

# Chapter 1

---

## Specifying and Implementing Secure Mobile Applications

1.1	Introduction .....	1
1.2	The Channel Ambient Calculus .....	4
1.3	The Channel Ambient Machine .....	17
1.4	The Channel Ambient Runtime .....	32
1.5	Agent Tracker Application .....	39
1.6	Related Work .....	44
1.7	Conclusion .....	47

---

### 1.1 Introduction

The Internet has grown substantially in recent years, and an increasing number of applications are now being developed to exploit this distributed infrastructure. Mobility is an important paradigm for such applications, where mobile code is supplied on demand and mobile components interact freely within a given network. However, mobile applications are difficult to develop. Not only do they involve complex parallel interactions between multiple components, but they must also satisfy strict security requirements. This chapter describes how the problem of specifying and implementing secure mobile applications can be addressed using a process calculus.<sup>1</sup>

The Internet is used for a wide variety of distributed that can benefit from mobile software. Specific examples include search engines, mining of data repositories, scripting languages for web browser animations, peer-to-peer file sharing systems, financial trading software, consumer auction sites and travel reservation systems. Unfortunately, these applications are often hindered by two phenomena that cannot be abstracted away in a distributed setting: *network latency* and *network failure* (see [29] for technical definitions and related discussion). Network latency refers to the interval of time between the departure of a message from one machine and its arrival on another machine. Two common factors of network latency are network congestion and the use of a slow network interface. Network failure, on the other hand, refers to a break in the connectivity between two machines on a network. This can occur for a

---

<sup>1</sup>The chapter is an extended version of [33] and is based on the author's PhD thesis [31]

variety of reasons. In some cases, network congestion can cause certain packets to be lost, resulting in a temporary disconnection between two machines. In other cases, a machine can be brought down by a direct attack from another machine and become isolated from the rest of the network. Alternatively, a machine can be physically unplugged from the network for a period of time.

In spite of the large increase in network bandwidth over the last few years [14, 28]<sup>2</sup>, network latency and failure are still very much an issue. This is due in part to the large increase in the size of files being transmitted over a network, such as audio and video content, particularly at peak times. Another reason is the increase in unsolicited network traffic. For example, certain viruses and worms can have a devastating effect on networks, albeit for a limited time period. Finally, with the growing popularity of mobile devices such as laptops, hand-helds and Internet-enabled mobile phones, the effects of network latency and failure are becoming increasingly apparent. This is because mobile devices can have comparatively slower connection speeds than fixed machines, and tend to connect and disconnect from networks more frequently.

In order to minimise the effects of network latency and failure, distributed applications are relying increasingly on *mobile software* in the form of *mobile code* and *mobile agents*. By definition, mobile code refers to program code that can be sent from one machine to another over a network. The code itself has no state, and can only begin executing after reaching its destination. A mobile agent, on the other hand, refers to an autonomous program that can stop executing, move through a network to a new machine, and continue executing at its new location. In the general case, a mobile agent can autonomously travel to an itinerary of multiple destinations, preserving its state after each move.

Mobile software, in the form of mobile code and mobile agents, can help to minimise the effects of network latency and failure in a number of distributed applications. Some of the benefits of mobile agents are described in [12]. It is somewhat ironic that many of the problems related to the use of mobile devices can in part be solved by mobile software.

Mobile applications can be roughly grouped into three main categories: *mobile code*, *resource monitoring* and *information retrieval*. Applets, Scripting languages and Data Mining are examples of mobile code, where a single piece of code is sent to a remote machine. In the case of data mining, agents are dispatched to data warehouses, such as those containing consumer information or news archives, to look for general trends in the data. Online Trading and Electronic Commerce are examples of resource monitoring, and Travel Agencies, Search Engines and Peer-to-Peer applications are examples of information retrieval. A broader survey of the various categories of mobile applications can be found in [41]. It is worth noting that a large proportion of

<sup>2</sup>Recent internet growth statistics are available from <http://www.dtc.umn.edu/mints/>

these applications involve mobile software agents travelling between hardware devices over wide-area networks such as the Internet. Even applications that only require mobile code can be expressed using the agent paradigm, giving programmers the flexibility to extend these applications and make full use of agents as appropriate. For example, in cases where a client uploads mobile code to a server, the client license may expire while the client is disconnected. If mobile agents are used, the agent responsible for uploading the code can move to a renewal site, negotiate the renewal of the license and then return to the server to continue executing the code. Perhaps one of the simplest categories of mobile applications is resource monitoring, and one such application is used as a running example for this chapter. Information retrieval applications are more complex, since they require sophisticated algorithms for allowing agents to communicate with each other as they move between data repositories. One such application is presented at the end of this chapter.

**Process Calculi for Mobile Applications** Process calculi have recently been proposed as a promising formalism for specifying and implementing secure mobile applications. Calculi can be thought of as simple programming languages, which provide a concise description of computation that facilitates rigorous analysis. They have a precise syntax and computable operational semantics that are both formally defined, together with an execution state that is implicit in the terms of the calculus. This contrasts with many alternative models of computation, including automata models, where the execution state needs to be given explicitly as a separate component.

Calculi have been used successfully for many years to model various forms of computation. An important example is the  $\lambda$ -calculus [13], which captures the essence of functional computation in a small number of terms. This enables a concise description of functional computation that supports formal reasoning about the correctness of algorithms. Many functional programming languages including Standard ML [27] have been based on the  $\lambda$ -calculus, which provides a solid theoretical foundation.

More recently, the  $\pi$ -calculus [26] has been developed as a model for concurrent computation. Many argue that the  $\pi$ -calculus achieves for concurrent computation what the  $\lambda$ -calculus does for functional computation, capturing the essence of computation in a small number of terms and facilitating formal reasoning.

With the advent of mobile programming, there has been considerable research on calculi for mobile computation. In particular, the Nomadic  $\pi$ -calculus [37] demonstrated the feasibility of using process calculi to specify and implement applications involving location-independent communication between mobile agents. The Ambient calculus [10] was also introduced to model the hierarchical topology of modern networks, and many variants of ambients were subsequently proposed. One variant of Ambients that seems well-suited to modelling mobile applications is the Boxed Ambient calculus.

In general, Boxed Ambient calculi have been used to model and reason about a variety of issues in mobile computing. The original paper on Boxed Ambients [5] shows how the Ambient calculus can be complemented with finer-grained and more effective mechanisms for ambient interaction. In [6] Boxed Ambients are used to reason about resource access control, and in [16] 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, the Safe Boxed Ambient calculus [23] uses co-capabilities to express explicit permissions for accessing ambients, and the NBA calculus [7] seeks to limit communication and migration interferences in Boxed Ambients. Boxed Ambient calculi can also benefit from many of the analysis techniques of the Ambient calculus, most notably Ambient Logics [9]. Although research on calculi for mobile applications is still in its early stages, already the potential benefits of such calculi are beginning to be widely recognized. In spite of these theoretical advances, there has been little research on how Boxed Ambient calculi can be correctly implemented in a distributed environment. Furthermore, Ambient calculi in general have not yet been used to model real-world mobile applications. This chapter demonstrates that it is feasible to develop a distributed programming language for mobile applications, based on a variant of the Ambient calculus. Furthermore, it is feasible to derive a provably correct algorithm for executing these applications, and to refine this algorithm to executable program code.

The chapter is structured as follows. Section 1.2 presents a novel calculus for specifying mobile applications, known as the Channel Ambient calculus (CA). The calculus is inspired by previous work on calculi for mobility, including the  $\pi$ -calculus, the Nomadic  $\pi$ -calculus and the Ambient calculus. Section 1.3 presents an abstract machine for the Channel Ambient calculus, known as the Channel Ambient Machine (CAM). The abstract machine is a formal specification of a runtime for executing calculus processes, which bridges a gap between the specification and implementation of mobile applications. Section 1.4 presents a runtime for executing processes of the Channel Ambient calculus, known as the Channel Ambient Runtime. The runtime is implemented by defining a direct mapping from the Channel Ambient Machine to functional program code. The Channel Ambient Language (CAL) is also presented, together with an example mobile application, in which a mobile agent monitors resources on a remote server. Finally, Section 1.5 presents an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network.

## 1.2 The Channel Ambient Calculus

Process calculi were described in Section 1.1 as a promising formalism for specifying and implementing secure mobile applications. In particular, Boxed Ambient calculi [7] were identified as a foundation for specifying applications that execute on networks with a hierarchical structure. Boxed Ambient calculi benefit from a range of analysis techniques originally developed for the Ambient calculus, including type systems, equational theories and Ambient Logics. They also enforce more rigid security measures by preventing ambient boundaries from being dissolved. This allows for new and richer security mechanisms to be developed, including various type systems and methods for control flow analysis. Many variants of Boxed Ambients have been defined, together with analysis techniques that can be used across multiple variants. Boxed Ambient calculi therefore seem a natural starting point when looking for a calculus to specify and implement secure mobile applications.

Although numerous variants of Boxed Ambients have been proposed, it can be argued that an essential feature of mobile applications is lacking from these variants, namely the existence of *channels* as first class entities. At the lowest level, channels are the building blocks of some of the most widely used protocols in mobile applications, including TCP/IP. Channels are also a fundamental programming abstraction, since they correspond to the notion of methods or *services* provided by an application component. In particular, channels allow a given component to provide multiple services, each of which is invoked using a separate service channel. From a security perspective channels are also fundamental, since they correspond to the notion of *keys* that can be used to regulate communication and migration of components in a mobile application. For example, a private channel can be used by a single component to establish a private communication with another component or to gain exclusive access to a location. Conversely, a public channel can be used to provide a public communication service or to enable public access to a location.

This section presents a calculus for specifying mobile applications, known as the Channel Ambient calculus (CA). The calculus was first described in an extended abstract [33], which formed the basis of the author's PhD thesis [31]. The calculus is inspired by previous work on calculi for mobility, including the  $\pi$ -calculus, the Nomadic  $\pi$ -calculus and the Ambient calculus. In many respects, the calculus can be considered a variant of Boxed Ambients in which channels are defined as first class entities, allowing ambients to communicate with each other and move in and out of each other over channels.

### 1.2.1 Syntax of Calculus Processes

The Channel Ambient calculus uses the notion of an *ambient*, first presented in [10], to model the components of a mobile application. An ambient is an abstract entity that can be used to model a machine, a mobile agent or a module. In this calculus ambients are named, arranged in a hierarchy and can interact by sending messages to each other and moving in and out of each other over channels.

---

$P, Q, R ::= \mathbf{0}$	Null	$\alpha ::= a \cdot x \langle v \rangle$	Sibling Output
$  P   Q$	Parallel	$  x^\dagger \langle v \rangle$	Parent Output
$  \nu n P$	Restriction	$  x(u)$	Internal Input
$  a \boxed{P}$	Ambient	$  x^\dagger(u)$	External Input
$  \alpha.P$	Action	$  \text{in } a \cdot x$	Enter
$  !\alpha.P$	Replication	$  \text{out } x$	Leave
		$  \overline{\text{in}} x$	Accept
		$  \overline{\text{out}} x$	Release

---

#### Definition 1.2.1 Syntax of CA

---

$\begin{array}{ c } \hline P \\ \hline \end{array}, \begin{array}{ c } \hline Q \\ \hline \end{array}, \begin{array}{ c } \hline R \\ \hline \end{array} ::= \begin{array}{ c } \hline \mathbf{0} \\ \hline \end{array}$	Null	$\begin{array}{ c } \hline P \\ \hline \end{array} \begin{array}{ c } \hline Q \\ \hline \end{array}$	Parallel	$\begin{array}{ c } \hline x : \begin{array}{ c } \hline P \\ \hline \end{array} \\ \hline \end{array}$	Restriction
$\begin{array}{ c } \hline a \\ \hline \end{array} \begin{array}{ c } \hline P \\ \hline \end{array}$	Ambient	$\begin{array}{ c } \hline \alpha.P \\ \hline \end{array}$	Action	$\begin{array}{ c } \hline !\alpha.P \\ \hline \end{array}$	Replication

---

#### Definition 1.2.2 Graphical Syntax of CA

The syntax of the Channel Ambient calculus is presented in Definition 1.2.1 in terms of processes  $P, Q, R$ , actions  $\alpha$  and values  $a, b, \dots, z$ . It is assumed that  $a, b, c$  represent ambient names,  $x, y, z$  represent channel names,  $n, m$  represent ambient or channel names and  $u, v$  represent arbitrary values. In an applied version of the calculus, these values can include names, constants, tuples etc. A corresponding graphical syntax is presented in Definition 1.2.2. The processes  $P, Q, R$  of the calculus have the following meaning:

**Null**  $\mathbf{0}$  does nothing and is used to represent the end of a process.

**Parallel**  $P | Q$  executes process  $P$  in parallel with process  $Q$ .

**Restriction**  $\nu n P$  executes process  $P$  with a private name  $n$ .

**Ambient**  $a \boxed{P}$  executes process  $P$  inside an ambient  $a$ . The ambient  $a$  represents a component of a mobile application, such as a machine, a mobile agent or a module.

**Action**  $\alpha.P$  tries to perform the action  $\alpha$  and then execute process  $P$ . The action can involve either a communication or a migration.

**Replication**  $! \alpha.P$  repeatedly tries to perform the action  $\alpha$  and then execute process  $P$ .

The graphical syntax is reminiscent of various informal representations for concurrent processes, in which each process or thread is represented as a vertical bar. The representation is particularly reminiscent of Message Sequence Charts [19], where each component is represented as a box above a process, labelled with the component name, and parallel composition is represented as a collection of adjacent processes. Unlike Message Sequence Charts, each vertical bar is also labelled with the current state of the process. In addition, restriction is represented as a dotted ring around a process, labelled with the restricted name. The actions  $\alpha$  of the calculus have the following meaning:

**Sibling Output**  $a \cdot x \langle v \rangle$  tries to send a value  $v$  on channel  $x$  to a sibling ambient  $a$ .

**Parent Output**  $x^\uparrow \langle v \rangle$  tries to send a value  $v$  on channel  $x$  to the parent ambient.

**External Input**  $x^\uparrow(u)$  tries to receive a value  $u$  on channel  $x$  from outside the current ambient.

**Internal Input**  $x(u)$  tries to receive a value  $u$  on channel  $x$  from inside the current ambient.

**Enter**  $\text{in } a \cdot x$  tries to enter a sibling ambient over channel  $x$ .

**Leave**  $\text{out } x$  tries to leave the parent ambient over channel  $x$ .

**Accept**  $\overline{\text{in}} x$  tries to accept a sibling ambient over channel  $x$ .

**Release**  $\overline{\text{out}} x$  tries to release a child ambient over channel  $x$ .

### 1.2.2 Reduction Rules for Executing Calculus Processes

The reduction rules of the Channel Ambient calculus describe how a calculus process can be executed. Each rule is of the form  $P \longrightarrow P'$ , which states that the process  $P$  can evolve to  $P'$  by performing an execution step. The reduction rules are presented in Definition 1.2.3, where  $P_{\{v/u\}}$  assigns the value  $v$  to the

$$a \boxed{b \cdot x \langle v \rangle . P \mid P'} \mid b \boxed{x^\dagger(u) . Q \mid Q'} \longrightarrow a \boxed{P \mid P'} \mid b \boxed{Q_{\{v/u\}} \mid Q'} \quad (1.1)$$

$$a \boxed{x^\dagger \langle v \rangle . P \mid P'} \mid x(u) . Q \longrightarrow a \boxed{P \mid P'} \mid Q_{\{v/u\}} \quad (1.2)$$

$$a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\overline{\text{in } x} . Q \mid Q'} \longrightarrow b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \quad (1.3)$$

$$b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \overline{\text{out } x} . Q \mid Q'} \longrightarrow b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \quad (1.4)$$

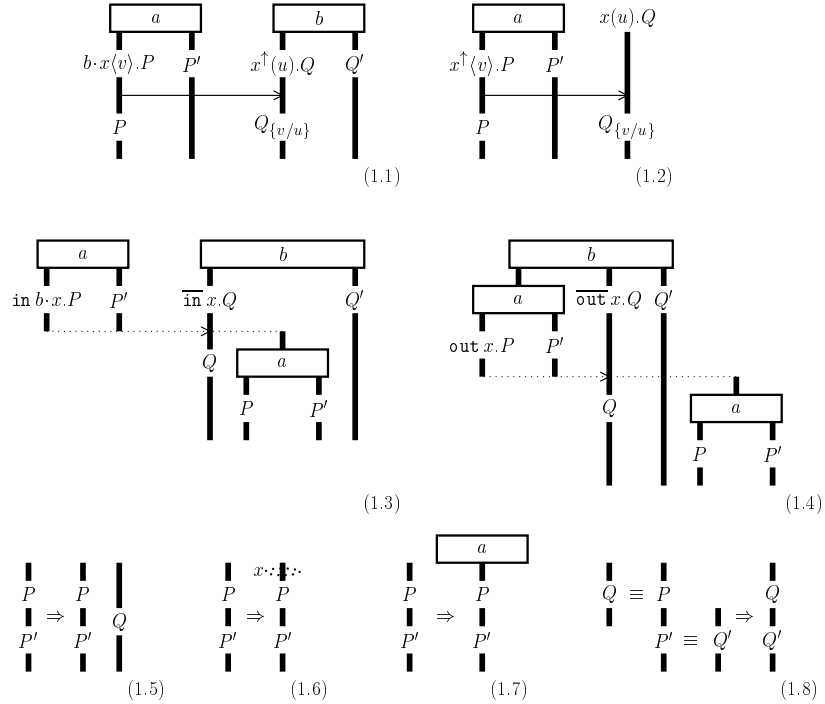
$$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q \quad (1.5)$$

$$P \longrightarrow P' \Rightarrow \nu n P \longrightarrow \nu n P' \quad (1.6)$$

$$P \longrightarrow P' \Rightarrow a \boxed{P} \longrightarrow a \boxed{P'} \quad (1.7)$$

$$Q \equiv P \longrightarrow P' \equiv Q' \Rightarrow Q \longrightarrow Q' \quad (1.8)$$

**Definition 1.2.3** *Reduction in CA*



**Definition 1.2.4** *Graphical Reduction in CA*



value  $u$  in process  $P$ , and  $P \equiv Q$  means that process  $P$  is equal to process  $Q$ . A corresponding graphical representation of the reduction rules is presented in Definition 1.2.4:

- (1.1) An ambient can send a value to a sibling over a channel. If an ambient  $a$  contains a sibling output  $b \cdot x \langle v \rangle . P$ , and there is a sibling ambient  $b$  with an external input  $x^\uparrow(u) . Q$ , then the value  $v$  can be sent to ambient  $b$  along channel  $x$ , and assigned to the value  $u$  in process  $Q$ .
- (1.2) An ambient can send a value to its parent over a channel. If an ambient  $a$  contains a parent output  $x^\uparrow \langle v \rangle . P$ , and there is an internal input  $x(u) . Q$  in parallel with  $a$ , then the value  $v$  can be sent along channel  $x$ , and assigned to the value  $u$  in process  $Q$ .
- (1.3) An ambient can enter a sibling over a channel. If an ambient  $a$  contains an enter  $\text{in } b \cdot x . P$ , and there is a sibling ambient  $b$  with an accept  $\text{in } x . Q$ , then  $a$  can enter  $b$  over channel  $x$ .
- (1.4) An ambient can leave its parent over a channel. If an ambient  $a$  contains a leave  $\text{out } x . P$ , and there is a parent ambient with a release  $\text{out } x . Q$ , then  $a$  can leave its parent over channel  $x$ .
- (1.5) A reduction can occur inside a parallel composition. If a process  $P$  can reduce to  $P'$ , then the reduction can also take place in parallel with a process  $Q$ .
- (1.6) A reduction can occur inside a restriction. If a process  $P$  can reduce to  $P'$  then the reduction can also take place if  $P$  has a restricted name  $n$ .
- (1.7) A reduction can occur inside an ambient. If a process  $P$  can reduce to  $P'$  then the reduction can also take place if  $P$  is inside an ambient  $a$ .
- (1.8) Equal processes can perform the same reduction. If a process  $P$  can reduce to  $P'$ ,  $Q$  is equal to  $P$  and  $Q'$  is equal to  $P'$ , then  $Q$  can reduce to  $Q'$ .

The graphical reduction rules are reminiscent of Message Sequence Charts, where time proceeds vertically downward, and communication between parallel components is represented by a solid arrow from a thread inside the sender to a thread inside the receiver. Unlike standard Message Sequence Charts, a given component can also move in or out of another component. This is represented as a dotted arrow from a thread inside the migrating component to a thread inside the destination component. In addition, each vertical bar is labelled with the current state of the thread, which can be modified as a result of an interaction.

### 1.2.3 Substitution of Free Values Inside Processes

Substitution in the Channel Ambient calculus is used to replace one free value by another. The expression  $P_\sigma$  denotes the application of a substitution  $\sigma$  to a process  $P$ , where  $\sigma$  is a substitution that maps a given value to another (different) value, and  $v_\sigma$  applies the substitution  $\sigma$  to the value  $v$ . An example of a substitution is  $P_{\{u', v'/u, v\}}$ , which replaces  $u$  with  $u'$  and  $v$  with  $v'$  in process  $P$ . If  $v$  is not in the domain of  $\sigma$  then  $v_\sigma = v$ . By convention, it is assumed that there is no overlap between the set of bound names of a process and the set of substituted values given by the range of  $\sigma$ .

The expression  $\text{fn}(P)$  denotes the set of free values  $\text{fn}(P)$  of a process  $P$  in the Channel Ambient calculus. The definition is standard, and relies on the definition of the set of bound values  $\text{bn}(P)$ , where restriction  $\nu n P$  binds the name  $n$  in process  $P$ , and internal input  $x(u).P$  and external input  $x^\dagger(u).P$  bind the value  $u$  in process  $P$ . The set of bound values  $\text{bn}(\alpha)$  of an action  $\alpha$  is defined as  $\{u\}$  for  $\alpha = x(u)$  and  $\alpha = x^\dagger(u)$ , and  $\emptyset$  otherwise. As usual, processes of the calculus are assumed to be equal up to renaming of bound values.

### 1.2.4 Structural Congruence Rules for Equating Calculus Processes

$$\mathbf{0} \mid P \equiv P \quad (1.9)$$

$$P \mid Q \equiv Q \mid P \quad (1.10)$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad (1.11)$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P \equiv \alpha.(P \mid !\alpha.P) \quad (1.12)$$

$$\nu n \mathbf{0} \equiv \mathbf{0} \quad (1.13)$$

$$\nu n \nu m P \equiv \nu m \nu n P \quad (1.14)$$

$$n \notin \text{fn}(Q) \Rightarrow (\nu n P) \mid Q \equiv \nu n (P \mid Q) \quad (1.15)$$

$$a \neq n \Rightarrow a[\nu n P] \equiv \nu n a[P] \quad (1.16)$$

$$\nu a a[\mathbf{0}] \equiv \mathbf{0} \quad (1.17)$$

---

#### Definition 1.2.5 Structural Congruence in CA

---

The structural congruence rules of the Channel Ambient calculus describe what it means for two processes to be equal. The rules are presented in Definition 1.2.5 and are mostly standard, apart from the rule for replication

(1.12). As usual, structural congruence is the least congruence that satisfies the rules in Definition 1.2.5.

### 1.2.5 Syntax Abbreviations for Frequently Used Processes

$$\begin{aligned} z \notin \text{fn}(x, v, P) \Rightarrow x\langle v \rangle.P &\triangleq \nu z (z \boxed{x^\dagger\langle v \rangle.z^\dagger\langle \rangle} \mid z().P) \\ z \notin \text{fn}(a, x, v, P) \Rightarrow a/x\langle v \rangle.P &\triangleq \nu z (z \boxed{a \cdot x\langle v \rangle.z^\dagger\langle \rangle} \mid z().P) \end{aligned}$$

**Definition 1.2.6** *Syntax Abbreviations in CA*

A number of convenient abbreviations can be defined for the Channel Ambient calculus, in order to improve the readability of the calculus syntax. Standard syntactic conventions are used, including writing  $\alpha$  as an abbreviation for  $\alpha.\mathbf{0}$  and assigning the lowest precedence to the parallel composition operator. In addition, local output  $x\langle v \rangle.P$  and child output  $a/x\langle v \rangle.P$  can be encoded using parent output and sibling output, respectively, as described in Definition 1.2.6.

Note that the structural congruence rule  $\nu a a \boxed{\mathbf{0}} \equiv \mathbf{0}$  allows the empty ambient  $z$  to be garbage-collected after the value  $v$  has been sent. The encodings of local output and child output are straightforward enough to justify the use of syntactic abbreviations, rather than extending the syntax and reduction rules of the calculus itself.

### 1.2.6 Using the Calculus Syntax to Model TCP/IP Networks

The Channel Ambient calculus can be used to model mobile applications that execute in a wide range of networks. Depending on the choice of network, the syntax of the calculus can be constrained to model the properties of the underlying network protocols. In this chapter, mobile applications are assumed to execute on networks that support the widely used TCP/IP version 4 protocol. Although TCP/IP networks are highly complex, it is possible to make a number of abstractions in order to obtain a high-level view of the main network properties. Based on this high-level view, the syntax of the calculus can be constrained to distinguish between two types of ambients: *sites*  $s$  and *agents*  $g$ . Sites represent hardware devices that are assumed to have a fixed network address, while agents represent software programs that can

move in and out of sites and other agents. In practice, the name of a site corresponds to an IP address, while the name of an agent corresponds to a simple identifier.

The following constraints can be placed on the syntax of the Channel Ambient calculus, in order to model the behaviour of sites and agents:

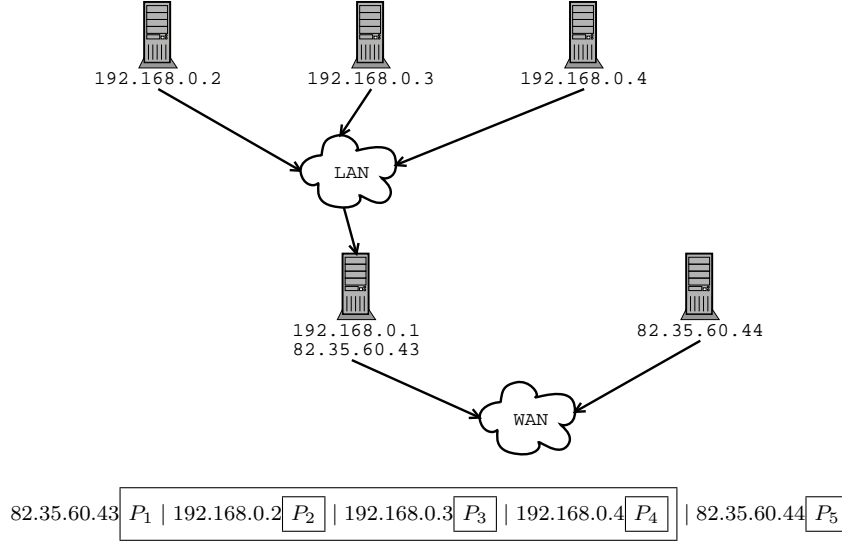
**Sites** A process inside a site  $s$  is constrained so that it cannot contain a sibling output to an agent. This reflects the assumption that agents do not have a network address, and therefore cannot be reached directly by sites over a network. In addition, a process inside a site  $s$  is constrained so that it cannot contain an enter or a leave. This reflects the assumption that sites have a fixed network address. Note that this constraint does not prevent a site from physically moving around in the network. It merely ensures that a given site remains in the same logical location with respect to other sites.

**Agents** A process inside an agent  $g$  is constrained so that it cannot contain a site. This reflects the assumption that hardware sites cannot be contained inside software agents.

In a flat network topology, all sites are assumed to execute in parallel with each other and a given site can potentially communicate with any other site in the network. In a hierarchical topology, sites can be logically contained inside other sites to form Local Area Networks (LANs). As a result, a given site is unable to communicate directly with another site that is not in the same LAN.

For such hierarchical networks it is useful to distinguish between two types of sites: ordinary sites and *gateways*. A gateway is a site that can logically contain other sites to form a Local Area Network, while an ordinary site cannot contain any other sites. In practice, a gateway acts as a bridge between two networks: the local network it contains and the global network in which it is contained. As a result, a given gateway is usually assigned two network addresses, one for the local network and one for the global network. By definition, the Channel Ambient calculus only allows a gateway to have a single name, which is used by other ambients to interact with the gateway. Based on this definition, the name of a gateway corresponds to its global address. The local address is not needed at the calculus level, since child ambients do not need to explicitly use the name of their parent in order to interact with it. Therefore, the local address is used merely as an implementation mechanism, to allow messages or agents from child ambients to be correctly routed to the parent gateway.

An example of how sites can be used to model the topology of Local and Wide Area Networks is illustrated in Figure 1.1. In this example, each site executes on a separate machine, with a given IP address. The sites 192.168.0.2 - 192.168.0.4 are part of a Local Area Network inside a gateway with local address 192.168.0.1 and global address 82.35.60.43. The ambients inside the



**Figure 1.1:** Hierarchical TCP/IP Networks and their corresponding calculus representation.

LAN will send messages or agents to a default parent, which will automatically be routed to the local address of the gateway. The ambients outside the LAN will send messages or agents to the global address of the gateway. Thus, the local and global addresses allow the gateway to distinguish between local and global interactions.

### 1.2.7 Using Reduction to Model Network Execution

The Channel Ambient calculus can be used to model the execution of mobile applications on networks that support the TCP/IP protocol. Note that the details of setting up of a TCP/IP session are below the level of abstraction of the calculus, which only models individual interactions. A host with address  $IP_1$  can be modelled as a site with name  $IP_1$ , a port number  $n$  can be modelled as a channel with name  $n$  and communication between hosts can be modelled using the communication primitives of the calculus. For example, a server with IP address 82.35.60.43 running an ftp service on port 21 and a telnet service on port 23 can be modelled as a site with name 82.35.60.43 containing replicated external inputs on channels 21 and 23. A corresponding client with IP address 82.35.60.44 that interacts with the server can be modelled as a site with name 82.35.60.44 containing a sibling output to the server on the corresponding channels:

$$82.35.60.43 \ !21^\dagger(args).Ftp \mid !23^\dagger(args).Telnet \mid Server$$

$$| 82.35.60.44 \boxed{82.35.60.43 \cdot 21 \langle v \rangle . P \mid Client} \mid Network$$

The migration of agents between hosts on the network can also be modelled using the primitives of the calculus. For example, a server with IP address 82.35.60.43 that accepts an agent on port 3001 can be modelled as a site with name 82.35.60.43 containing an accept on channel 3001. In general, when two ambients interact over a network the channel corresponds to a port number, and when two ambients interact locally the channel corresponds to a simple identifier.

A private communication channel established between two hosts can be modelled using channel restriction. For example, a private *ssh* channel that was established between a client and a server using the SSH protocol can be modelled as:

$$\nu ssh (client \boxed{server \cdot ssh \langle n \rangle . P \mid Client} \mid server \boxed{ssh^\dagger(m).Q \mid Server}) \mid Network$$

The use of restriction to limit the scope of the *ssh* channel between client and server guarantees, at an abstract level, that other entities in the network cannot interfere with communication on this channel. For a more detailed model of establishing private channels over a network, encryption and decryption mechanisms can be added as an extension to the calculus, in the style of [1, 4].

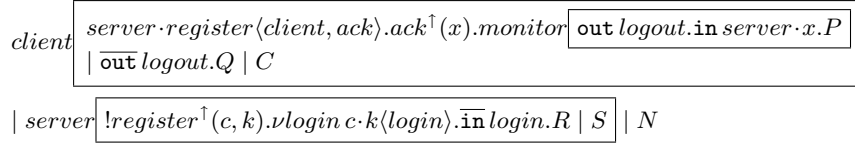
Up to this point, TCP/IP networks have been modelled in the Channel Ambient calculus by mapping IP addresses to sites and port numbers to channels. Implicitly, this approach assumes that each site corresponds to a separate device on the network, with its own IP address. An alternative, more flexible approach is to map each site to a *socket address*, where a socket address consists of an IP address and a port number. This allows multiple sites to run on the same device, where each site uses a separate port on the device. Arbitrary string names can then be used as channels and multiple channels can share the same port, resulting in a significantly more flexible interaction model. This can be implemented by adding a thin layer of multiplexing above the TCP/IP protocol, in order to allow a given site to interact on an arbitrary number of named channels.

### 1.2.8 Resource Monitoring Application

This section describes how the Channel Ambient calculus can be used to specify an example mobile application, in which a mobile agent monitors a resource on a remote server.

Figure 1.2 presents a formal specification of the resource monitoring application, expressed as a process of the Channel Ambient calculus:

- The *client* tries to send its name and an acknowledgement channel *ack* to the server on the *register* channel. After sending the registration request, the client waits for a login channel *x* on the acknowledgement

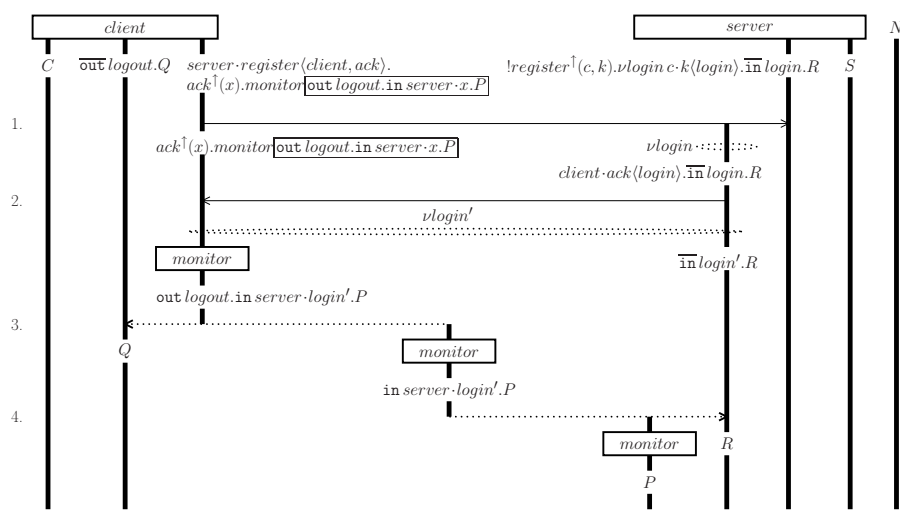
**Figure 1.2:** Resource Monitoring Specification

channel. After receiving the login channel, the client creates a new *monitor* agent, which tries to leave on the *logout* channel, enter the server on the login channel and then execute the process  $P$ . In parallel, the client tries to release an agent on the logout channel and then execute the process  $Q$ . The client can also execute other processes in parallel, represented by the process  $C$ .

- The *server* continually listens on the *register* channel for a client name  $c$  and an acknowledgement channel  $k$ . Each time a registration request is received, the server creates a new *login* channel. The server tries to send the login channel to the client on the acknowledgement channel, accept an agent on the login channel and then execute the process  $R$ . The server can handle multiple requests concurrently, represented by the process  $S$ .
- The network can contain arbitrary agents and machines, represented by the process  $N$ . These agents and machines can potentially try to interfere with the interactions between the client and the server.

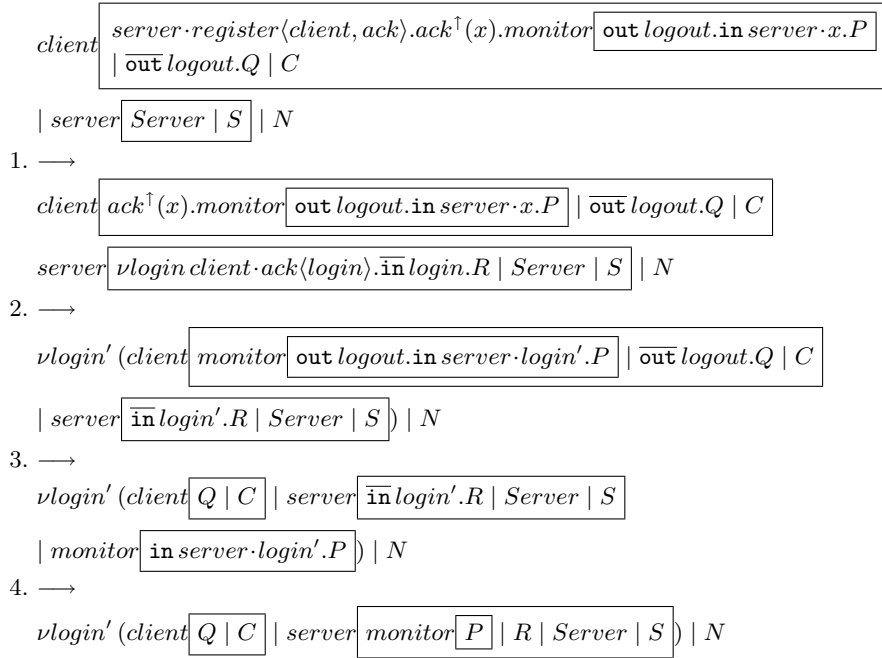
Figure 1.3 uses the graphical Channel Ambient calculus to formally describe an execution scenario for the resource monitoring application. The corresponding textual execution scenario is presented in Figure 1.4:

1. The client sends its name and an acknowledgement channel to the server on the register channel.
2. The server creates a new *login* channel and sends it to the client on the acknowledgement channel.
3. The client creates a new monitor agent, which leaves the client on the logout channel.
4. The monitor agent enters the server on the login channel and executes the process  $P$ , which monitors the resource on the server. In parallel, the server executes the process  $R$ , which forwards information about the resource to the monitor.



**Figure 1.3:** Graphical Calculus Execution Scenario, where  $\text{login}' \notin \text{fn}(P, R)$

$$\text{Server} \triangleq !\text{register}^\dagger(c, k).v\text{login } c.k(\text{login}).\overline{\text{in}} \text{login}.R$$



**Figure 1.4:** Calculus Execution Scenario, where  $\text{login}' \notin \text{fn}(P, Q, R, S, C)$



Note that the graphical representation offers greater flexibility in displaying the scope of restricted names than the corresponding textual representation. In particular, the scope of the ring for the *login'* channel can be graphically adjusted to represent the fact that  $\text{login}' \notin \text{fn}(Q, S, C)$ .

This resource monitoring example illustrates how the Channel Ambient calculus can be used to specify various security mechanisms for mobile applications. On receiving a registration request, the server creates a fresh login channel and sends this to the client over an acknowledgement channel. The login channel acts as a key, which the client can use to send a monitor agent to the server. The server will only allow a single monitor to enter using this key, thereby ensuring strict access control to the server. Similarly, the monitor agent can only leave the client on the logout channel. This prevents other agents that do not know the name of the logout channel from leaving the client without permission. Furthermore, other processes inside the client that do not know the name of the acknowledgement channel cannot interfere with communication from the server, and therefore will be unable to acquire the login channel. More generally, a wide range of analysis techniques that have been developed for related calculi can also be applied to the Channel Ambient calculus, in order to reason about the security properties of applications. Some of these techniques are discussed in [31], including proving safety to prevent runtime errors, using channel types to ensure reliable communication, and using syntactic constraints to prevent ambient impersonation.

---

### 1.3 The Channel Ambient Machine

The Channel Ambient calculus was designed as a high-level formalism for specifying mobile applications. A given application can first be specified as a process of the calculus, and a number of security properties can then be verified for the specification. Once an application has been formally specified in this way, the next stage is to develop a corresponding implementation. One way to achieve this is to use a runtime to execute calculus processes. This approach bridges a gap between specification and implementation by allowing the specification to be executed directly. In addition, the approach ensures that any security properties of the calculus specification are preserved during execution, provided the runtime is implemented correctly. Support for language interoperability can also be provided in the runtime, allowing the mobile and distributed aspects of an application to be specified in the calculus, and the remaining local and functional aspects to be written in a chosen target language. 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 [3]. More importantly, the abstract machine provides an efficient way of executing a large number of parallel threads that constantly synchronise and move between machines. In order to ensure that the runtime is implemented correctly, an abstract machine can be defined as a formal specification of how the runtime should behave. The correctness of the abstract machine can then be verified with respect to the underlying calculus. This section presents an abstract machine for the Channel Ambient calculus, known as the Channel Ambient Machine (CAM). A proof of correctness is given in Section 1.3.8.

### 1.3.1 Syntax of Machine Terms

---


$$V ::= \nu n \, V \text{ Restriction} \quad (1.18) \qquad z ::= \underline{z} \text{ Blocked} \quad (1.23)$$

$$\mid A \text{ List} \quad (1.19) \qquad \mid z \text{ Unblocked} \quad (1.24)$$

$$A, B, C ::= [] \text{ Empty} \quad (1.20)$$

$$\mid \alpha.P :: C \text{ Action} \quad (1.21)$$

$$\mid a[A] :: C \text{ Ambient} \quad (1.22)$$

**Definition 1.3.1** *Syntax of CAM, where  $(:)$  denotes list composition. For convenience,  $!\alpha.P$  is written as syntactic sugar for an expanded replicated action  $\alpha.(P \mid !\alpha.P)$ .*

---

The syntax of the Channel Ambient Machine is defined using terms  $U, V$ , where each term represents a corresponding calculus process. In general, a machine term is a list of actions  $\alpha.P \dots \alpha'.P'$  and ambients  $a[A] \dots a'[A']$ , with a number of top-level restricted names  $n \dots n'$ . The actions and ambients in the list can be either blocked or unblocked, and each ambient contains its own list. The notation  $\alpha.P$  denotes either a blocked action  $\underline{\alpha}.P$  or an unblocked action  $\alpha.P$ , while the notation  $a[A]$  denotes either a blocked ambient  $\underline{a}[A]$  or an unblocked ambient  $a[A]$ . The full syntax of the Channel Ambient Machine is presented in Definition 1.3.1. Therefore, a machine term can be viewed as a tree of actions and ambients with a number of top-level restricted names, where the nodes of the tree are ambients, and the leaves are actions:

$$\nu n \dots \nu n' \, \alpha.P :: \dots :: \alpha'.P' :: a[A] \dots a'[A'] :: []$$

The machine executes a given term by scheduling an unblocked action  $\alpha.P$  somewhere in the tree. If there is a corresponding blocked co-action then

the two actions can interact and a reduction can occur. If there is no corresponding blocked co-action, then the scheduled action is blocked. If all of the actions in a given ambient  $a[A]$  are blocked then the ambient itself is blocked to  $\underline{a}[A]$ . The machine continues scheduling actions in this way until all the actions and ambients in the tree are blocked, and no more reductions can occur.

### 1.3.2 Using Construction to Encode a Process to a Machine Term

In order for a process to be executed by the Channel Ambient Machine, it must first be converted to a corresponding machine term. This can be achieved by defining a suitable *encoding function*. The encoding function  $\llbracket P \rrbracket$  encodes a given process  $P$  to a corresponding machine term using a *construction operator*, as described in Definition 1.3.2. The construction  $P : []$  adds the process  $P$  to the empty list  $[]$ .

---


$$\llbracket P \rrbracket \triangleq P : []$$

**Definition 1.3.2** *Encoding CA to CAM*

---

$$n \notin \text{fn}(P) \Rightarrow P : (\nu n V) \triangleq \nu n (P : V) \quad (1.25)$$

$$\mathbf{0} : A \triangleq A \quad (1.26)$$

$$(P \mid Q) : A \triangleq P : Q : A \quad (1.27)$$

$$n \notin \text{fn}(P : A) \Rightarrow (\nu n P) : A \triangleq \nu n (P_{\{n/m\}} : A) \quad (1.28)$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P : A \triangleq \alpha.(P \mid !\alpha.P) : A \quad (1.29)$$

$$a[\underline{P}] : A \triangleq a[\underline{P : []}] : A \quad (1.30)$$

$$\alpha.P : A \triangleq \alpha.P :: A \quad (1.31)$$

$$n \notin \text{fn}(a, V) \Rightarrow a[\underline{\nu n U}] : V \triangleq \nu n (a[\underline{U}] : V) \quad (1.32)$$

$$n \notin \text{fn}(a, A) \Rightarrow a[\underline{A}] : \nu n V \triangleq \nu n (a[\underline{A}] : V) \quad (1.33)$$

$$a[\underline{A}] : C \triangleq a[\underline{A}] :: C \quad (1.34)$$

**Definition 1.3.3** *Construction in CAM*

---

In general, construction is used to add a process to a machine term or to add an ambient and a term to a machine term. The construction  $P:V$  adds the process  $P$  to the term  $V$ , and the construction  $a[\overline{U}]:V$  adds the ambient  $a$  and the term  $U$  to the term  $V$ . Note that the symbol  $(:)$  is overloaded, and carries a different type depending on the context in which it is used. In the first case it takes a process and a term as arguments, and in the second case it takes an ambient name and two terms as arguments. Also note the distinction between the function  $(:)$ , which expands processes and terms so they can be added to a list, and the syntactic construct  $(:)$ , which denotes an action or ambient at the head of a list. The full definition of construction is given in Definition 1.3.3, where  $\text{fn}(V)$  is the set of free names in  $V$ .

A process is added to a term by extending the scope of any restricted names in the term to the top-level, and then adding the process to the list inside the scope of the restricted names (1.25). A process is added to a list by discarding the null process (1.26), adding each process in a parallel composition separately (1.27), expanding replicated actions (1.29) and extending the scope of any restricted names in the process to the top level (1.28). Any processes inside an ambient are first encoded to corresponding terms, which can then be added to the list (1.30). It is worth emphasising here that a parallel composition  $P_1 \mid \dots \mid P_N$  will ultimately be added to a term  $A$  as a list of processes  $P_1 : \dots : P_N : A$ , in sequence. Individual processes will then be selected for execution using a suitable scheduling algorithm.

An ambient and a term are added to a term by extending the scope of any restricted names in both terms to the top-level (1.33), and then placing the ambient at the head of the list inside the scope of the restricted names (1.34).

### 1.3.3 Using Selection to Schedule a Machine Term

---


$$A@q.P::A' \succ q.P::A@A' \quad (1.35)$$

$$A@b[\overline{B}]:A' \succ b[\overline{B}]:A@A' \quad (1.36)$$

$$A \succ A' \Rightarrow q[\overline{A}]:C \succ q[\overline{A'}]:C \quad (1.37)$$

---

#### Definition 1.3.4 Selection in CAM

---

In order to execute a term of the Channel Ambient Machine, the term must be matched with the left-hand side of one of the execution rules. This can be achieved by defining a selection function, which re-arranges a given term to match another term. The selection function is a simplification of

the structural congruence rules of the calculus, which takes into account the additional constraints on the syntax of machine terms.

The full definition of selection is given in Definition 1.3.4, where the append function  $A@A'$  is used to concatenate the list  $A$  with the list  $A'$ . A list can match itself, an action or ambient inside a list can be brought to the front of the list (1.35)-(1.36), and a list can be re-arranged inside an ambient (1.37). Unlike structural congruence, the selection function is neither symmetric nor transitive. This minimises the amount of re-arranging that is needed to match a term with the left-hand side of one of the execution rules, thereby increasing the efficiency of the machine.

### 1.3.4 Using Unblocking to Prevent Deadlocks During Execution

---


$$[\ ] \triangleq \ ] \quad (1.38)$$

$$[\alpha.P::C] \triangleq \alpha.P::[C] \quad (1.39)$$

$$[a[A]::C] \triangleq a[A]::[C] \quad (1.40)$$

---

**Definition 1.3.5** *Unblocking in CAM*

---

The Channel Ambient Machine distinguishes between blocked and unblocked actions, in order to efficiently schedule a sequence of actions to execute. In principle, the execution rules of the machine are similar to those of the calculus, except that a given interaction takes place between an action and a corresponding blocked co-action. As a result, the machine needs to prevent both an action and a corresponding co-action from being blocked simultaneously. If this happens, the blocked action and co-action may remain deadlocked, each waiting indefinitely for the other to unblock. Such deadlocks could arise when an ambient containing a blocked action moves to a new location containing a corresponding blocked co-action. In order to ensure that deadlocks do not occur during a migration, an unblocking function is used to unblock the contents of an ambient when it moves to a new location. This allows the ambient to *re-bind* to its new environment, by giving any blocked actions in the ambient the chance to interact with their new location.

The unblocking function  $[A]$  unblocks all of the top-level actions in a given list  $A$ . The full definition of unblocking is given in Definition 1.3.5. The function is used to unblock the contents of a given ambient  $a[A]$  when the ambient moves to a new location. Since the execution rules only allow adja-

cent ambients to interact, nested ambients inside  $A$  cannot interact directly with the new environment. Therefore, only the top-level actions in  $A$  need to be unblocked. For example, suppose the ambient  $a$  contains a blocked parent output, and has just moved inside an ambient  $b$ , which contains a corresponding blocked internal input. Unblocking the contents of  $a$  allows it to check its new environment for potential interactions, such as communicating with its new parent:

$$b \left[ \underline{x(m)}.Q :: a \left[ \underline{x^\uparrow \langle n \rangle}.P :: A \right] \right] :: B :: D$$

On the other hand, suppose ambient  $a$  contains a child  $c$  with a blocked parent output. In this case the contents of  $c$  do not need to be unblocked, since its immediate environment has not changed. In particular, there will be no blocked actions inside  $a$  that were not already present before the migration took place:

$$b \left[ \underline{x(m)}.Q :: a \left[ \underline{c \left[ \underline{x^\uparrow \langle n \rangle}.P :: C \right] :: A} \right] \right] :: B :: D$$

### 1.3.5 Using Reduction to Execute a Machine Term

The reduction rules of the Channel Ambient Machine describe how a machine term can be executed. By definition, the relation  $V \longrightarrow V'$  is true if the machine can reduce the term  $V$  to the term  $V'$  in a single execution step. The reduction rules of the machine are derived from the reduction rules of the calculus. In the calculus reduction rules, an ambient can send a value to a sibling or to its parent over a channel, and can enter a sibling or leave its parent over a channel. Each of these rules is mapped to four corresponding reduction rules in the machine, to allow an action to interact with a corresponding blocked co-action and vice-versa, and to block an action or co-action if no interaction can take place. As with the calculus, there is also a rule to allow reduction inside a restriction and inside an ambient, and to allow matching terms to perform the same reductions. However, unlike the calculus, there is no rule to allow reduction inside a list composition, since each rule is defined over the entire length of a list. Note that the entire list needs to be checked before an action can be blocked, since the machine needs to ensure that there is no corresponding blocked co-action. If the entire list is not checked, then a given action could be blocked even though a suitable blocked co-action may be present in another part of the list. This would result in a form of execution deadlock, in which both an action and a corresponding co-action are blocked simultaneously.

The full definition of reduction is given in Definition 1.3.6, where  $A \succ A'$  means that the term  $A$  can be re-arranged to match the term  $A'$ , and  $[A]$  unblocks any top-level blocked actions in  $A$ , as defined previously.

$$C \succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x(v).P :: A} :: C \longrightarrow a \boxed{P:A} : b \boxed{Q_{\{v/u\}}:B} : C' \quad (1.41)$$

$$C \not\succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x(v).P :: A} :: C \longrightarrow a \boxed{b \cdot x(v).P :: A} :: C \quad (1.42)$$

$$C \succ a \boxed{b \cdot x(v).P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow a \boxed{P:A} : b \boxed{Q_{\{v/u\}}:B} : C' \quad (1.43)$$

$$C \not\succ a \boxed{b \cdot x(v).P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow b \boxed{x^\dagger(u).Q :: B} :: C \quad (1.44)$$

$$C \succ x(u).Q :: C' \Rightarrow a \boxed{x^\dagger(v).P :: A} :: C \longrightarrow a \boxed{P:A} : Q_{\{v/u\}} : C' \quad (1.45)$$

$$C \not\succ x(u).Q :: C' \Rightarrow a \boxed{x^\dagger(v).P :: A} :: C \longrightarrow a \boxed{x^\dagger(v).P :: A} :: C \quad (1.46)$$

$$C \succ a \boxed{x^\dagger(v).P :: A} :: C' \Rightarrow x(u).Q :: C \longrightarrow a \boxed{P:A} : Q_{\{v/u\}} : C' \quad (1.47)$$

$$C \not\succ a \boxed{x^\dagger(v).P :: A} :: C' \Rightarrow x(u).Q :: C \longrightarrow x(u).Q :: C \quad (1.48)$$

$$C \succ b \boxed{\text{in } x.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow b \boxed{Q : a \boxed{P:[A]} : B} : C' \quad (1.49)$$

$$C \not\succ b \boxed{\text{in } x.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \quad (1.50)$$

$$C \succ a \boxed{\text{in } b \cdot x.P :: A} :: C' \Rightarrow b \boxed{\text{in } x.Q :: B} :: C \longrightarrow b \boxed{Q : a \boxed{P:[A]} : B} : C' \quad (1.51)$$

$$C \not\succ a \boxed{\text{in } b \cdot x.P :: A} :: C' \Rightarrow b \boxed{\text{in } x.Q :: B} :: C \longrightarrow b \boxed{\text{in } x.Q :: B} :: C \quad (1.52)$$

$$B \succ \overline{\text{out}} x.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{Q:B'} : a \boxed{P:[A]} : C \quad (1.53)$$

$$B \not\succ \overline{\text{out}} x.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \quad (1.54)$$

$$B \succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\text{out } x.Q :: B} :: C \longrightarrow b \boxed{Q:B'} : a \boxed{P:[A]} : C \quad (1.55)$$

$$B \not\succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\text{out } x.Q :: B} :: C \longrightarrow b \boxed{\text{out } x.Q :: B} :: C \quad (1.56)$$

$$V \longrightarrow V' \Rightarrow \nu n V \longrightarrow \nu n V' \quad (1.57)$$

$$B \succ A \wedge A \longrightarrow V' \Rightarrow B \longrightarrow V' \quad (1.58)$$

$$A \longrightarrow V' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{V'} : C \quad (1.59)$$

$$A \not\succ \alpha.P :: A' \wedge A \not\succ b \boxed{B} :: A' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{A} : C \quad (1.60)$$

**Definition 1.3.6** *Reduction in CAM*

### 1.3.6 Modelling Execution on TCP/IP Networks

The reduction rules of the Channel Ambient Machine can be used to model the execution of mobile applications on TCP/IP networks with broadcast. The main rules are explained below:

- (1.41) An ambient  $a$  can send a value  $v$  to a site  $b$  on channel  $x$  using TCP/IP. The blocked external input on channel  $x$  inside site  $b$  can be implemented by binding a socket to a port  $x$  inside a machine with IP address  $b$ , and waiting for a connection on port  $x$ . The sibling output to  $b$  can be implemented by connecting a socket to IP address  $b$  on port  $x$ , and then sending the value  $v$  over the network from  $a$  to  $b$ .
- (1.45) A site  $a$  can send a value  $v$  to its parent over a channel  $x$  using TCP/IP. The blocked external input on channel  $x$  inside the parent can be implemented by binding a socket to a port  $x$  inside the parent machine. The output to the parent can be implemented by connecting a socket to the IP address of the parent on port  $x$  and sending the value  $v$  over the network from  $a$  to its parent.
- (1.49) An agent  $a$  can enter a site  $b$  on channel  $x$  using TCP/IP. The blocked accept on channel  $x$  inside site  $b$  can be implemented by binding a socket to a port  $x$  inside a machine with IP address  $b$ , and waiting for a connection on port  $x$ . The enter to  $b$  can be implemented by connecting a socket to IP address  $b$  on port  $x$ , and then sending the agent  $a$  over the network in serialised form to  $b$ . The agent  $a$  can resume execution on arrival.
- (1.58) The machine can non-deterministically select a given ambient or action to be executed. This rule reflects the inherent non-determinism of distributed networks, in which parallel reductions can occur in any order.
- (1.59) The machine can execute the contents of a site independently of other sites. This rule is fundamental for distribution, since it allows sites to be independently executed on different physical machines, by different runtimes. The different sites can then interact with each other using the protocols of the underlying network.

Although the above reduction rules can be readily applied to arbitrary TCP/IP networks, the rules (1.43), (1.47) and (1.51) require additional support for network broadcast. In particular, rule (1.43) allows an external input inside a given site  $b$  to interact with a blocked sibling output inside a remote site  $a$ . By definition, the rule requires site  $b$  to poll all the sites in the network until it finds a site  $a$  with a suitable blocked sibling output. This is because an external input does not specify a particular site with which to interact, and can therefore potentially interact with any of the sites in a network. Such interactions can be readily implemented in a local area network by broadcasting



to all the sites in the network, but do not scale to wide area networks with potentially large numbers of sites. More precisely, assume  $S$  is the number of sites in a wide area network,  $N$  is the number of external inputs in  $b$  and  $R$  is the number of corresponding sibling outputs to  $b$  over a given time period. In cases where  $S \times N \gg R$  it is significantly more efficient for the external inputs in site  $b$  to block and wait for corresponding sibling outputs to  $b$  from remote sites, rather than polling all the sites in the network. This is particularly apparent on the Internet, where  $S$  is of the order of millions and  $R$  is typically of the same order as  $N$ . One way to enforce this constraint is to prevent sibling outputs to remote sites from blocking. This ensures that a given external input inside a site will always block first, and wait for a corresponding sibling output. A similar argument can be applied to the rule that allows a value to be received from a child site (1.47) and the rule that allows an agent to be accepted from a remote site (1.51).

A number of constraints can be placed on the reduction rules of the Channel Ambient Machine, in order to model the execution of mobile applications on TCP/IP networks without broadcast. The constraints avoid the use of network broadcast by ensuring that a sibling output to a site, an enter to a site and a parent output to a site are never blocked. Instead, these actions repeatedly try to interact with a corresponding blocked co-action in a remote site until a synchronisation can occur. In some cases, a synchronisation may never occur and the action will remain unblocked indefinitely. In these cases, the interval between synchronisation attempts can be increased exponentially over time in order to avoid causing a denial of service attack.

### 1.3.7 Resource Monitoring Application

This section describes how the Channel Ambient Machine can be used to execute an example application, in which a mobile agent monitors resources on a remote site.

Figure 1.5 shows how the calculus specification of Figure 1.2 can be encoded to a corresponding machine term, using the encoding function  $\llbracket P \rrbracket = P : []$ . The unguarded *client* and *server* ambients in the calculus process are encoded to unblocked *client* and *server* ambients in the corresponding machine term. All unguarded parallel compositions are encoded to list compositions. Since there are no unguarded restrictions in the calculus process, the scope of restricted names remains unchanged in the corresponding machine term. The replicated external input on the register channel is expanded according to the corresponding construction rule. For convenience, the replicated action is represented in its unexpanded form.

Once the calculus process has been encoded to a corresponding machine term, it can then be executed by the machine. Figure 1.6 uses a graphical representation of the Channel Ambient Machine to describe an execution scenario for the encoded term. The representation is based on the graphical syntax of the Channel Ambient calculus presented in Section 1.2. In addi-

Figure 1.5: Application Encoding and Execution



1. The server blocks a replicated external input on the *register* channel,

waiting to receive a value on this channel.

2. The client blocks a release on the *logout* channel, waiting to release an agent on this channel.
3. The client sends its name and an acknowledgement channel to the server on the register channel.
4. The client blocks an external input on the acknowledgement channel, waiting to receive a value on this channel.
5. After creating a globally unique *login'* channel, the server sends this channel to the client on the acknowledgement channel.
6. The server blocks an accept on the login channel, waiting to accept an agent on this channel.
7. After the client creates a *monitor* agent, the agent leaves the client on the logout channel, and the client executes the process  $Q$ .
8. The monitor agent enters the server on the login channel and then executes the process  $P$ , which monitors the resource on the server. In parallel, the server executes the process  $R$ , which forwards information about the resource to the monitor.

### 1.3.8 Correctness of the Channel Ambient Machine

This section proves the correctness of the Channel Ambient Machine with respect to the Channel Ambient calculus. Additional proof details are given in [31].

**Proving Safety to Prevent Runtime Errors** Safety ensures that the machine always produces a valid term after each execution step. This ensures that the machine does not produce any runtime errors when executing a given term. Theorem 1.3.7 (Reduction Safety) states that if the machine reduces a term  $V$  to  $V'$  then  $V'$  is a valid machine term.

**Theorem 1.3.7** (*Reduction Safety*)  $\forall V.V \in \text{CAM} \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}$

**PROOF** By induction on Definition 1.3.6 of reduction in CAM. □

$$\llbracket \nu n V \rrbracket \triangleq \nu n \llbracket V \rrbracket \quad (1.61)$$

$$\llbracket [] \rrbracket \triangleq \mathbf{0} \quad (1.62)$$

$$\llbracket \alpha.P :: C \rrbracket \triangleq \alpha.P \mid \llbracket C \rrbracket \quad (1.63)$$

$$\llbracket a[A] :: C \rrbracket \triangleq a[\llbracket A \rrbracket] \mid \llbracket C \rrbracket \quad (1.64)$$

---

**Definition 1.3.8** *Decoding CAM to CA*

---

**Proving Soundness to Ensure Valid Execution Steps** Soundness ensures that each execution step in the machine corresponds to a valid execution step in the calculus. This ensures that the machine always performs valid execution steps when executing a given term. The correspondence between machine execution and calculus execution is defined using a *decoding function*  $\llbracket V \rrbracket$ , which maps a given machine term  $V$  to a corresponding calculus process (see Definition 1.3.8). In general, the decoding function maps the null term to the null process, list construction to parallel composition, blocked machine actions to calculus actions and blocked machine ambients to calculus ambients. Restricted names, unblocked actions and unblocked ambients are preserved by the mapping.

Once a decoding function from machine terms to calculus processes has been defined in this way, it is possible to prove the soundness of the machine. Theorem 1.3.9 (Reduction Soundness) states that if the machine reduces a term  $V$  to  $V'$  then the calculus can reduce the decoding of  $V$  to the decoding of  $V'$  in at most one step.

**Theorem 1.3.9** (*Reduction Soundness*)  $\forall V.V \in \text{CAM} \wedge V \longrightarrow V' \Rightarrow \llbracket V \rrbracket \longrightarrow \llbracket V' \rrbracket \vee \llbracket V \rrbracket \equiv \llbracket V' \rrbracket$

**PROOF** By induction on Definition 1.3.6 of 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.  $\square$

**Proving Completeness to Ensure Accurate Execution** Completeness ensures that each execution step in the calculus can be matched by a corresponding sequence of execution steps in the machine, up to re-ordering of machine terms. This ensures that the machine can match all possible execution steps of the calculus when executing the encoding of a given process. The re-ordering of terms can be defined using a structural congruence relation  $V \equiv U$ , which allows a given term  $V$  to be re-ordered to match a term  $U$ . In general, the structural congruence relation allows segments of a list to be

permuted, successive restricted names to be permuted and unused restricted names to be discarded. Theorem 1.3.10 (Reduction Completeness) states that if the calculus can reduce a process  $P$  to  $P'$  then the machine can reduce the encoding of  $P$  to the encoding of  $P'$  in two steps, up to structural congruence.

**Theorem 1.3.10** (*Reduction Completeness*)  $\forall P. P \in \text{CA} \wedge P \longrightarrow P' \Rightarrow \llbracket P \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \rrbracket$

**PROOF** By induction on Definition 1.2.3 of reduction in CA □

**Proving Liveness to Prevent Deadlocks** Liveness ensures that the machine always produces a deadlock-free term after each execution step. This ensures that the machine does not deadlock when executing a given term. Intuitively, a term is deadlocked if it is unable to match a reduction of the corresponding calculus process. In practice, a term is deadlocked if it contains both a blocked action and a corresponding blocked co-action. For example, the following term contains an ambient  $a$  with a blocked sibling output to ambient  $b$  on channel  $x$ . It also contains a sibling ambient  $b$  with a blocked external input on channel  $x$ :

$$a \boxed{b \cdot x \langle n \rangle . P :: A} :: b \boxed{x^\dagger(m) . Q :: B} :: C$$

Since the sibling output and external input are both blocked they cannot interact, because there is no rule that allows an interaction between two blocked actions. In contrast, the interaction is possible in the corresponding calculus process:

$$a \boxed{b \cdot x \langle n \rangle . P \mid \llbracket A \rrbracket} \mid b \boxed{x^\dagger(m) . Q \mid \llbracket B \rrbracket} \mid \llbracket C \rrbracket \longrightarrow a \boxed{P \mid \llbracket A \rrbracket} \mid b \boxed{Q_{\{n/m\}} \mid \llbracket B \rrbracket} \mid \llbracket C \rrbracket$$

A term is also deadlocked if it contains an unblocked action or ambient inside a blocked ambient. For example, the following term contains a blocked ambient  $a$  with an internal input on channel  $x$ . Ambient  $a$  also contains an ambient  $b$  with a parent output on channel  $x$ :

$$a \boxed{b \boxed{x^\dagger \langle n \rangle . P :: B} :: x(m) . Q :: A} :: C$$

Since ambient  $a$  is blocked the parent output and external input cannot interact, because there is no rule that allows an interaction inside a blocked ambient. In contrast, the interaction is possible in the corresponding calculus process:

$$a \boxed{b \boxed{x^\dagger \langle n \rangle . P \mid \llbracket B \rrbracket} \mid x(m) . Q \mid \llbracket A \rrbracket} \mid \llbracket C \rrbracket \longrightarrow a \boxed{b \boxed{P \mid \llbracket B \rrbracket} \mid Q_{\{n/m\}} \mid \llbracket A \rrbracket} \mid \llbracket C \rrbracket$$

Conversely, a term is deadlock-free if it is able to match all reductions of the corresponding calculus process. In general, a term is deadlock-free if it does not contain both a blocked action and a corresponding blocked co-action, and if it does not contain an unblocked action or ambient inside a blocked ambient. The set of deadlock-free terms is denoted by  $\text{CAM}^\vee$  and is defined by placing constraints on the set of machine terms  $\text{CAM}$  (see Definition 1.3.11).

---


$$V ::= \nu n V \quad \text{Restriction} \quad (1.65)$$

$$\mid A \quad \text{List} \quad (1.66)$$

$$A, B, C ::= [] \quad \text{Empty} \quad (1.67)$$

$$\mid \alpha.P :: C \quad \text{Action, } \alpha.P = \underline{x(m)}.P \Rightarrow C \not\prec q \boxed{x^\dagger \langle n \rangle . P :: A} :: C' \quad (1.68)$$

$$\mid q \boxed{A} :: C \quad \text{Ambient, } q = \underline{a} \Rightarrow (A \not\prec \alpha.P :: A' \wedge A \not\prec b \boxed{B} :: A') \quad (1.69)$$

$$A \succ \overline{\text{out}} x.Q :: A' \Rightarrow A' \not\prec b \boxed{\text{out } x.P :: B} :: A''$$

$$A \succ \underline{b \cdot x \langle n \rangle} . P :: A' \Rightarrow C \not\prec b \boxed{x^\dagger \langle m \rangle . Q :: B} :: C'$$

$$A \succ \underline{x^\dagger \langle m \rangle} . P :: A' \Rightarrow C \not\prec b \boxed{a \cdot x \langle n \rangle . Q :: B} :: C'$$

$$A \succ \underline{x^\dagger \langle n \rangle} . P :: A' \Rightarrow C \not\prec x(m).Q :: C'$$

$$A \succ \underline{\text{in } b \cdot x} . P :: A' \Rightarrow C \not\prec b \boxed{\text{in } x.Q :: B} :: C'$$

$$A \succ \underline{\text{in } x} . P :: A' \Rightarrow C \not\prec b \boxed{\text{in } a \cdot x.Q :: B} :: C'$$

---

**Definition 1.3.11** *Syntax of Deadlock-Free Terms*  $\text{CAM}^\vee$

---

Once the set of deadlock-free terms has been defined in this way, it is possible to prove that the machine is deadlock-free. Lemma 1.3.12 (Deadlock-Free Reduction) ensures that reduction cannot cause a term to deadlock. The lemma states that if the machine reduces a deadlock-free term  $V$  to  $V'$  then  $V'$  is deadlock-free.

**Lemma 1.3.12** (*Deadlock-Free Reduction*)  $\forall V.V \in \text{CAM}^\vee \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}^\vee$

**PROOF** By induction on Definition 1.3.6 of reduction in  $\text{CAM}$ . □

The main property of deadlock-free terms is that they should be able match

all reductions of the corresponding calculus process. In order to prove this, it is first necessary to prove certain properties about the relationship between reduction, decoding and encoding.

Lemma 1.3.13 (Decoding Reduction) ensures that machine terms with the same decoding can perform corresponding reductions. The lemma states that if terms  $U, V$  are deadlock-free and have the same decoding, and if  $V$  can reduce to  $V'$  then  $U$  can reduce to a term that has the same decoding as  $V'$ .

**Lemma 1.3.13** (*Decoding Reduction*)  $\forall U, V, U', V' \in \text{CAM}^\vee \wedge \llbracket U \rrbracket = \llbracket V \rrbracket \wedge V \longrightarrow V' \Rightarrow \exists U'. U \longrightarrow^* U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket$

**PROOF** By induction on Definition 1.3.6 of reduction in CAM.  $\square$

Lemma 1.3.14 (Decoding Encoding) ensures that a process is structurally congruent to the decoding of its encoding.

**Lemma 1.3.14** (*Decoding Encoding*)  $\forall P. P \in \text{CA} \Rightarrow \llbracket \llbracket P \rrbracket \rrbracket \equiv P$

**PROOF** Follows by induction on the definition of construction in CAM.  $\square$

Lemma 1.3.15 (Encoding Decoding) ensures that a term and the encoding of its decoding both have the same decoding. Note that by Definition 1.3.8, if two terms have the same decoding then they are equal up to blocking of actions and ambients.

**Lemma 1.3.15** (*Encoding Decoding*)  $\forall V. V \in \text{CAM} \Rightarrow \llbracket \llbracket \llbracket V \rrbracket \rrbracket \rrbracket = \llbracket V \rrbracket$

**PROOF** By induction on Definition 1.3.8 of decoding in CAM.  $\square$

Once these properties have been proved for reduction, decoding and encoding, it is possible to prove the liveness of the machine. Theorem 1.3.16 (Liveness) ensures that the machine can match all reductions of the calculus. The theorem states that if a given term  $V$  is deadlock-free and the decoding of  $V$  can reduce to  $P'$  then  $V$  can reduce to a term  $V'$  that decodes to  $P'$ , up to structural congruence.

**Theorem 1.3.16** (*Liveness*)  $\forall V. V \in \text{CAM}^\vee \wedge \llbracket V \rrbracket \longrightarrow P' \Rightarrow \exists V'. V \longrightarrow^* V' \wedge \llbracket V' \rrbracket \equiv P'$

**PROOF** By Lemma 1.3.14 (Decoding Encoding), Lemma 1.3.15 (Encoding Decoding), Lemma 1.3.13 (Decoding Reduction) and by Theorem 1.3.10 (Reduction Completeness).  $\square$

### 1.3.9 Proving Termination to Prevent Livelocks

Termination ensures that a given machine term will always terminate, provided the corresponding calculus process also terminates. This ensures that the machine does not livelock when executing a given term. According to Theorem 1.3.17 (Termination), if the decoding of a given term  $V$  cannot reduce, then  $V$  will be unable to reduce after a finite number of steps.

**Theorem 1.3.17 (Termination)**  $\forall V \in \text{CAM}. \llbracket V \rrbracket \not\rightarrow \Rightarrow \exists V'. V \rightarrow^* V' \wedge V' \not\rightarrow$

**PROOF** The reduction rules of the machine can be divided into two types of rules: blocking rules  $V \rightarrow_b V'$  and interactive rules  $V \rightarrow_i V'$ . From the proof of Theorem 1.3.9 it can be shown that:

$$\begin{aligned} V \rightarrow_i V' &\Rightarrow \llbracket V \rrbracket \rightarrow \llbracket V' \rrbracket \\ V \rightarrow_b V' &\Rightarrow \llbracket V \rrbracket \equiv \llbracket V' \rrbracket \end{aligned}$$

Therefore, if  $\llbracket V \rrbracket \not\rightarrow$  then either  $V \not\rightarrow$  or  $V \rightarrow_b V'$ . Furthermore, if  $V \rightarrow_b V'$  then  $\llbracket V \rrbracket \equiv \llbracket V' \rrbracket$  and  $\llbracket V' \rrbracket \not\rightarrow$ . By definition, a given term can only perform a finite number of consecutive blocking reductions, since each term can only contain a finite number of actions or ambients to block. Therefore, by induction  $V$  will be unable to reduce after a finite number of steps.  $\square$

Note that the termination property does not hold in the case where actions to a remote site are prevented from blocking. This is because a given action will continue polling until a synchronisation occurs, which may be indefinitely.

---

## 1.4 The Channel Ambient Runtime

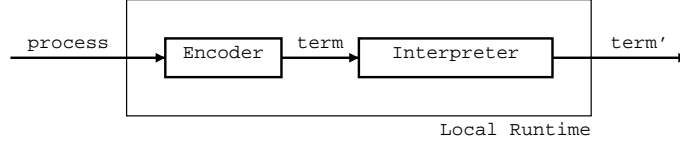
This section presents a runtime for executing processes of the Channel Ambient calculus, known as the Channel Ambient Runtime. The runtime is implemented by defining a mapping from the Channel Ambient Machine to functional program code, using the OCaml language [21].

### 1.4.1 Local Runtime Implementation

The Channel Ambient Machine can be used to implement a local runtime, which executes a given calculus process on a single physical device.

The architecture of the local runtime is described in Figure 1.7. The main components of the runtime are an *encoder* and an *interpreter*, where a given process of the Channel Ambient calculus is executed by the local runtime as





**Figure 1.7:** Architecture of the Local Runtime

follows. First, the process is encoded to a corresponding runtime term by the encoder. The resulting term is then executed by the interpreter in steps, according to a reduction relation. At each step, the interpreter transforms the initial term into an updated term. Execution continues until no more reductions are possible.

The implementation of the local runtime is almost a direct mapping from the Channel Ambient Machine to functional program code. Full details of the mapping are given in [31].

### 1.4.2 Distributed Runtime Implementation

The Channel Ambient Machine can also be used to implement a distributed runtime, which executes a given calculus process over multiple devices in a hierarchical TCP/IP network. A runtime term containing multiple sites is executed by mapping each site in the term to a separate distributed runtime. For example, the following term is executed using three separate runtimes, one for each of the sites  $s_0, s_1, s_2$ :

$$\nu \tilde{z} s_0 \boxed{S_0 :: s_1 \boxed{S_1} :: s_2 \boxed{S_2} :: []} :: []$$

The top-level restricted values are assumed to span all of the distributed runtimes and do not need to be implemented explicitly, since they form part of an implicit set of global values in the network.

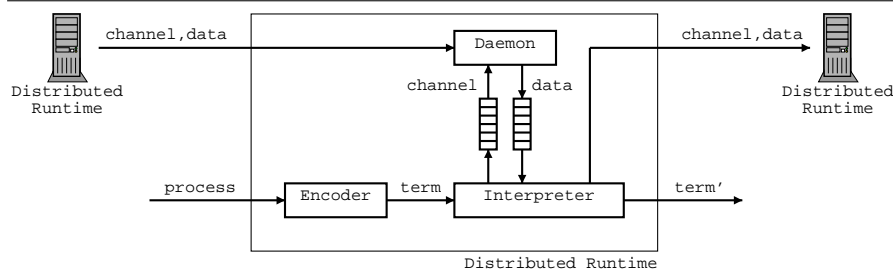
The terms inside separate distributed runtimes can interact with each other using standard network protocols. This is achieved by configuring each distributed runtime to act as a server on the address of the site it is executing. The tree structure of the network is preserved by adding a link from each site to its parent. These links are implemented by placing each site inside a *proxy* of its parent, using the local address of the parent. If no parent is specified, then a default root parent is used. For example, the above term is implemented by executing each of the following three terms on a separate distributed runtime:

$$\text{root} \boxed{s_0^{m_0} \boxed{S_0} :: []} :: [], \quad m_0 \boxed{s_1 \boxed{S_1} :: []} :: [], \quad m_0 \boxed{s_2 \boxed{S_2} :: []} :: []$$

The site  $s_0$  is placed inside a proxy of the default root parent, and each of the sites  $s_1$  and  $s_2$  is placed inside a separate proxy  $m_0$  of the site  $s_0$ . The name  $m_0$  of the proxy corresponds to the local address of  $s_0$ , which is used for receiving messages and agents from sites that are logically contained in  $s_0$ . Although each proxy site is initially empty, during the course of execution it can be used to temporarily store and execute agents that are in transit between sites. Thus, in addition to providing a link to the parent site, a proxy can also be used to decentralise the execution of the contents of the parent. For example, the following runtime term represents a site  $s$  with contents  $S$ , inside a proxy site  $m$  with child agents  $g_1, \dots, g_N$ :

$$m \left[ s \left[ S \right] :: g_1 \left[ G_1 \right] :: \dots :: g_N \left[ G_N \right] :: [] \right] :: []$$

Although the child agents  $g_1, \dots, g_N$  have already left  $s$ , they can still temporarily remain on the runtime inside the proxy  $m$  while in transit to their next destination. If one of the agents needs to perform a local interaction inside the parent then it will be moved to the parent on demand. Otherwise, it will remain inside the proxy until it moves to its next destination.



**Figure 1.8:** Architecture of the Distributed Runtime

The architecture of the distributed runtime is described in Figure 1.8. The main components of the runtime are an *encoder*, an *interpreter*, a *daemon* and shared *data* and *channel buffers*. A given process of the Channel Ambient calculus is executed by the distributed runtime as follows:

1. First, the process is encoded to a corresponding runtime term by the encoder. The process is assumed to be of the form  $m \left[ s \left[ P \right] \right]$ , where  $s$  is the main site to be executed by the runtime,  $P$  is contents of the site and  $m$  is a proxy of the parent site. The process is encoded to a runtime term of the form  $m \left[ s \left[ P : [] \right] :: [] \right] :: []$

2. The resulting term is then executed by the interpreter in steps, according to a reduction relation. At each step, the interpreter transforms the initial term into an updated term.
3. Since the interpreter is single-threaded, a separate daemon is needed in order to receive data from distributed runtimes. The data can be either a message received by the daemon or a mobile agent accepted by the daemon over a given channel. Any data received by the daemon is added to the data buffer. After each execution step, the interpreter checks this buffer for incoming data, which is added to the site  $s$ .
4. During execution, the interpreter can instruct the daemon to receive data on a given channel. The type of data that can be received is determined by the type of the channel, where different channel types are used to distinguish between incoming messages and agents. The interpreter can also send data to distributed runtimes directly over a given channel.
5. Execution continues until no more reductions are possible, after which the interpreter goes into a blocked state, waiting for an interrupt from the data buffer to signal the arrival of new data. As soon as any data arrives it is added to the site  $s$ , allowing the interpreter to resume executing.

The reduction rules of the Channel Ambient Machine are implemented in a distributed setting by making a distinction between sites and agents, as described in Section 1.2, and by taking into account the constraints described in Section 1.3. Although the changes required to go from the local reduction rules to the distributed reduction rules are relatively minor, the corresponding changes in the implementation are more substantial. In particular, the Channel Ambient Machine defines a single reduction rule for performing a sibling output, parent output or enter to an ambient, but the distributed implementation needs to distinguish between performing these actions to a remote site or to a local agent. Similarly, a single reduction rule is defined for blocking an external input or accept inside an ambient, but the distributed implementation needs to distinguish between blocking these actions inside a remote site or inside a local agent. As a result, additional functions are required to enable distributed runtimes to interact over a network. Full details are provided in [31].

### 1.4.3 Enhanced Runtime Implementation

The distributed runtime has been enhanced in a number of ways in order to improve the efficiency of process execution, while conserving network bandwidth:

- The runtime terms are implemented using a map data structure instead of a list, in order to improve the efficiency of lookups when searching for a blocked co-action.
- A deterministic selection algorithm is used in order to improve the efficiency of runtime execution, while ensuring a suitable notion of fairness.
- Network masks are used to conserve bandwidth, by preventing spurious attempts to interact with inaccessible network addresses.

Full details of the optimisations are described in [31].

An indication of runtime performance can be obtained by characterising the efficiency of the enhanced runtime execution algorithm. Essentially, each site in a distributed application is executed by a separate runtime, which can be stopped and started independently. This allows a large number of parallel runtimes to be used in a given application with low initialisation overhead. Each individual runtime executes in a loop, as described in Section 1.4.2, where the state  $A$  of the runtime consists of a queue of active actions, a queue of blocked actions and a queue of ambients, recursively:

$$\alpha_1.P_1 :: \dots :: \alpha_i.P_i, \alpha_j.P_j :: \dots :: \alpha_k.P_k, a_1 \boxed{A_1} :: \dots :: a_N \boxed{A_N}$$

Each blocked or active action corresponds to a separate, self-contained thread, allowing a large number of threads to be executed in parallel with low overhead. The runtime scheduling algorithm simply picks the first action in the active queue, which takes constant time, and then checks whether there is a corresponding blocked co-action. The check will depend on the type of co-action that is required. If the co-action is in a separate runtime, the action is either blocked immediately or sent over the network to the runtime. If the co-action is in the same runtime it will either be inside the same ambient, the parent, a child, or a sibling. For a co-action inside the same ambient or inside the parent, the lookup time is  $O(N)$ , where  $N$  is the number of blocked co-actions inside the ambient. In practice, the machine state is further refined by grouping blocked co-actions of the same type (e.g. all the in a x actions to a given ambient  $a$  on a given channel  $x$ ) into a map data structure, so that the lookup time is  $O(\log N)$ , where  $N$  is the number of different types of blocked co-actions inside the ambient. For co-actions inside a sibling or child ambient, the lookup time is  $O(N_A \times \log N)$ , where  $N_A$  is the number of sibling or child ambients. This gives us an upper bound on the cost of computing a given interaction. If no co-action is found then the chosen action is blocked. If no actions are left in the active queue then an action inside the first active ambient in the ambient queue is chosen, recursively. In the worst case, the time to schedule an action will be  $O(N_T)$ , where  $N_T$  is the total number of active ambients in the runtime. If no active actions are found anywhere inside an ambient or its children then the ambient is blocked, allowing the ambient to be skipped in future scheduling. If all the actions or ambients in a runtime






are blocked, then the entire runtime goes into a blocked state, waiting for messages from the network. This combination of lookup and scheduling will determine the cost of executing a given program, and will vary depending on the number of ambients and actions in the program. As an example, the full agent tracker application in Section 1.5 took 0.136s when all the sites were executed as local agents on the same runtime. This includes the disk I/O time for writing the initial and final states of the runtime to separate html files. When each site is executed on a separate runtime, the execution time is dominated by the properties of the underlying network.

#### 1.4.4 The Channel Ambient Language

The Channel Ambient Calculus has been used as the basis for a programming language for mobile applications, known as the Channel Ambient Language (CAL). The language extends the calculus with various programming constructs such as data structures, process definitions and system calls, and the execution rules of the language are based largely on the reduction rules of the Channel Ambient Machine. Although the language itself is merely a prototype, it gives a useful indication of how next-generation programming languages for mobile applications can be based on a formal model. The language and runtime system are available from [30].

**Execution** The Channel Ambient Runtime can be used to execute a file *source.ca* by typing the command *cam.exe source.ca* in a console. The file *source.ca* contains the code for a single site with a given network address. Before the runtime executes the program, it checks to see whether the program code is well-typed. The type system for the Channel Ambient Language is defined in [31] and ensures that only values of the correct type can be sent and received over channels. When the runtime executes a given program, its internal state is modified with each execution step, according to the reduction rules of the Channel Ambient Machine.

The execution state of the runtime is regularly streamed to the file *state.html*, which can be viewed in a web browser and periodically refreshed to display the latest state information. For reference, the initial state of the runtime is stored in the file *start.html*.

The state of the runtime looks very much like a source file, and contains any program code that is currently being executed by the runtime. Each currently executing thread is displayed next to a *thread* icon , where icons of blocked threads are underlined . Each currently executing ambient is enclosed in a box with an *open* folder icon  next to the ambient's name, where icons of blocked ambients are underlined . The state of the ambient can be hidden by clicking on this icon, which collapses the contents of the ambient and displays a *closed* folder icon  next to the ambient's name. If

the entire contents of the runtime are blocked then execution is suspended until the runtime receives an interrupt from the network, announcing the arrival of new messages or ambients to be executed.

**System Calls** A given runtime executes a single site, where the top-level process inside the site is specified by the user, but new agents can enter and leave the site dynamically. The security model only allows the top-level process inside the site to perform a system call, by doing a parent output on one of the pre-defined *System* channels. These include channels for printing on a console, reading and writing to files, executing external programs and forwarding messages to remote sites. If a parent output on a *System* channel is performed inside a nested agent then it is treated as an ordinary parent output. These parent outputs can be forwarded to the parent site through the use of private forwarding channels. Where necessary, a separate forwarding channel can be defined for each agent or group of agents, allowing fine-grained access permissions to be implemented for each system call. If no forwarding channels are implemented, then none of the agents inside a site will be able to perform a system call.

The runtime also allows files to be stored in an ambient in binary form and sent over channels like ordinary values. 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.

#### 1.4.5 Resource Monitoring Application

The Channel Ambient language can be used to program the resource monitoring example of Section 1.2. The code for the client and server is given in Figure 1.9. The syntax of the code is similar to the syntax of the calculus, 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. 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 server runs on port 3001 of IP address 192.168.0.2, while the client runs on port 3000 of IP address 192.168.0.3. In this example the *Report*, *Monitor* and *Continue* processes are defined as simple outputs. In general they can be defined as arbitrarily complex processes.

---

```

192.168.0.3:3000                                192.168.0.2:3001
[ let server = 192.168.0.2:3001 in                [ let Report() = print<report> in
let client = 192.168.0.3:3000 in                    let register =
let register =                                     register:<site,<<migrate>>> in
    register:<site,<<migrate>>> in                    !register^(c,k);
let logout = logout:<migrate> in                      new login:<migrate>
let Monitor() = print<monitor> in                    c.k<login>;
let Continue() = print<continue> in                  -in login;
( new ack:<<migrate>>                                Report<> ]
  server.register<client,ack>;
  ack^(x);
  monitor
  [ out logout;
    in server.x;
    Monitor<>
  ]
| -out logout;
  Continue<>
) ]

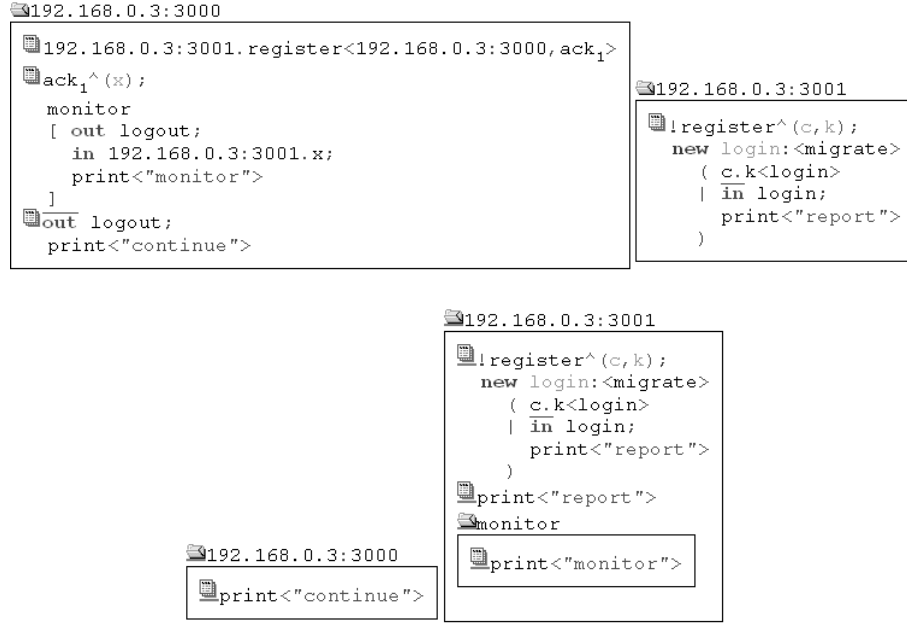
```

---

**Figure 1.9:** Program code for the Resource Monitoring Application

---

The two programs are executed on separate client and server sites by separate runtimes. Initially, each runtime parses the program code and substitutes any value, type or process definitions. Top-level parallel compositions are expanded accordingly and the client creates a new acknowledgement channel, where  $ack_1$  is an abbreviation for a private channel name. All occurrences of the name  $ack$  are substituted with the  $ack_1$  inside the client. The initial internal state of the client and server runtimes are shown in Figure 1.10 (top). These states correspond to HTML output that is automatically generated by the runtimes after each execution step. The final state of the client and server runtimes are shown in Figure 1.10 (bottom). The monitor agent has entered the server on the login channel and executes a process to monitor the resource on the server. In parallel, the server executes a process to forward information about the resource to the monitor. Further examples can be downloaded from [30], along with debugging information about the initial and final states of the runtime.



**Figure 1.10:** Initial (top) and final (bottom) states for the server and client defined in Figure 1.9

## 1.5 Agent Tracker Application

The Channel Ambient Language can be used to develop an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network. The application is inspired by previous work on location-independence, studied in the context of the Nomadic  $\pi$ -calculus [41] and the Nomadic Pict programming language [42]. Algorithms for reliably tracking the location of agents are fundamental in many distributed applications. One example is in the area of information retrieval, where multiple agents visit specialised data repositories to perform computation-intensive searches, periodically communicating with each other to update their search criteria based on high-level goals. A simple example of an information retrieval application is the organisation of a conference using mobile agents. Each agent is given a dedicated task, such as flight and train reservations, hotel reservations, sightseeing tours, restaurant bookings etc. The various agents then move between dedicated sites in order to achieve



---

$Server(s_i) \triangleq$   
 $!register(client, ack). \nu tracker \nu send \nu move \nu deliver \nu lock$   
 $(tracker \boxed{Tracker(client, send, move, deliver, lock)})$   
 $| client/ack(tracker, send, move, deliver, lock)$   
 $| tracker/lock\langle s_i \rangle$

$Tracker(client, send, move, deliver, lock) \triangleq$   
 $(!send^\uparrow(x, m).lock^\uparrow(s).fwd^\uparrow\langle s, client, deliver, (s, x, m) \rangle)$   
 $| !move^\uparrow(s').s'^\uparrow\langle \rangle.lock^\uparrow(s).fwd^\uparrow\langle s, client, lock, s' \rangle)$

$Site() \triangleq$   
 $(!child^\uparrow(a, x, m).a/x\langle m \rangle)$   
 $| !fwd(s, a, x, m).s \cdot child\langle a, x, m \rangle$   
 $| !\overline{in} login | !\overline{out} logout$   
 $| !s_0() | \dots | !s_N()$

$Client(home, tracker, deliver, lock) \triangleq$   
 $(!lock^\uparrow(s).out logout.in s \cdot login.fwd^\uparrow\langle home, tracker, lock, s \rangle.moved\langle s \rangle)$   
 $| !deliver^\uparrow(s, x, m).fwd^\uparrow\langle home, tracker, lock, s \rangle.x\langle m \rangle)$

```

let Server(home:site)=
!register(client,ack); new tracker:agent  new move:<site>
  new deliver:<site,<'a>,'a>  new lock:<site>
  ( tracker[ Tracker<client,move,deliver,lock> ]
  | client/ack<tracker,move,deliver,lock>
  | tracker/lock<home> )

let Tracker(client:agent,move:<site>,deliver:<site,<'a>,'a>,lock:<site>)=
( !send^(x,m); lock^(s); forward^<s,client,deliver,(s,x,m)>
| !move^(s1); s1^<>; lock^(s); forward^<s,client,lock,s1>)

let Site()=
( !child^(a,x,m); a/x<m>
| !forward(s,a,x,m); s.child<a,x,m>
| !-in login | !-out logout)

let Client(home:site,tracker:agent,deliver:<site,<'a>,'a>,lock:<site>)=
( !lock^(s);out logout;in s.login;forward^<home,tracker,lock,s>;moved<s>
| !deliver^(s,x,m); forward^<home,tracker,lock,s>; x<m>)

```

**Figure 1.11:** Agent Tracker Specification (top) and Implementation (bottom)

---

their specific goals, periodically communicating with each other to resolve conflicts in scheduling or availability. In general, algorithms for tracking the location of migrating agents are a key feature of mobile agent platforms, and are part of the Mobile Agent System Interoperability Facility (MASIF) standard for mobile agent systems [35]. Many agent platforms rely on tracking algorithms to forward messages between migrating agents, including for example the JoCaml system. In this respect, the Channel Ambient Language can be used to develop secure, extensible platforms for mobile agents, using tracker algorithms similar to the one presented.

This section describes a decentralised version of the agent tracker algorithm given in [40]. 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 agent tracker application is specified using the *Server*, *Tracker*, *Site* and *Client* processes defined in Figure 1.11. The corresponding program code for these definitions is also presented. The *Site* process describes the services provided by each trusted site in the network. An agent at a trusted site can receive a message  $m$  on channel  $x$  from a remote agent via the *child* channel. Likewise, it can forward a message  $m$  on channel  $x$  to a remote agent  $a$  at a site  $s$  via the *fwd* channel. Visiting agents can enter and leave a trusted site via the *login* and *logout* channels respectively. An agent at a trusted site can check whether a given site  $s$  is known to be trusted by sending an output on channel  $s$ . The *Server* process describes the behaviour of a home server that keeps track of the location of multiple clients in the network. A *client* agent can register with a server site via the *register* channel, which creates a new *tracker* agent to keep track of the location of the client. The *Tracker* and *Client* processes describe the services provided by the tracker and client agents, respectively.

Figure 1.12 describes a scenario in which a client registers with its home site. The scenario uses a simplified version of the graphical representation for the Channel Ambient calculus presented in Section 1.2, 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. 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 agent 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 agent  $t_c$  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  $P$ .

Figure 1.13 describes a scenario in which a tracker agent sends a message

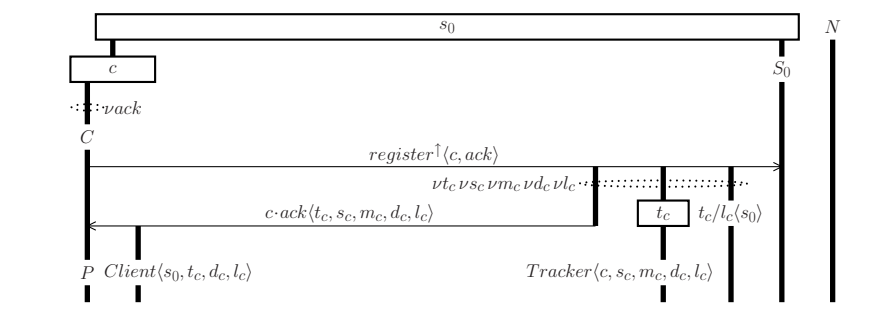


Figure 1.12: Tracker Registration

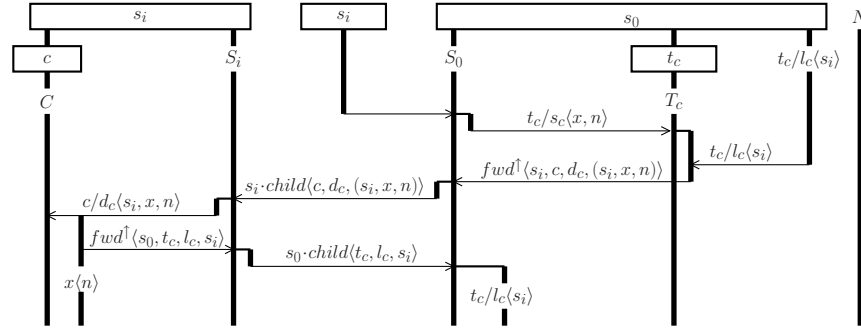


Figure 1.13: Tracker Delivery

to its client. A message can be sent to the client by sending a request to its corresponding tracker agent on the home site  $s_0$ . The request is sent to the tracker agent  $t_c$  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 client. When the client receives the request, it forwards its current location to the tracker on the lock channel, thereby releasing the lock, and 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 agent on the move channel  $m_c$ . The tracker agent first checks whether this site is trusted by trying to send an output on channel  $s_j$ . If the output succeeds, the tracker 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 agent on the lock channel, thereby releasing the lock.

The above definitions can be used to specify a distributed instance of the tracker application, in which multiple client agents communicate with each other as they move between trusted sites in a network. The following specification describes three client agents  $a, b, c$  which start out on a home site  $s_0$ .

$$s_0 \left[ \text{Server}\langle s_0 \rangle \mid \text{Site}\langle \rangle \mid a \boxed{A} \mid b \boxed{B} \mid c \boxed{C} \right] \mid s_1 \boxed{\text{Site}\langle \rangle} \mid s_2 \boxed{\text{Site}\langle \rangle} \mid s_3 \boxed{\text{Site}\langle \rangle}$$

The processes  $A, B, C$  are used to describe how the clients register with the home site  $s_0$ , locally exchange tracker names and then embark on their respective journeys through the network. The clients are able to communicate with each other as they move between the trusted sites  $s_0, \dots, s_4$  by sending messages to their respective tracker agents on the home site  $s_0$ . The trackers then forward the messages to the appropriate client. This example was programmed in the Channel Ambient Language by executing the following four sites on four different runtimes:

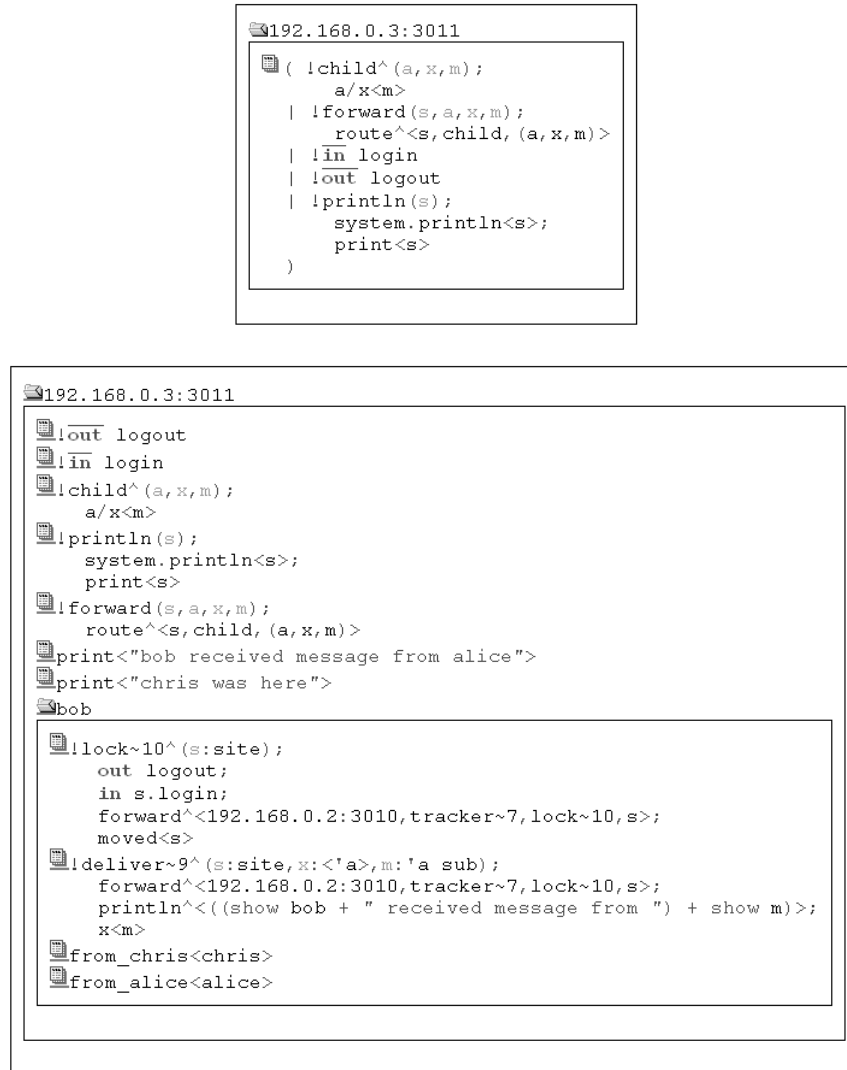
```
192.168.0.2:3010
[ Site<> | Server<192.168.0.2:3010>
  | alice[Alice<>] | bob[Bob<>] | chris[Chris<>]
]
192.168.0.3:3011[ Site<> ]
192.168.0.4:3012[ Site<> ]
192.168.0.5:3013[ Site<> ]
```

The full code for the application is available online at [30]. During execution, each runtime periodically produced an html output of its execution state. The state of site  $s_1$  at the beginning and end of execution is shown in Figure 1.14. Initially, the site executes code to provide a number of basic services, but does not contain any agents. By the end of the program execution, the site has been visited by both the *bob* and *chris* agents, and the bob agent is still present on the site. Multiple runs of the application show that all of the messages are reliably delivered to the respective agents as they move around between sites.

---

## 1.6 Related Work

The Channel Ambient calculus is inspired by previous work on calculi for mobility, including the Ambient calculus, the Nomadic  $\pi$ -calculus and variants of the Boxed Ambient calculus. The main differences with Boxed Ambients



**Figure 1.14:** Initial and final Runtime states for site  $s_1$  at address 192.168.0.3:3011

are that ambients in CA can interact using named channels and that sibling ambients can communicate directly. Sibling communication over channels is inspired by the Nomadic  $\pi$ -calculus and channel communication is also used in the Seal calculus [11], although sibling seals cannot communicate directly. The use of channels for mobility is inspired by the mechanism of passwords, first introduced in [24] and subsequently adopted in [7]. The main advantage of CA over the Nomadic  $\pi$ -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  $\pi$ . In the Safe Ambient calculus [22], co-actions are used to allow an ambient to enter, leave or open another ambient. However, there is no way of controlling which ambients are allowed to perform the corresponding actions. In [41] the authors of Nomadic  $\pi$  argue that some form of local synchronous communication is fundamental for programming mobile applications. In contrast, Boxed Ambient calculi typically require all communication between sibling agents to take place via the parent. However, there is nothing to prevent one or both of these agents from migrating while the message is still in transit, leaving the undelivered message stuck inside the parent.

The Channel Ambient Machine is inspired by the Pict abstract machine [38], which is used as a basis for implementing the asynchronous  $\pi$ -calculus. Like the Pict machine, CAM uses a list syntax to represent the parallel composition of processes. In addition, CAM extends the 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, which allows mobile ambients in CAM to re-bind to new environments. In addition, CAM requires an explicit notion of restriction in order to manage the scope of names across ambient boundaries. The Pict machine does not require such a notion of restriction, since all names in Pict are local to a single machine. By definition, the Pict abstract machine is deterministic and, although it is sound with respect to the  $\pi$ -calculus, it is not complete. In contrast, CAM is non-deterministic and is both sound and complete with respect to the Channel Ambient calculus.

A number of abstract machines have also been defined for variants of the Ambient calculus. In [8] an informal abstract machine for Ambients is presented, which has not been proved sound or complete. In [18] a distributed abstract machine for Ambients is described, based on a formal mapping from the Ambient Calculus to the Distributed Join calculus. However, it is not clear how such a translation can be applied to the Channel Ambient calculus, which uses more high-level communication primitives. In addition, the translation is tied to a particular implementation language (JoCaml), whereas the Channel Ambient Machine uses more a low-level approach that can be implemented in any language with support for function definitions. Furthermore,

the abstract machine in [18] separates the logical structure of ambients from the physical structure of the network, whereas the Channel Ambient Machine uses ambients to directly model the hierarchical topology of the network. This approach assumes that the network topology is part of the application specification, which leads to a simplified implementation. In [36] 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 Channel Ambients, where the *open* primitive is non-existent.

The Nomadic Pict runtime [42] is an extension of the Pict runtime with support for the migration of mobile agents between runtimes over a network. Unlike the Channel Ambient Runtime, the Nomadic Pict runtime is not based on a formal abstract machine, although such a machine could in principle be defined using the approach described in Section 1.3. Another feature of the Nomadic Pict runtime is that all the agents in a given application are typically launched from a single runtime, and then migrated to other runtimes in the network. This is because the constructs for agent creation require each agent to be created with a unique identity that is only known within a limited scope.

The JoCaml runtime [15] is implemented based on a high-level abstract machine for the Join calculus [17] and a compilation of the calculus to a suitable target language [20]. In JoCaml, migrating agents communicate by sending messages to each other irrespective of their location, which requires complete network transparency. Therefore, in order to fully implement the semantics of the Join calculus, sophisticated distributed algorithms are required, such as the information retrieval infrastructure described in this chapter. In general, the JoCaml runtime relies on a small set of powerful primitives that provide complete network transparency, whereas the Channel Ambient runtime relies on a small set of low-level primitives that can be directly implemented above standard network protocols, together with an easy way of encoding more expressive primitives on top.

In [41] Nomadic Pict is used to program an infrastructure for reliably forwarding messages to mobile agents, and a centralised version of this algorithm is proved correct in [40]. This chapter shows how similar applications can be programmed using the Ambient paradigm. One of the advantages of Ambients is that they can directly model computation within nested locations, which is not possible in Nomadic Pict. In addition, the Channel Ambient Language provides constructs for regulating access to a location by means of named channels, whereas Nomadic Pict assumes that all locations are freely accessible.

## 1.7 Conclusion

The agent tracker application for information retrieval relies on distributed algorithms previously presented in [41], which also describes a variety of algorithms for fault tolerance, load-balancing, large-scale parallel computation and event-driven mobility. In future, some or all of these algorithms could be specified in the Channel Ambient calculus and used to implement a range of mobile applications in the Channel Ambient Language.

From a security perspective, perhaps the most interesting area for future work lies in the use of Ambient Logics for reasoning about the security properties of mobile applications. Such logics are a powerful tool that can express a much broader range of properties than process equivalences [9]. In general, there has been a significant amount of research on security mechanisms for Ambient calculi. Although only a fraction of this research has been applied to the Channel Ambient calculus, much more could be applied in future. For the time being, this chapter focuses on ensuring that, if a given security property holds for the calculus specification of an application, then it will also hold for its implementation.

From an implementation perspective, a number of improvements can be made to both the Channel Ambient Runtime and the Channel Ambient Language. Thanks to the completeness of the Channel Ambient Machine, a number of optimisations can be introduced for the scheduling of processes, while still preserving the correctness of the runtime. The Channel Ambient Language is merely a prototype, and a number of extensions to the language can be envisaged. The Pict language demonstrated how high-level encodings of functions, procedures and data structures could be readily embedded in a  $\pi$ -calculus language. Since the Channel Ambient calculus is an extension of the  $\pi$ -calculus, a full embedding of a functional language can be incorporated into the Channel Ambient language in a similar way.

From a specification perspective, a number of extensions to the calculus can also be envisaged, including a notion of time and non-deterministic choice. A promising approach to modelling time in the  $\pi$ -calculus is presented in [2], which can also be readily applied to Ambient calculi. A possible extension for modelling non-deterministic choice is described in the Bio Ambient calculus [34]. The combination of these two extensions could be readily used to program timeouts and other exception mechanisms. For example,  $ack^\dagger(k).P + \tau_n.Q$  could be used to model a choice (+) between receiving an acknowledgement  $k$  on channel  $ack$  and then executing  $P$ , or waiting for time  $n$  and then executing  $Q$ . In this example, if time  $n$  elapses before the acknowledgement is received, the process  $Q$  is executed. Such an extension would be relatively straightforward – for comparison, a notion of stochastic choice with timeout has already been implemented in [32]. So far, the Channel Ambient calculus has been used to specify mobile applications for networks that



support the TCP/IP protocol. In principle, the calculus could also be used to specify applications on a range of networks types. For example, wireless networks with mobile devices can be readily modelled using the synchronisation primitives of the calculus. Indeed, mobile wireless networks were some of the first network types to be targeted by process calculi. For example, [25] uses the  $\pi$ -calculus to describe a simple protocol for switching mobile phones between base stations.

This chapter investigates to what extent a variant of the Ambient calculus can be used for specifying and implementing secure mobile applications. In particular, the chapter investigates whether Ambients can be used as the basis for a distributed, mobile programming language. Over the years there has been substantial theoretical research on the Ambient calculus and its many variants, but there has been comparatively little research on the implementation of Ambients. In this regard, a number of lessons can be learned from the Nomadic  $\pi$ -calculus, for which the underlying theory [39] and implementation [41] were developed in concert, in order to produce the beginnings of a programming language and runtime system for developing real mobile applications [42]. Ambient calculi have matured substantially over the years, and it is perhaps time to seriously address how the theory of Ambients can be turned into practice, in order to reap the benefits of the last decade of theoretical research. This follows on from one of the early papers on Ambients [10], which states: “On this foundation, we can envision new programming methodologies, programming libraries and programming languages for global computation”.



---

## Bibliography

- [1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999.
- [2] Martin Berger. Basic theory of reduction congruence for two timed asynchronous  $\pi$ -calculi. In *CONCUR'04*, volume 3170 of *LNCS*, pages 115–130. Springer, January 2004.
- [3] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Xklaim and klava: Programming mobile code. In Marina Lenisa and Marino Miculan, editors, *ENTCS*, volume 62. Elsevier, 2002.
- [4] Karthikeyan Bhargavan, Cedric Fournet, and Andrew D. Gordon. A semantics for web services authentication. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–209, New York, NY, USA, 2004. ACM Press.
- [5] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01*, number 2215 in *LNCS*, pages 38–63. Springer, 2001.
- [6] 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.
- [7] 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.
- [8] Luca Cardelli. Mobile ambient synchronization. Technical Report SRC-TN-1997-013, Digital Equipment Corporation, July 25 1997.
- [9] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of POPL '00*, pages 365–377. ACM, January 2000.
- [10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *FoSSaCS '98*: 140–155.
- [11] Giuseppe Castagna, Jan Vitek, and Francesco Zappa. The seal calculus. 2003. Available from <ftp://ftp.di.ens.fr/pub/users/castagna/seal.ps.gz>.

- [12] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), Yorktown Heights, New York, 1994.
- [13] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [14] K. G. Coffman and Andrew M. Odlyzko. Internet growth: Is there a "moore's law" for data traffic? In Panos M. Pardalos and Mauricio Resende, editors, *Handbook of Massive Data Sets*, pages 47–93. Kluwer, 2002.
- [15] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.
- [16] S. Crafa, M. Bugliesi, and G. Castagna. Information flow security for boxed ambients. In *F-WAN'02*, number 66(3) in ENTCS, 2002.
- [17] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, January 1996.
- [18] Cédric Fournet, Jean-Jacques Lévy, and Alain Schmitt. An asynchronous distributed implementation of mobile ambients. In J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors, *Proceedings of TCS 2000*, volume 1872 of *LNCS*, pages 348–364. IFIP, Springer, August 2000.
- [19] ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 1996.
- [20] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
- [21] X. Leroy, D. Doligez, J. Garrigue, D. Remy, and J. Vouillon. The Objective Caml system, release 3.08, Documentation and user's manual. INRIA, <http://caml.inria.fr/ocaml/>, 2004.
- [22] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*. ACM Press, 2000.
- [23] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *CONCUR'02*, volume 2421 of *LNCS*, pages 304–320. Springer, 2002.
- [24] Massimo Merro and Matthew Hennessy. Bisimulation congruences in safe ambients. In *POPL'02*, pages 71–80. ACM Press, 2002.

- [25] Robin Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. Cambridge University Press, May 1999.
- [26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.
- [27] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [28] Andrew M. Odlyzko and K. G. Coffman. Growth of the internet. In T. Li and I. P. Kaminow, editors, *Optical Fiber Telecommunications IV B: Systems and Impairments*, pages 17–56. Academic Press, 2002.
- [29] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Elsevier, 4 edition, 2007.
- [30] Andrew Phillips. *The Channel Ambient System*, 2005. Runtime and documentation available from <http://research.microsoft.com/~aphillip/cam/>.
- [31] Andrew Phillips. *Specifying and Implementing Secure Mobile Applications in the Channel Ambient System*. PhD thesis, Imperial College London, April 2006.
- [32] Andrew Phillips and Luca Cardelli. Efficient, correct simulation of biological processes in the stochastic pi-calculus. In *Computational Methods in Systems Biology*, volume 4695 of *LNCS*, pages 184–199. Springer, September 2007.
- [33] Andrew Phillips, Nobuko Yoshida, and Susan Eisenbach. A distributed abstract machine for boxed ambient calculi. In *ESOP'04*, *LNCS*. Springer, April 2004.
- [34] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Shapiro. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, September 2004.
- [35] Dejan S. Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. MASIF: The OMG Mobile Agent System Interoperability Facility. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA'98), Stuttgart (Germany), September 1998*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer-Verlag, 1999.
- [36] D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. In *ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.

- [37] Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location independence for mobile agents. In *ICCL'98*.
- [38] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis.
- [39] Asis Unyapoth. Nomadic  $\pi$ -calculi: Expressing and verifying communication infrastructure for mobile computation. Technical Report UCAM-CL-TR-514, University of Cambridge, Computer Laboratory, June 2001.
- [40] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *POPL'01*, pages 116–127, 2001.
- [41] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, June 2000.
- [42] Paweł T. Wojciechowski. *The Nomadic Pict System*, 2000. Available electronically as part of the Nomadic Pict distribution.