\ldots)$) on the channels $open_x$, out_x, in_x, respectively, where x stands for any name. A transition of the last type turns out to be the first transition of the $n+1$-th trace.

The proof proceeds by cases on the type of the possible transitions the $n+1$-th trace can start with. To simplify the presentation, we do not precisely specify the list of restricted names, that can be deduced by the definition of the encoding function $\mathcal{T}(\ldots)$, and we use a generic $(\nu\ \tilde{m})$.

case open

By hypothesis, $Q' \equiv (\nu\ \tilde{m})(R \,|\, Open(a, openn) \,|\, Amb(b, g, \tilde{n}))$, for some process R, and some names a, b, g; Q' can execute the first two transitions of the n+1-th trace, involving processes $Open(a, openn)$ and $Amb(b, g, \tilde{n})$:

$Q' \xrightarrow{(\nu\ pc)\langle\overline{openn}(pc), openn(x)\rangle} Q_1 \xrightarrow{\langle\overline{pc}(b,g), pc(a_x, p_x)\rangle} Q_2$, where

$Q_2 \equiv (\nu\ \tilde{m})(R \,|\, Link1_{open}(a, pc, b, g, openn) \,|\, Busy(b, g, pc, open, \tilde{n}))$

Now, process $Link1_{open}(\ldots)$ (see code in Table A4) tries to communicate along channel a, and there are the following two possibilities:

a. A sub-process $Amb(a, p, \tilde{s})$ exists in R, for some names p and s;

b. A sub-process $Amb(a, p, \tilde{s})$ does not exist in R, for some names p and s.

First, we show the **case b.**, as it is simpler, then we focus on the **case a**.

case b.

By definition of process $Amb(\ldots)$ (see code in Table 7), there must exist one sub-process $Opened(\ldots)$ that has substituted the $Amb(a, p, \tilde{s})$ process as the effect of the simulation of the *open a* capability. Please note that there could be a chain of $Opened(\ldots)$ subprocesses involved in the $n+1$-th trace. Thus, we have to prove that the process Q_2 executes some transitions involving the $Opened(\ldots)$ sub-processes and that the computation falls in the **case a**. The proof proceeds by induction on the number of times o_p that any $Opened(\ldots)$ sub-process is involved in the computation.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

base case $o_p = 1$
By hypothesis, we can write Q_2 as
$Q_2 \equiv (\nu\ \tilde{m})(R' \mid Link1_{open}(a, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}) \mid Opened(a, a') \mid Amb(a', p', \tilde{s}))$,
for some names a', p', s, with $R = R' \mid Opened(a, a') | Amb(a', p', \tilde{s})$;
now, the process Q_2 can execute two transitions, between processes
$Link1_{open}(a, pc, b, g, openn)$ and $Opened(a, a')$, for an updating of the coordinates:
$Q_2 \xrightarrow{(\nu\ pc1)\langle \overline{a}(pc1), a(pc_x)\rangle} Q_2' \xrightarrow{(\overline{pc1}(a', a'), pc1(a_{x1}, f_{x1}))} Q_2''$, where
$Q_2'' \equiv (\nu\ \tilde{m})(R' \mid Link1_{open}(a', pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}) \mid Opened(a, a') \mid Amb(a', p', \tilde{s}))$.
Now, the first parameter of $Link1_{open}(a', pc, b, g, openn)$ is updated and the computation can proceed as in the following case **a**.

inductive case $o_p + 1$
By hypothesis, we can write Q_2 as
$Q_2 \equiv (\nu\ \tilde{m})(R' \mid Link1_{open}(a, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}) \mid Opened_1(a, a^1) \mid \ldots \mid Opened_n(a^n, a^{n+1}) \mid Amb(a^{n+1}, p, \tilde{s})$, for some names $a^1, \ldots, a^{n+1}$, p, s, where
$R = R'' \mid Opened_1(a, a^1) \mid \ldots \mid Opened_n(a^n, a^{n+1}) \mid Amb(a^{n+1}, p, \tilde{s})$, and also by hypothesis, o_p $Opened(\ldots)$ sub-processes have already been involved in the computation, thus we have
$Q_2' \equiv (\nu\ \tilde{m})(R' \mid Link1_{open}(a^n, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}) \mid Opened_1(a, a^1) \mid \ldots \mid Opened_n(a^n, a^{n+1}) \mid Amb(a^{n+1}, p, \tilde{s})$.
Now, Q_2' can proceed with the two transitions involving $Opened_{n+1}(a^n, a^{n+1})$ and $Link1_{open}(a^n, pc, b, g, openn)$:
$Q_2' \xrightarrow{(\nu\ pc1)\langle \overline{a^n}(pc1), a^n(pc_x)\rangle} Q_2'' \xrightarrow{\langle \overline{pc1}(a^{n+1}, a^{n+1}), pc1(a_{x1}, p_{x1})\rangle} Q_2'''$, where
$Q_2''' \equiv (\nu\ \tilde{m})(R' \mid Link1_{open}(a^{n+1}, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}) \mid Opened_1(a, a^1) \mid \ldots \mid Opened_n(a^n, a^{n+1}) \mid Amb(a^{n+1}, p, \tilde{s}))$.
Now, the first parameter of $Link1_{open}(a^{n+1}, pc, b, g, openn)$ is updated and we can say that the computation can proceed as in the following case **a**.

case a.
There are three further sub-cases:

a1. The sub-process $Amb(a, p, \tilde{s})$ appears in Q_2, i.e. it is not involved in any of the previous n traces;
a2. The sub-process $Amb(a, p, \tilde{s})$ has evolved in its busy configuration, i.e. it appears in Q_2 in its form $Busy(\ldots)$;
a3. The sub-process $Amb(a, p, \tilde{s})$ appears in Q_2 neither in its $Amb(\ldots)$ configuration nor in its $Busy(\ldots)$ configuration. This happens when one of the input prefixes in the first five rows of $Amb(\ldots)$ code or in the two last rows of $Busy(\ldots)$ code in Table 7 has been fired. In this case, no deadlock can occur, as it is always possible to execute the subsequent output prefix on the private channel pc. After that, the computation follows either the case **a1.** or the case **a2.**, depending if process $Amb(a, p, \tilde{s})$ is involved or not in one of the previous n traces.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

First, we show the **case a2.** as it is simpler, then we proceed with the **case a1**.

case a2.

By hypothesis, the process $Amb(a, p, \tilde{s})$ is involved in one of the first n previous traces and appears in Q_2 in its *busy* configuration, i.e. $Busy(a, p, pc', cap, \tilde{s})$, for some name pc'. We can write
$Q_2 \equiv (\nu \quad \tilde{m})(R' \mid Busy(a, p, pc', cap, \tilde{s}) \mid Link1_{open}(a, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}))$, where $R = R' \mid Busy(a, p, pc', cap, \tilde{s})$, and $cap \in \{in, out, open\}$.
The only possible transitions for the n+1-th trace are the interactions between processes $Busy(a, p, pc', cap, \tilde{s})$ and $Link1_{open}(a, pc, b, g, openn)$:
$Q_2 \xrightarrow{(\nu \ pc1)\langle \overline{a}(pc1), a(pc_x)\rangle} Q_3 \xrightarrow{\langle \overline{pc1}(busy, busy), pc1(a_{x1}, p_{x1})\rangle} Q_4 \equiv$
$(\nu\tilde{m})(R' \mid Busy(a, p, pc', cap, \tilde{s}) \mid Release1(a, pc, openn) \mid Busy(b, g, pc, open, \tilde{n}))$.
As the process $Busy(a, p, pc', cap, \tilde{s})$ has sent the message 'busy', the computation of the n+1-th trace ends as an aborting trace and no more interferences with the other traces are possible:
$Q_4 \xrightarrow{\langle \overline{pc}(release, release), pc(a_{x1}, p_{x1})\rangle} Q_5 \equiv (\nu\tilde{m})(R' \mid Busy(a, p, pc', cap, \tilde{s}) \mid Amb(b, g, \tilde{n}) \mid Open(a, openn))$.
The computation has backtracked and the process has returned to a state congruent with Q': $Q_5 \equiv Q'$.

case a1.

By hypothesis, we have
$Q_2 \equiv (\nu\tilde{m})(R' \mid Amb(a, p, \tilde{s}) \mid Link1_{open}(a, pc, b, g, openn) \mid Busy(b, g, pc, open, \tilde{n}))$, where $R = R' \mid Amb(a, p, \tilde{s})$.
Now, the only two possible transitions for the n+1-th trace are between processes, $Amb(a, p, \tilde{s})$ and $Link1_{open}(a, pc, b, g, openn)$, and we have the following:
$Q_2 \xrightarrow{(\nu \ pc1)\langle \overline{a}(pc1), a(pc_x)\rangle} Q_3 \xrightarrow{\langle \overline{pc1}(a, p), pc1(a_{x1}, p_{x1})\rangle} Q_4 \equiv$
$(\nu\tilde{m})(R' \mid Busy(a, p, pc1, nocap, \tilde{s}) \mid \quad Check1_{open}(a, p, pc, b, g, pc1, a, p, openn) \mid Busy(b, g, pc, open, \tilde{n}))$.
The process $Check1_{open}(a, p, pc, b, g, pc1, a, p, openn)$ can evolve to process $Link3_{open}(a, p, pc, b, g, pc1, openn)$ as the received name a is the expected one, see the code of $Check1_{open}(\ldots)$ process in Table A4; thus, we have
$Q_4 \equiv (\nu\tilde{m})(R' \mid Busy(a, p, pc1, nocap, \tilde{s}) \mid \quad Link3_{open}(a, p, pc, b, g, pc1, openn) \mid Busy(b, g, pc, open, \tilde{n}))$.
Now, there are three possibilities for the coordinate g, i.e. the parent of b:

a1.1. the coordinate g is updated and the structural condition for the capability to be executed is satisfied, i.e. $g = a$;

a1.2. the coordinate g is updated and the structural condition for the capability to be executed is not satisfied, i.e. $g \neq a$;

a1.3. the coordinate g is not updated. This case requires the same reasoning made in the previous case **b.**, where the trace involves one or more $Opened(\ldots)$ processes in order

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

to update the coordinates, and we fall again in either case **a1.1.** or case **a1.2**. Thus, we skip this case.

```
case a1.1.
```

By hypothesis, $g = a$, thus, we can substitute g with a:
$Q_4 \equiv (\nu\tilde{m})(R' \mid Busy(a, p, pc1, nocap, \tilde{s}) \mid Link3_{open}(a, p, pc, b, a, pc1, openn) \mid Busy(b, a, pc, open, \tilde{n}))$.
The n+1-th trace evolves by executing the two following transitions between processes: $Busy(a, p, pc1, nocap, \tilde{s})$ and $Link3_{open}(a, p, pc, b, a, pc1, openn)$:
$Q_4 \xrightarrow{(\nu\ pc2)\langle\overline{a}(pc2,pc2)a(pc_x,pc'_x)\rangle} Q_5 \xrightarrow{\langle\overline{pc2}(a,p),pc2(a_{x2},p_{x2})\rangle} Q_6 \equiv (\nu\tilde{m})(R' \mid$
$Busy(a, p, pc1, nocap, \tilde{s}) \mid Check3_{open}(a, p, pc, b, a, pc1, a, p, pc2, openn) \mid Busy(b, a, pc, open, \tilde{n}))$.
Please note that the process $Check3_{open}(a, p, pc, b, a, pc1, a, p, pc2, openn)$ can evolve to process $SimulateOpen(a, p, pc, pc1)$ as the received name a verifies the mismatch $a \neq busy$, and the two matches $a = a$ (see the code of processes $Link3_{open}(\dots)$, $Check3_{open}(\dots)$, and $Verify_{open}(\dots)$ in Table A4).
Then, the n+1-th trace concludes the capability simulation with the two following transitions:
$Q_6 \xrightarrow{\langle\overline{pc}(a,p),pc(a_{new},p_{new})\rangle} Q_7 \xrightarrow{\langle\overline{pc1}(a,p),pc1(a_{new},p_{new})\rangle} Q_8 \equiv (\nu\tilde{m})(R' \mid Amb(a, p, \tilde{s}) \mid C \mid Opened(b, a))$,
where C stands for the continuation of the sub-process $Open(a, openn)$.
We have shown how the $n+1$-th trace concludes its computation simulating the *open n* capability without any interference.

```
case a1.2.
```

By hypothesis, $g \neq a$, thus there exists a process $Amb(g, g', \tilde{r})$ for some g' and r. There are three further sub-cases:

— case **a1.2.1.**, the process $Amb(g, g', \tilde{r})$ appears in Q_4, i.e. it is not involved in any of the previous n traces;
— case **a1.2.2.**, the process $Amb(g, g', \tilde{r})$ has evolved in its busy configuration, i.e. it appears in Q_4 in its form $Busy(\dots)$;
— case **a1.2.3.**, the process $Amb(g, g', \tilde{r})$ appears in Q_4 neither in its $Amb(\dots)$ configuration nor in its $Busy(\dots)$ configuration. This happens when one of the input prefixes in the first five rows of $Amb(\dots)$ code or the two last rows of $Busy(\dots)$ code in Table 7 has been fired. In this case, no deadlock can occur, as it is always possible to execute the subsequent output prefix on the private channel pc. After that, the computation follows either the case **a1.2.1.** or the case **a1.2.2.**, depending if process $Amb(g, g', \tilde{r})$ is involved or not in one of the previous n traces.

In the second case **a1.2.2.**, the $n+1$-th trace aborts, and the proof is similar to the case **a2.**; thus, we skip it.
In the first case **a1.2.1.**, $Amb(g, g', \tilde{r})$ is not involved in other traces and the computation of the $n+1$-th trace proceeds from process Q_4:
$Q_4 \equiv (\nu\tilde{m})(R'' \mid Busy(a, p, pc1, nocap, \tilde{s}) \mid Link3_{open}(a, p, pc, b, g, pc1, openb) \mid Busy(b, g, pc, open, \tilde{n}) \mid$
$Amb(g, g', \tilde{r}))$, where $R' = R'' \mid Amb(g, g', \tilde{r})$.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Q_4 can evolve by executing the two following transitions between processes:
$Link3_{open}(a, p, pc, b, g, pc1, openn)$ and $Amb(g, g', \tilde{s})$:
$Q_4 \xrightarrow{(\nu\ pc2)\langle \overline{g}(pc2), g(pc_x)\rangle} Q_5 \xrightarrow{\langle \overline{pc2}(g,g'), pc2(a_{x2},p_{x2})\rangle} Q_6 \equiv$
$(\nu\tilde{m})(R'' \mid Busy(a, p, pc1, nocap, \tilde{s}) \mid \quad Release3(a, p, pc, b, g, pc1, pc2, openn) \mid \quad Busy(b, g, pc, open, \tilde{n}) \mid$
$\quad Busy(g, g', pc2, nocap, \tilde{r}))$.
Please note that process $Link3_{open}(a, p, pc, b, g, pc1, openb)$ evolves to process
$Release3(a, p, pc, b, g, pc1, pc2, openn)$ as the received name g verifies the mismatch $g \neq busy$, the match $g = g$, and the mismatch $a \neq g$ (see the code of processes $Link3_{open}(\dots)$, $Check3_{open}(\dots)$, and $Verify_{open}(\dots)$ in Table A4). At this point, the $n+1$-th trace can only abort by performing the three following transitions:
$Q_6 \xrightarrow{\langle \overline{pc}(release,release), pc(a_{new},p_{new})\rangle} Q_7 \xrightarrow{\langle \overline{pc1}(release,release), pc1(a_{new},p_{new})\rangle}$
$Q_8 \xrightarrow{\langle \overline{pc2}(release,release), pc2(a_{new},p_{new}\rangle} Q_9 \equiv (\nu\tilde{n})(R'' \mid Amb(a, p, \tilde{s}) \mid Open(a, openn) \mid Amb(b, g, \tilde{n}) \mid Amb(g, g', \tilde{s}))$.
Here, we have shown that the $n+1$-th trace aborts as there are not the structural conditions for the *openn* capability to be simulated.

cases in, out

The proofs for the cases **in** and **out** are similar to **case open**, i.e. we have to show all the possible evolutions for the two traces similarly to the previous **case open**; thus, we skip the two proofs.
We have shown two facts: (1) for any possible evolution of the **n+1**-th trace, no interference with the previous **n** traces is possible, and (2) any transition performed by an encoding process can only be part of a trace. □

Proposition 4.4 (trace properties).
Let P be a MA process, and let ξ be a trace executed by the process $Q = \mathcal{T}(P)$, then

1. the sub-process of the form $In(\dots)$, $Out(\dots)$, $Open(\dots)$, involved in the first transition of a trace ξ, will be involved in all the transitions of ξ. We will call this sub-process the sub-process characterising the trace ξ;
2. in the last transition of a trace ξ: $Q \rightarrow_{\xi} Q'$, the sub-process C in Q, characterising the trace, in Q' will evolve into a composition of sub-processes $In(\dots)$, $Out(\dots)$, $Open(\dots)$, $Amb(\dots)$, $Opened(\dots)$, and $\mathbf{0}$.
3. the number of the transitions of a trace is limited.

Proof.

1. The proof consists in executing all the possible evolutions of each type of traces (a simulating trace, mimicking an in or an out or open capability, or an aborting trace), and it is similar to the one in Proposition 4.3.
2. The proof consists in executing all the possible evolutions of each type of traces (a simulating trace, mimicking an in or an out or open capability, or an aborting trace), focussing the attention on how the involved sub-processes evolve from the source state to the target state of each transition. Thus, it is similar to the one in Proposition 4.3.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

3. The proof consists in executing all the possible evolutions of each type of traces (a simulating trace, mimicking an in or an out or open capability, or an aborting trace), and it is similar to the one in Proposition 4.3.
   By counting the number of transitions of each trace, we can also be more precise and declare the exact number of transitions for each kind of trace:
   — a trace simulating an **in** capability has a length of $11 + 2N$;
   — a trace simulating an **out** or an **open** capability has a length of $8 + 2N$;
   — an aborting trace,
     – using the recovery routine Release1, has a length of $5 + 2N$;
     – using the recovery routine Release2, has a length of $8 + 2N$;
     – using the recovery routine Release3, has a length of $11 + 2N$;
     – using the recovery routine Release4, has a length of $12 + 2N$;
   where N is the number of the $Opened(\dots)$ processes involved in the computation of the trace. □

Theorem 4.1 (operational correspondence 1).
Let P be a MA process, and let $Q \equiv \mathcal{T}(P)$ be its encoding. If $Q \rightarrow^* Q'$, then $\exists\ P' \in \mathcal{P}_{MA}$, and $Q'' \in A_\pi$ such that $P \rightarrow_p^* P'$, $Q' \rightarrow^* Q''$, and $Q'' \equiv \mathcal{T}(P')$.

Proof. In Proposition 4.3, we have shown that process Q can only execute transitions which are part of a trace. Thus, we can provide a proof by induction on the number n of the traces involved in the computation of Q.

base case n=0
By hypothesis, Q executes no traces, thus we may also say that Q executes no transitions; in fact, if $Q \rightarrow Q'$, by Proposition 4.2, it would be the first move of a trace. Then, we let P execute no transition and we set $Q = Q''$, and $P = P'$. The thesis follows.

base case n=1
By hypothesis, in the computation $Q \rightarrow^* Q'$, only one trace is involved. There are two possibilities: **(a)** it is an aborting trace or **(b)** a simulating one.
In the case **(a)**, we have to prove that there is a process $Q'' \in A_\pi$ such that $Q \rightarrow^* Q''$, and $Q'' \equiv Q$. The proof consists in showing the execution of any possible aborting trace, and proceeds similarly to one of the cases **(a1.2)**, **(a.1.2.1)**, **(a.1.2.2)**, and **(a2)** of the Proposition 4.3. The thesis follows by letting P execute no transition, and by setting $P' \equiv P$.
In the second case **(b)**, there are three possibilities: the trace simulates an *in*, an *out*, or an *open* capability.
case in
By hypothesis, Q can execute a simulating trace mimicking an in capability; thus, it follows that
$Q \equiv (\nu\tilde{m})(In(b, inn)|S|Amb(b, d, \tilde{s})|N|Amb(c, d, \tilde{n})|R|Amb(d, t, \tilde{r})|D)$, where (b, d), (c, d), (d, t) are the coordinates assigned by the translation function to the π-calculus sub-processes

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

encoding of the ambients s, n, r, respectively; and where S, N, and R, are the encoding of the MA sub-processes located within the ambients s, n, and r, respectively, and where D is the encoding of the context where the ambient d is located. Now, the proof continues by structural induction on Q, where the base case is when $D \equiv \mathbf{0}$ and $d = t = top$, i.e. when the context is null.

base case $R \equiv \mathbf{0}$ **and** $d = t = top$

We can rewrite Q as $Q \equiv (\nu\tilde{m})(In(b, inn)|S|Amb(b, top, \tilde{s})|N|Amb(c, top, \tilde{n})|Amb(top)|\mathbf{0})$.
For simplicity, we only consider the case where no $Opened(\ldots)$ process is involved in the computation, as the case where one or more $Opened(\ldots)$ processes participate in the computation evolves similarly to case **(b)** of the Proposition 4.3.
The process Q evolves in Q'' by simulating the $in\,n$ capability:
$Q \rightarrow^* Q' \rightarrow^* Q'' \equiv (\nu\tilde{m})(S|C|Amb(b, c, \tilde{s})|N|Amb(c, top, \tilde{n})|Amb(top))$, where C is the encoding of the continuation of $In(b, inn)$, i.e. $In(b, inn)@C \equiv \mathcal{T}_{ma}(in\,nP_1, b)$, for some MA process P_1. We skip all the details of the above computation as they are similar to case **a1.1.** of the Proposition 4.3; the only difference is that we should proceed by cases on the number of transitions that Q has already executed evolving in Q': $Q \rightarrow^* Q'$.
By definition of the encoding function, $\mathcal{T}(P) = Q$, we derive
$P \equiv s[\mathsf{in}\, n.P_1|P_3]|n[P_2]$, with $S = \mathcal{T}_{ma}(P_3, s)$, $N = \mathcal{T}_{ma}(P_2, n)$.
At this point, the process P executes the in capability as follows:
$P \rightarrow_p P' \equiv n[s[P_1|P_3]]|P_2]$. The encoding of P' is: $\mathcal{T}(P') \equiv$
$(\nu\tilde{m})(\mathcal{T}_{ma}(P_1, b)|\mathcal{T}_{ma}(P_3, b)|Amb(b, c, \tilde{s})|Amb(c, top, \tilde{n})|Amb(top)|\mathcal{T}_{ma}(P_2, c))$.
Now, we have the thesis: Q'' is congruent to $\mathcal{T}(P')$ as it correctly records the changing of the ambient hierarchy: After the computation, process encoding ambient n, identified by name c, has become the parent of process encoding ambient s, identified by the name b.

inductive case $R \not\equiv \mathbf{0}$ **and** $d \neq top, t \neq top$

By the Proposition 4.3, in the previous base case, along the computation for the simulation of the $in\,n$ capability, there is no possibility of synchronisation with other sub-processes in the context R (when the sub-processes $Opened(\ldots)$ are involved in the computation, we fall again in the case already shown in the case **b.** of Proposition 4.3); thus, no side effects are produced on the context R. In particular, $d \neq top$ and $t \neq top$ means that the interacting processes are nested in the hierarchical ambient structure, and this situation does not change the execution of the computation.
Then, we skip the inductive case of the structural induction, as it is similar to the base case.

case *out*

The proof proceeds similarly to case *in*.

case *open*

By hypothesis, Q can execute a simulating trace mimicking an open capability; thus, it follows that:
$Q \equiv (\nu\tilde{m})(R|Open(b, openm)|S|Amb(b, d, \tilde{s})|N|Amb(c, b, \tilde{n}))$, where (b, d) and (c, b) are the coordinates of translation of the ambients s and n, respectively, and where S and N are the translation of processes within the ambients s and n, respectively, and R is the encoding of the context.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

As before, the proof continues by structural induction on Q, where the base case is when $R \equiv \mathbf{0}$ and $d = top$, i.e. when the context is null.

base case $R \equiv \mathbf{0}$ **and** $b = top$
We can rewrite Q as $Q \equiv (\nu\tilde{m})(Open(top, openn)|S|Amb(top)|N|Amb(c, top, \tilde{n}))$. For simplicity, we only consider the case where no $Opened(\dots)$ process is involved in the computation, as the case where one or more $Opened(\dots)$ processes participate in the computation evolves similarly to case **b.** of the Proposition 4.3.
Now, the process Q can execute the simulation of an open capability, evolving in Q'', as follows: $Q \rightarrow^* Q' \rightarrow^* Q'' \equiv (\nu\tilde{m})(C|S|Amb(top)|N|Opened(c, top))$, where C is the encoding of the continuation of $Open(b, openn)$, i.e. $Open(b, openn)@C \equiv \mathcal{T}_{ma}(open\, n.P_1, b)$ for some MA process P_1; the computation is similar to case **a1.1.** of the Proposition 4.3; thus, we skip all the steps of the computation.
By definition of the encoding function we have, $P \equiv open\, n.P_1|P_3|n[P_2]$, where $S = \mathcal{T}_{ma}(P_3, top)$, and $N = \mathcal{T}_1(P_2, n)$.
Now, the process P executes the *open* capability as follows: $P \rightarrow_p P' \equiv P_1|P_3|(\!|P_2|\!)$.
The translation of P' is
$\mathcal{T}(P') = (\nu\ \tilde{m})(\mathcal{T}_{ma}(P_1, top)|\mathcal{T}_{ma}(P_3, top)|Amb(top)|Opened(c, top)|\mathcal{T}_{ma}(P_2, c))$, where c is a restricted name (corresponding to restricted name c in the process Q'').
The process $Opened(c, top)$, that has replaced the process $Amb(c, top, \tilde{n})$, corresponds to the translation of the new *passive context* surrounding P_2, that has been created after the execution of the open capability in P'.
The thesis follows.

inductive case $R \not\equiv \mathbf{0}$ **and** $b \neq top$
As before, by the Proposition 4.3, the context R is not altered by the simulation of the *open* capability, and the interacting processes can also be located in a nested position in the hierarchical ambient structure, i.e. $b \neq top$ with no changes in the execution of the computation; thus, we skip the inductive case of the structural induction.

inductive case n+1
By hypothesis, the computation $Q \rightarrow^* Q'$ involves n traces. Then, we may also assume that process Q' starts the $n+1$-th trace.
By Proposition 4.3, we can say that the $n+1$-th trace can correctly proceed its computation with no interferences with the previous n traces. Now, as in the **base case** ($n = 1$), there are two possibilities: **(a)** the $n+1$-th trace is an aborting trace; **(b)** the $n+1$-th trace is a simulating trace, and in the case **(b),** we have three choices: the simulation of an *in*, or of an *out*, or of an *open* capability. The proof proceeds similarly to the previous base case, and by inductive hypothesis we get the thesis. □

Theorem 4.2 (operational correspondence 2).
Let P be a MA process, if $P \rightarrow_p^* P'$, then there exists a π-calculus process Q' such that $Q = \mathcal{T}(P) \rightarrow^* Q'$ and $\mathcal{T}(P') \equiv Q'$.

Proof. The proof is by induction on the number n of the transitions executed by P.
base case $n = 0$

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

By hypothesis, P executes no transition, and we let Q execute no transition; it follows that $\mathcal{T}(P) = Q$, and we have the thesis.

base case $n = 1$

The proof proceeds by cases on the type of capability that process P executes: *in*, *out*, or *open*.

case open

By hypothesis $P \equiv \mathcal{G}\{\mathbb{D}(open\, n.P_1)|\mathbb{C}(n[P_2])\} \rightarrow_p P' \equiv \mathcal{G}\{\mathbb{D}(P_1)|\mathbb{C}((\!|P_2|\!))\}$.

For simplicity, we assume that the context $\mathcal{G}$ contains an ambient t, where the sub-process $\mathbb{D}(open\, a.P_1)|\mathbb{C}(a[P_2])$ lies.

The encoding of process P:

$\mathcal{T}(P) = Q \equiv (\nu\, \tilde{m})(R|Open(b, openn)|Amb(b, d, \tilde{t})|Amb(a, b, \tilde{n})|\mathcal{T}_{ma}(P_2, a))$, where (a, b), and (b, d) are the coordinates associated to the translation of the ambients n and t, respectively; R is the encoding of the contexts $\mathcal{G}, \mathbb{D}$, and $\mathbb{C}$.

Now, Q can execute the open simulating trace:

$Q \xrightarrow{\tilde{\tau}}{}^{*} (\nu\, \tilde{m})(R|C|Amb(b, d, \tilde{t})|Opened(a, b)|\mathcal{T}_{ma}(P_2, a))$, where C is the encoding of the process P_1: $Open(a, openn)@C \equiv \mathcal{T}_{ma}(open\, n.P_1, a)$. We will skip the details of the computation, as the execution is similar to the ones in the proof of Proposition 4.3.

The direct encoding of P' is

$\mathcal{T}(P') = Q_1 \equiv (\nu\, \tilde{m})(R|C|Amb(b, d, \tilde{t})|Opened(a, b)|\mathcal{T}_{ma}(P_2, a))$.

The process Q_1 is congruent to Q', up to α-conversion, and the thesis follows.

cases in and out

The others in and out cases are simpler, as no new passive context is created.

inductive case $n + 1$

By hypothesis, $P \rightarrow_p^n P_1$. The proof proceeds similarly to the base case, by also considering that, by Proposition 4.3, the interleaved execution of $n + 1$ (simulating or aborting) traces causes no side-effects on the previous n computations. □

References

Bodei, C., Brodo, L. and Bruni, R. (2013). Open multiparty interaction. In: Martí-Oliet, N. and Palomino, M. (eds.) *Recent Trends in Algebraic Development Techniques,* Lecture Notes in Computer Science, volume 7841, Springer, Berlin Heidelberg, 1–23.

Bodei, C., Brodo, L., Bruni, R. and Chiarugi, C. (2014). A flat process calculus for nested membrane interactions. *Scientific Annals of Computer Science* **24** (1) 91–136.

Brodo, L. (2011). On the expressiveness of the *pi*-calculus and the mobile ambients. In: Johnson, M. and Pavlovic, D. (eds.) *AMAST-Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, volume 6486, Springer-Verlag, 44–59.

Brodo, L., Degano, P. and Priami, C. (2003). Reflecting mobile ambients into the π-calculus. In: Priami, C. (ed.) *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, Lecture Notes in Computer Science, volume 2874, Springer, Berlin Heidelberg, 25–56.

Busi, N., Gabbrielli, M. and Zavattaro, G. (2009). On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science* **19** (6) 1191–1222.

Cardelli, L. (2005). Brane calculi - interactions of biological membranes. In: Danos, V. and Schachter, V. (eds.) *Computational Methods in Systems Biology,* Lecture Notes in Computer Science, volume 3082, Springer, Berlin Heidelberg, 257–278.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256

Cardelli, L. and Gordon, A. (2000). Mobile ambients. *Theoretical Computer Science* **240** (1) 177–213.

Cenciarelli, P., Talamo, I. and Tiberi, A. (2005). Ambient graph rewriting. *Electronic Notes in Theoretical Computer Science*, volume 117, Elsevier, 335–351.

Ciobanu, G. and Zakharov, V.A. (2007). Encoding mobile ambients into the π-calculus. In: Virbitskaite, I. and Voronkov, A. (eds.) *Perspectives of Systems Informatics,* Lecture Notes in Computer Science, volume 4378, Springer, Berlin Heidelberg, 148–165.

Fournet, C., Lévy, J.-J. and Schmitt, A. (2000). An asynchronous, distributed implementation of mobile ambients. In: Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D. and Ito, T. (eds.) *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics,* Lecture Notes in Computer Science, volume 1872, Springer, Berlin Heidelberg, 348–364.

Gadducci, F. and Monreale, G.V. (2010). A decentralized implementation of mobile ambients. *The Journal of Logic and Algebraic Programming* **80** (2) 113–136.

Gorla, D. (2010). A taxonomy of process calculi for distribution and mobility. *Distributed Computing* **23** (4) 273–299.

Levi, F. and Sangiorgi, D. (2003). Mobile safe ambients. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **25** (1) 1–69.

Milner, R., Parrow, J. and Walker, D. (1992). A calculus of mobile processes, part 1–2. *Information and Computation* **100** (1) 1–77.

Palamidessi, C. (2003). Comparing the expressive power of the synchronous and the asynchronous pi-calculus. *Mathematical Structures in Computer Science* **15** (5) 685–719.

Parrow, J. (2001). An introduction to the π-calculus. In: *Handbook of Process Algebra*, Elsevier Science, 479–543.

Păun, Gh. (2000). Computing with membranes. *Computer and System Sciences* **61** (1) 108–143.

Phillips, I. and Vigliotti, M.G. (2008). Symmetric electoral systems for ambient calculi. *Information and Computation* **206** (1) 34–72.

Sangiorgi, D. (1996). Pi-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science* **167** (1&2) 235–274.

Zimmer, P. (2003). On the expressiveness of the pure safe ambients. *Mathematical Structures in Computer Science* **13** (5) 721–770.

Downloaded from https://www.cambridge.org/core. IP address: 3.236.122.9, on 16 May 2021 at 17:48:50, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129516000256