An Asynchronous, Distributed Implementation of Mobile Ambients

Cédric Fournet[1], Jean-Jacques Lévy[2], and Alan Schmitt[2⋆]

[1] Microsoft Research
[2] INRIA Rocquencourt

Extended abstract submitted to TCS'2000
Track (2): Logic, Semantics, Specification and Verification.

Abstract We present a first distributed implementation of the Cardelli-Gordon's ambient calculus. We use Jocaml as an implementation language and we present a formal translation of Ambients in the distributed Join Calculus, the process calculus associated with Jocaml. We prove the correctness of the translation.

The operational semantics of Ambients requires that complex migration steps be performed atomically. As a result, direct implementations rely on the serialization of migrations via a centralized control. In contrast, our implementation is fully asynchronous and allows concurrent reduction steps. It relies on a message-passing protocol between an ambient and its parent. Only the actions involving an ambient migrating or being opened are blocked during a reduction step.

The proof of correctness of this implementation is two staged. First, an extended ambient calculus with a refined semantics is introduced, and the two semantics are related using coupled-simulations. Then, a bisimulation is shown to exist between processes in the extended ambient calculus and their distributed join calculus translations. In combination, these results imply that our translation does not affect a variety of global observation predicates. In the final section, the actual implementation in Jocaml is discussed. As a whole, our paper gives an insight in the implementability of ambients.

In an appendix, the semantics of ambients and the join calculus is recalled, with an overview of both calculi.

Contact Author: Jean-Jacques Lévy, INRIA Rocquencourt
78153 – Le Chesnay Cedex, France
tel: +33 1 39 63 56 89
fax: +33 1 39 63 54 69
e-mail: jean-jacques.levy@inria.fr

⋆ This work is partly supported by the RNRT project MARVEL 98S0347

1 Introduction

This paper presents a highly concurrent distributed implementation of the Cardelli-Gordons's calculus of Mobile Ambients [5] in Jocaml [13]. The ambient calculus is a simple and very esthetic model for distributed mobile computing. However, until now, it did not have a distributed implementation. Such an implementation may seem easy to build, especially with a language with distribution and strong migration (Jocaml), but we encountered several difficulties and design choices.

Ambients are nested. Their dynamics is defined by three atomic steps: an ambient may move into a sibling ambient (IN), it may move out of its parent ambient (OUT), or it may open one of its child ambients (OPEN). Each atomic migration step may involve several ambients, possibly on different sites. For instance, the source and destination ambients participate to an IN-step; similarly the source and parent ambients take part to an OUT-step; the target ambient participates to an OPEN-step. Each atomic step of the ambients calculus can be decomposed in two parts: checking whether the local structure of the ambient tree enables the step, and actually performing the migration.

The first part imposes some distributed synchronization. One may use a global synchronous primitive existing at the operating system or networking level, but such a solution is not much portable, nor does exist in large scale networks. A first range of solutions can be designed by considering locks and critical sections in order to serialize the implementation of atomic steps. For instance, the two ambients participating to a reduction step are temporary locked, and the locks are released at the end of the step. However this solution cannot be symmetric, in the same way as there is no symmetric distributed solution to the Dining Philosophers problem. Some ambients have to be distinguished, for instance, one ambient could be the synchronizer of all ambients. Naturally, the nested structure of ambients can be used, for instance each ambient controls the synchronization of its direct subambients. In both cases, one has to be careful to avoid deadlocks or too much serialization. This solution would be similar to Cardelli's centralized implementation of an earlier variant of the ambient calculus in Java™ [2,3]. One advantage of a serialized solution is the ease of the correctness proof of the implementation. On the negative side, each attempt to perform a step takes several locks higher up in the ambient hierarchy; these locks may be located at remote sites, leading to long delays before these locks are released for other local steps. Moreover, due to the mobility discipline of the ambient calculus, an ambient that migrates from one point to another in the ambient hierarchy has to travel through an ambient enclosing both the origin and the destination, thus inducing global bottlenecks.

A different set of solutions is fully asynchronous. Atomic steps of ambients are decomposed into several elementary steps, each involving only local synchronization. In this approach, each ambient step is implemented as a run of a protocol involving several messages. Concurrency is higher, as only the moving ambient might not be available for other reduction steps. For instance, our solution never blocks steps involving parents of a moving ambient. Our solution even allows

ambients to move out of a moving ambient; or to complete an IN-move towards a moving ambient; but it does not allow ambients to initiate such IN-moves. The implementation of migration towards a mobile target may be problematic, but can be handled independently of the implementation of ambient synchronization, e.g., using a forwarding mechanism. In our case, we simply rely on the strong migration primitive of Jocaml. On the negative side, the correctness proof is more involved.

In this paper, we present an asynchronous distributed algorithm for implementing ambients, we make it precise as a translation into the Join Calculus—the process calculus which is a model of Jocaml[9], and we refine this translation into a distributed implementation of ambients written in Jocaml. The algorithm provides an insight into the implementability of ambients. The Jocaml prototype is a first, lightweight, distributed implementation of ambients. The translation is proved correct in two stages: first we use barbed coupled-similarity for the correctness of the algorithm, then we use an hybrid barbed bisimulation for the actual translation to the join calculus. Technically, the first stage is a first application of coupled-simulations [18] in a reduction-based, asynchronous setting; it relies on the introduction of an auxiliary ambient calculus extended with transient states; it does not depend of the target language. The second stage is a challenging application of the decreasing diagram technique [17]. In combination, these results imply that the translation does not affect a variety of global observation predicates.

The paper is organized as follows. In section 2, we present the asynchronous algorithm and we show a formal translation from ambient processes to join calculus processes. In section 3, we discuss the correctness of the translation in terms of observations. In section 4, we focus on the operational correspondence between a process and its translation; to this end, we refine the ambient calculus to express additional transient states induced by the translation. In section 5, we state our main technical results and give an idea of their proofs. In section 6, we describe more practical aspects of the Jocaml implementation. We conclude in section 7. In an appendix, we recall the operational semantics of the distributed join calculus and of the calculus of mobile ambients, and we give an overview of both calculi. Additional discussions, technical details, and all the proofs appear in the full version of this paper [10].

2 From ambients to the join calculus

We describe the asynchronous algorithm, then we specify it as a translation from ambients to the join calculus. We begin with a fragment of the ambient calculus given by the grammar $P ::= \; a[\,P\,] \;\mid\; \mathtt{in}\; a.P \;\mid\; \mathtt{out}\; a.P \;\mid\; P \mid P' \;\mid\; \mathbf{0}$. In a second stage, we incorporate OPEN steps and other ambient constructs.

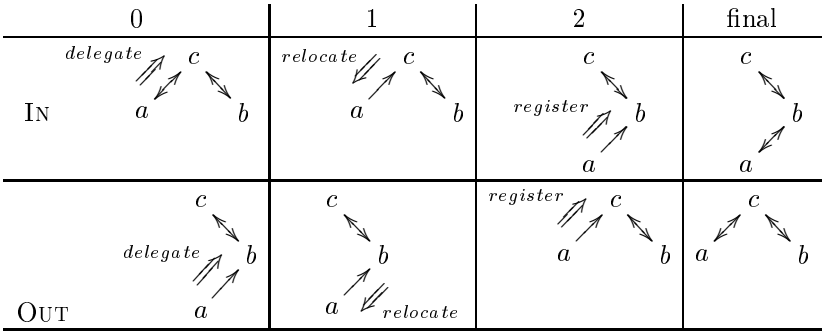
2.1 An asynchronous algorithm

The dynamic tree structure of ambients is represented by a doubly linked tree. Each node in the tree implements an ambient: each node contains non-ambient

processes such as $\texttt{in}\ b.P$ or $\texttt{out}\ a.c[Q]$ running in parallel; each node also hosts an *ambient manager* that controls the steps performed in this ambient and in its direct subambients. Different nodes may be running at different physical sites, so their ambient managers should use only asynchronous messages to communicate with one another. Since several ambients may have the same name, each node is also associated with a unique identifier. (Informally, we still refer to ambients by name, rather than unique identifier.)

Each ambient points to its subambients and to its parent ambient. The down links are used for controlling subambients, the up link is used for proposing new actions. The parent of the moving ambient for an In-step knows the destination ambient; the parent also knows the destination ambient—its own parent—for an Out-step; it controls the opened ambient for an Open-step. Hence, the decision to perform a step will always be taken by the parent of the affected ambient.

Ambient moves of a in and out of b correspond to three successive steps, depicted below. Single arrows represent current links; double arrows represent messages in transit.



We detail the dynamics of an In-step, e.g., $c[\ a[\,\texttt{in}\ b.Q\ ]\ \mid\ b[\,0\,]\,] \rightarrow c[\ b[\ a[\,Q\,]\ ]\ ]$.

0-step: initially, a delegates the migration request In b to its current parent (here c); to this end, it uses its current up link to send a message to c saying that a is willing to move into an ambient named b.

1-step: the enclosing ambient c matches a's request with a's and b's down links. Atomically, a's request and the down link to a are erased, and a relocation message is sent to a; this message contains the address of b, so that a will be able to relocate to b, and also a descriptor of a's successful action, so that a can complete this step by triggering its guarded process.

2-step: the moving ambient a receives c's relocation message, relocates to b's site, and updates its up link to point to b. It also sends a message to b that eventually registers a as a subambient of b.

The 1-step may preempt other actions delegated by a to its former parent c. Such actions should now be delegated to its new parent b. For that purpose, a's ambient manager keeps a log of the pending actions delegated in 0-steps, and, as it completes one of these action in a 2-step, it re-delegates all other actions towards its new parent. The log cannot be maintained by the parent, because

delegation messages may arrive long after a's departure. Moreover, in the case an ambient moves back into a former parent, former delegation messages may still arrive, and should not be confused with fresh ones. Such stale messages must be deleted. This is not directly possible in an asynchronous world, but equivalently each migration results in a modification of the unique identifier of the moving ambient, each delegation message is tagged with this identifier, and the parent discards every message with an old identifier.

An OUT-step of a out of b corresponds to the same series of three steps. The main different is in step 1, as the enclosing ambient b matches a's request with a's down link and its own name b, and passes its own up link to c in the relocation message sent back to a.

2.2 A simple translation

The compositional translation $[\![\cdot]\!]_e$ appears in Figure 1. It uses a (meta) function $AM_{a,e}(\cdot)$ implementing the ambient manager. Overall, the tree of nested ambients is mapped to an isomorphic tree of nested locations. Each ambient is mapped to a join calculus location containing the definition of the channel names that form the ambient interface, and containing processes that represent the ambient state. The definition D is composed of three groups of rules D_0, D_1, and D_2 that respectively implement 0, 1, and 2-steps of the algorithm.

To represent the data structure used in the algorithm of section 2.1, an ambient is represented by an interface e, which is a record that contains fields *here*, *amb*, sub_{in}, sub_{out}, *reloc*, *in*, and *out*. The *here*-field is the name of the location hosting the ambient, whereas the other fields are channel names used to interact with this ambient. The translation is parameterized by the interface e of the current enclosing ambient. A down link to a subambient named b with interface e_b and unique identifier (uid) j is represented as a message $amb(j, b, e_b)$. For every ambient, the up link to its parent ambient is represented by the parent interface e, which is stored in the state message $s(a, i, e, l)$. In addition, the state message contains the name a and the current uid i of the ambient, and the log l of IN and OUT actions that have been delegated to the parent ambient using e.

We resume our study of an IN-action, considering the role of each message in the translation of $c[\, a[\, \texttt{in}\, b.Q\,]\, |\, b[\, 0\,]]$. Initially, the translation defines a continuation κ for Q and issues a message $in(b, \kappa)$ in a, which is a subjective migration request into an ambient named b.

The first step consists of delegating the request to the parent ambient. Using the first rule of a's D_0, the messages $s(a, i, e, l)$ and $in(b, \kappa)$ are consumed, the request is recorded in a's log as an entry IN b κ, and the request is forwarded to the enclosing ambient c described by the interface e. The ambient a remains active, with new state $s(a, i, e, l \cup \{\text{IN}\ b\ \kappa\})$. In parallel, the message $e.sub_{in}(i, b, \kappa)$ is a subambient move request sent to c, with the explicit identifier i of the requester a.

The second step is performed by c's ambient manager. The message $sub_{in}(i, b, \kappa)$ may be consumed using the first rule of D_1. The rule also requires that both the ambient that issued the request and another destination ambient with name b

$$
\begin{aligned}
[\![\, a[\, P\,]\,]\!]_e &= \texttt{def}\ AM_{a,e}(P)\ \texttt{in}\ \mathbf{0} \\
[\![\texttt{in}\ a.P]\!]_e &= \texttt{def}\ \kappa() \triangleright [\![P]\!]_e\ \texttt{in}\ e.in(a,\kappa) \\
[\![\texttt{out}\ a.P]\!]_e &= \texttt{def}\ \kappa() \triangleright [\![P]\!]_e\ \texttt{in}\ e.out(a,\kappa) \\
[\![P \mid Q]\!]_e &= [\![P]\!]_e \mid [\![Q]\!]_e \\
[\![\mathbf{0}]\!]_e &= \mathbf{0}
\end{aligned}
$$

where the ambient manager $AM_{a,e}(P)$ is defined as:

$$
\begin{aligned}
D_0 \stackrel{\text{def}}{=}\ & s(a,i,e,l) \mid in(b,\kappa) \triangleright s(a,i,e,l \cup \{\text{In } b\ \kappa\}) \mid e.sub_{in}(i,b,\kappa) \\
\wedge\ & s(a,i,e,l) \mid out(b,\kappa) \triangleright s(a,i,e,l \cup \{\text{Out } b\ \kappa\}) \mid e.sub_{out}(i,b,\kappa) \\
D_1 \stackrel{\text{def}}{=}\ & s(a,i,e,l) \mid amb(j,b,e_b) \mid amb(k,c,e_c) \mid sub_{in}(k,b,\kappa) \triangleright \\
& s(a,i,e,l) \mid amb(j,b,e_b) \mid e_c.reloc(e_b,\kappa) \\
\wedge\ & s(a,i,e,l) \mid amb(j,b,e_b) \mid sub_{out}(j,a,\kappa) \triangleright s(a,i,e,l) \mid e_b.reloc(e,\kappa) \\
D_2 \stackrel{\text{def}}{=}\ & s(a,i,e,l) \mid reloc(e',\kappa) \triangleright \texttt{go}(e'.here); (I_{a,e_h,e'} \mid \kappa() \mid Flush(l,in,out,\kappa)) \\
D \stackrel{\text{def}}{=}\ & D_0 \wedge D_1 \wedge D_2
\end{aligned}
$$

$$
I_{a,e_h,e} \stackrel{\text{def}}{=} \texttt{def uid}\ i\ \texttt{in}\ s(a,i,e,\emptyset) \mid e.amb(i,a,e_h)
$$

$$
AM_{a,e}(P) \stackrel{\text{def}}{=} here\left[D\ :\ I_{a,e_h,e} \mid [\![P]\!]_{e_h}\right]
$$

with the record notation $e_h \stackrel{\text{def}}{=} \left\{ \begin{array}{l} here = here, amb = amb, sub_{in} = sub_{in}, \\ sub_{out} = sub_{out}, reloc = reloc, in = in, out = out \end{array} \right\}$.

Figure1. Translation from In/Out ambient processes to the join calculus

be actually present. This step removes the down link for the moving ambient—the message $amb(i,a,e_a)$—, thus blocking other actions competing for the same message, whereas the destination ambient remains available for concurrent steps. A relocation message $e_a.reloc(e_b,\kappa)$ is emitted, signalling to the requesting ambient a that it must migrate to the ambient with interface e_b, with continuation κ.

The third step, using a's rule D_2, consumes the message on $reloc$ and the current state message, performs a join calculus migration to the location of the destination ambient, then resumes its activities using the new interface e_b. To this end, the process I_{a,e_a,e_b} restores an active state: it generates a fresh uid i', issues a local state message $s(a,i',e_b,\emptyset)$ representing the up link, and sends to the new parent a message $e_b.amb(i',a,e_a)$ representing the down link. (Since no down link will ever mention the previous uid i, previous delegation messages mentionning i will never match a rule of D_1. In our implementation, these stale messages are actually discarded.) In addition, the message $\kappa()$ trigger the continuation. Finally, the process $Flush(l \cup \{\text{In } b\ \kappa\}), in, out, \kappa)$ restarts any preempted actions appearing in the log. As defined in appendix 7, this process emits a message $in(d,\kappa')$ or $out(d,\kappa')$ in a for every entry In $d\ \kappa'$ or Out $d\ \kappa'$ appearing the log l. These entries correspond to actions preempted by the migration; they will be delegated to a new parent through other iterations of 0-steps of the algorithm.

$$
\begin{aligned}
[\![\texttt{open}\ a.P]\!]_e &= \texttt{def}\ \kappa() \triangleright [\![P]\!]_e\ \texttt{in}\ e.open(a,\kappa)\\
[\![\langle n\rangle]\!]_e &= e.snd(n)\\
[\![(n).P]\!]_e &= \texttt{def}\ \kappa(n) \triangleright [\![P]\!]_e\ \texttt{in}\ e.rcv(\kappa)\\
[\![!P]\!]_e &= \texttt{def}\ \kappa() \triangleright [\![P]\!]_e \mid \kappa()\ \texttt{in}\ \kappa()\\
[\![\nu a.P]\!]_e &= \texttt{def}\ \texttt{fresh}\ a\ \texttt{in}\ [\![P]\!]_e
\end{aligned}
$$

with additional rules in the definition of $AM_{a,e}(P)$:

$$
\begin{aligned}
D_1' &\stackrel{\text{def}}{=} s(a,i,e,l) \mid amb(j,b,e_b) \mid open(b,\kappa) \triangleright s(a,i,e,l) \mid e_b.opening(\kappa)\\
D_2' &\stackrel{\text{def}}{=} s(a,i,e,l) \mid opening(\kappa) \triangleright f(e) \mid \kappa() \mid Flush(l, e.in, e.out, \kappa)\\
D_C &\stackrel{\text{def}}{=} s(a,i,e,l) \mid rcv(\kappa) \mid snd(n) \triangleright s(a,i,e,l) \mid \kappa(n)\\
D_F &\stackrel{\text{def}}{=} f(e) \mid in(b,\kappa) \triangleright f(e) \mid e.in(b,\kappa)\\
&\quad \wedge\ f(e) \mid out(b,\kappa) \triangleright f(e) \mid e.out(b,\kappa)\\
&\quad \wedge\ f(e) \mid open(b,\kappa) \triangleright f(e) \mid e.open(b,\kappa)\\
&\quad \wedge\ f(e) \mid amb(j,b,e_b) \triangleright f(e) \mid e.amb(j,b,e_b)\\
&\quad \wedge\ f(e) \mid sub_{in}(k,b,\kappa) \triangleright f(e) \mid e.sub_{in}(k,b,\kappa)\\
&\quad \wedge\ f(e) \mid sub_{out}(k,b,\kappa) \triangleright f(e) \mid e.sub_{out}(k,b,\kappa)\\
&\quad \wedge\ f(e) \mid rcv(\kappa) \triangleright f(e) \mid e.rcv(\kappa)\\
&\quad \wedge\ f(e) \mid snd(b) \triangleright f(e) \mid e.snd(b)
\end{aligned}
$$

$$
D \stackrel{\text{def}}{=} D_0 \wedge D_1 \wedge D_1' \wedge D_2 \wedge D_2' \wedge D_C \wedge D_F
$$

with the extended record notation

$$
e_h \stackrel{\text{def}}{=} \left\{ \begin{array}{l} here = here, amb = amb, sub_{in} = sub_{in}, sub_{out} = sub_{out}, open = open, \\ reloc = reloc, in = in, out = out, opening = opening, rcv = rcv, snd = snd \end{array} \right\}
$$

Figure2. Additional clauses for the full translation

Similarly, an OUT-step is performed according to the algorithm by using the second rule of D_0 of the moving ambient, the second rule of D_1 of the enclosing ambient, and finally the rule D_2 of the moving ambient.

2.3 Dealing with other ambient constructs

The translation of Figure 2 generalizes the translation above to the full ambient calculus. For each additional construct, we add a clause to the compositional translation $[\![\ \cdot\ ]\!]_e$. We also upgrade $AM_{a,e}(\,\cdot\,)$, and use a larger environment e with extra fields for *open*, *opening*, *rcv*, and *snd*. New constructs to consider are:

Values and Scopes. Ambient names are mapped to identical names in the join calculus. The two calculi rely on similar lexical scope disciplines, with scope extrusion performed by structural equivalence (rule SCOPE in join, rules R1 and R2 in ambients). Thus, it suffices to translate the creation of local ambient names $\nu a.P$ into the creation of binders `fresh` a in join-calculus definitions.

Communication. Ambient communication is implemented by supplementing every ambient manager with a rule D_C that binds two channels *snd* and *rcv* and synchronizes message outputs and message requests. This encoding is much like the encoding of pi-calculus channels into the join calculus (see [8]).

Replication. Each replicated process $!P$ is coded using a standard recursive encoding of infinite loops in the join calculus.

Open. Ambient processes may dissolve ambient boundaries using the `open` a capabilities. In contrast, join calculus names are statically bound to their defining location, and location boundaries never disappear. We thus lose the one-to-one mapping from ambients to locations, and distinguish two states for each AM location: either the ambient is still running—and the message s is present, or it has been opened—and henceforth messages sent to its interface are forwarded to the ambient that performed the open. The indirection is achieved through the use of message f. This leads to complications in the proofs, as one must prove that these opened locations do not interfere with the rest of the translation.

3 Correctness of the translation

The distributed synchronization algorithm significantly departs from the operational semantics of ambients; besides, the translation of nested ambients may yield arbitrarily large terms with numerous instances of the algorithm running in parallel. This makes the correctness of the distributed implementation problematic. On the other hand, both calculi have a reduction-based semantics, which can be equipped with standard notions of observables. This provides a precise setting for establishing correctness on the translation, rather than on an abstraction of the algorithm. (Of course, there are still minor discrepancies between the translation and the actual code in Jocaml; see section 6.)

We first define a syntactic notion of observation. For each calculus, we use a family of predicates on processes indexed by names, written $P \downarrow_b$.

- In the ambient calculus, $P \downarrow_b$ when b is free in P and $P \equiv \nu\widetilde{v}.\, b[\,Q\,] \mid R$.
- In the join calculus, $P \downarrow_b$ when b is free in P and $P \equiv \alpha[D' : b(\widetilde{v}) \mid P'] \wedge D$.

Next, we express the correctness of the translation in terms of the following predicates, for both ambient and join processes:

- A process P has a *weak barb* on b (written $P \Downarrow_b$) when $P \to^* P' \downarrow_b$.
- A process P *diverges* (written $P \Uparrow$) when P has an infinite series of steps.
- A process P has a *fair-must barb* on b (written $\Box P \Downarrow_b$) when for all P' such that $P \to^* P'$, we have $P' \Downarrow_b$.

In combination, these predicates give a precise contents to the informal notion of correctness: "the translation should neither suppress existing behaviors, nor introduce additional behaviors." The minimal notion of correctness for an implementation is the reflection of weak barbs, which rules out spurious behaviors; the converse direction states that the implementation does not discard potential

behaviors; for instance, it rules out deadlocks, or even an empty translation. The preservation of convergence is of pragmatic importance. In addition, correctness for fair-must tests relates infinite computations, and rules out implementations with restrictive scheduling policies.

Since top-level ambients are not translated into messages on free names, the observation of translated ambients requires some special care. To this end, we supplement the top-level translation with a mechanism that can test for a particular barb $\Downarrow_b$ in the source calculus. For a given ambient interface e, we let D and Q be defined by $[\![a[\,\mathbf{0}\,]]\!]_e = \mathtt{def}\ here[D : Q]\ \mathtt{in}\ \mathbf{0}$, and let e_0 be the record obtained by substituting the empty word for $e_0.here$ in the interface e_h defined by D. We use the definition $D_t \stackrel{\text{def}}{=} s(a,i,e,l) \mid amb(j,b,e_b) \mid t(b) \triangleright s(a,i,e,l) \mid amb(j,b,e_b) \mid yes()$ and write $[\![\cdot]\!]^b$ for the translation that maps every process P to the process $[D \wedge D_t : Q \mid t(b) \mid [\![P]\!]_{e_0}]$. Without loss of generality, we always assume that the names in a, t, e, yes do not clash with free names of P, and that the location and channel names of the translation do not clash with any name of P.

We are now ready to state that all the derived observations discussed above are preserved and reflected by the translation:

Theorem 1. *For every ambient process P and name b, we have $P \Downarrow_b$ if and only if $[\![P]\!]^b \Downarrow_{yes}$; $P \Uparrow$ if and only if $[\![P]\!]^b \Uparrow$; and $\Box P \Downarrow_b$ if and only if $\Box[\![P]\!]^b \Downarrow_{yes}$.*

While correctness is naturally expressed in terms of observables along the reduction traces of processes, its proof is challenging. In particular, a direct approach clearly leads to intractable inductions on both the syntax of the source process and on the series of reductions.

4 A calculus of Ambients extended with transient states

In order to prove theorem 1, we introduce an intermediate calculus of ambients with constructs that materialize the key transient states of the algorithm of section 2.1, and we equip this calculus with a reduction semantics in direct correspondence with the algorithm. For instance, atomic In steps are decomposed into series of 1- and 2-steps. (However, 0-steps are not represented, inasmuch as requests are always eventually delegated to the current parent.) In the next section, we rely on the extended calculus to establish correctness as the composition of two equivalences. First, we will use *coupled simulations* [18] to relate the two semantics for ambients; then, we will use bisimulations to relate ambients equipped with the extended semantics to their join calculus translations.

The grammar for the extended calculus appears in Figure 3. It has new processes representing ambients which are committed to move or to be opened, as the result of their father's 1-step—we call such transient ambients *stubs*—and also new processes marking the future position of migrating ambients—we call such precursors *scions*. Pairs of stubs and scions are syntactically connected by a marker i.

$P ::=$		extended ambient process
	$\ldots$	*all the constructors of Figure 5* on page 17
	$\mid\ Xa[\,P\,]$	stub
	$\mid\ i$	scion
	$\mid\ \nu i.P$	marker restriction
$X ::=$		state extension
	$\overline{\imath}\{P\}$–	the stub is committed to move to i
	$\mid\ \mathtt{o}\{P\}$–	the stub is being opened

Well-formed conditions: stubs and scions may occur only in extended evaluation contexts; restricted markers i must be used linearly (exactly one stub and one scion).

Figure3. Syntax for an ambient calculus extended with transient states

Extended evaluation contexts $E(\cdot)$ are defined by the grammar

$$E(\cdot) ::= \ \cdot \ \mid\ P \mid E(\cdot) \ \mid\ E(\cdot) \mid P \ \mid\ X^{=}a[\,E(\cdot)\,] \ \mid\ \nu a.E(\cdot) \ \mid\ \nu i.E(\cdot)$$

Structural equivalence $\equiv$ is the smallest equivalence relation on processes that is closed by application of extended evaluation contexts and by α-conversion, and that satisfies the axioms P0, P1, P2, C0, C1, R1, R2 of figure 6 and:

$$\text{R2X}\ \frac{a \neq n \quad a \text{ not free in } X}{Xn[\nu a.P] \equiv \nu a.Xn[P]}$$

Labeled transitions $\xrightarrow{\alpha}$ is the smallest family of relations closed by application of restriction-free extended evaluation contexts and such that

$$\textsc{Stub}\ \ \overline{\imath}\{Q\}\text{–}n[R] \xrightarrow{\overline{\imath}.n[Q|R]} \mathbf{0} \qquad \textsc{Scion}\ \ i \xrightarrow{i.P} P \qquad \textsc{Scope}\ \ \frac{P \xrightarrow{\alpha} P' \quad v \text{ not free in } \alpha}{\nu v.P \xrightarrow{\alpha} \nu v.P'}$$

Reduction steps $\rightarrow$ are defined as in Figure 6 with extended evaluation contexts. *Initial steps* $\rightarrow_1$, *Completion steps* $\rightarrow_2$, *and Other steps* $\rightarrow_C$ are the smallest relations closed by structural equivalence, by application of extended evaluation contexts, and such that:

$$\textsc{In 1}\ \ \begin{array}{c} m[P] \mid n[\mathtt{in}\ m.Q \mid R] \\ \rightarrow_1 \nu i.\ m[i \mid P] \mid \overline{\imath}\{Q\}\text{–}n[R] \end{array} \qquad \textsc{Out 1}\ \ \begin{array}{c} X^{=}m\big[n[\mathtt{out}\ m.Q \mid R] \mid P\big] \\ \rightarrow_1 \nu i.\ i \mid X^{=}m\big[\overline{\imath}\{Q\}\text{–}n[R] \mid P\big] \end{array}$$

$$\textsc{Move 2}\ \ \frac{P \xrightarrow{\overline{\imath}.S} P' \quad Q \xrightarrow{i.S} Q'}{\nu i.(P \mid Q) \rightarrow_2 P' \mid Q'}$$

$$\textsc{Open 1}\ \ \mathtt{open}\ n.Q \mid n[R] \rightarrow_1 \mathtt{o}\{Q\}\text{–}n[R] \qquad \textsc{Open 2}\ \ \mathtt{o}\{Q\}\text{–}n[R] \rightarrow_2 Q \mid R$$

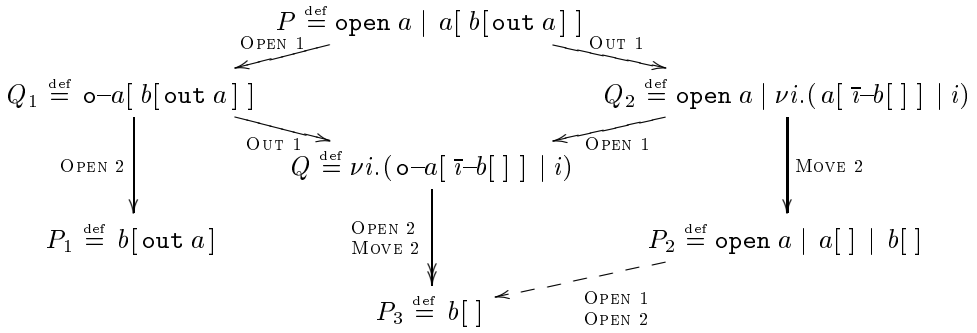
$$\textsc{Recv}\ \ \langle a\rangle \mid (x).P \rightarrow_C P\{{}^{a}/_{x}\} \qquad \textsc{Repl}\ \ !P \rightarrow_C P \mid !P$$

Figure4. Semantics for an ambient calculus extended with transient states

The extended operational semantics appears in Figure 4. (The term $X^{=}a[P]$ stands for either $Xa[P]$ or $a[P]$.) It is a reduction-based semantics with auxiliary labels for stubs and scions. Each of the reduction steps In, Out, and Open is decomposed in two steps. Initial steps $\rightarrow_1$ introduce stubs and scions; completion steps $\rightarrow_2$ consume them. The reduction rules for Recv and Repl are those of the original semantics, except that we write $\rightarrow_C$ instead of $\rightarrow$. Overall, we obtain a reduction system for extended ambient processes with steps $\rightarrow_{12C} \stackrel{\text{def}}{=} \rightarrow_1 \cup \rightarrow_2 \cup \rightarrow_C$. For the sake of comparison, we also extend the original reduction semantics and the observation predicates from ambients to extended ambients.

Initially, stubs and scions are neighbors, but they may drift apart as the result of other steps before performing the matching completion step, so an auxiliary labeled transition system is used to match stubs and scions. The completion of a deferred migration is thus rendered as a global communication step between processes residing in two different ambient contexts, and syntactically linked by the names $\overline{\imath}$ and i. This may seem difficult to implement, but actually scions i have no operational contents; they represent the passive target for a strong migration.

To illustrate the extended calculus, we give below the reductions for a process with a critical pair. Processes P, P_1, P_2, P_3 are regular ambient processes; processes Q, Q_1, Q_2 are transient processes reflecting intermediate states of our algorithm.



5 Coupled simulations and operational correspondence

We continue our study of correctness in terms of equivalences based upon weak barbs. These equivalences are essential to obtain a modular proof. As a side benefit, they also provide a finer account of correctness. (See [7] for a discussion of equivalences and encodings in process calculi.) Instead of equivalences, we actually often use relations ranging over different domains, equipped with different notions of reduction steps and observations.

Definition 1 (Barbed Bisimulations). *A relation $\mathcal{R} \in \mathcal{P} \times \mathcal{P}'$ is a weak barbed simulation when, for all $P\ \mathcal{R}\ Q$, we have*

1. *if* $P \to^* P'$, *then there exists* Q' *such that* $Q \to^* Q'$ *and* $P' \mathcal{R} Q'$;
2. *if* $P \Downarrow_b$, *then* $Q \Downarrow_b$.

$\mathcal{R}$ *is a barbed bisimulation when* $\mathcal{R}$ *and its converse* $\mathcal{R}^{-1}$ *are barbed simulations.*

Bisimulations come with effective proof techniques that consider only a few steps at a time, rather than whole execution traces. Unfortunately, barbed bisimilarity $\approx$—the largest barbed bisimulation closed by application of evaluation contexts—is too discriminating for our protocol. Transient processes such as Q_1 in the example above account for a partially-committed internal choice: Q_1 may reduce to P_1 and P_3, but not to P_2. In some contexts, they are not bisimilar to any derivative of P in the original ambient semantics. To address this issue of gradual commitment, Parrow and Sjödin proposed coarser relations called coupled simulations [18,16]. We liberally adapt their definition to ambients:

Definition 2 (Coupled simulations). *The relations* $\leqslant \in \mathcal{P} \times \mathcal{P}'$ *and* $\leqslant \in \mathcal{P}' \times \mathcal{P}$ *form a pair of* barbed coupled simulations *when* $\leqslant$ *and* $\leqslant$ *are barbed simulations that meet the coupling conditions:*

1. *if* $P \leqslant Q$, *then for some* Q' *we have* $Q \to^* Q'$ *and* $Q' \leqslant P$;
2. *if* $Q \leqslant P$, *then for some* P' *we have* $P \to^* P'$ *and* $P' \leqslant Q$.

The discrepancy between $\leqslant$ and $\geqslant$ is most useful for handling transient states such as Q_1. The coupling conditions guarantee that every transient state can be mapped both to a less advanced state and to a more advanced one. In our case, we would have $Q_1 \leqslant P$ and $P_i \leqslant Q_1$ for $i = 1, 3$.

Correctness of the asynchronous algorithm. The first stage of our correctness argument is expressed as a coupled-simulation between ambient processes equipped with the original and the extended semantics. In the statement below, related processes have the same syntax, but live in different calculi, equipped with different reduction semantics.

Theorem 2. *Let* $\lesssim$ *be the union of* $\leqslant \cap \geqslant$ *for all barbed coupled simulations between ambient and extended ambient processes that are closed by application of evaluation contexts. For all ambient processes* P, *we have* $P \lesssim P$.

The proof appears in [10]; it makes apparent some subtleties of our algorithm due to additional concurrency. After a first series of results on partial commutation properties for extended steps, the key lemmas establish that, for any ambient process P and extended process Q, if $P \to^*_{12C} Q$, then (1) for some ambient process P' we have $Q \to^*_{12C} P'$ and (2) for any such process P', we also have $P \to^* P'$ in the original semantics.

A sketch of the proof for (2) is as follows: by induction on the number of extended steps, we use partial commutation properties of 1- and 2-steps to bring forward a pair of matching steps $\to_1 \to_2$, which can trivially be replaced with $\to$. We rely on an iterative procedure for selecting an adequate pair of steps. When the series begin with a 1-step, we partition the following steps into external steps

(steps that involve ambients that are external to the stub created by the first step), entering steps (MOVE 2-steps traversing this stub), and internal steps. If there are internal 1-steps, we show that the first one commutes with previous steps and can be brought forward at the beginning of the series. We repeat the procedure for this new 1-step. Eventually, we obtain a 1-step followed by external steps only, followed by a matching $\rightarrow_2$ step, and we conclude by showing that this $\rightarrow_2$ can be brought in second position.

Operational correspondence. The second stage of the proof relates ambients equipped with the extended semantics to their join calculus translations. It is simpler than the first one, in principle, but its proof is complicated because the translation makes explicit many details of the implementation that are inessential to the algorithm.

In order to express the correspondence of observations across the translation, we supplement the top-level translation $[\![\cdot]\!]^b$ of theorem 1 with an external choice of the ambient barb to be tested. With the same notations, we write $[\![\cdot]\!]^t$ for the translation that maps every process P to the process $[D \wedge D_t : Q \mid p(t) \mid [\![P]\!]_{e_0}]$. As before, we assume that names in a, e, p, t, and yes do not clash with names free in P. At any point, we can use the evaluation context $T_b(\cdot) \stackrel{\text{def}}{=} h_T[p(t) \triangleright t(b) : \mathbf{0}] \wedge (\cdot)$ to test a translated ambient barb on b by testing the plain join calculus barb $T_b(\cdot) \Downarrow_{yes}$. (We have $[\![P]\!]^b \approx T_b([\![P]\!]^t)$ in the join calculus.)

Theorem 3 (Correctness of the translation). *Let $\approx^{aj}$ be the largest bisimulation between extended ambient processes with reductions $\rightarrow_{12C}$ and join processes such that $Q \approx^{aj} R$ implies $Q \Downarrow_b$ iff $T_b(R) \Downarrow_{yes}$. For all ambient processes P, we have $P \approx^{aj} [\![P]\!]^t$.*

In the long version of the paper [10], a more precise *strong bisimulation up to bookkeeping* result is proved for the translations of all reachable extended ambient processes. Since every significant transient state induced by the translation has been lifted to the extended ambient calculus, its proof essentially amounts to an operational correspondence between the two calculi. We partition reductions in the join calculus according to the static rule of the translation being used. For instance, we let $\rightarrow_1$ steps in the join calculus be the steps using a rule of a definition $D_1 \wedge D_1'$ of figure 2; these steps create a *reloc* or an *opening* message, and are in direct operational correspondence with source $\rightarrow_1$ steps. We obtain two main classes of join calculus steps: steps $\rightarrow_{12C}$ that can be traced back to extended ambient steps, and "bookkeeping" steps $\rightarrow_B$, which are auxiliary steps used to trigger continuations, manage the logs, or unfold new ambient managers.

The main lemmas describe dynamic simplifications of derivatives of the translation, which are required to obtain translations of derivatives in the extended source calculus. These lemmas are expressed as elementary commutation diagrams between simplification relations and some families of reduction steps. For instance, one lemma states that "stale messages" can be discarded; another, more complex lemma states that locations and ambient managers representing opened ambients can be eliminated, effectively merging the contents of opened ambients

with the contents of their previously-enclosing ambient. To conclude, we exhibit a bisimulation relation between extended ambients equipped with steps $\rightarrow_{12C}$ and global translations of these extended ambients equipped with steps $\rightarrow_B^* \rightarrow_{12C} \rightarrow_B^*$. The whole proof is structured using the decreasing diagram technique of [17], whose conditions guarantee that every weak simulation diagram in the final proof can be obtained by gluing previously-established diagrams.

6 Distributed Implementation

In this section, we briefly describe the actual implementation in Jocaml. The initialization and the dynamics of distribution for ambient processes among several Jocaml runtimes lead to some design choice, discussed in the long version of the paper [10]. We also refer to [11] for the source code, setup instructions, and examples.

Our implementation closely follows the translation given in figures 1 and 2. Since Jocaml already provides support for mobility, local synchronization, and run-time distribution, our code is very compact—less than 400 lines for the interpreter, less than 40k in bytecode for the object files. The main differences between the formal translation and the code are given below:

- Messages in the implementation may pass names, but also arbitrary chains of capabilities, as in $\langle \texttt{in}\ a.\texttt{out}\ b \rangle \mid !(x).x.\langle x \rangle$ (cf. [5]).
- The implementation is an interpreter, rather than a global translation. Accordingly, it maintains an environment for local variables, and it performs dynamic type checking when a value is used either as a name or as a capability.
- The translation relies on non-linear join-patterns, which are not available in Jocaml. More explicitly, the implementation relies on hash tables to cache some messages: when a message arrives on sub_{in}, sub_{out}, amb, or $open$, either it is immediately used in combination with the message cache, or it is added to the cache.
- The formal translation of replication always yields a diverging computation. More reasonably, the interpreter unfolds replication on demand: since every ambient reduction involves at most two copies of a replicated process, it suffices to initially unfold two copies, then to unfold an additional copy whenever a fresh copy is used or modified. Hence, $!\,a[\,]$ does not diverge, while $!\,a[\,\texttt{in}\ a\,]$ still does.

7 Conclusions

We translated Mobile Ambients into the join calculus, and gave a first, asynchronous, distributed implementation of the ambient calculus in Jocaml, with a high level of concurrency. The synchronization mechanisms of Ambients turned out to be challenging first to implement, then to prove correct. This provides an insight into the ambient calculus as a model of concurrency. At the same time,

this shows how Jocaml and its formal model can be used to tackle distributed and mobile implementations. For instance, the translation takes full advantage of join patterns to describe complex local synchronization steps, while a more traditional language would decompose these steps into explicit series of reads and updates protected by locks.

In order to get a safer and more efficient implementation, one should care about typing information for passed values and mobility capabilities [6,4]. Our implementation insures dynamic type-checking on values, whereas it would be preferable to use the static type-checking discipline of Jocaml. Similarly, static knowledge of the actions that can appear in a given ambient can lead to more efficient, specialized ambient managers.

Finally, little is known about actual programming in ambient, or the relevant abstractions to build a high-level language on top of the ambient calculus. While we did not consider changing the source language, we believe that our implementation provides an adequate platform for experimenting with ambient-based language design. For instance, our translation would easily accomodate the co-capabilities proposed in [14].

Acknowledgments. This work benefited from discussions with Luca Cardelli, Fabrice Le Fessant, and Luc Maranget.

References

1. G. Boudol. Asynchrony and the π-calculus (note). Technical Report 1702, INRIA Sophia-Antipolis, May 1992.
2. L. Cardelli. Ambit, 1997. Available from http://www.luca.demon.co.uk/Ambit/Ambit.html.
3. L. Cardelli. Mobile ambient synchronization. Technical note 1997-013, Digital Systems Research Center, July 1997.
4. L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *ICALP'99*, LNCS, 1999.
5. L. Cardelli and A. Gordon. Mobile ambients. In *FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155, 1998.
6. L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL'99*, pages 79–92. ACM, Jan. 1999.
7. C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming.* PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998. INRIA, TU-0556.
8. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL '96*, pages 372–385. ACM, Jan. 1996.
9. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR '96*, volume 1119 of *LNCS*, pages 406–421, Aug. 1996.
10. C. Fournet, J.-J. Lévy, and A. Schmitt. A distributed implementation of Ambients. Long version of this paper, available from http://join.inria.fr/ambients.html, 1999.
11. C. Fournet and A. Schmitt. An implementation of Ambients in JoCAML. Software available from http://join.inria.fr/ambients.html, 1999.
12. A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. In *FoSSaCS'99*, volume 1578 of *LNCS*, pages 212–226, 1999.

13. F. Le Fessant. The JoCAML system prototype. Software and documentation available from http://join.inria.fr/jocaml, 1998.
14. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*. ACM, Jan. 2000.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, Sept. 1992.
16. U. Nestmann and B. C. Pierce. Decoding choice encodings. In *CONCUR '96*, volume 1119 of *LNCS*, pages 179–194, Aug. 1996. Revised full version as report ERCIM-10/97-R051, 1997.
17. V. Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126:259–280, 1994.
18. J. Parrow and P. Sjödin. Multiway synchronization verified with coupled simulation. In *CONCUR '92*, volume 630 of *LNCS*, pages 518–533, 1992.

Appendix: Two notions of mobile computations

We define our syntax and semantics for the calculus of mobile ambients and for the distributed join calculus, then we give overviews of the two calculi.

Operational semantics for ambients Our syntax and semantics of the calculus of ambients are given in figures 5 and 6.

This presentation slightly differs from Cardelli and Gordon's [5] on several counts. The structural equivalence is more restrictive here; it does not introduce or remove νx binders; it operates only in evaluation contexts. The operational semantics of ambients represents the unfolding of replication as a silent reduction step rather than a structural law. Also, communicated values are just ambient names, rather than both names and chains of capabilities. (Chains of capabilities are fully supported in the Jocaml implementation, but their encoding is heavy.) In spirit, our semantics is actually closer to the harness semantics of [12].

Operational semantics for the join calculus Our syntax and semantics for the join calculus are given in figures 7 and 8.

A *path* α is a string of location names $a, b, \ldots$. *Active locations* are locations not under a `def`. They can be nested. The path of an active sublocation $a[D : P]$ is $\alpha.a$, where α is the path of its enclosing location. A distributed program is a conjunction of top-level locations such that all every location has a unique name and that every channel is defined in at most one location. In a distributed program, a location with path $\alpha.a$ is *frozen* when it is the only top-level location whose path contains a.

Names can be bound either as parameters $\widetilde{y}$ in a message pattern $x(\widetilde{y})$ or as names defined in D by `def` D `in` P. The definition $a[D' : P]$ defines a and names defined in D'. A definition containing a rule with message pattern $x(\widetilde{y})$ also defines x.

To simplify the translation of section , we supplement the join calculus with some convenient extensions, which are easily encoded in the plain join calculus. We use finite sets, interpreted in the standard mathematical sense. We supplement definitions with new constructs `uid` i and `fresh` a that bind names i and a, which we use to generate unique identifiers—we could use instead dummy rules such as $i() \triangleright 0$. We also use a record notation as a shortcut for tuples of names passed in a consistent order, and an algebraic notation In b κ, Out b κ for log entries with tags In, Out and names b, κ.

Rather than making explicit a standard encoding of list iterators for implementing the process $Flush(l, in, out, \kappa)$, we supplement the operational semantics of the join calculus with a rule for flushing logs of messages:

$$\textsc{Flush}\quad Flush(l, in, out, \kappa) \to \prod_{\textsc{In}\ d\ \kappa' \in l | \kappa \neq \kappa'} in(d, \kappa') \quad | \prod_{\textsc{Out}\ d\ \kappa' \in l | \kappa \neq \kappa'} out(d, \kappa')$$

$P ::=$		ambient process
	$a[\,P\,]$	ambient
$\mid$	$P \mid P'$	parallel composition
$\mid$	$C.P$	guarded process
$\mid$	$\nu a.P$	name restriction
$\mid$	$\langle a \rangle$	asynchronous message
$\mid$	$(x).P$	message reception
$\mid$	$!P$	replication
$\mid$	$\mathbf{0}$	inert process
$C ::=$		capability
	$\texttt{in}\ a$	ingoing migration
$\mid$	$\texttt{out}\ a$	outgoing migration
$\mid$	$\texttt{open}\ a$	ambient dissolution

Figure5. Syntax for the ambient calculus

Evaluation contexts $E(\cdot)$ are defined by the grammar

$$E(\cdot) ::= \ \cdot \ \mid \ P \mid E(\cdot) \ \mid \ E(\cdot) \mid P \ \mid \ a[\,E(\cdot)\,] \ \mid \ \nu a.E(\cdot)$$

Structural equivalence $\equiv$ is the smallest equivalence relation closed by application of evaluation contexts, by α-conversion, and such that

$$\begin{array}{ll} \text{P0} & P \mid \mathbf{0} \equiv P \\ \text{P1} & P \mid P' \equiv P' \mid P \\ \text{P2} & (P \mid P') \mid P'' \equiv P \mid (P' \mid P'') \end{array} \qquad \begin{array}{ll} \text{R1} & \dfrac{a \text{ not free in } P}{P \mid \nu a.Q \equiv \nu a.(P \mid Q)} \\ \text{R2} & \dfrac{a \neq n}{a[\nu n.P] \equiv \nu n.a[P]} \end{array}$$

Ambient reduction $\rightarrow$ is the smallest relation closed by structural equivalence, by application of evaluation contexts, and such that

$$\textsc{In} \quad \begin{array}{l} m[\,P\,] \mid n[\,\texttt{in}\ m.Q \mid R\,] \\ \rightarrow\ m\big[\,P \mid n[\,Q \mid R\,]\,\big] \end{array} \qquad \textsc{Out} \quad \begin{array}{l} m\big[\,n[\,\texttt{out}\ m.R \mid R'\,] \mid Q\,\big] \\ \rightarrow\ m[\,Q\,] \mid n[\,R \mid R'\,] \end{array}$$

$$\textsc{Open} \quad \texttt{open}\ n.Q \mid n[\,R\,] \rightarrow Q \mid R$$

$$\textsc{Recv} \quad \langle a \rangle \mid (x).P \rightarrow P\{^{a}/_{x}\} \qquad\qquad \textsc{Repl} \quad !P \rightarrow P \mid !P$$

Figure6. Operational semantics for the ambient calculus

$P ::=$		join calculus process
	$\mathbf{0}$	inert process
$\mid$	$P \mid P'$	parallel composition
$\mid$	$x(\widetilde{y})$	asynchronous message
$\mid$	$\mathtt{go}(a); P$	migration request
$\mid$	$\mathtt{def}\ D\ \mathtt{in}\ P$	local definition
$D ::=$		join calculus definition
	$\top$	void definition
$\mid$	$D \wedge D'$	composition
$\mid$	$J \triangleright P$	reaction rule
$\mid$	$a[D : P]$	sub-location (named a, running D and P)
$\mid$	$\alpha[D : P]$	top-level location (with path α, running D and P)
$J ::=$		join pattern
	$x(\widetilde{y})$	message pattern
$\mid$	$J \mid J'$	synchronization

Figure7. Syntax for the distributed join calculus

Structural equivalence $\equiv$ (on both processes and definitions) is the smallest equivalence relation closed by application of contexts $\cdot \wedge \cdot$, $\cdot \mid \cdot$ and $\alpha[\ \cdot : \cdot\ ]$, by α-conversion on bound names, and such that:

$$\begin{array}{llll}
\text{P0} & P \mid \mathbf{0} \equiv P & \text{D0} & D \wedge \top \equiv D \\
\text{P1} & P \mid P' \equiv P' \mid P & \text{D1} & D \wedge D' \equiv D' \wedge D \\
\text{P2} & (P \mid P') \mid P'' \equiv P \mid (P' \mid P'') & \text{D2} & (D \wedge D') \wedge D'' \equiv D \wedge (D' \wedge D'')
\end{array}$$

$$\textsc{Tree}\ \alpha\big[a[D' : P'] \wedge D : P\big] \equiv \alpha.a[D' : P'] \wedge \alpha[D : P] \qquad \textsc{Scope}\ \frac{\text{names defined in } D' \text{ are fresh}}{\alpha[D : P \mid \mathtt{def}\ D'\ \mathtt{in}\ P'] \equiv \alpha[D \wedge D' : P \mid P']}$$

Join calculus reduction $\rightarrow$ is the smallest relation closed by structural equivalence, by application of contexts $D \wedge \cdot$, and such that:

$$\textsc{Comm}\ \frac{x \text{ is defined in } D'}{\alpha[D : P \mid x(\widetilde{v})] \wedge \beta[D' : P'] \rightarrow \alpha[D : P] \wedge \beta[D' : P' \mid x(\widetilde{v})]} \qquad \textsc{Join}\ \frac{\sigma \text{ operates on message contents of } J}{\alpha[D \wedge J \triangleright Q : P \mid J\sigma] \rightarrow \alpha[D \wedge J \triangleright Q : P \mid Q\sigma]}$$

$$\textsc{Go}\ \frac{a \text{ frozen}}{\beta.b[D' : P'] \wedge \alpha.a[D : P \mid \mathtt{go}(b); Q] \rightarrow \beta.b[D' : P'] \wedge \beta.b.a[D : P \mid Q]}$$

Figure8. Operational semantics for the distributed join calculus

Overview for Ambients The calculus of ambients [5] describes movements of processes through administrative domains. An ambient process P may be either the inert process $\mathbf{0}$, or a composed process $P \mid P'$, or a replicated process $!P$, or an ambient $n[\,P\,]$ with name n, or a process $\nu a.P$ with local name a, or a process $C.P$ guarded by C, or a message reception $(x).P$, or an asynchronous emission $\langle a \rangle$ of name a. Capabilities C represent potential IN/OUT/OPEN actions.

A message $\langle a \rangle$ is communicated locally inside the current ambient, where any receptor $(x).P$ can catch it. In order to send a message to a receptor in a remote ambient, the message needs to be wrapped in an ambient which moves to and is opened by the receiving ambient. As in the join calculus, ambients have a tree structure. The (invisible) root represents the network. Deeper subambients are physical or logical mobile entities. Movements are local; they depend on the local tree structure. An ambient may move to the position of its parent sibling, or of a child of its sibling ambients. It can also open a sibling ambient. An ambient migrating to a remote ambient needs to be aware of the current tree structure to perform elementary moves toward the target ambient.

As an example, consider process $Q = \langle 4 \rangle$ which sends 4 to a secure printer $P = !(x).\mathit{Print}_x$ (assume Print_x is a built-in process printing any value assigned to x). Take the following protocol derived from the firewall example in [5]

$$(\nu w.\, w[\; k[\mathtt{out}\; w.\mathtt{in}\; k'.\mathtt{in}\; w\;] \mid \mathtt{open}\; k'.P]) \;\mid\; k'[\mathtt{open}\; k.Q\,]$$

The firewall sends the subambient of name k to Q to check its key. Then the subambient comes back into the firewall with Q and allows the interaction between P and Q.

As in the π-calculus, the structural equivalence allows scope extrusion for bound ambient names. The equivalence has also commutative and monoidal rules for $\mid$.

$$\begin{aligned}
& \nu w.\, w[\; k[\mathtt{out}\; w.\mathtt{in}\; k'.\mathtt{in}\; w\;] \mid \mathtt{open}\; k'.P] \;\mid\; k'[\mathtt{open}\; k.Q\,] && (1)\\
\to\; & \nu w.\, w[\mathtt{open}\; k'.P\,] \;\mid\; k[\mathtt{in}\; k'.\mathtt{in}\; w] \;\mid\; k'[\mathtt{open}\; k.Q\,] && (2)\\
\to\; & \nu w.\, w[\mathtt{open}\; k'.P\,] \;\mid\; k'[\; k[\mathtt{in}\; w\;] \mid \mathtt{open}\; k.Q] && (3)\\
\to\; & \nu w.\, w[\mathtt{open}\; k'.P\,] \;\mid\; k'[\mathtt{in}\; w \mid Q\,] && (4)\\
\to\; & \nu w.\, w[\mathtt{open}\; k'.P \mid k'[Q\;]\,]\\
\to\; & \nu w.\, w[P \mid Q\,]\\
\to\; & \nu w.\, w[P \mid (x).\mathit{Print}_x \;\mid\; \langle 4 \rangle\,] && (5)\\
\to\; & \nu w.\, w[P \mid \mathit{Print}_4\,] && (6)
\end{aligned}$$

The previous process is equivalent to (1). The k-subambient moves out of the firewall, and the process OUT-reduces to (2). Now the k-ambient can move into the k'-ambient by an IN-reduction (3). Finally the k-ambient is opened (4). The calculation continues with IN/OPEN/REPL-reductions (5). Now P and Q are inside the same w-ambient and can exchange values (6).

Hence, ambient movements are sequences of IN/OUT/OPEN reductions, which modify the tree structure of ambients. As in the join calculus, an ambient moves with all its subambients. Each IN/OUT reduction moves a subtree of the tree

of ambients, but an open reduction removes a single node, and attach sons of the canceled node to their grandparent. Communication with the RECV rule is a communication on an anonymous channel inside the current ambient.

Overview of the join calculus The join calculus [8] is a variant of the asynchronous π-calculus [15,1] with a distributed implementation and a uniform treatment of local and remote communication channels [13]. In this paper, we consider only the distributed join calculus without failures (named join calculus thereafter) [9,7].

Join-terms are either processes or definitions. A process P can be either the inert process $\mathbf{0}$, or a composed process $P_1 \mid P_2$, or the asynchronous sending $x(\widetilde{y})$ of message $\widetilde{y}$ onto channel x, or a migration request $\mathtt{go}(a); Q$ to move the current location to location a and then do Q, or the process Q with local definition D, written $\mathtt{def}\ D\ \mathtt{in}\ Q$.

Definitions of channels give the receptive behavior of new channels. A definition D may be either the void definition $\top$, or a composed definition $D_1 \wedge D_2$, or a reaction $J \triangleright P$ consuming messages that match the pattern J and producing P instead, or a location $\alpha[D : P]$ with path α, active definition D, and running process P. Join-patterns express reception and synchronization of messages. Namely, the join-pattern $x(\widetilde{y})$ waits for the reception of any message $\widetilde{y}$ on channel x. Similarly $J \mid J'$ waits for passing J and J'.

For instance, a counter allocator is expressed as the following join calculus process:

```
def counter (x, κ) ▷
    def count(n) | inc(κ′) ▷ count(n+1) | κ′()
     ∧  count(n) | get(κ′) ▷ count(n) | κ′(n)
    in count(x) | κ(get, inc)
in P
```

The definition of channels `get`, `inc`, and `count` is local to the definition of `counter`. In P, access to `counter` is permitted, and one may pass to it an initial value x, and a continuation κ. A possible value for P is

```
def κ1 (g, i) ▷
    def κ2 () ▷ 0 in i(κ2) | i(κ2) | g(print)
in counter(3, κ1) | counter(10, κ1)
```

where `print` is a built-in channel printing its parameter on a terminal. Two counters are created with initial values 3 and 10, and two values $n \in \{3, 4, 5\}$ and $n' \in \{10, 11, 12\}$ are printed in any order. (Abbreviations for avoiding intensive use of continuations are possible, but will not be considered in this paper.)

Our counter may be localized and mobile. In

```
def counter (user, x, κ) ▷
    def a [    count(n) | inc(κ′) ▷ count(n+1) | κ′()
            ∧  count(n) | get(κ′) ▷ count(n) | κ′(n)
             : go(user); count(x) | κ(get, inc)] in 0
in Q
```

local channels `get`, `inc`, and `count` are now bundled on a new location a and will move with it. Here the target destination `user` is a new parameter of `counter`.

Locations are organized as a tree. Intuitively, the root location is the Internet; first level sub-locations are IP addresses; deeper sub-locations are logical entities ready to move. Here, location a is a sub-location of the current location of `counter`, but will become a sub-location of `user` after moving with `go(user)` and before executing `count(x)` | κ`(get,inc)`.

To explore the dynamics of our example, rewrite the localized-mobile counter above as `def` D_c `in` Q, where D_c is of the form

```
counter (user, x, κ) ▷
def a [D_{i,g} : go(user); count(x) | κ(get, inc)] in 0
```

and where $D_{i,g}$ consists of the two rules defining `count`, `inc`, and `get`. Suppose our example is located at the anonymous root location and that Q contains a first-level location u running the process P.

```
[⊤ : def D_c in def u[⊤ : P] in 0]
```

By erasing the `def` binder (rule Scope), this definition is equivalent to

```
[D_c : def u[⊤ : P] in 0]
```

which again by erasing the `def` binders of u, then κ_1, is equivalent to

```
[D_c ∧ u[D_{κ1,κ2} : counter(u,3,κ1) | counter(u,10,κ1) ] : 0]
```

which by considering the tree-structure (rule Tree) is also equivalent to

```
[D_c : 0] ∧ u[D_{κ1,κ2} : counter(u,3,κ1) | counter(u,10,κ1) ]
```

These steps are part of the structural equivalence defined on join-terms. It requires some subtleties in the treatment of bound variables introduced by the `def` construct, which binds all definitions and locations appearing in it. The structural equivalence takes also into account the monoidal laws of | in processes and $\wedge$ in definitions, and the tree structure of locations.

There are also three reduction rules. The first one (Comm) moves a message on some channel to the (unique) location of its receptor. In our example, the previous definition reduces to

```
[D_c : counter(u,3,κ1)] ∧ u[D_{κ1,κ2} : counter(u,10,κ1)]
```

The second rule (Join) replaces a join-pattern by its definition body. The example definition now reduces to

```
[D_c : def a[D_{i,g} : go(u); count(3) | κ1(get, inc)] in 0]
∧ u[D_{κ1,κ2} : counter(u,10,κ1)]
```

which is structurally equivalent to (by rules Scope and Tree)

$[D_c$: **0**]
$\wedge$ a[$D_{i,g}$: go(u); count(3) | κ_1(get, inc)]
$\wedge$ u[D_{κ_1,κ_2} : counter(u,10,κ_1)]

The third rule (Go) is a subjective move of the location containing the go operator to a sub-location (with same name) of the target location. The example definition reduces to

$[D_c$: **0**]
$\wedge$ u.a[$D_{i,g}$: count(3) | κ_1(get, inc)]
$\wedge$ u[D_{κ_1,κ_2} : counter(u,10,κ_1)]

which is structurally equivalent to

$[D_c$: **0**]
$\wedge$ u[D_{κ_1,κ_2} $\wedge$ a[$D_{i,g}$: count(3) | κ_1(get, inc)] : counter(u,10, κ_1)]

At this point, we see that our counter definition D_c is still at the Internet level, but an instance of the counter is at sub-location a of location u. The calculation proceeds in parallel with the second counter, or by executing continuation κ_1.

The previous three reduction rules express the full dynamics of the join calculus. It is important to notice that mobility affects the locality of terms, but not their functionality, since bindings are preserved through the reduction rules. Another important remark is about the linearity condition of receptors and locations. In any localized definition $a[D : P]$, channel names are defined in a single location and there should be a single definition of each sub-location. This is essential in a distributed implementation, since one always knows where to send any message. The calculus has been implemented in a language named Jocaml [13].