A Distributed Abstract Machine for Boxed Ambient Calculi

Andrew Phillips, Nobuko Yoshida, and Susan Eisenbach

Department of Computing, 180 Queen's Gate,
Imperial College London, SW7 2AZ, UK
{anp,yoshida,sue}@doc.ic.ac.uk

Abstract. Boxed ambient calculi have been used to model and reason about a wide variety of problems in mobile computing. Recently, several new variants of Boxed Ambients have been proposed, which seek to improve on the original calculus. In spite of these theoretical advances, there has been little research on how such calculi can be correctly implemented in a distributed environment. This paper bridges a gap between theory and implementation by defining a distributed abstract machine for a variant of Boxed Ambients with channels. The abstract machine uses a *list semantics*, which is close to an implementation language, and a *blocking semantics*, which leads to an efficient implementation. The machine is proved sound and complete with respect to the underlying calculus. A prototype implementation is also described, together with an application for tracking the location of migrating ambients. The correctness of the machine ensures that the work done in specifying and analysing mobile applications is not lost during their implementation.

1 Introduction

Boxed ambient calculi have been used to model and reason about a wide variety of problems in mobile computing. The original paper on Boxed Ambients [2] shows how the Ambient calculus can be complemented with finer-grained and more effective mechanisms for ambient interaction. In [3], Boxed Ambients are used to reason about resource access control, and in [9] a sound type system for Boxed Ambients is defined, which provides static guarantees on information flow. Recently, several new variants of Boxed Ambients have been proposed, which seek to improve on the foundations of the original calculus. In particular, [14] introduces Safe Boxed Ambients, which uses co-capabilities to express explicit permissions to access ambients, and [4] introduces the NBA calculus, which seeks to limit communication and migration interferences in Boxed Ambients.

In spite of these theoretical advances, there has been little research on how boxed ambient calculi can be correctly implemented in a distributed environment. In general, such an implementation can be achieved by defining an Application Programming Interface (API), which maps the constructs of the calculus to a chosen programming language. Examples of this approach include [5], which describes a Java API for the Ambient calculus, and [17], which describes a Java

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 155–170, 2004.

API for a variant of Boxed Ambients. The main advantage of the approach is that the API can be smoothly integrated into an existing programming language. The main disadvantage, however, is that the constructs of the chosen language often extend beyond the scope of the calculus model. Therefore, when method calls to the API are combined with arbitrary program code, e.g. code which generates exceptions, the resulting program is unpredictable. As a result, most of the benefits of using a calculus model are lost, since any security or correctness properties that hold for the model may not hold for its implementation.

An alternative approach for implementing boxed ambient calculi uses an interpreter to execute calculus expressions. Although this approach constrains the programmer to use a custom language (which may not be compatible with existing code or libraries), support for language interoperability can be provided in the implementation, allowing the non-distributed aspects of a mobile application to be written in any chosen language, and the mobile and distributed aspects to be written in the calculus. The interaction between these two aspects can be achieved using the communication primitives of the calculus, allowing a clean separation of concerns in the spirit of modern coordination languages such as [1]. More importantly, this approach ensures that any security or correctness properties of the calculus model are preserved during execution, provided the interpreter is implemented correctly. This can be achieved by defining an *abstract machine* to specify how the interpreter should behave. The correctness of the machine can then be verified with respect to the underlying calculus.

This paper presents a distributed abstract machine for a variant of Boxed Ambients with channels, known as the Channel Ambient calculus (CA). [1]To our knowledge, a correct abstract machine for a variant of Boxed Ambients has not yet been implemented in a distributed environment. The remainder of the paper is structured as follows: Section 2 introduces the Channel Ambient calculus, and Section 3 defines the syntax and semantics of the Channel Ambient Machine (CAM), an abstract machine for CA. Section 4 outlines the proof of correctness of CAM with respect to the calculus CA. Due to space limitations the full proofs have been omitted, but can be found in [15]. Section 5 describes a distributed runtime system that has been implemented based on CAM, and Section 6 describes an example application. Finally, Section 7 compares CA and CAM with related calculi and abstract machines.

2 The Channel Ambient Calculus

The Channel Ambient calculus is a variant of the Boxed Ambient calculus in which ambients can interact using named channels. The main constructs of CA are illustrated in the following example:

$$Network \mid client\boxed{C \mid \nu login\,(server{\cdot}request\langle client, login\rangle \mid \overline{\mathtt{in}}\,login.P)}$$
$$\mid server\boxed{S \mid !\overline{\mathtt{out}}\,logout \mid !request^{\uparrow}(c,x).service\boxed{\mathtt{out}\,logout.\mathtt{in}\,c{\cdot}x.Q}}$$

[1] The work presented here forms part of the first author's forthcoming PhD thesis [16]

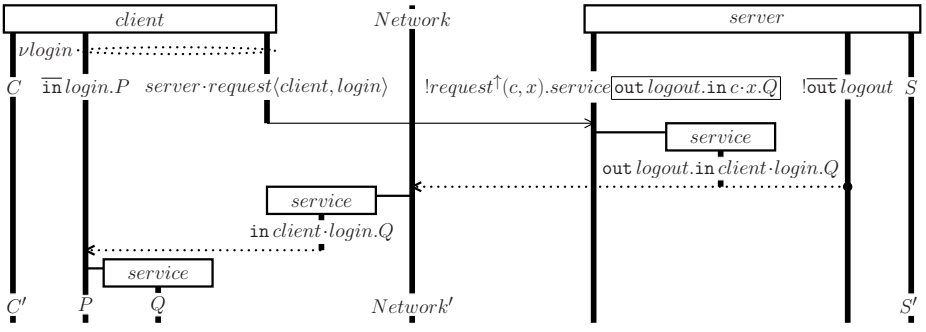

Fig. 1. Execution Scenario

The example uses a polyadic variant of CA to model a *client* machine, which downloads a *service* ambient from a *server* machine over a *Network*. An execution scenario for this system is illustrated in Fig. 1, in which the vertical lines represent parallel processes, the boxes represent ambients, the horizontal arrows represent interaction, and the flow of time proceeds from top to bottom. Initially, the client creates a new *login* channel and sends its name and login channel to the server on the *request* channel. In parallel, it allows an ambient to enter via the login channel. The server receives the request and creates a new *service* ambient, which leaves via the *logout* channel and enters the client via the login channel. Once inside the client, the service ambient executes process Q, and the client executes process P. In parallel, the *Network* evolves to *Network*′, and the processes C and S inside the client and server evolve to C' and S' respectively. The system has a high degree of parallelism. In particular, the network can contain multiple clients and the server can handle multiple requests simultaneously.

The full syntax of CA is defined in terms of processes P, Q, R and actions α:

$P, Q, R ::=$	$\mathbf{0}$	Null	$\alpha ::=$	$a{\cdot}x\langle n\rangle$	Sibling Output
	$P \mid Q$	Parallel		$x^{\uparrow}\langle n\rangle$	Parent Output
	$\nu n\, P$	Restriction		$x(m)$	Internal Input , $x \neq m$
	$a[\,P\,]$	Ambient		$x^{\uparrow}(m)$	External Input , $x \neq m$
	$\alpha.P$	Action		$\mathtt{in}\, a{\cdot}x$	Enter
	$!\alpha.P$	Replication		$\mathtt{out}\, x$	Leave
				$\overline{\mathtt{in}}\, x$	Accept
				$\overline{\mathtt{out}}\, x$	Release

Processes in CA have the same structure as processes in the Ambient calculus [7], except that replicated actions $!\alpha.P$ are used instead of general replication $!P$. The definition of the set of free names $fn(P)$ of a process P is standard, where

restriction $\nu m\, P$ and inputs $x(m).P$ and $x^{\uparrow}(m).P$ act as binders for the name m. Standard notational conventions are used, including assigning the lowest precedence to the parallel composition operator, and writing α as syntactic sugar for $\alpha.\mathbf{0}$. In addition, local output $x\langle n\rangle$ and child output $a/x\langle n\rangle$ are written as syntactic sugar for $\nu b\, b\boxed{x^{\uparrow}\langle n\rangle}$ and $\nu b\, b\boxed{a{\cdot}x\langle n\rangle}$ respectively, where $b \notin \{a, x, n\}$.

The semantics of CA is defined in terms of structural congruence ($\equiv$) and reduction ($\longrightarrow$). The definition of structural congruence is mostly standard:

$$P \mid \mathbf{0} \equiv P \quad (1) \qquad n \notin fn(P) \Rightarrow P \mid \nu n\, Q \equiv \nu n\, (P \mid Q) \quad (8)$$

$$P \mid Q \equiv Q \mid P \quad (2) \qquad a \neq n \Rightarrow a\boxed{\nu n\, P} \equiv \nu n\, a\boxed{P} \quad (9)$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad (3) \qquad n \notin fn(P) \Rightarrow \nu m\, P \equiv \nu n\, P_{\{n/m\}} \quad (10)$$

$$!\alpha.P \equiv \alpha.(P \mid !\alpha.P) \quad (4) \qquad n \notin fn(P) \Rightarrow x(m).P \equiv x(n).P_{\{n/m\}} \quad (11)$$

$$\nu n\, \mathbf{0} \equiv \mathbf{0} \quad (5) \qquad n \notin fn(P) \Rightarrow x^{\uparrow}(m).P \equiv x^{\uparrow}(n).P_{\{n/m\}} \quad (12)$$

$$\nu n\, n\boxed{\mathbf{0}} \equiv \mathbf{0} \quad (6) \qquad P \equiv Q \Rightarrow a\boxed{P} \equiv a\boxed{Q} \quad (13)$$

$$\nu n\, \nu m\, P \equiv \nu m\, \nu n\, P \quad (7) \qquad P \equiv Q \Rightarrow P \mid R \equiv Q \mid R \quad (14)$$

Rule (6) allows empty ambients to be garbage-collected, and (11) and (12) allow bound names to be substituted, where $P_{\{n/m\}}$ substitutes the name n for m in process P. The only non-standard rule is (4), which allows a replicated action $!\alpha.P$ to be expanded to $\alpha.(P \mid !\alpha.P)$. This ensures that only a single copy of $\alpha.P$ can be created at a time, since $!\alpha.P$ is unable to spawn a new copy while inside the prefix α. This is useful from an implementation perspective, and differs from the standard rule for replication, $!\alpha.P \equiv \alpha.P \mid !\alpha.P$, which allows an infinite number of copies of $\alpha.P$ to be created. The definition of reduction is also standard:

$$(Q \equiv P \wedge P \longrightarrow P' \wedge P' \equiv Q') \Rightarrow Q \longrightarrow Q' \quad (15)$$

$$(P \longrightarrow P') \Rightarrow P \mid Q \longrightarrow P' \mid Q \quad (16)$$

$$(P \longrightarrow P') \Rightarrow \nu n\, P \longrightarrow \nu n\, P' \quad (17)$$

$$(P \longrightarrow P') \Rightarrow a\boxed{P} \longrightarrow a\boxed{P'} \quad (18)$$

$$a\boxed{b{\cdot}x\langle n\rangle.P \mid P'} \mid b\boxed{x^{\uparrow}(m).Q \mid Q'} \longrightarrow a\boxed{P \mid P'} \mid b\boxed{Q_{\{n/m\}} \mid Q'} \quad (19)$$

$$a\boxed{x^{\uparrow}\langle n\rangle.P \mid P'} \mid x(m).Q \longrightarrow a\boxed{P \mid P'} \mid Q_{\{n/m\}} \quad (20)$$

$$a\boxed{\mathtt{in}\, b{\cdot}x.P \mid P'} \mid b\boxed{\overline{\mathtt{in}}\, x.Q \mid Q'} \longrightarrow b\boxed{Q \mid Q' \mid a\boxed{P \mid P'}} \quad (21)$$

$$b\boxed{a\boxed{\mathtt{out}\, x.P \mid P'} \mid \overline{\mathtt{out}}\, x.Q \mid Q'} \longrightarrow b\boxed{Q \mid Q'} \mid a\boxed{P \mid P'} \quad (22)$$

Rule (15) allows structurally congruent processes to perform the same reductions. Rules (16) - (18) allow a reduction to take place inside a parallel composition, inside a restriction or inside an ambient. Rules (19) and (20) allow an ambient to send a message to a sibling or to its parent on a given channel. Rules (21) and (22) allow an ambient to enter a sibling or to leave its parent on a given channel.

A brief comparison of CA with related calculi is given in Section 7. One distinguishing feature of CA is that it allows sibling ambients to communicate synchronously. This is often desirable within a single machine, or between machines in a local area network. Even over a wide area network, certain protocols such as TCP/IP provide a useful abstraction for synchronous communication. In cases where asynchronous communication is required, such as the UDP protocol, sibling ambients can communicate via an intermediate *router* ambient.

3 The Channel Ambient Machine

The Channel Ambient Machine is inspired by the Pict abstract machine [19], which is used as a basis for implementing the asynchronous π-calculus. Like the Pict machine, CAM uses a list syntax to represent the parallel composition of processes. It also uses a notion of *blocked processes*, which are a generalisation of the *channel queues* in Pict.

The machine executes a given process P by extending the scope of each unguarded restriction in P to the top level and converting each unguarded parallel composition in P to a list form. The resulting term is of the form $\nu n ... \nu n' A$, where A is a tree of the form $\alpha_1.P_1 :: ... \alpha_N.P_N :: a_1\boxed{A_1} :: ... :: a_M\boxed{A_M}$. The leaves of the tree are actions $\alpha_i.P_i$ and the nodes are ambients $a_i\boxed{A_i}$. Once a process has been converted to this form, the machine attempts to execute an action somewhere in the tree. If the chosen action $\alpha.P$ is able to interact with a corresponding *blocked* co-action then a reduction is performed. If not, the action $\alpha.P$ is *blocked* to $\underline{\alpha.P}$. The machine then non-deterministically schedules a different action to be executed. If all the actions in a given ambient $a\boxed{A}$ are blocked then the ambient itself is blocked to $\underline{a}\boxed{A}$. Execution terminates when all the actions and ambients in the tree are blocked.

Blocking significantly improves the efficiency of the machine by labelling those ambients and actions that are not able to initiate a reduction. This partitions the search space, allowing the machine to ignore the entire contents of a blocked ambient when looking for an unblocked action to schedule. Blocking also simplifies the condition for termination, since the machine only needs to check for the presence of an unblocked action to know whether a reduction is possible.

The machine can be used to execute the example from Section 2. First, it converts the calculus process to the following term, where $login \notin fn(S, C, Network)$:

$$\nu login\ (server\boxed{!\overline{\mathtt{out}}\, logout :: !request^{\uparrow}(c,x).service\boxed{\mathtt{out}\, logout.\mathtt{in}\, c{\cdot}x.Q} :: S}$$
$$:: client\boxed{server{\cdot}request\langle client, login\rangle :: \overline{\mathtt{in}}\, login.P :: C} :: Network)$$

Then, the machine tries to execute one of the actions in the tree. For example, it can try to execute the replicated release $!\overline{\mathtt{out}}\, logout$. If this action is unable to interact with a corresponding blocked co-action it will block to $\underline{!\overline{\mathtt{out}}\, logout}$. Similarly, the replicated input on the *request* channel will block if there is no

corresponding blocked output. The machine can then execute the sibling output $server{\cdot}request\langle client, login\rangle$, which will interact with the blocked input on the server. The model scales smoothly to the case where the server and client processes are executing on two different machines. In this case, the interaction between machines is implemented by socket libraries, which provide an interface to the physical network layer. When a given machine can no longer perform any reductions it goes into a blocked state, waiting for an interrupt from the network announcing the arrival of new messages or ambients.

The full syntax of CAM is defined below, in terms of machine terms V, U, lists A, B, C and blocked lists $\underline{A}, \underline{B}, \underline{C}$:

$$
\begin{array}{rcll}
V, U & ::= & A & \text{List} \\
 & | & \nu n\, V & \text{Restriction} \\
\\
\underline{A}, \underline{B}, \underline{C} & ::= & [] & \text{Empty List} \\
 & | & \underline{a}\boxed{\underline{A}} :: \underline{C} & \text{Blocked Ambients} \\
 & | & \underline{\alpha.P} :: \underline{C} & \text{Blocked Actions}
\end{array}
\qquad
\begin{array}{rcll}
A, B, C & ::= & [] & \text{Empty List} \\
 & | & a\boxed{A} :: C & \text{Ambient} \\
 & | & \underline{a}\boxed{\underline{A}} :: C & \text{Blocked Ambient} \\
 & | & P :: C & \text{Process} \\
 & | & \underline{\alpha.P} :: C & \text{Blocked Action}
\end{array}
$$

A machine term V is a list with a number of restricted names. Each list A can contain processes, blocked actions, ambients or blocked ambients. By definition, blocked ambients can only contain blocked actions or blocked sub-ambients. In addition, $\underline{!\alpha.P}$ is written as syntactic sugar for $\underline{\alpha.(P \mid !\alpha.P)}$ and $\underline{\underset{\cdot\cdot}{a}}$ is used to denote either a blocked ambient $\underline{a}$ or an unblocked ambient a.

The semantics of CAM is defined in Fig. 2 in terms of structural congruence ($\equiv$) and reduction ($\longrightarrow$). The structural congruence rules are used to find an element in a list that matches a specific pattern. Rule (23) allows an element X at the head of a list to be placed at the back of the list, where ($@$) denotes the list append function, and (24) allows the list A inside an ambient to be replaced with a list B that is structurally congruent to A.

The reduction rules of the machine are derived from the reduction rules of the calculus, and use standard definitions for free names $fn(A)$ and substitution $A_{\{n/m\}}$. Each rule is defined over the entire length of a list and there is no rule to allow reduction across the ($::$) operator. Rule (28) ensures that all restricted names are moved to the top-level, by substituting each restricted name with a globally unique name generated by the machine. In practice, such a name can be created using a secret key, together with a time stamp or a suitable counter. Rule (29) is used to label an ambient as blocked, indicating that it does not contain any unblocked actions. Rules (31) - (34) allow CA processes to be converted to list form, and rules (35) - (42) allow an action to interact with a blocked co-action. The latter rules are derived from rules (19) - (22) in CA, where each rule in CA is mapped to two corresponding rules in CAM. Rules (39) - (42) make use of an *unblocking function* $\lfloor A \rfloor$, which unblocks all of the top-level actions in a given list A. The main rule for unblocking is given by $\lfloor \underline{\alpha.P} :: C \rfloor \triangleq \alpha.P :: \lfloor C \rfloor$, and the full definition is given in Section 4. The unblocking function is used to unblock the contents of an ambient when it moves to a new location, allowing the ambient to *re-bind* to its new environment by giving any blocked actions it contains the chance to interact with their new context. Additionally, for each

$$X :: A \;\equiv\; A@X \tag{23}$$

$$A \equiv B \Rightarrow \underline{a}\boxed{A} :: C \;\equiv\; \underline{a}\boxed{B} :: C \tag{24}$$

$$(U \equiv V \wedge V \longrightarrow V') \Rightarrow U \longrightarrow V' \tag{25}$$

$$(V \longrightarrow V') \Rightarrow \nu n\, V \longrightarrow \nu n\, V' \tag{26}$$

$$(A \longrightarrow A') \Rightarrow a\boxed{A} :: C \longrightarrow a\boxed{A'} :: C \tag{27}$$

$$n \notin fn(a\boxed{A'} :: C) \wedge A \longrightarrow \nu m\, A' \Rightarrow a\boxed{A} :: C \longrightarrow \nu n\, (a\boxed{A'_{\{n/m\}}} :: C) \tag{28}$$

$$a\boxed{\underline{A}} :: C \longrightarrow \underline{a}\boxed{\underline{A}} :: C \tag{29}$$

$$!\alpha.P :: C \longrightarrow \alpha.(P \mid !\alpha.P) :: C \tag{30}$$

$$\mathbf{0} :: C \longrightarrow C \tag{31}$$

$$(P \mid Q) :: C \longrightarrow P :: Q :: C \tag{32}$$

$$n \notin fn(P :: C) \Rightarrow (\nu m\, P) :: C \longrightarrow \nu n\, (P_{\{n/m\}} :: C) \tag{33}$$

$$a\boxed{P} :: C \longrightarrow a\boxed{P :: []} :: C \tag{34}$$

$$C \equiv \underline{b}\boxed{\underline{x^{\uparrow}(m).Q} :: B} :: C' \Rightarrow a\boxed{b{\cdot}x\langle n\rangle.P :: A} :: C \longrightarrow a\boxed{P :: A} :: b\boxed{Q_{\{n/m\}} :: B} :: C' \tag{35}$$

$$C \equiv \underline{a}\boxed{\underline{b{\cdot}x\langle n\rangle.P} :: A} :: C' \Rightarrow b\boxed{x^{\uparrow}(m).Q :: B} :: C \longrightarrow b\boxed{Q_{\{n/m\}} :: B} :: a\boxed{P :: A} :: C' \tag{36}$$

$$C \equiv \underline{x(m).Q} :: C' \Rightarrow a\boxed{x^{\uparrow}\langle n\rangle.P :: A} :: C \longrightarrow a\boxed{P :: A} :: Q_{\{n/m\}} :: C' \tag{37}$$

$$C \equiv \underline{a}\boxed{\underline{x^{\uparrow}\langle n\rangle.P} :: A} :: C' \Rightarrow x(m).Q :: C \longrightarrow Q_{\{n/m\}} :: a\boxed{P :: A} :: C' \tag{38}$$

$$C \equiv \underline{b}\boxed{\underline{\overline{\mathtt{in}}\, x.Q} :: B} :: C' \Rightarrow a\boxed{\mathtt{in}\, b{\cdot}x.P :: A} :: C \longrightarrow b\boxed{Q :: a\boxed{P :: \lfloor A \rfloor} :: B} :: C' \tag{39}$$

$$C \equiv \underline{a}\boxed{\underline{\mathtt{in}\, b{\cdot}x.P} :: A} :: C' \Rightarrow b\boxed{\overline{\mathtt{in}}\, x.Q :: B} :: C \longrightarrow b\boxed{Q :: a\boxed{P :: \lfloor A \rfloor} :: B} :: C' \tag{40}$$

$$B \equiv \underline{\overline{\mathtt{out}}\, x.Q} :: B' \Rightarrow b\boxed{a\boxed{\mathtt{out}\, x.P :: A} :: B} :: C \longrightarrow b\boxed{Q :: B'} :: a\boxed{P :: \lfloor A \rfloor} :: C \tag{41}$$

$$B \equiv \underline{a}\boxed{\underline{\mathtt{out}\, x.P} :: A} :: B' \Rightarrow b\boxed{\overline{\mathtt{out}}\, x.Q :: B} :: C \longrightarrow b\boxed{Q :: B'} :: a\boxed{P :: \lfloor A \rfloor} :: C \tag{42}$$

Fig. 2. Machine Semantics

of the rules (35) - (42) there is a corresponding rule to block an action that is unable to interact with a corresponding co-action. The blocking rule for (35) is as follows:

$$C \not\equiv \underline{b}\boxed{\underline{x^{\uparrow}(m).Q} :: B} :: C' \Rightarrow a\boxed{b{\cdot}x\langle n\rangle.P :: A} :: C \longrightarrow a\boxed{\underline{b{\cdot}x\langle n\rangle.P} :: A} :: C \tag{43}$$

The blocking rules for the remaining actions are defined in a similar fashion.

The abstract machine can be used in both local and distributed settings. Local execution of a term A on a device a is modeled as $a\boxed{A}$. Distributed execution of terms $A_1, ..., A_N$ on devices $a_1, ..., a_N$, respectively, is modeled as $network\boxed{a_1\boxed{A_1}::...::a_N\boxed{A_N}}$, where each identifier a_i corresponds to the address of a device in the network. Nested ambients inside a given term A_i can be executed independently, since only top-level processes in A_i can interact with other devices in the network. These interactions between devices are implemented using socket libraries, according to the reduction rules of CAM. For example, in a local area network a device a performs a sibling output to a device b by checking if b contains a corresponding blocked input, according to (35). Similarly, a performs an external input by checking if any of the devices in the network contain a corresponding blocked output, according to (36). In a wide area network, devices send all sibling outputs to an intermediate router device, and therefore only need to check the router for blocked outputs in order to perform an external input. Note also that CA, and ambient calculi in general, allow migrating ambients to interact while in transit between devices. Such interactions can either be prevented using type systems for single-threaded migration [18], or they can be implemented by defining a default *network* device where migrating ambients can execute while waiting to be admitted to a new host.

4 Correctness of the Channel Ambient Machine

The correctness of the Channel Ambient Machine is expressed in terms of three main properties: soundness, completeness and termination. Due to space limitations the full proofs have been omitted, but can be found in [15].

The soundness of the machine relies on a decoding function $[V]$, which maps a given machine term V to a calculus process, and the unblocking function $\lfloor V \rfloor$ described in Section 3. Decoding and unblocking are defined as follows:

$$
\begin{array}{rcl}
[\nu n\, V] & \triangleq & \nu n\, [V] \\
[[]] & \triangleq & \mathbf{0} \\
[\underline{a}\boxed{A}::C] & \triangleq & a\boxed{[A]} \mid [C] \\
[P::C] & \triangleq & P \mid [C] \\
[\underline{\alpha.P}::C] & \triangleq & \alpha.P \mid [C]
\end{array}
\qquad
\begin{array}{rcl}
\lfloor \nu n\, V \rfloor & \triangleq & \nu n\, \lfloor V \rfloor \\
\lfloor [] \rfloor & \triangleq & [] \\
\lfloor \underline{a}\boxed{A}::C \rfloor & \triangleq & \underline{a}\boxed{A}::\lfloor C \rfloor \\
\lfloor P::C \rfloor & \triangleq & P::\lfloor C \rfloor \\
\lfloor \underline{\alpha.P}::C \rfloor & \triangleq & \alpha.P::\lfloor C \rfloor
\end{array}
$$

Proposition 1 states that reduction in CAM is sound with respect to reduction in CA. The proof relies on Lemma 1, which states that machine terms are reduction-closed.

Proposition 1. $\forall V.V \in \mathrm{CAM} \wedge V \longrightarrow^* V' \Rightarrow [V] \longrightarrow^* [V']$

Proof. By Lemma 1 and by induction on reduction in CAM. The decoding function is applied to the left and right hand side of each reduction rule and the result is shown to be a valid reduction in CA. □

Lemma 1. $\forall V.V \in \mathrm{CAM} \wedge V \longrightarrow V' \Rightarrow V' \in \mathrm{CAM}$

Proof. By definition of structural congruence, by definition of unblocking, and by induction on reduction in CAM. □

The completeness of the Channel Ambient Machine relies on a notion of well-formedness to describe invariants on machine terms. The set of well-formed terms $\mathrm{CAM}^{\surd}$ is defined with respect to the set of ill-formed terms $\mathrm{CAM}^{\times}$, such that $V \in \mathrm{CAM}^{\surd} \Rightarrow V \in \mathrm{CAM} \wedge V \notin \mathrm{CAM}^{\times}$. Ill-formed terms $V^{\times}$ are defined as follows:

$$
\begin{array}{lll}
V^{\times} ::= & A^{\times} & \text{Bad List} \\
\mid & \nu n\, V^{\times} & \text{Bad Restriction}
\end{array}
$$

$$
\begin{array}{lll}
A^{\times} ::= & \underline{a}\boxed{A^{\times}} :: C & \text{Bad Ambient} \\
\mid & C & \text{Bad Enter}, C \equiv \underline{a}\boxed{\underline{\mathtt{in}\, b \cdot x.P} :: A} :: C' \wedge C' \equiv \underline{b}\boxed{\underline{\overline{\mathtt{in}}\, x.Q} :: B} :: C'' \\
\mid & C & \text{Bad Leave}, C \equiv \underline{b}\boxed{\underline{a}\boxed{\underline{\mathtt{out}\, x.P} :: A} :: B} :: C' \wedge B \equiv \underline{\overline{\mathtt{out}}\, x.Q} :: B' \\
\mid & C & \text{Bad Sibling}, C \equiv \underline{a}\boxed{\underline{b \cdot x\langle n\rangle.P} :: A} :: C' \wedge C' \equiv \underline{b}\boxed{\underline{x^{\uparrow}(m).Q} :: B} :: C'' \\
\mid & C & \text{Bad Parent}, C \equiv \underline{a}\boxed{\underline{x^{\uparrow}\langle n\rangle.P} :: A} :: C' \wedge C' \equiv \underline{x(m).Q} :: C''
\end{array}
$$

A term is ill-formed if it contains both a blocked action $\underline{\alpha.P}$ and a corresponding blocked co-action. Therefore, a well-formed term cannot contain a blocked action $\underline{\alpha.P}$ that could participate in a reduction if it were unblocked to $\alpha.P$.

Proposition 2 states that reduction in $\mathrm{CAM}^{\surd}$ is complete with respect to reduction in CA. The proof relies on Lemma 2, which states that well-formed machine terms are reduction-closed. The proof of reduction closure relies on Lemma 3, which states that a term V' inside an arbitrary context K cannot become ill-formed as a result of a reduction.

Proposition 2. $\forall V.(V \in \mathrm{CAM}^{\surd} \wedge [V] \longrightarrow^{*} P') \Rightarrow \exists V'.V \longrightarrow^{*} V' \wedge [V'] \equiv P'$

Proof. By Lemma 2 and by induction on reduction in CA. For each reduction rule in CA, any machine term that corresponds to the left hand side of the rule can reduce to a term that corresponds to the right hand side. □

Lemma 2. $\forall V.V \in \mathrm{CAM}^{\surd} \wedge V \longrightarrow V' \Rightarrow V' \in \mathrm{CAM}^{\surd}$

Proof. By Lemma 3 and by definition of $\mathrm{CAM}^{\surd}$. □

Lemma 3. $\forall V.\forall K.V \longrightarrow V' \wedge K(V') \in \mathrm{CAM}^{\times} \Rightarrow K(V) \in \mathrm{CAM}^{\times}$

Proof. By definition of unblocking and by induction on reduction in CAM. □

Proposition 3 states that reduction in CAM$^{\surd}$ terminates if reduction in CA terminates. For a given well-formed machine term V, if the corresponding calculus process $[V]$ is unable to reduce then V will be unable to reduce after a finite number of steps.

Proposition 3. $\forall V.V \in \text{CAM}^{\surd} \wedge [V] \not\longrightarrow \Rightarrow V \not\longrightarrow^*$

Proof. Reductions in CAM can be classified into housekeeping, interaction and blocking reductions. By definition of well-formedness, an action is blocked if it cannot interact with a suitable co-action. Therefore, if no interactions are possible then all the actions in a given machine term will eventually block and execution will terminate after a finite number of housekeeping reductions. □

5 Implementation

The Channel Ambient Machine has been used to implement a runtime system, which executes programs written in the Channel Ambient language. The runtime is implemented in OCaml, and the language is based on polyadic CA with built-in base types from OCaml. The language also uses a type system for channel communication based on the polymorphic type system of Pict [19]. The runtime is invoked by the command `cam.exe sourcefile portnumber`. This starts a new runtime, which executes the contents of `sourcefile.ca` and accepts connections on `portnumber` of the host machine. Before a given source file is executed, it is statically checked by the CA type-checker, which reports any type errors to the user. The current state of the runtime is regularly streamed to `sourcefile.html`, which can be viewed in a web browser and periodically refreshed to display the latest state information. A beta version of the runtime can be downloaded from [15], together with a user guide.

The main functionality of the runtime is illustrated by the following example, which lists the contents of two source files, *server.ca* and *client.ca*, respectively:

```
    let request = request:<site,<void>>
in
    let service = service:ambient in
    let logout = logout:<void> in
    (  !request(c:site,x:<void>);
         applet[out logout; in c.x; Q<>]
    || !-out logout )
```

```
    let server = 192.168.0.2(3145) in
    let client = 192.168.0.3(3145) in
    let request = request:<site,<void>>
in
    new login:<void>
    (  server.request<client,login>
    || -in login; P<> )
```

These files contain source code corresponding to the *server* and *client* ambients of the example described in Section 2. The syntax of the code is similar to the syntax of CA, with minor variations such as using a semi-colon instead of a dot for action prefixes. The code also contains type annotations of the form $n : T$, where n is a variable name and T is a type expression, and value declarations of the form $let\ n\ = V\ in\ P$, where V is a value expression. Site values are of the form $IP(i)$, where IP is an IP address and i is a port number. Channel values are of the form $n : \langle T \rangle$, where n is a name and T is the type of values carried by

the channel, and ambient values are of the form $n : ambient$ where n is a name. The additional type information helps to preserve type safety when remote machines interact over global channels, since two channels are only equal if both their names and types coincide. The code also contains process macros $P\langle\rangle$ and $Q\langle\rangle$, whose definitions are omitted. The example is executed by invoking the commands `cam.exe server 3145` and `cam.exe client 3145` on two different machines with IP addresses 192.168.0.2 and 192.168.0.3 respectively. The behaviour of the program is as described in Section 2: after a certain number of executions steps, the client receives a service ambient from the server.

The runtime also provides support for system calls, which are modeled using a sibling output of the form $system \cdot call\langle parameters\rangle$. For example, the call $system{\cdot}print\langle n\rangle$ prints the value n on the runtime console. For security reasons, system calls can only be executed by the top-level ambient. This ambient can then provide an interface to a given system call by means of forwarder channels. A separate forwarder channel can be defined for each user or group of users, enabling fine-grained security policies to be implemented for each system call. The runtime also allows files to be stored in an ambient in binary form and sent over channels like ordinary values. This feature is used in the call $system \cdot java\langle file\rangle$, which invokes the Java runtime with the specified file. This can be used to program a wide range of applications. For example, it can be used to program an ambient that moves to a remote site, retrieves a Java class file and then moves to a new site to execute the retrieved file. A similar approach can be used to coordinate the execution of multiple Prolog queries on different sites, or coordinate the distribution and retrieval of multiple HTML forms. Section 6 describes an ambient tracker application, which illustrates the kind of application that can be executed by the runtime system.

6 Ambient Tracker Application

The Channel Ambient calculus can be used to model an *ambient tracker* application, which keeps track of the location of registered client ambients as they move between trusted sites $s_0, ..., s_N$ in a network. The application is inspired by previous work on location-independence, studied in the context of the Nomadic π-calculus [21] and the Nomadic Pict programming language [22]. This section describes a decentralised version of the algorithm given in [20]. The algorithm uses multiple *home servers* to track the location of mobile clients, and relies on a locking mechanism to prevent race conditions. The locking mechanism ensures that messages are not forwarded to a client while it is migrating between sites, and that the client does not migrate while messages are in transit.

The application is modeled using the *Site*, *Tracker* and *Client* processes defined in Fig. 3. The *Site* process describes the services provided by each trusted site in the network. An ambient at a trusted site can receive a message m on channel x from a remote ambient via the *child* channel. Likewise, it can forward a message m on channel x to a remote ambient a at a site s via the fwd channel. Visiting ambients can enter and leave a trusted site via the *login* and *logout* channels respectively. An ambient at a trusted site can check whether a given site s is trusted by sending an output on channel s. A *client* ambient can register

$$
\begin{aligned}
&Site(s_i) \triangleq \\
&(!child^{\uparrow}(a,x,m).a/x\langle m\rangle \\
&\mid !fwd(s,a,x,m).s\cdot child\langle a,x,m\rangle \\
&\mid !\overline{\texttt{in}}\, login \mid !\overline{\texttt{out}}\, logout \\
&\mid !s_0() \mid ... \mid !s_N() \\
&\mid !register(client,ack).\nu tracker\, \nu send\, \nu move\, \nu deliver\, \nu lock \\
&\qquad (tracker\, \boxed{Tracker\langle client, send, move, deliver, lock\rangle} \\
&\qquad \mid client/ack\langle tracker, send, move, deliver, lock\rangle \\
&\qquad \mid tracker/lock\langle s_i\rangle))
\end{aligned}
$$

$$
\begin{aligned}
&Tracker(client, send, move, deliver, lock) \triangleq \\
&(!send^{\uparrow}(x,m).lock^{\uparrow}(s).fwd^{\uparrow}\langle s, client, deliver, (s,x,m)\rangle \\
&\mid !move^{\uparrow}(s').s'^{\uparrow}\langle\rangle.lock^{\uparrow}(s).fwd^{\uparrow}\langle s, client, lock, s'\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&Client(home, tracker, deliver, lock) \triangleq \\
&(!lock^{\uparrow}(s).\texttt{out}\, logout.\texttt{in}\, s\cdot login.fwd^{\uparrow}\langle home, tracker, lock, s\rangle.moved\langle s\rangle \\
&\mid !deliver^{\uparrow}(s,x,m).fwd^{\uparrow}\langle home, tracker, lock, s\rangle.x\langle m\rangle)
\end{aligned}
$$

Fig. 3. Tracker Definitions

with a trusted site via the *register* channel, which creates a new *tracker* ambient to keep track of the location of the client. The *Tracker* and *Client* processes describe the services provided by the tracker and client ambients respectively.

Figure 4 describes a scenario in which a client registers with its home site. The client c sends a message to its home site s_0 on the *register* channel, consisting of its name and an acknowledgement channel *ack*. The site creates a new ambient name t_c and new send, move, deliver and lock channels s_c, m_c, d_c, l_c respectively. It then sends these names to the client on channel *ack*, and in parallel creates a new tracker ambient t_c for keeping track of the location of the client. The tracker is initialised with the *Tracker* process and the current location of the client is stored as an output to the tracker on the lock channel l_c. When the client receives the acknowledgement it spawns a new *Client* process in parallel with process P.

A message can be sent to the client by sending a request to its corresponding tracker ambient on the home site s_0. The request is sent to the tracker ambient t_c on the send channel s_c, asking the tracker to send a message n to its client on channel x. The tracker then inputs the current location s_i of its client via the lock channel l_c, thereby acquiring the lock and preventing the client from moving. The tracker then forwards the request to the deliver channel d_c of the

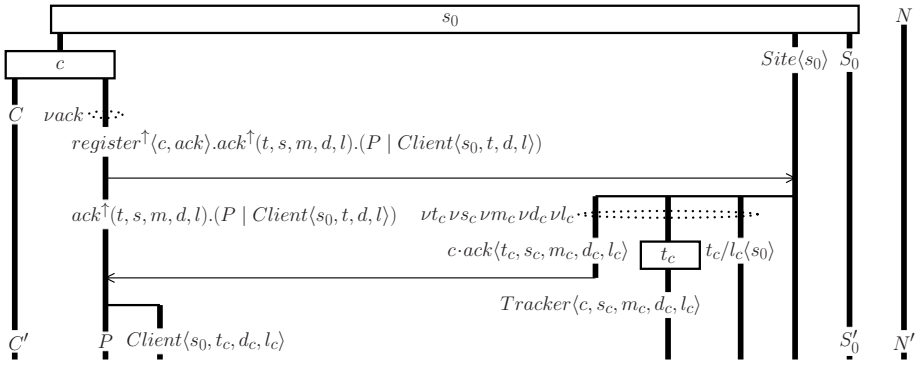

Fig. 4. Tracker Registration

client. When the client receives the request, it forwards its current location to the tracker on the lock channel, thereby releasing the lock, and then locally sends the message n on channel x.

When the client wishes to move to a site s_j, it forwards the name s_j to the tracker ambient on the move channel m_c. The tracker ambient first checks whether this site is trusted by trying to send an output on channel s_j. If the output succeeds, the tracker then inputs the current location of its client on the lock channel, thereby acquiring the lock and preventing subsequent messages from being forwarded to the client. It then forwards the name s_j to the client on the lock channel, giving it permission to move to site s_j. When the client receives permission to move, it leaves on the *logout* channel and enters site s_j on the *login* channel. It then forwards its new location to the tracker ambient on the lock channel, thereby releasing the lock.

The above application has been implemented in the channel ambient programming language, and preliminary trials have been performed. Multiple clients were initialised on a number of different machines and registered with a tracker service. These clients were programmed to migrate between the various sites and perform local tasks, including retrieving files and launching Java applications, whilst communicating with each other on the move.

7 Comparison with Related Work

The Channel Ambient calculus is a variant of Boxed Ambients inspired by recent developments in the design of process calculi for mobility. The calculus uses guarded replication, which is present in modern variants of Boxed Ambients such as [4]. Actions in CA are similar to actions in Boxed Ambients [9], except that ambients in CA can interact using named channels and sibling ambients can communicate directly. This form of sibling communication is inspired by the

Nomadic π-calculus, although agents in Nomadic π are not nested. A similar form of channel communication is also used in the Seal calculus [8], although sibling seals cannot communicate directly. The use of channels for mobility is inspired by the mechanism of passwords, first introduced in [13] and subsequently adopted in [4]. The main advantage of CA over existing variants of Boxed Ambients is its ability to express high-level constructs such as channel-based interaction and sibling communication directly. These constructs seem to be at the right level of abstraction for writing mobile applications such as the tracker application outlined in Section 6. The main advantage of CA over the Nomadic π-calculus is its ability to regulate access to an ambient by means of named channels, and its ability to model computation within nested locations, both of which are lacking in Nomadic π.

The Channel Ambient Machine extends the list semantics of the Pict machine, to provide support for nested ambients and ambient migration. Pict uses channel queues in order to store blocked inputs and outputs that are waiting to synchronise. In CAM these channel queues are generalised to a notion of blocked processes, in order to allow both communication and migration primitives to synchronise. A notion of unblocking is also defined, to allow mobile ambients in CAM to re-bind to new environments. The Pict machine does not require a notion of restriction, since all names in Pict are local to a single machine. In contrast, CAM requires an explicit notion of restriction in order to manage the scope of names across ambient boundaries, as given by rules (26), (28) and (33). By definition, the Pict abstract machine is deterministic and, although it is sound with respect to the π-calculus, it is not complete. In contrast, CAM is non-deterministic and is both sound and complete with respect to CA.

A number of abstract machines have also been defined for variants of the Ambient calculus. In [5] an abstract machine for Ambients is presented, which has not been proved sound or complete. In [11] a distributed implementation of Ambients is described, which uses JoCaml as an implementation language. The implementation relies on a formal translation of Ambients into the Distributed Join calculus. However, it is not clear how such a translation can be adapted to boxed ambient calculi, which use more sophisticated communication primitives. Furthermore, the implementation is tied to a particular programming language (JoCaml), which limits the scope in which it can be used. In [18] a distributed abstract machine for Safe Ambients is presented, which uses logical forwarders to represent mobility. Physical mobility can only occur when an ambient is *opened*. However, such an approach is not applicable to boxed ambient calculi, where the *open* primitive is non-existent.

The ambient tracker application described in Section 6 is inspired by work on mobile applications presented in [21]. This work identifies a key role for process calculi in modelling communication infrastructures for mobile agent systems. Such infrastructures can be used as a foundation for building robust applications in the areas of data mining and resource monitoring, among others. In [21] the Nomadic π-calculus is used to model an infrastructure for reliably forwarding messages to mobile agents, and a centralised version of this algorithm is proved correct in [20]. In Section 6, a decentralised version of this algorithm is modeled

in CA. The main advantage of Channel Ambients over Nomadic Pict lies in the correctness of the Channel Ambient Machine, which ensures that any properties satisfied by the calculus model will also hold in its implementation.

8 Conclusions and Future Work

In this paper we presented a distributed abstract machine for a variant of the Boxed Ambient calculus with channels, and proved the soundness and completeness of the machine with respect to the calculus. We then described a prototype implementation, together with an application for tracking the location of mobile ambients. The prototype is a useful tool for experimenting with the development of mobile applications based on a formal model, which we hope will provide insight into the design of future programming languages for mobile computing. To our knowledge, this is the first time that a correct abstract machine for a variant of Boxed Ambients has been implemented in a distributed environment. The correctness of the machine ensures that the work done in specifying and analysing mobile applications is not lost during their implementation.

We are currently using the techniques outlined in this paper to define abstract machines for other variants of boxed ambients, such as those described in [9] and [4]. We also believe that similar techniques can be used to define an abstract machine for Safe Ambients [12]. It would be interesting to see how the resulting machine compares with those defined in [11] and [18].

Another area for future research is the choice between a deterministic and a non-deterministic implementation. Non-determinism guarantees completeness, but requires a random scheduling algorithm to be implemented. Determinism is more efficient, but leads to weaker correctness properties. The machine defined in this paper allows both alternatives to be implemented. It would be interesting to investigate whether the more efficient deterministic machine provides sufficient guarantees for correctness.

The main motivation for defining an abstract machine for CA, as opposed to another variant of boxed ambients, was the desire to express high-level constructs such as channel-based interaction and sibling communication directly. In future, we plan to investigate whether these constructs can be encoded in other variants of Boxed Ambients. We also intend to ascertain whether existing type theories for boxed ambient calculi can be smoothly to applied to CA.

More generally, we plan to investigate whether existing program analysis techniques can be used to prove properties of applications written in CA. Examples of such techniques include equivalence-based analysis of mobile applications [20], model-checking techniques based on ambient logic [6] and security types for mobile ambients [10].

Acknowledgements. The authors would like to thank the ESOP referees and the members of the Imperial College Concurrency group for their comments and suggestions. Nobuko Yoshida is partially supported by EPSRC grants GR/R33465 and GR/S55538.

References

1. L. Bettini, R. D. Nicola, and R. Pugliese. Xklaim and klava: Programming mobile code. In M. Lenisa and M. Miculan, editors, *ENTCS*, volume 62. Elsevier, 2002.
2. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01*, number 2215 in LNCS, pages 38–63. Springer, 2001.
3. M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *CONCUR'01*, number 2154 in LNCS, pages 102–120. Springer, 2001.
4. M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 71–84. Springer, 2002.
5. L. Cardelli. Mobile ambient synchronization. Technical Report SRC-TN-1997-013, Hewlett Packard Laboratories, July 25 1997.
6. L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of POPL '00*, pages 365–377. ACM, Jan. 2000.
7. L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Comput. Sci.*, 240(1):177–213, 2000. An extended abstract appeared in *FoSSaCS '98*: 140–155.
8. G. Castagna, J. Vitek, and F. Zappa. The seal calculus. 2003. Available from ftp://ftp.di.ens.fr/pub/users/castagna/seal.ps.gz.
9. S. Crafa, M. Bugliesi, and G. Castagna. Information flow security for boxed ambients. In *F-WAN'02*, number 66(3) in ENTCS, 2002.
10. M. Dezani-Ciancaglini and I. Salvo. Security types for mobile safe ambients. In *ASIAN'00*, volume 1961 of *LNCS*, pages 215–236. Springer, 2000.
11. C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Proceedings of TCS 2000*, volume 1872 of *LNCS*, pages 348–364. IFIP, Springer, Aug. 2000.
12. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*. ACM Press, 2000.
13. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *POPL'02*, pages 71–80. ACM Press, 2002.
14. M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *CONCUR'02*, volume 2421 of *LNCS*, pages 304–320. Springer, 2002.
15. A. Phillips. *The Channel Ambient System*, 2003. Runtime system and related documentation available from http://www.doc.ic.ac.uk/~anp/ca.html.
16. A. Phillips. *The Channel Ambient Calculus: From Process Algebra to Mobile Code*. PhD thesis, Imperial College London, 2004. Forthcoming.
17. A. Phillips, S. Eisenbach, and D. Lister. From process algebra to java code. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.
18. D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. In *ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.
19. D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).
20. A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *POPL'01*, pages 116–127, 2001.
21. P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, June 2000. Also appeared as Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.
22. P. T. Wojciechowski. *The Nomadic Pict System*, 2000. Available electronically as part of the Nomadic Pict distribution.