

A distributed abstract machine for Safe Ambients

Davide Sangiorgi
INRIA Sophia-Antipolis, France

Andrea Valente
Università di Torino, Italy

Abstract

An abstract machine for a distributed implementation of an ambient calculus is presented, and proved operationally correct. The abstract machine is different from, and simpler than, previous implementations of ambient-like calculi, mainly because: the underlying calculus is typed Safe Ambients rather than the untyped Ambient calculus; the logical structure of an ambient system and its physical distribution are separated. A sketch of an implementation of the abstract machine in Java is given.

1 Introduction

The Ambient calculus [4] is a model for mobile distributed computing. An ambient is the unit of movement. Processes within the same ambient may exchange messages; ambients may be nested, so to form a hierarchical structure. The three primitives for movement allow: an ambient to enter another ambient, $n[\text{in } m.P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$; an ambient to exit another ambient, $m[n[\text{out } m.P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$; a process to dissolve an ambient boundary thus obtaining access to its content, $\text{open } n.P \mid n[Q] \longrightarrow P \mid Q$.

Several studies of the basic theory of the Ambient calculus have recently appeared, concerning for instance behavioural equivalences, types, logics, static analysis techniques [5, 6, 1, 7, 12]. In comparison, little attention has been given to implementations. The only implementations of Ambients we are aware of are Cardelli's [2, 3], and Fournet, Lévy and Schmitt's [9]. The latter, formalised as a translation of Ambients into the distributed Join Calculus, is the only distributed implementation. Although ingenious, the algorithms that these implementations use for simulating the ambient reductions are fairly complex.

One of the difficulties of a distributed implementation of an ambient-like language is that each movement operation involves ambients on different hierarchical levels. For instance, the ambients affected by an `out` operation are the moving ambient, and its initial and its final parent; at the beginning they reside on three different levels. In [2, 3] locks are used to achieve a synchronisation among all ambients affected by a movement. In a distributed setting, however, this lock-based policy can be expensive. For instance, the serialisations introduced diminish the parallelism of the whole system. In [9] the synchronisations are simulated by means of protocols of asynchronous messages. The problems of implementation have been a restraint to the development of programming languages based on Ambients and to experimentation of Ambients on concrete examples. In our opinion, implementation is one of the aspects of Ambients that most need investigations.

In this paper we study an abstract machine for a distributed implementation of an ambient-like calculus. The algorithms of our abstract machine are quite different from, and simpler than, those of [2, 3, 9], mainly for two reasons. The first – the most important – is that the calculus that we actually take is typed Safe Ambients [11] (SA) rather than untyped Ambients. SA is a variant of the original calculus that eliminates certain forms of interference in ambients, the grave interferences. They are produced when an ambient tries to perform two different movement operations at the same time, as for instance $n[\text{in } h.P \mid \text{out } n.Q \mid R]$. The control of mobility is obtained in SA by a modification of the syntax and a type system. In [11] the absence of grave interferences is used to develop an algebraic theory and prove the correctness of some examples. One of the contributions of this paper is to show that the absence of grave interferences also brings benefits in implementations.

The second reason for the differences in our abstract machine is the separation between the logical structure of an ambient system and its physical distribution. Exploiting this, the interpretation of the movement associated to the capabilities is reversed: the movement of the `open` capability is physical, that is, the location of some processes changes, whereas that of `in` and `out` is only logical, that is, some hierarchical dependencies among ambients may change, but not their physical location. Intuitively, `in` and `out` are acquisition of access rights, and `open` is exercise of them.

The differences also show up in the correctness proof of the abstract machine, which is much simpler than the correctness proof of the Join implementation.

Of course another difference is that our algorithms are formulated as an abstract machine. The machine is independent of a specific implementation language, and can thus be used as a basis for implementations on different languages. In the paper we present one such implementation, written in Java.

2 Safe Ambients: syntax and semantics

We briefly describe typed Safe Ambient (SA), from [11]. In the reduction rules of the original Ambient calculus, mentioned in Section 1, an ambient may enter, exit, or open another ambient. The second ambient undergoes the action; it has no control on *when* the action takes place. In SA this is rectified: *coactions* $\overline{\text{in}}n, \overline{\text{out}}n, \overline{\text{open}}n$ are introduced with which any movement takes place only if both participants agree. The syntax of SA is presented in Table 1. Expressions that are not variables or names are the *capabilities*. We often omit the trailing $\mathbf{0}$ in processes $M.\mathbf{0}$. Parallel composition has the least syntactic precedence, thus $m[M.P \mid Q]$ reads $m[(M.P) \mid Q]$. An ambient, or a parallel composition, or variable, is *unguarded* if it is not underneath a capability or an abstraction. In a recursion $\text{rec } X.P$, the recursion variable X should be guarded in P . For simplicity of presentation we omit path expressions in the syntax.

Below are the reduction axioms: those for movement, and the communication rule (communication is asynchronous, takes place inside ambients, and is anonymous—it does not use channel or process names):

$$\begin{array}{ll}
n[\text{in } m.P_1 \mid P_2] \mid m[\overline{\text{in}}m.Q_1 \mid Q_2] & \longrightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2] & \text{[R-IN]} \\
m[n[\text{out } m.P_1 \mid P_2] \mid \overline{\text{out}}m.Q_1 \mid Q_2] & \longrightarrow n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2] & \text{[R-OUT]} \\
\text{open } n.P \mid n[\overline{\text{open}}n.Q_1 \mid Q_2] & \longrightarrow P \mid Q_1 \mid Q_2 & \text{[R-OPEN]} \\
\langle M \rangle \mid (x)P & \longrightarrow P\{M/x\} & \text{[R-MSG]}
\end{array}$$

The inference rules allow a reduction to occur underneath a restriction, a parallel composition, and inside an ambient. Moreover, the structural congruence relation (\equiv) can be applied before a reduction step. Structural congruence is used to bring the participants of a potential interaction into contiguous positions; its definition is standard, and includes rules for commuting the positions of parallel components, for stretching the scope of a restriction, for unfolding recursions. We write \Longrightarrow for the reflexive and transitive closure of \longrightarrow . The use of coactions, in the syntax and operational rules, is the only difference between (untyped) SA and the original Ambient calculus.

Up to structural congruence, every ambient in a term can be rewritten into a normal form

$$n[P_1 \mid \dots \mid P_s \mid m_1[Q_1] \mid \dots \mid m_r[Q_r]]$$

where P_i ($i = 1..s$) does not contain unguarded ambients or unguarded parallel compositions. In this case, P_1, \dots, P_s are the *local processes* of the ambient, and $m_1[Q_1] \mid \dots \mid m_r[Q_r]$ are the *subambients*.

SA has two main kinds of types: *single-threaded* and *immobile*. We consider them separately. We begin with the single-threaded types, which we informally describe below. We consider immobility types in Section 6.

h, k, \dots, n, m	<i>Names</i>	x, y, z	<i>Variables</i>
		X, Y, Z	<i>Recursion variables</i>
	<i>Expressions</i>		<i>Processes</i>
M, N	$:= x$ (<i>variable</i>)	P, Q, R	$:= \mathbf{0}$ (<i>nil</i>)
	n (<i>name</i>)	$P \mid Q$ (<i>parallel</i>)	
	$\mathbf{in} M$ (<i>enter</i>)	$(\nu n) P$ (<i>restriction</i>)	
	$\overline{\mathbf{in}} M$ (<i>allow enter</i>)	$M.P$ (<i>prefixing</i>)	
	$\mathbf{out} M$ (<i>exit</i>)	$M[P]$ (<i>ambient</i>)	
	$\overline{\mathbf{out}} M$ (<i>allow exit</i>)	$\langle M \rangle$ (<i>value message</i>)	
	$\mathbf{open} M$ (<i>open</i>)	$(x)P$ (<i>abstraction</i>)	
	$\overline{\mathbf{open}} M$ (<i>allow open</i>)	X (<i>recursion variable</i>)	
		$\mathbf{rec} X.P$ (<i>recursive process</i>)	

Table 1: The syntax of Safe Ambients

The capabilities of the local processes of an ambient control the activities of that ambient. In an untyped (or immobile) ambient such control is distributed over the local processes: any of them may exercise a capability. In a *single-threaded* (ST) ambient, by contrast, at any moment at most *one* process has the control thread, and may therefore use a capability. An ST ambient n is willing to engage in at most one interaction at a time with external or internal ambients. Inside n , however, several activities may take place concurrently: for instance, a subambient may reduce, or two subambients may interact with each other. Thus, if an ambient n is ST, the following situation, where at least two local processes are ready to execute a capability, cannot occur: $n[\mathbf{in} m.P \mid \mathbf{out} h.Q \mid R]$. The control thread may move between processes local to an ST ambient by means of an open action. Consider, for instance, a reduction

$$n[\mathbf{open} m.P \mid m[\overline{\mathbf{open}} m.Q]] \longrightarrow n[P \mid Q] \quad (1)$$

where n and m are ST ambients. Initially $\mathbf{open} m.P$ has the control thread over n , and $\overline{\mathbf{open}} m.Q$ over m . At the end, m has disappeared; the control thread over n may or may not have moved from P to Q , depending on the type of m . If the movement occurs, Q can immediately exercise a capability, whereas P cannot; to use further capabilities within n , P will have to get the thread back.

For simplicity, we assume here a strong notion of ST, whereby a value message $\langle M \rangle$ never carries the thread. In [11] a weaker notion is used, where also messages may carry the thread. In this case, the control thread over an ambient may move, other than by an open operation, as a result of the consumption of a value by an abstraction. The results in our paper can be adapted to this weaker notion. In the remainder, all processes are assumed to be well-typed, and closed (i.e., without free variables).

3 The abstract machine, informally

We describe the data structures and the algorithms of the abstract machine, called PAN. PAN separates between the logical and the physical distribution of the ambients. The logical distribution is given by the tree structure of the ambient syntax. The physical distribution is given by the association of a location to each ambient.

In PAN, an ambient named n is represented as a *located ambient* $h:n[P]_k$, where h is the location, or site, at which the ambient runs, k is the location of the parent of the ambient, and P collects the processes local to the ambient. While the same name may be assigned to several ambients, a location univocally identifies an ambient; it can be thought of as its physical address.

A tree of ambients is rendered, in PAN, by the parallel composition of the (unguarded) ambients in the tree. In this sense, the physical and the logical topology are separated: the space of physical locations is flat, and each location hosts at most one ambient, but each ambient knows the location at which its parent resides. For instance, an SA term $n[P_1 \mid P_2 \mid m_1[Q_1] \mid m_2[Q_2]]$, where P_1 and P_2 are the local processes of n , and Q_i ($i = 1, 2$) is a local process of m_i (i.e., m_i has no subambients), becomes in PAN:

$$h:n[P_1 \mid P_2]_{\text{root}} \parallel k_1:m_1[Q_1]_h \parallel k_2:m_2[Q_2]_h$$

where h, k_1, k_2 are different location names, **root** is a special name indicating the outermost location, and \parallel is parallel composition of located ambients. (The above configuration is actually obtained after two creation steps, in which the root ambient spawns off the two ambients located at k_1 and k_2 .) Since ambients may run at different physical sites, they communicate with each other by means of *asynchronous* messages.

All the actions (**in**, **out**, and **open**) can modify the logical distribution. Only **open**, however, can modify the physical distribution. The algorithms that PAN adopts to model reduction in SA are based on 3 steps: first, a *request* message is sent upward, from a child ambient that wants to move (logically or physically) to its parent; second, a *match* is detected by the parent itself; third, a *completion* message is sent back to the child, for its relocation. The only exception is the algorithm for **open**, where a further message is needed to migrate the child's local processes to the parent. These steps are sketched in Figures 1-3, where a, b, c represent three ambients, a straight line represents a pointer from an ambient to its parent, and a curved line represents the sending of a message. Thus in Figure 1, at the beginning b and c are sibling ambients and a is their parent. This figure illustrates an R-IN reduction in which b becomes a child of c . In the first phase, b demands to enter c (precisely, if n is the name of c , then b demands of entering an ambient with name n), and c accepts an ambient in. For this, b and c send requests **in** and $\bar{\text{in}}$ to their parent a (the actual messages may also contain the name and location of the sender; these are not shown in the figures). In the second phase, a sees that two matching requests have been sent and authorises the movement. Finally, in the third phase, a sends completion messages to b and c . The message sent to b also contains the location of c , which b will use to update its parent field. An ambient that has sent a request to its parent but has not yet received an acknowledgement back, goes into a *wait* state, in which it will not send further requests. In the figures, this situation is represented by a circle that encloses the ambient. An ambient in a *wait* state, however, can still receive and answer requests from its children and can perform local communications.

Figure 2 sketches an R-OUT reduction. In the first phase, ambient a demands its parent b to exit. When b authorises the movement (phase 2), it sends a an acknowledgement containing the location of the parent of b , namely c , and upon receiving this message (phase 3) a updates its parent field. Note that the grandparent ambient c is not affected by the dialog between a and b . Figure 3 sketches an R-OPEN reduction. Ambient a accepts to be opened, and thus notifies its parent c . If a matching capability exists, that is, one of the processes local to c demands to open a , then c authorises a to migrate its local processes into c . Ambient a then becomes a forwarder ($a \triangleright c$ in the figure) whose job is just to forward any messages sent to a on to c . Such a forwarder is necessary, in general, because a may have subambients, which would run at different locations and which would send their requests of movement to a . (With appropriate optimisations, forwarders can be further simplified, or even altogether removed; see the discussion in Section 9.)

The other reduction rule of SA, the R-MSG rule, is an interaction between two processes local to the same ambient. In PAN, this is simulated by essentially the same rule: no messages need to be exchanged between locations.

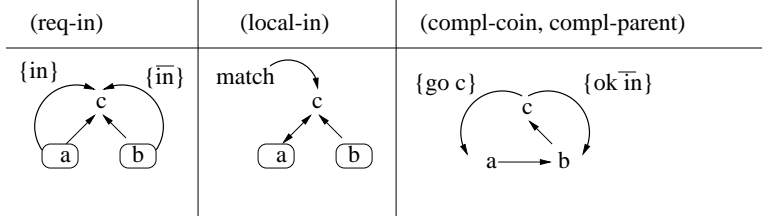


Figure 1: The simulation of the reduction R-IN in PAN

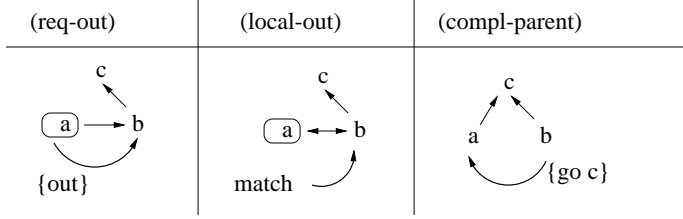


Figure 2: The simulation of the reduction R-OUT in PAN

Using R-OPEN, rather than R-IN or R-OUT, for the physical movements may appear counterintuitive. One should however bear in mind that, in an ambient-like formalism, entering and exiting ambients is not very useful without opening some ambients. The processes local to an ambient a can interact with the processes local to ambient b only if one of the ambients moves inside the other *and* is opened. For instance, suppose we want to model a traveller that starts on a base site, goes to distant server, interacts with its services, and then comes back reporting a result. For simplicity, we suppose that interaction just consists in loading a value v that is also the final result. The distant server is defined thus:

$$SERVER \stackrel{\text{def}}{=} s[\overline{\text{in}} s. \text{open } go. \overline{\text{out}} s. P \mid \langle v \rangle]$$

The base site and the traveller T are:

$$\begin{aligned} BASE &\stackrel{\text{def}}{=} n[T \mid \overline{\text{out}} n. \overline{\text{in}} n. \text{open } \text{return}. (x) R] \\ T &\stackrel{\text{def}}{=} go[\text{out } n. \text{in } s. \overline{\text{open}} go. (x) \text{return}[\text{out } s. \text{in } n. \overline{\text{open}} \text{return}. \langle x \rangle]] \end{aligned}$$

We have:

$$SERVER \mid BASE \implies s[P] \mid n[R\{v/x\}] \quad (2)$$

In the PAN execution, the initial configuration is (omitting some local processes of s and n)

$$h: s[\dots]_{\text{root}} \parallel k: n[T \mid \dots]_{\text{root}}$$

and the first step is to spawn off a new location hosting the traveller, obtaining:

$$h: s[\dots]_{\text{root}} \parallel k: n[\dots]_{\text{root}} \parallel \ell: go[\dots]_k$$

In the reductions in (2), two R-OPEN are executed. When this is modeled in PAN, in the first R-OPEN the (derivative of the) traveller physically moves into the distant server so to load v ; in the second, the (derivative of the) traveller comes back to deliver its result to R .

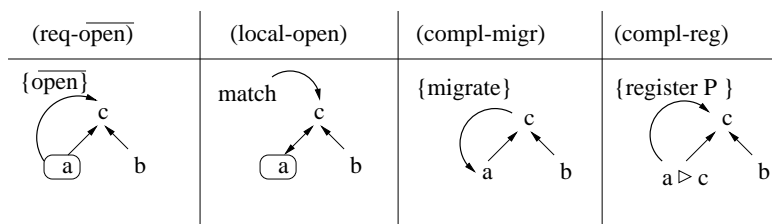


Figure 3: The simulation of the reduction R-OPEN in PAN

4 The abstract machine, formally

Syntax. The syntax of PAN is shown in Table 2. A term of PAN, a *net*, is the parallel composition of *agents* and *messages*, with some names possibly restricted. An agent can be a *located ambient* or a *forwarder*. Located ambients are the basic unit of PAN, and represent ambients of SA with their local processes. A located ambient becomes a forwarder when opened. Messages are of two forms: *requests* and *completions*. The former are the messages that an ambient sends to its parent to request a movement operation; the latter are the messages that the parent sends back to the children to complete the operation. An **open** needs however a third completion message (a **register** message) in which the processes local to a child are migrated to its parent. The syntax of the processes inside located ambients is similar to that of processes in SA. The only additions are: the prefix **wait**. P , which appears in an ambient when this has sent a request to its parent but has not received an answer yet; and the requests, which represent messages received from the children and not yet served. We use A to range over nets.

Semantics. The reduction relation of PAN, \mapsto , from nets to nets, is defined by the rules below. The axioms are divided into five groups, for, respectively: reductions local to an ambient; creation of new ambients and new restrictions; forwarding; consumption of request messages; emission of request messages; consumption of completion messages. There is finally a group of inference rules. The rules for local reductions, and the associated inference rule PAR-PROC, have a special format. We write

$$P \xrightarrow[h:n]{k} Q \gg \widetilde{Msg}$$

to mean a process P , local to an ambient n that is located at h , and whose parent is located at k , becomes Q and, as a side effect, the messages in \widetilde{Msg} are generated. We use \widetilde{Msg} to indicate a possibly empty parallel composition of messages. For instance, if $P \xrightarrow[h:n]{k} Q \gg \widetilde{Msg}$, then, using PROC-AGENT and PAR-AGENT, we have, for any net A :

$$A \parallel h:n[P]_k \mapsto A \parallel h:n[Q]_k \parallel \widetilde{Msg}$$

When n or h or k are unimportant, we replace them with $-$, as in $P \xrightarrow[-:n]{k} Q \gg \widetilde{Msg}$. The inference rule STRUCT-CONG make use of the structural congruence relation \equiv , whose definition is similar to that for SA:

Definition 4.1 Structural congruence is the smallest congruence \equiv such that:

1. parallel composition and $\mathbf{0}$ form an abelian monoid (on processes as well as on agents);
2. $\nu p(A_1) \parallel A_2 \equiv \nu p(A_1 \parallel A_2)$ if p not free in A_2 , and
 $\nu n(P_1) \mid P_2 \equiv \nu p(P_1 \mid P_2)$ if n not free in P_2 ;
3. $\nu p \mathbf{0} \equiv \mathbf{0}$, and $\nu p \nu q P \equiv \nu q \nu p P$;
4. $\text{rec } X. P \equiv P \{ \text{rec } X. P / X \}$.

	$a, b, \dots \in \text{Names}$	$h, k, \dots \in \text{Locations}$		$p, q, \dots \in \text{Names} \cup \text{Locations}$
A	$:=$	$\mathbf{0}$	$ $	$Agent$
		$Agent$	$ $	$h \triangleright k$
		$h\{MsgBody\}$	$ $	$h: n[P]_k$
		$A_1 \parallel A_2$	$ $	$Request$
		$(\nu p)A$	$ $	$Completion$
		(empty)		(forwarder)
		(agent)		(located ambient)
		(message)		(request)
		(composition)		(completion)
		(restriction)		(completion)
		$Request$		$Message\ body$
		$Completion$		(request)
		$in\ n, h$		$(\text{the agent at } h \text{ wants to enter } n)$
		$\overline{in}\ n, h$		$(\text{the agent at } h, \text{ named } n, \text{ accepts someone in})$
		$out\ n, h$		$(\text{the agent at } h \text{ wants to go out of } n)$
		$\overline{open}\ n, h$		$(\text{the agent at } h, \text{ named } n, \text{ accepts to be opened})$
		$go\ h$		$(\text{change the parent to be } h)$
		$0\overline{Kin}$		$(\text{request } \overline{in} \text{ accepted})$
		$migrate$		$(\text{request } \overline{open} \text{ accepted})$
		$register\ P$		$(\text{add } P \text{ to the local processes})$

Process-related syntax:

P	$:=$	$\mathbf{0}$	$ $	M	$:=$	x
		$P_1 \mid P_2$	$ $	n		n
		$(\nu n)P$	$ $	$in\ M$		$in\ M$
		$M.P$	$ $	$\overline{in}\ M$		$\overline{in}\ M$
		$M[P]$	$ $	$out\ M$		$out\ M$
		$\langle M \rangle$	$ $	$\overline{out}\ M$		$\overline{out}\ M$
		$(x)P$	$ $	$open\ M$		$open\ M$
		X	$ $	$\overline{open}\ M$		$\overline{open}\ M$
		$rec\ X.P$				
		$wait.P$				
		$\{Request\}$				

Table 2: The syntax of PAN

The side condition of rule PAR-PROC ensures that all subambients of an ambient are activated as soon as possible, before any local reduction takes place (here we exploit the fact that recursions are guarded, otherwise there could be an infinite number of ambients to create). We write \Longrightarrow for the reflexive and transitive closure of \mapsto .

Local reductions

$$\begin{aligned} \langle M \rangle \mid (x). P &\xrightarrow[-;:-]{\bar{-}} P\{M/x\} \gg \mathbf{0} && \text{[LOCAL-COM]} \\ \{\text{in } n, h\} \mid \{\overline{\text{in}} \ n, k\} &\xrightarrow[-;:-]{\bar{-}} \mathbf{0} \gg h\{\text{go } k\} \mid k\{\mathbf{0K}\overline{\text{in}}\} && \text{[LOCAL-IN]} \\ \{\text{out } n, h\} \mid \overline{\text{out}} \ n. P &\xrightarrow[-:n]{k} P \gg h\{\text{go } k\} && \text{[LOCAL-OUT]} \\ \text{open } n. P \mid \{\overline{\text{open}} \ n, h\} &\xrightarrow[-;:-]{\bar{-}} \text{wait}. P \gg h\{\text{migrate}\} && \text{[LOCAL-OPEN]} \end{aligned}$$

Creation

$$\begin{aligned} h: n[m[P] \mid Q]_{h'} &\mapsto h: n[Q]_{h'} \parallel \nu k(k: m[P]_h) && \text{[NEW-LOCAMB]} \\ h: n[\nu m P]_k &\mapsto \nu m(h: n[P]_k) && \text{[NEW-RES]} \end{aligned}$$

Forwarder

$$h \triangleright k \parallel h\{\text{MsgBody}\} \mapsto h \triangleright k \parallel k\{\text{MsgBody}\} \quad \text{[FW-MSG]}$$

Consumption of request messages

$$h: n[P]_{h'} \parallel h\{\text{Request}\} \mapsto h: n[P \mid \{\text{Request}\}]_{h'} \quad \text{[CONSUME-REQ]}$$

Emission of request messages (should be $h \neq \text{root}$)

$$\begin{aligned} \text{in } m. P &\xrightarrow[h:-]{k} \text{wait}. P \gg k\{\text{in } m, h\} && \text{[REQ-IN]} \\ \overline{\text{in}} \ n. P &\xrightarrow[h:n]{k} \text{wait}. P \gg k\{\overline{\text{in}} \ n, h\} && \text{[REQ-COIN]} \\ \text{out } m. P &\xrightarrow[h:-]{k} \text{wait}. P \gg k\{\text{out } m, h\} && \text{[REQ-OUT]} \\ \overline{\text{open}} \ n. P &\xrightarrow[h:n]{k} \text{wait}. P \gg k\{\overline{\text{open}} \ n, h\} && \text{[REQ-COOPEN]} \end{aligned}$$

Consumption of completion messages

$$\begin{aligned} h: n[P \mid \text{wait}. Q]_k \parallel h\{\text{go } h'\} &\mapsto h: n[P \mid Q]_{h'} && \text{[COMPL-PARENT]} \\ h: n[P \mid \text{wait}. Q]_k \parallel h\{\mathbf{0K}\overline{\text{in}}\} &\mapsto h: n[P \mid Q]_k && \text{[COMPL-COIN]} \\ h: n[P \mid \text{wait}. Q]_k \parallel h\{\text{migrate}\} &\mapsto h \triangleright k \parallel k\{\text{register } P \mid Q\} && \text{[COMPL-MIGR]} \\ h: n[P \mid \text{wait}. Q]_k \parallel h\{\text{register } R\} &\mapsto h: n[P \mid Q \mid R]_k && \text{[COMPL-REG]} \end{aligned}$$

Inference rules

$$\begin{aligned} \frac{P \xrightarrow[h:n]{k} P' \gg \widetilde{\text{Msg}} \quad Q \text{ does not have unguarded ambients}}{P \mid Q \xrightarrow[h:n]{k} P' \mid Q \gg \widetilde{\text{Msg}}} &&& \text{[PAR-PROC]} \\ \frac{P \xrightarrow[h:n]{k} P' \gg \widetilde{\text{Msg}}}{h: n[P]_k \mapsto h: n[P']_k \parallel \widetilde{\text{Msg}}} &&& \text{[PROC-AGENT]} \\ \frac{A \mapsto A'}{A \parallel B \mapsto A' \parallel B} &&& \text{[PAR-AGENT]} \end{aligned}$$

$$\frac{A \mapsto A'}{\nu p A \mapsto \nu p A'} \quad [\text{RES-AGENT}]$$

$$\frac{A \equiv A' \quad A' \mapsto A'' \quad A'' \equiv A'''}{A \mapsto A'''} \quad [\text{STRUCT-CONG}]$$

5 Correctness of the abstract machine

The *name* of a located ambient $h: n[P]_k$ is n , the *home location* is h , the *parent location* (of h) is k . If a `wait` prefix occurs in P , the ambient is in *wait state*. The *home location* of a forwarder $h \triangleright k$ is h , and the *parent location* is k . The *target location* of a message $h\{MsgBody\}$ is h . The *source location* of a request $\{M, k\}$ (that can be part of a message or of a local process) is k . There is no source location in a completion message. A special location is `root`, indicating the outermost (logical) location. By convention, the ambient located at `root` has name `rootname` and parent `rootparent`. We can extract a relation among locations from a net A , whereby two locations are related if they are the home and the parent location of the same agent. We call this the *dependency relation on the locations of A* .

Since PAN separates between the logical and physical distribution of ambients, we need to make sure that the two are consistent. For instance, the graph of the dependencies among locations in the physical distribution, which represents the logical structure, should be a tree. We also need conditions that ensure that the `wait` state is used as described informally in previous sections. We therefore introduce the notion of well-formedness.

Definition 5.1 (well-formedness) *A net A is well-formed if `root` and `rootparent` are the only free locations of A , and the following conditions are true for A (by alpha-conversion, we assume that all restricted locations and names are different from each other and from free locations and names):*

1. *The dependency relation on the locations of A form a a tree.*
2. *There is a special located ambient, called the root, located at `root`, with name `rootname`, and with parent `rootparent`. The name `rootname` and the location `rootparent` cannot appear anywhere else. The location `root` cannot be*
 - *the source of a request message,*
 - *the target of a completion message that is not a register.*
3. *For every location h , there is at most one agent located at h .*
4. *For every forwarder $h \triangleright k$, the home location h does not appear as source location of a request message or a request process.*
5. *A located ambient with home location h is in `wait` state iff h appears exactly once in at most one of the following ways:*
 - (a) *as the source of a request (in a message or in a local process);*
 - (b) *as the target of a completion message;*
 - (c) *as the parent of the target of a migrate message.*
6. *A located ambient in `wait` state does not contain unguarded ambients.*

Lemma 5.2 *Suppose A is well-formed and $A \mapsto A'$. Then also A' is well-formed.*

Invariant under reductions for condition (6) of Definition 5.1 is ensured by the side condition in rule `PAR-PROC`. In the remainder, all nets we write are supposed to be well-formed.

We write $A \Downarrow_n$ if A is *observable at n* ; this means, intuitively, that A contains an agent n that accepts interactions with the external environment. Formally:

Definition 5.3 (observability) *We write $A \Downarrow_n$ if $A \equiv \nu \tilde{p}(h: n[\mu. Q_1 \mid Q_2]_{\text{root}} \parallel A')$ where $\mu \in \{\overline{\text{in}}\ n, \overline{\text{open}}\ n\}$ and $n \notin \tilde{p}$. We write $A \Downarrow_n$ if $A \Longrightarrow \Downarrow_n$.*

Using the reduction relation \Longrightarrow and the predicates $A \Downarrow_n$ we can define reduction-based behavioural equivalences, such as barbed bisimulation or barbed expansion. The latter, written \preceq , is an asymmetric variant of barbed bisimulation; $A' \succeq A$ means that A' and A are barbed bisimilar and, in addition, A' has at least as many reduction steps as A .

The lemma below shows that forwarders behave as substitutions.

Lemma 5.4 $\nu h (h \triangleright k \mid A) \succeq A\{k/h\}$.

A reduction $A \longmapsto A'$ is *administrative* if its derivation proof does not use the axioms of local reductions. A key lemma for the correctness proof is:

Lemma 5.5 *If $A \longmapsto A'$ is administrative, then $A \succeq A'$.*

The proof proceeds by a case analysis on the reduction axiom used to infer $A \longmapsto A'$. The hardest cases are those of the rules `COMPL-REG` and `COMPL-MIGR`, and of the rules for consuming messages and for creation. For this, we exploit the fact that once an ambient request to be opened has been accepted by its parent, both ambients are in a `wait` state. We also exploit the fact that, by Definition 5.1(6), a located ambient that can create a new located ambient cannot be in a `wait` state and, therefore, cannot receive migration messages (which would make it into a forwarder). Some proofs are carried out using the technique of “barbed expansion up to expansion” [13].

Definition 5.6 *A net A is in normal form if it cannot perform an administrative reduction.*

Lemma 5.7 *For every A there is a normal form A^* such that*

1. $A \succeq A^*$, and $A \Longrightarrow A^*$ by means of a sequence of administrative reductions.

Moreover, the normal form A^ is unique up to \equiv , in the sense that if there is another process A' which satisfies (1) and (2) as A^* , then $A' \equiv A^*$.*

Let $\llbracket \cdot \rrbracket$ be the translation of terms of SA into terms of PAN, so defined:

$$\llbracket P \rrbracket \stackrel{\text{def}}{=} \text{root:rootname}[P]_{\text{rootparent}}$$

We write $\llbracket P \rrbracket^*$ for the normal form of $\llbracket P \rrbracket$.

Lemma 5.8 *Let P be a well-typed SA process.*

1. if $P \longrightarrow P'$ then $\llbracket P \rrbracket^* \longmapsto \succeq \llbracket P' \rrbracket^*$.
2. if $\llbracket P \rrbracket^* \longmapsto A$ then there is P' such that $P \longrightarrow P'$ and $A \succeq \llbracket P' \rrbracket^*$.

Theorem 5.9 *Let P be a well-typed SA process. Let P be a well-typed SA process..g*

1. if $P \Longrightarrow P'$ then $\llbracket P \rrbracket^* \Longrightarrow \succeq \llbracket P' \rrbracket^*$.
2. if $\llbracket P \rrbracket^* \Longrightarrow A$ then there is P' such that $P \Longrightarrow P'$ and $A \succeq \llbracket P' \rrbracket^*$.

Observability in SA is defined similarly as for PAN: $P \Downarrow_n$ if $P \Longrightarrow P'$, for some P' such that $P' \equiv \nu \tilde{n} (n[\mu. Q_1 \mid Q_2] \mid Q_3)$ where $\mu \in \{\overline{\text{in}}n, \overline{\text{open}}n\}$ and $n \notin \tilde{n}$.

Corollary 5.10 (adequacy) *Let $P \in SA$. It holds that $P \Downarrow_n$ iff $\llbracket P \rrbracket \Downarrow_n$, for all n .*

6 Immobile ambients

The other important type of ambients in SA are the *immobile* ambients. (A typed SA program may therefore contain both single-threaded *and* immobile ambients.) These are ambients that: (i) cannot jump into or out of other ambients; (ii) cannot be opened. Thus the only capabilities that an immobile ambient can exercise are $\overline{\text{in}}n$, $\overline{\text{out}}n$, and $\text{open}n$; several of them can be ready for execution at the same time.

The same rules for the abstract machine in Section 4 could be adopted for immobile ambients. This has however the following problem. Consider the process

$$P \stackrel{\text{def}}{=} n[\text{rec } X. (\overline{\text{in}} n \mid \nu m (\text{open } m. X \mid m[\overline{\text{open}} m]))]$$

(Using replication, the behaviour of P can be expressed as $n[!\overline{\text{in}} n]$.) With the rules of Section 4, ambient n could flood its parent with $\overline{\text{in}}$ requests. To avoid the problem, we modify PAR-PROC:

$$\frac{\begin{array}{l} P \xrightarrow[h:n]{k} P' \gg \widetilde{Msg} \\ n \text{ is an immobile ambient} \\ Q \text{ does not have unguarder ambients} \\ Q \text{ or } P' \text{ do not contain any wait} \end{array}}{P \mid Q \xrightarrow[h:n]{k} P' \mid Q \gg \widetilde{Msg}} \quad [\text{IMM-PAR-PROC}]$$

We then have to modify also LOCAL-OPEN and PAR-PROC, so that an immobile ambient does not go into a `wait` state while opening a child ambient:

$$\frac{n \text{ is an immobile ambient}}{\text{open } m. P \mid \{\overline{\text{open}} m, h\} \xrightarrow[-:n]{-} P \gg h\{\text{migrate}\}} \quad [\text{IMM-LOCAL-OPEN}]$$

$$\frac{n \text{ is an immobile ambient}}{h:n[P]_k \parallel h\{\text{register } R\} \mapsto h:n[P \mid R]_k} \quad [\text{IMM-COMPL-REG}]$$

The original rules LOCAL-OPEN, PAR-PROC, and COMPL-REG are now used only for ST ambients, therefore the corresponding side conditions is added.

With the new rules, the following property holds (for both ST and immobile ambients): an agent can send only one request message at a time to its parent. An immobile ambient can exercise several capabilities at the same time. Sending one request at a time to the parent is correct because the only capability that may produce a request from an immobile ambient named n to its parent is $\overline{\text{in}} n$ (the protocol for $\overline{\text{in}}$ can however be executed in parallel with several protocols for $\overline{\text{out}}$ and `open` operations).

With the new rules, the addition of immobile ambients requires few modifications to the correctness proof of Section 5; in particular, Theorem 5.9 and Corollary 5.10 continue to hold.

7 Comparisons and remarks

Cardelli [2, 3] has produced the first implementation, called *Ambit*, of an ambient-like language; it is a single-machine implementation of the untyped Ambient calculus, and is written in Java. The algorithms are based on locks: all the ambients involved in a movement (three ambients for an `in` or `out` movement, two for an `open`) have to be locked for the movement to take place. More recently, Fournet, Lévy and Schmitt [9] have presented a distributed implementation of the untyped Ambient calculus, as a translation of the calculus into Joacaml [10] (a programming language based on the distributed Join Calculus [8]). Our abstract machine is quite different from the above mentioned implementations mainly because:

- (i) We are implementing a variant of the Ambient calculus (the Safe Ambients) that has coactions and types for single-threadness and immobility.
- (ii) We separate the logical and physical distribution of an ambient system.

The combination of (i) and (ii) allows us considerable simplifications, both in the abstract machine and in its correctness proof. We are not aware of correctness proofs for *Ambit*. The correctness proof for the Join implementation is very ingenious and makes use of sophisticated techniques, such as coupled simulation and decreasing diagram techniques. Below, we focus on the differences with the Join implementation, which is a distributed implementation, and which we will refer to as AtJ (Ambients to Join).

- In AtJ `open` is by far the most complex operation, because the underlying Jocaml language does not have primitives with a similar effect. In AtJ, every ambient has a manager that collects the requests of operations from the subambients and from the local processes. If the ambient is opened, its manager becomes a forwarder of messages towards the parent ambient. The processes local to the opened ambient are not moved.

As a consequence, in AtJ the processes local to an ambient can be distributed on several locations. Therefore, also the implementation of the communication rule R-MSG may require exchange of messages among sites, which does not occur in PAN, where forwarders are always empty.

- In AtJ, forwarders are also introduced with `in` and `out` operations, to cope with possible asynchronous messages still travelling after the move is finished. These forwarders are not needed in PAN.
- In PAN, the presence of coactions dispenses us from having backward pointers from an ambient to its children. In the example of Figure 1, without `in`, the ambient a would not know the location of c and therefore could not communicate this location to b . Backward pointers, as in AtJ, make bookkeeping and correctness proof more complex.

In PAN, the absence of backward pointers and the presence of coactions make the implementation of forms of dynamic linking straightforward: new machines hosting ambients can be connected to existing machine running an ambient system; it suffices that the new machines know the location of one of the running ambients; no modifications or notifications is needed to the running ambients themselves.

- In PAN, since any moving ambient (an ambient that tries to enter or exit another ambient, or that can be opened) is single-threaded, each ambient requests at most one operation at a time to its parent. By contrast, in AtJ an ambient can send an unbounded number of requests to the parent (an example is $n[!in\ m_1 \mid !out\ m_2]$).

Moreover, due to this property, in PAN no ambient needs a log of pending requests received from a given children or sent to the parent. Without the property, both forms of log are needed, as it happens in AtJ. To see why, consider two ambients a and b , where b is the parent of a . If moving ambients can request several operations concurrently, b must of course keep a log of the pending requests from a . A copy of the same log must however be kept by a , because messages exchanged among ambients are asynchronous and therefore the following situation could arise. Suppose a requests two operations, say `in` n and `in` m . The request for `in` n could reach b first. The request for `in` m could reach b only when the movement for `in` n has been completed (indeed, a might have completed other movements). The request `in` m must now be resent to the new parent of a , but b does not possess this information. This task must therefore be accomplished by a , which, for this, must have stored `in` m in its log of pending requests to the parent.

The example also shows that, aside from message retransmission in forwarders, some requests may have to be retransmitted several times, to different parents (in the example, `in` m); in PAN every request is sent at most once.

- In PAN, any movement for a given ambient is requested to the parent, which (assuming this is not a forwarder) makes decisions and gives authorisations; the grandparent is never contacted. This omogeneity property breaks in presence of backward pointers from an ambient to its children. For instance, the simulation of the `out` reduction of Figure 2 would then need also the involvement of the grandparent c .
- In AtJ, the domain of physical distribution is a tree. The `in` and `out` operations produce physical movements in which an ambient, and all its tree of subambients, must move. To achieve this, the tree of ambients is first “frozen” so that all the activities in the ambients of the tree stop while the movement takes place. In PAN, where the domain of physical distribution is flat, `in` and `out` only give logical movement; no freezing of ambients is required. On the other hand, in PAN, but not in AtJ, `open` gives physical movement.
- PAN is an abstract machine, and is therefore independent of a specific target language.

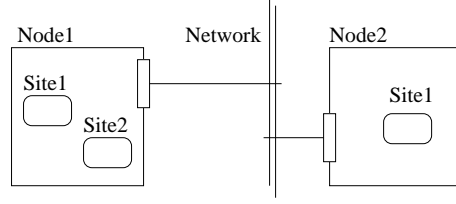


Figure 4: The three layers: agents, nodes and network

8 Implementation architecture

Our implementation, written in Java, closely follows the definition of the abstract machine. Perhaps the main difference is that the implementation allows clustering of agents on the same IP node (i.e. a physical machine). Therefore the implementation is made of three layers: agents, nodes and the network (Figure 4). The address k of an agent is composed of the *IP-name* of the node on which it resides, plus a suffix, which is different for each agent in that node. Each agent is executed by an independent Java thread; the processes local to an ambient are scheduled using a round-robin policy. Each agent knows its name, its address, its parent's address, and keeps a link to its node.

From a physical point of view, the messages exchanged between agents are of two kinds: local, when both agents reside on the same node, and remote, when two distinct nodes are involved. In each node a special Java *RMI object*, with its own thread of execution, takes care of inter-node communications. For this, nodes act alternatively as clients (requiring that a message is sent to another computer) and as servers (receiving a message and pushing it into a local mail-box). The node layer is implemented using Java RMI and *serialization*, and the network layer simply provides IP-name registry for RMI communications to take place (using Java *RMIregistry*).

An agent acts as an interpreter for the ambient expressions that constitute its local processes. When the agent wants to create a subambient, it sends a special message to its node, which will spawn a new agent hosting the subambient code. We also allow *remote creation* of new agents: an agent may send a message to a node different from its own, to demand the creation of a subambients. This corresponds to the addition of a primitive `create $n[P]$ at h` , where h is the IP-name of a node, to the abstract machine. When the execution of an ambient expression begins on a given node, the first action is the local creation of a *root* agent. An agent resides on the same node until it is opened; then, its processes are serialised and sent via RMI to the parent agent.

The implementation also allows dynamic linking of ambients, as hinted at in Section 7.

9 Further developments

In the abstract machine presented, a message may have to go through a chain of forwarders before getting to destination. A (partial) solution to this problem is a modification of the rules that guarantees the following property: every agent sends a message to a given forwarder at most once. The modification consists in adding the source field to the completion messages $h\{\overline{OKin}\}$, which thus becomes $h\{\overline{OKin}, k\}$, where k is the ambient that is authorising the move. Thus the rules LOCAL-IN and COMPL-COIN become

$$\{\text{in } n, h\} \mid \{\overline{\text{in}} n, k\} \xrightarrow[h';-]{-} \mathbf{0} \gg h\{\text{go } k\} \parallel k\{\overline{OKin}, h'\} \quad [\text{LOCAL-IN2}]$$

$$h: n[P \mid \text{wait}. Q]_k \parallel h\{\overline{OKin}, h'\} \mapsto h: n[P \mid Q]_{h'} \quad [\text{COMPL-COIN2}]$$

The reason why these rules may be useful is that the parent of an ambient that has sent a $\overline{\text{in}}$ request may have become a forwarder; thus the real parent is another ambient further up in the hierarchy. With the new rules, the parent of the ambient that has sent the $\overline{\text{in}}$ request is updated and hence this ambient will not go through the forwarder afterwards. With the other capabilities that may originate a request from an ambient to its parent (`open`, `out`, `in`), the issue does not arise, because either the requesting ambient is dissolved (`open`), or its parent is anyway modified (`out`, `in`).

Even with the rules above, however, the forwarder introduced in an `open` operation is permanent. We plan to study the problem of the garbage-collection of forwarders. We also plan to experiment the addition of backwards pointers, from an ambient to its children; this should avoid the introduction of forwarders in an `open`, but may complicate other parts of the abstract machine.

In the abstract machine, `open` is the only operation that gives movement of terms. Although at present we do not see the need of enhancing this, the modifications for allowing movement of terms also with `in` and `out` would be simple. The main price is the introduction of additional forwarders, as we have now in the `open` case.

We hope that work presented will be helpful for the design of efficient implementations of programming languages based on ambients. We intend to pursue this direction, taking advantage also of the experience of π -calculus-based programming languages such as Pict and Join.

Acknowledgements. We have benefitted from comments by Jean-Jacques Lévy and Alan Schmitt.

References

- [1] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. 28th POPL*. ACM Press, 2001.
- [2] L. Cardelli. Ambient. <http://www.luca.demon.co.uk/Ambit/Ambit.html> 1997.
- [3] L. Cardelli. Mobile ambient synchronisation. Technical Report 1997-013, Digital SRC, 1997.
- [4] L. Cardelli and A.D. Gordon. Mobile ambients. *Proc. FoSSaCS '98*, LNCS 1378, pages 140–155. Springer Verlag, 1998.
- [5] L. Cardelli and A.D. Gordon. Equational properties of mobile ambients. *Proc. FoSSaCS'99*, LNCS 1578, pages 212–226. Springer Verlag, 1999.
- [6] L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proc. 26th POPL*, pages 79–92. ACM Press, 1999.
- [7] L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. 27th POPL*. ACM Press, 2000.
- [8] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile processes. *Proc. CONCUR '96*, LNCS 1119, pages 406–421. Springer Verlag, 1996.
- [9] C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous distributed implementation for mobile ambients. *IFIP TCS2000*, LNCS 1872, pages 348–364. Springer Verlag, 2000.
- [10] F. Le Fessant. The Jocaml system prototype. <http://join.inria.fr/jocaml>. 1998.
- [11] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proc. 27th POPL*. ACM Press, 2000.
- [12] F. Nielson and H.R. Nielson. Shape analysis for mobile ambients. In *Proc. 27th POPL*, pages 142–154, N.Y., January 19–21 2000. ACM Press.
- [13] D. Sangiorgi and R. Milner. The problem of “Weak Bisimulation up to”. *Proc. CONCUR '92*, LNCS 630, pages 32–46. Springer Verlag, 1992.