

University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Specifying and Implementing Secure Mobile Applications in the Channel Ambient System

Andrew Nicholas John Brojer Phillips

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, October 2005

Abstract

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. One could argue that the development of such applications requires a rigorous means of describing and reasoning about mobile computation, through the use of an appropriate model.

Foundational research by Cardelli and Gordon on the Ambient Calculus has shown that process calculi are a promising approach to modelling mobile computation. This thesis builds on more recent research in the field of process calculi, and presents a new model of computation known as the Channel Ambient calculus, which can be used both to specify mobile applications and to reason about their security properties. The primitives of the model were developed with real-world applications in mind, and are designed to be at a level of abstraction suitable for an application programmer.

The thesis also bridges a gap between theory and implementation by defining a distributed abstract machine for the Channel Ambient calculus. The abstract machine uses a list semantics, which is close to an implementation language, and a blocking semantics, which leads to an efficient implementation. The machine is proved sound and complete with respect to the underlying calculus. A prototype implementation is also described, together with an application for tracking the location of migrating ambients. The correctness of the machine ensures that the work done in specifying and analysing mobile applications is not lost during their implementation.

To Sandra

Acknowledgements

I would like to thank my supervisors, Susan Eisenbach and Nobuko Yoshida, for their dedication and support during my time at Imperial and throughout the writing of this thesis. Both have shown tremendous patience in reading and evaluating my work, and have been an invaluable source of encouragement, providing the guidance and advice I needed to bring this research to fruition. Susan's expertise in distributed programming languages was instrumental in building the foundations for this research, allowing me to identify and implement programming abstractions for mobile applications. I also benefited from discussions with the SLURP programming group, which helped to place this research in the broader context of modern programming languages for distributed computing. Nobuko's expertise in process calculi was essential in developing the theoretical foundations of this research, and I am grateful for her patience in teaching me the theoretical background I needed to design and implement a calculus for mobile computation. I also benefited from numerous discussions with the members of the Concurrency Group, which provided an ideal environment for exchanging ideas and exploring the technical challenges of process calculi. I am particularly grateful to Sergio Maffei and Maria Vigliotti, not only for lively discussion and sound technical advice, but also for their friendship and support.

I would also like to thank my co-supervisor, Bashar Nuseibeh, without whom none of this research would have been possible. His expertise in the area of Software Engineering provided an ideal background for studying the design and implementation of mobile applications. I also benefited from discussions with the Requirements Engineering group, which helped to crystallise some of the challenges faced by modern software developers and to identify potential areas for research.

I am particularly grateful to my external examiners, Andrew Gordon and Julian Rathke, for their thorough examination of this thesis and their invaluable suggestions for improvement. I am also grateful to my advisers, Luca Cardelli and Stephen Emmott at Microsoft Research Cambridge, for giving me the time I needed to complete this thesis.

During my time at Imperial I was fortunate to work in an ideal environment. I am very grateful to my colleagues and friends in the Department of Computing, who helped to make my teaching

and research activities that much more enjoyable. I have particularly fond memories of organising weekly PhD cake meetings with Xiang Feng, which provided an ideal setting for discussing and exchanging ideas.

My interest in process calculi was initially sparked by a research project at Cambridge University, under the supervision of Peter Sewell. His pragmatic views on the delicate balance between theory and practice were an inspiration for my research. During this time, I also had the privilege of working with Pawel Wojciechowski, who shared with me his experience in developing the Nomadic Pict system, and provided invaluable insight on the design and implementation of process calculi.

Throughout my time in the UK, my parents John and Mary, and my dearest sister Rosemary have always been near to my heart, encouraging me every step of the way. I am also grateful to friends and extended family for their support. Last but not least, I would like to thank my wife, Sandra, who has been with me from the very beginning of this work, and whose love and kindness have helped me reach this far.

This research was supported by an ORS scholarship, and also by a research studentship from the Department of Computing, Imperial College.

Declaration

This thesis conforms to the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College.

The text of this thesis does not exceed 100,000 words, excluding the bibliography and appendices.

The research presented in this thesis has not been submitted for any degree or comparable award of this or any other university or institution.

This thesis constitutes my own work, and is not the outcome of work done in collaboration. The thesis was supervised by Susan Eisenbach, Nobuko Yoshida and Bashar Nuseibeh. Some of the research presented in this thesis was previously published in the following article, of which I was the main author:

Andrew Phillips, Nobuko Yoshida, and Susan Eisenbach. *A distributed abstract machine for boxed ambient calculi*. In proceedings of the European Symposium on Programming, LNCS. Springer, 2004.

Contents

1	Introduction	14
1.1	Problem Description	14
1.1.1	Specifying and Implementing Secure Mobile Applications	14
1.1.2	Example Mobile Applications	14
1.1.3	Security Properties of Mobile Applications	17
1.1.4	Implementing Mobile Applications	18
1.1.5	Specifying Mobile Applications	18
1.2	Related Work	19
1.2.1	Process Calculi for Mobile Applications	19
1.2.2	The π -calculus	20
1.2.3	The Nomadic π -calculus	25
1.2.4	The Ambient Calculus	29
1.2.5	The Boxed Ambient Calculus	33
1.3	Thesis Outline	34
1.3.1	Thesis Statement	34
1.3.2	Thesis Contribution	34
1.3.3	The Channel Ambient Calculus	34
1.3.4	The Channel Ambient Machine	35
1.3.5	The Channel Ambient Runtime	35
1.3.6	The Channel Ambient Language	36
2	The Channel Ambient Calculus	37
2.1	Introduction	37
2.2	Design Principles	38
2.2.1	Components of Mobile Applications	39
2.2.2	Interaction Between Mobile Application Components	40
2.2.3	Communication Between Mobile Application Components	43

2.2.4	Migration of Mobile Application Components	44
2.3	Definition	45
2.3.1	Syntax of Calculus Processes	45
2.3.2	Reduction Rules for Executing Calculus Processes	47
2.3.3	Structural Congruence Rules for Equating Calculus Processes	49
2.3.4	Substitution of Free Values Inside Processes	50
2.3.5	Syntax Abbreviations for Frequently Used Processes	51
2.4	Distributed Model	52
2.4.1	Constraining the Calculus Syntax to Model TCP/IP Networks	52
2.4.2	Using Sites to Model a Hierarchical Network Topology	54
2.4.3	Using Reduction to Model Network Execution	55
2.4.4	Allowing Multiple Sites to Share the Same IP Address	57
2.5	Resource Monitoring Application	58
2.5.1	Examples of Resource Monitoring Applications	59
2.5.2	Informal Specification of an Application	60
2.5.3	Formal Calculus Specification	61
2.5.4	Calculus Execution Scenario	62
2.6	Security Properties	64
2.6.1	Proving Safety to Prevent Runtime Errors	64
2.6.2	Using Channel Types to Ensure Reliable Communication	65
2.6.3	Using Syntactic Constraints to Prevent Ambient Impersonation	68
2.7	Related Work	72
2.8	Conclusion	74
3	The Channel Ambient Machine	77
3.1	Introduction	77
3.2	Design Principles	79
3.2.1	Problems with Using Calculus Reduction to Implement a Runtime	79
3.2.2	Using a Normal Form to Execute Reductions up to Structural Congruence	80
3.2.3	Using Blocking to Efficiently Execute Successive Reductions	82
3.2.4	Justification for Defining an Abstract Machine	82
3.3	Definition	83
3.3.1	Syntax of Machine Terms	84
3.3.2	Using Construction to Encode a Process to a Machine Term	85
3.3.3	Using Selection to Schedule a Machine Term	88
3.3.4	Using Unblocking to Prevent Deadlocks During Execution	88

3.3.5	Using Reduction to Execute a Machine Term	89
3.4	Distributed Execution	93
3.4.1	Summary of Distributed Constraints for the Channel Ambient Calculus . .	93
3.4.2	Modelling Execution on TCP/IP Networks With Broadcast	94
3.4.3	Modelling Execution on TCP/IP Networks Without Broadcast	96
3.5	Resource Monitoring Application	97
3.5.1	Encoding the Application Specification to a Machine Term	97
3.5.2	Graphical Execution Scenario for the Application	99
3.5.3	Using the Application to Evaluate the Channel Ambient Machine	100
3.6	Correctness	103
3.6.1	Proving Safety to Prevent Runtime Errors	103
3.6.2	Proving Soundness to Ensure Valid Execution Steps	104
3.6.3	Proving Completeness to Ensure Accurate Execution	106
3.6.4	Proving Liveness to Prevent Deadlocks	108
3.6.5	Proving Termination to Prevent Livelocks	111
3.7	Related Work	112
3.8	Conclusion	113
4	The Channel Ambient Runtime	116
4.1	Introduction	116
4.2	Local Runtime Implementation	117
4.2.1	Architecture of the Local Runtime	118
4.2.2	Implementing Runtime Terms as Functional Data Structures	119
4.2.3	Implementing Substitution to Bind Values to Variables	119
4.2.4	Implementing Construction to Encode a Process to a Runtime Term	121
4.2.5	Implementing Selection to Schedule a Runtime Term	123
4.2.6	Implementing Unblocking to Prevent Deadlocks During Execution	125
4.2.7	Implementing Reduction to Execute a Runtime Term	125
4.3	Distributed Runtime Implementation	130
4.3.1	Implementing Runtime Terms in a Distributed Setting	130
4.3.2	Architecture of the Distributed Runtime	132
4.3.3	Implementing Reduction in a Distributed Setting	134
4.3.4	Implementing a Daemon to Manage Network Interactions	138
4.4	Enhanced Runtime Implementation	139
4.4.1	Using a Map Data Structure to Improve Selection	140
4.4.2	Using Deterministic Selection to Improve Efficiency	141

4.4.3	Using Network Masks to Conserve Bandwidth	142
4.4.4	Using Modules to Structure the Runtime Code	144
4.5	Related Work	145
4.6	Conclusion	148
5	The Channel Ambient Language	151
5.1	Introduction	151
5.2	Runtime Execution	152
5.2.1	Execution State	152
5.2.2	Networking	154
5.2.3	Debugging	154
5.3	Language Definition	155
5.3.1	Programs	155
5.3.2	Processes	156
5.3.3	Actions	158
5.3.4	Patterns	166
5.3.5	Types	167
5.3.6	Values	168
5.4	Resource Monitoring Application	170
5.4.1	Program Code	170
5.4.2	Initial State	171
5.4.3	Execution	172
5.4.4	Final State	173
5.5	Agent Tracker Application	174
5.6	Related Work	179
5.7	Conclusion	181
6	Conclusion	184
6.1	Thesis Summary	184
6.2	Future Work	186
A	Proofs	191
A.1	Security of the Channel Ambient Calculus	191
A.1.1	Safety	195
A.1.2	Typing	197
A.1.3	Authenticity	199
A.2	Correctness of the Channel Ambient Machine	202

<i>CONTENTS</i>	11
A.2.1 Safety	205
A.2.2 Soundness	208
A.2.3 Completeness	212
A.2.4 Liveness	216
B Language Syntax	221

List of Figures

1.1	Examples of Distributed Applications that can Benefit from Mobile Software . . .	15
1.2	Using the π -calculus to Model the Creation of Web Services	24
2.1	Hierarchical Components	40
2.2	Hierarchical Network Topology	40
2.3	Adjacent Components (i) and the Possible Interactions Between Components (ii) .	41
2.4	Symmetric Communication and Migration	42
2.5	Hierarchical TCP/IP Networks and their corresponding calculus representation. .	54
2.6	Multiplexed TCP/IP Networks and their corresponding calculus representation. .	58
2.7	Cartoon Scenario	60
2.8	Graphical Scenario	61
2.9	Calculus Specification	61
2.10	Graphical Calculus Execution Scenario, where $login' \notin \text{fn}(P, R)$	63
2.11	Calculus Execution Scenario, where $login' \notin \text{fn}(P, Q, R, S, C)$	63
3.1	Application Specification	98
3.2	Application Encoding	98
3.3	Graphical Machine Scenario, where $login' \notin \text{fn}(P, Q, R)$	99
3.4	Machine Scenario, where $login' \notin \text{fn}(P, Q, R)$	101
4.1	Architecture of the Local Runtime	118
4.2	Hierarchical Network Topology	131
4.3	Architecture of the Distributed Runtime	132
4.4	Protocol for Interaction between Sibling Runtimes	139
4.5	Modular Implementation	145
5.1	Specification and Implementation of the Resource Monitoring Application	170
5.2	Program code for the Resource Monitoring Application	170
5.3	Agent Tracker Specification and Implementation	174

5.4	Tracker Registration	176
5.5	Tracker Delivery	176
5.6	Initial and final Runtime states for site s_1 at address 192.168.0.3:3011	180
A.1	Proof Notations	191

Chapter 1

Introduction

1.1 Problem Description

1.1.1 Specifying and Implementing Secure Mobile Applications

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 thesis addresses the problem of how to specify and implement secure mobile applications.

1.1.2 Example Mobile Applications

The Internet is used across the globe for a wide variety of distributed applications. Some of these applications are illustrated in Figure 1.1, including search engines, data mining, applets, scripting, peer-to-peer systems, online trading and electronic commerce. Unfortunately, many of these applications are often hindered by two phenomena that cannot be abstracted away in a distributed setting: network *delay* and *disconnection*. Network delay refers to the interval of time between the departure of a message from one machine and its arrival on another machine. Two common causes of network delay are network congestion and the use of a slow network interface. Network disconnection, on the other hand, refers to a break in the connectivity between two machines on a network. This can occur for a 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

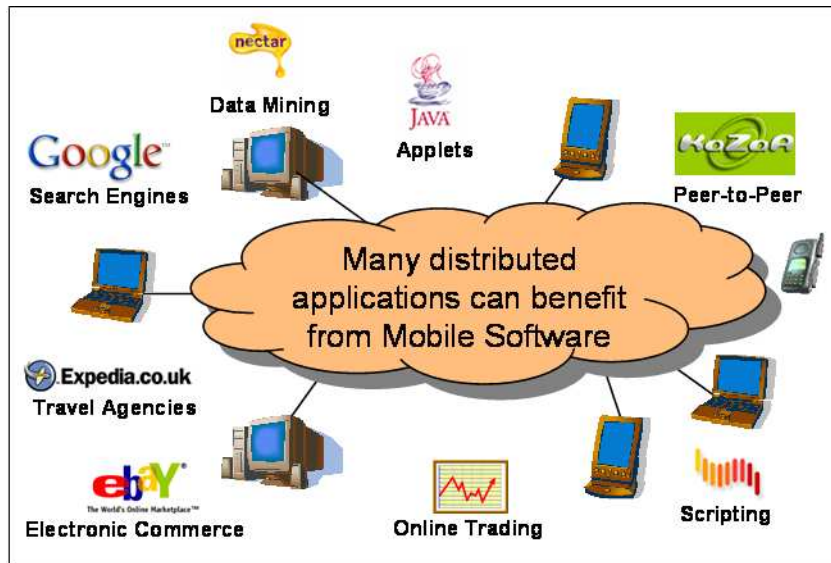


Figure 1.1: Examples of Distributed Applications that can Benefit from Mobile Software

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, network delay and disconnection 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 during peak hours. 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 delay and disconnection 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 delay and disconnection, 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 delay and disconnection in a number of distributed applications, including those

described in Figure 1.1. Some of the benefits of mobile agents are described in [24]. It is somewhat ironic that many of the problems related to the use of mobile devices can in part be solved by mobile software:

Travel Agencies agents visit different sites to plan a holiday according to certain high-level criteria, such as the cost of airline tickets, the weather forecast, hotel and car hire availability, flight times and connecting trains. A number of agents can be programmed with specific goals and then dispatched to dedicated sites in parallel, saving time and conserving network bandwidth. The agents can communicate with each other and update their search criteria in order to compile a shortlist of the best holidays, which can then be sent back to the client.

Search Engines agents crawl the web looking for data relating to certain keywords. The agents perform local, computation-intensive searches on large data sets, thereby reducing the consumption of network bandwidth. Agents can also provide a unifying interface to potentially heterogeneous querying environments, such as the range of data repositories containing large volumes of genetic information.

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.

Applets JavaTM applets are downloaded and executed on demand for a wide variety of applications. Applets provide a uniform programming abstraction and execution environment for implementing a range of downloadable applications.

Scripting various web applications send program code written in a specialised scripting language to client machines, in order to allow local interactions between the client and the application.

Peer-to-Peer agents are used to perform intelligent retrieval of files, in order to conserve bandwidth for peer-to-peer applications that share text, audio and video content.

Online Trading agents are dispatched to remote sites to respond to certain events with minimal delay, such as selling shares when stock prices reach a certain threshold. Agents are able to overcome the problems of network congestion by residing on the same site as the trading server. This allows the agent to respond as soon as information about price changes becomes available, even during peak trading hours when the network may be congested.

Electronic Commerce agents are used to negotiate a purchase on behalf of a client. The negotiations can be performed by the agent over long periods of time, e.g. for lengthy auctions that span several days. Agents can also be pre-programmed to meet certain timing constraints, such as bidding start times and closures that may conflict with a client's schedule or may be impractical in certain time zones.

These examples of 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. On-line 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 [83]. It is worth noting that a large proportion of 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 thesis. 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 in Chapter 5 of this thesis.

1.1.3 Security Properties of Mobile Applications

Security is of paramount importance for mobile applications. This stems from the fact that, in most cases, not all the members of a given network can be trusted. Security concerns for mobile applications can be expressed in terms of *Resources*, *Applications* and *Mobility*.

In general, a network is a means of sharing resources, but there is also a need to protect such resources from untrusted parties. The kinds of resources one may wish to protect can be expressed in terms of *data* and *services*:

Data includes passwords, secret keys, confidential documents or other media files, etc.

Services include licensed applications, restricted procedures, system calls, use of hardware resources such as printers, etc.

The well-known principle of *data encapsulation* unifies these two concepts by requiring all data to be accessed through services. A means of regulating access to these shared services is essential for the security of mobile applications.

Application security refers to the protection of an application from malicious attack. This includes the protection of resources within an application, but also extends to protecting the

behaviour of the application itself. There is a need to limit the extent to which this behaviour can be compromised by arbitrary attacks. One could argue that such attacks may only take place via access to application resources. As a result, it is necessary not only to carefully control access to resources, but also to limit the potential effects of such access on application behaviour.

Mobility allows an agent to move from one location to another. Typically these locations are either physical machines, or logical administrative domains. In each case, movement of an agent to a new location can result in new resources becoming available to both parties. Furthermore, abuse of these resources can compromise the security of applications that depend on them. Therefore, the movement of an agent to a new location needs to be carefully controlled, in order to protect the resources and applications of both the mobile agent and its host.

1.1.4 Implementing Mobile Applications

A major concern of mobile application developers is how to implement applications in a way that preserves the security properties of their specification.

Traditional approaches for developing mobile applications tend to first specify the application using high-level tools and then ascertain that the specification meets the desired requirements, either by prototyping or by formal analysis. The specification is then discarded and the application is implemented in a chosen target language. The main drawback with this approach is that any desirable properties of the specification will not necessarily hold in the implementation, since various security loopholes can be introduced during coding. Although it will never be possible to eliminate all the security flaws from an implementation, tools for generating program code directly from formal specifications are a step in the right direction. This helps to ensure that the work done during the specification of a mobile application is not lost during its implementation.

For the purposes of this thesis, a number of broad assumptions are made about the networks in which mobile applications can be implemented. In particular, applications are assumed to execute on networks that support the TCP/IP version 4 protocol, which is the most widely used Internet protocol to date. Applications are also assumed to execute in structured networks with a hierarchical topology. A distinction is made between Local Area Networks and Wide Area Networks, where machines inside a LAN can be protected from machines in a WAN by a firewall.

1.1.5 Specifying Mobile Applications

Before a mobile application can be implemented, one of the main challenges to overcome is how to produce a detailed specification of the application. This specification should be *rigorous*, *concise* and *suitably abstract* with respect to the mobile application being specified.

The specification should be rigorous, in order to allow reasoning about the properties of the

mobile application. A distinct advantage would be the ability to prove such properties in a formal manner. The specification should also be concise, remaining as clear as possible to avoid ambiguity. It is difficult to reason about a model that is not readily understandable. Finally, the specification should be suitably abstract with respect to the mobile application being specified, so that there is a close correspondence between the basic elements of the specification and the basic elements of the application. This will enable results obtained through reasoning about the specification to be readily applied to the application.

1.2 Related Work

1.2.1 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 a 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 λ -calculus [25], 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 [50] have been based on the λ -calculus, which provides a solid theoretical foundation.

More recently, the π -calculus [49] has been developed as a model for concurrent computation. Many argue that the π -calculus achieves for concurrent computation what the λ -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 π -calculus [71] demonstrated the feasibility of using process calculi to specify and implement applications involving location-independent communication between mobile agents. The Ambient calculus [21] was also introduced to model the hierarchical topology of modern networks, and many variants of ambients were subsequently proposed. In particular, Boxed Ambients [11] were developed to provide finer-grained and more effective mechanisms for ambient interaction. 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.

1.2.2 The π -calculus

Background It has been argued that a *choice-free* variant of the π -calculus [40, 10] is a suitable foundation on which to build a formalism for specifying and implementing secure mobile applications [69].

Definition The syntax of the π -calculus is summarised in Definition 1.2.1 in terms of processes P, Q, R , channels x, y, z and values u, v . A corresponding graphical syntax is presented in Definition 1.2.4, where each parallel process is represented as a vertical bar labelled with the actions that the process can perform. The graphical syntax is novel, but is reminiscent of various informal representations of concurrent processes, in which parallel processes are represented as adjacent vertical lines. The syntax of the π -calculus is defined as follows:

Null 0 terminates the execution of a process.

Parallel Composition $P \mid Q$ executes process P in parallel with process Q .

Restriction $\nu x P$ executes process P with a private channel x .

Output $x\langle v \rangle.P$ tries to send a value v on channel x and then execute process P .

Input $x(u).P$ tries to receive a value u on channel x and then execute process P .

Replicated Input $!x(u).P$ repeatedly tries to receive a value u on channel x and then execute process P .

The execution of a π -calculus process is defined using *reduction rules* of the form $P \longrightarrow P'$. The reduction rules of the π -calculus are summarised in Definition 1.2.2, where each rule describes how a given process P can evolve to a process P' by performing a single execution step. A corresponding graphical definition of reduction is presented in Definition 1.2.5, in which communication between parallel processes is represented as a horizontal arrow from sender to receiver. The portion of the diagram above the arrow represents the state of the system before the communication takes place, and the portion below represents the resulting state after the communication has occurred. The graphical representation is novel, but is reminiscent of Message Sequence Charts [42], in which a sequence of message exchanges is represented by successive horizontal arrows between sender and receiver processes, and time proceeds vertically downwards. The reduction rules of the π -calculus are defined as follows:

$P, Q, R ::=$	$\mathbf{0}$	Null	$ $	$x\langle v \rangle.P$	Output
	$ $	$P Q$	Parallel Composition	$ $	$x(u).P$ Input
	$ $	$\nu x P$	Restriction	$ $	$!x(u).P$ Replicated Input

Definition 1.2.1. Syntax of the π -calculus

$$x\langle v \rangle.P | x(u).Q \longrightarrow P | Q_{\{v/u\}} \quad (1.1)$$

$$x\langle v \rangle.P | !x(u).Q \longrightarrow P | Q_{\{v/u\}} | !x(u).Q \quad (1.2)$$

$$P \longrightarrow P' \Rightarrow P | Q \longrightarrow P' | Q \quad (1.3)$$

$$P \longrightarrow P' \Rightarrow \nu x P \longrightarrow \nu x P' \quad (1.4)$$

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

Definition 1.2.2. Reduction in the π -calculus

$$\mathbf{0} | P \equiv P \quad (1.6)$$

$$P | Q \equiv Q | P \quad (1.7)$$

$$P | (Q | R) \equiv (P | Q) | R \quad (1.8)$$

$$\nu x \mathbf{0} \equiv \mathbf{0} \quad (1.9)$$

$$\nu x \nu y P \equiv \nu y \nu x P \quad (1.10)$$

$$x \notin \text{fn}(Q) \Rightarrow Q | \nu x P \equiv \nu x (Q | P) \quad (1.11)$$

Definition 1.2.3. Structural Congruence in the π -calculus

$\begin{array}{ c } \hline P \\ \hline \end{array}, \begin{array}{ c } \hline Q \\ \hline \end{array}, \begin{array}{ c } \hline R \\ \hline \end{array} ::= \begin{array}{ c } \hline \mathbf{0} \\ \hline \end{array}$	Null	$\begin{array}{ c } \hline P \\ \hline \end{array} \begin{array}{ c } \hline Q \\ \hline \end{array}$	Parallel	$x::\begin{array}{ c } \hline P \\ \hline \end{array}$	Restriction
$\begin{array}{ c } \hline x\langle v \rangle.P \\ \hline \end{array}$	Output	$\begin{array}{ c } \hline x(u).P \\ \hline \end{array}$	Input	$\begin{array}{ c } \hline !x(u).P \\ \hline \end{array}$	Replication

Definition 1.2.4. Graphical Syntax of the π -calculus

$$\begin{array}{|c|} \hline x\langle v \rangle.P \\ \hline \end{array} \begin{array}{|c|} \hline x(u).Q \\ \hline \end{array} \longrightarrow \begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline Q_{\{v/u\}} \\ \hline \end{array} \quad (1.1)$$

$$\begin{array}{|c|} \hline x\langle v \rangle.P \\ \hline \end{array} \begin{array}{|c|} \hline !x(u).Q \\ \hline \end{array} \longrightarrow \begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline Q_{\{v/u\}} \\ \hline \end{array} \quad (1.2)$$

$$\begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline P' \\ \hline \end{array} \Rightarrow \begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline Q \\ \hline \end{array} \quad (1.3)$$

$$\begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline P' \\ \hline \end{array} \Rightarrow \begin{array}{|c|} \hline x::\begin{array}{|c|} \hline P \\ \hline \end{array} \\ \hline \end{array} \begin{array}{|c|} \hline P' \\ \hline \end{array} \quad (1.4)$$

$$\begin{array}{|c|} \hline Q \\ \hline \end{array} \equiv \begin{array}{|c|} \hline P \\ \hline \end{array} \begin{array}{|c|} \hline P' \\ \hline \end{array} \equiv \begin{array}{|c|} \hline Q' \\ \hline \end{array} \Rightarrow \begin{array}{|c|} \hline Q \\ \hline \end{array} \quad (1.5)$$

Definition 1.2.5. Graphical Reduction in the π -calculus

- (1.1) A parallel output and input process can communicate over a common channel. If there is an output $x\langle v \rangle.P$ executing in parallel with an input $x(u).Q$ then the value v can be sent over channel x and assigned to the value u in process Q , written $Q_{\{v/u\}}$. After the communication takes place, processes P and Q execute in parallel.
- (1.2) Similarly, a parallel output and replicated input process can communicate over a common channel. If there is an output $x\langle v \rangle.P$ executing in parallel with a replicated input $!x(u).Q$ then the value v can be sent over channel x and assigned to the value u in process Q , written $Q_{\{v/u\}}$. After the communication takes place, processes P and Q execute in parallel with the original replicated input $!x(u).Q$. In many respects, the replicated input $!x(u).Q$ can be viewed as a *server* process, which repeatedly receives values on channel x . For each value u received on x a new process Q is spawned in parallel.
- (1.3) 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 .
- (1.4) 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 private channel x .
- (1.5) 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' .

Equality between two processes is defined using *structural congruence rules* of the form $P \equiv Q$. The structural congruence rules are presented in Definition 1.2.3, where each rule describes how a given process P is equal to a process Q . In general, processes are equal up to re-ordering of parallel compositions and re-ordering of bound names. A corresponding graphical representation of structural congruence can be defined in a straightforward manner. Further details on structural congruence and the π -calculus in general can be found in [68, 48].

Example The expressive power of the π -calculus comes from its ability to model the creation and sharing of new channels. In the following example, the calculus is used to model the creation of a new web service, which is subsequently made available to a client. First, a new url is created and a server is set up to accept connections on this url. The url is then communicated to a client, which connects to the url, downloads a copy of the content over a secure channel and then interacts with its own private instance of the web service. The client can also send this url to other clients that would not have known about it beforehand. The example is specified in the π -calculus as follows:

$$\begin{aligned}
Server &\triangleq \nu url (c\langle url \rangle \mid !url(x).x\langle content \rangle.S) \\
Client &\triangleq \nu r c(u).u\langle r \rangle.r(m).C
\end{aligned}$$

Server The server creates a new *url* and sends it to a client on a public channel *c*. In parallel, the server continually listens on the *url* for a channel *x*. For each channel *x* received on the *url*, the server sends the corresponding content to channel *x* and spawns a new copy of the web service *S*.

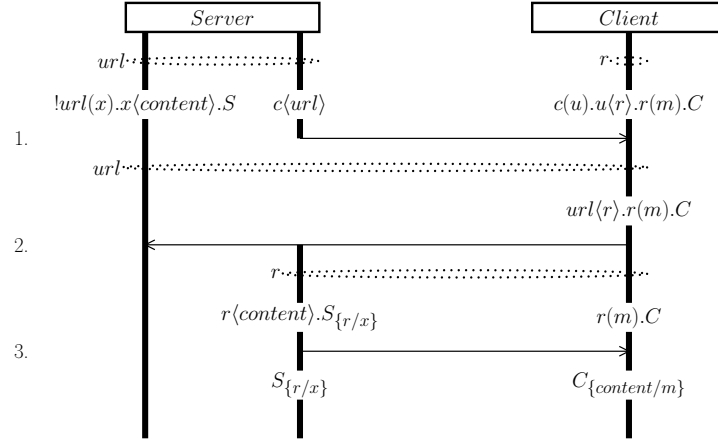
Client The client creates a new response channel *r* and listens for a *url* on the public channel *c*. Once a *url* has been received, the client sends the channel *r* on the *url* and then waits for the content on *r*. Once the content has been received, the client executes the process *C*, which can interact with the web service.

An execution scenario for this example is presented in Figure 1.2, together with the corresponding graphical representation. Initially, the server and client are executing in parallel:

1. The server sends a fresh *url* to the client on channel *c*.
2. The client sends a fresh response channel *r* to the *url*.
3. The server sends the content of the *url* to the client on the response channel, and then executes a copy of the web service *S*. In parallel the client executes the process *C*, which interacts with the web service.

The above scenario illustrates how restriction in the π -calculus can be used to explicitly model the privacy of communication. For example, initially the *url* is only known by the server, but after (1) it becomes known by both the server and the client. The scope of the *url* can be further extended if it is communicated to other clients in the network. Similarly, after (2) the interaction between the web service and the client takes place without interference on a private channel *r*.

Evaluation From a specification perspective, the π -calculus enables a concise description of concurrent computation and offers a good level of abstraction for specifying concurrent applications. Channels in the π -calculus bear a close resemblance to many entities in distributed computing, including IP addresses, TCP sockets and local ports. They can be used to model *data* including URLs, references to documents or other media files, and *services* such as methods, functions, remote procedure calls, servers and ports for hardware devices such as printers. Nevertheless, a number of extensions to the π -calculus are needed in order to make it suitable for specifying mobile applications. For example, the π -calculus treats all parallel processes as being at the same level, and therefore cannot express the logical and physical boundaries that characterise



- $$\begin{aligned}
 & \nu url (c\langle url \rangle \mid !url(x).x\langle content \rangle.S) \mid \nu r c(u).u\langle r \rangle.r(m).C \\
 1. \quad & \longrightarrow \nu url (!url(x).x\langle content \rangle.S \mid \nu r url\langle r \rangle.r(m).C) \\
 2. \quad & \longrightarrow \nu url (!url(x).x\langle content \rangle.S) \mid \nu r (r\langle content \rangle.S_{\{r/x\}} \mid r(m).C) \\
 3. \quad & \longrightarrow \nu url (!url(x).x\langle content \rangle.S) \mid \nu r (S_{\{r/x\}} \mid C_{\{content/m\}})
 \end{aligned}$$

Figure 1.2: Using the π -calculus to Model the Creation of Web Services

most networks. In addition, the π -calculus requires global synchronisation over channels in order for parallel processes to communicate, which violates one of the basic principles of distributed computing.

From an implementation perspective, the π -calculus has been used as the basis for the Pict programming language, in which a wide range of concurrent applications have been developed. The Pict runtime was implemented based on a formal specification, which was proved sound with respect to the π -calculus. However, the π -calculus cannot be readily implemented in a distributed setting, since it requires global synchronisation over channels in a network.

From a security perspective, a large volume of research has already been published on security in the π -calculus, including numerous type systems, encodings and theories of equivalence. By using the π -calculus as the basis for modelling mobile computation, much of this associated theory can be re-used. In particular, the π -calculus can facilitate reasoning about security properties of concurrent applications by means of process equivalences. This is achieved by using a π -calculus process to describe an abstract specification of a concurrent application, for which the proofs of certain behavioural and security properties are straightforward. The approach is then to prove that the complete π -calculus specification is equivalent, at a suitable level, to this abstract specification. Various analysis tools for checking equivalences in the π -calculus have already been developed, such as the mobility workbench [79]. Ongoing research indicates that a similar approach can be used to

reason about the security of mobile applications [77]. More recently, logics for the π -calculus have been defined as a more general alternative to the use of process equivalences [15]. Types can also be used to guarantee security properties, by constraining the types of data that can be sent over a given channel. In addition, processes themselves can be typed according to specific behaviour patterns, and dynamic checks can be used to ensure that processes are well-behaved with respect to their type. The π -calculus has also been used to reason about communication protocols for network security. In particular, the spi-calculus [2] extends the π -calculus with primitives for encryption and decryption. More recently, the π -calculus has also been used to reason about security properties of web services [9].

A number of variations on the π -calculus have been proposed, in order to overcome some of its limitations. The Join calculus [28] was developed as the foundation for a concurrent, distributed programming language. The calculus can be considered a simplification of the π -calculus in which restriction, replication and input are merged into a single *definition* construct, where a given definition can synchronise with multiple outputs simultaneously. The Explicit Fusion calculus [34] was proposed as the basis for a distributed abstract machine for the π -calculus. The Distributed π -calculus [61, 62] is a distributed extension of the π -calculus, which incorporates the notions of remote execution, migration and site failure. The calculus uses explicitly located communication channels, such that both the channel name and its location are required for communication. A survey of some of the variants of the π -calculus and their application to distributed programming is given in [69].

1.2.3 The Nomadic π -calculus

Background The Nomadic π -calculus [71, 83] was developed as a formalism for studying the interaction between mobile agents. According to [71], the calculus considers “the design, semantic definition and implementation of communication primitives by which mobile agents can interact”. According to [83], the authors of Nomadic π “were looking for a calculus which would lay down a foundation for a distributed programming language, suitable for describing infrastructure algorithms for mobile agent systems”. They recognized that the π -calculus provided an abstract model of concurrent computation, based on a reduced set of concepts. However, they also recognized that the π -calculus did not provide support for modelling *distributed programming*, since it was unable to express the notion of “computer nodes, allocation of resources to these nodes, process mobility and system failures.”

The main focus of the Nomadic π -calculus was to address the problem of location-independent communication over a wide-area network. The calculus was used to specify a central server application for keeping track of the location of a number of mobile clients. This application was proved correct using novel techniques for process equivalences, as outlined in [77]. In addition,

the Nomadic Pict programming language and runtime system [84] were developed based on the underlying calculus.

$P, Q ::=$	0	Null		agent $a = P$ in Q	Agent Creation
	$P \mid Q$	Parallel Composition		migrate to $s \rightarrow P$	Agent Migration
	$x!v$	Output		if $u = v$ then P else Q	Equality Testing
	$x?u \rightarrow P$	Input		iflocal $\langle a \rangle x!v \rightarrow P$ else Q	Test-and-Send
	$x?*u \rightarrow P$	Replicated Input			
	new x in P	Restriction			

Definition 1.2.6. Syntax of the Nomadic π -calculus

$$\Gamma, @_a \mathbf{agent} \ b = P \ \mathbf{in} \ Q \longrightarrow \Gamma, \mathbf{new} \ b @ \Gamma(a) \ \mathbf{in} \ (@_b P \mid @_a Q) \quad (1.12)$$

$$\Gamma, @_a \mathbf{migrate to} \ s \rightarrow P \longrightarrow (\Gamma \oplus a \mapsto s), @_a P \quad (1.13)$$

$$\begin{aligned} \Gamma, @_a \mathbf{iflocal} \ \langle b \rangle x!v \rightarrow P \ \mathbf{else} \ Q &\longrightarrow \Gamma, @_b x!v \mid @_a P && \text{if } \Gamma(a) = \Gamma(b) \\ &\longrightarrow \Gamma, @_a Q && \text{if } \Gamma(a) \neq \Gamma(b) \end{aligned} \quad (1.14)$$

$$\Gamma, @_a (x!v \mid x?u \rightarrow P) \longrightarrow \Gamma, @_a \{v/u\} P \quad (1.15)$$

$$\Gamma, @_a (x!v \mid x?*u \rightarrow P) \longrightarrow \Gamma, @_a (\{v/u\} P \mid x?*u \rightarrow P) \quad (1.16)$$

$$\begin{aligned} \Gamma, @_a \mathbf{if} \ u = v \ \mathbf{then} \ P \ \mathbf{else} \ Q &\longrightarrow \Gamma, @_a P && \text{if } u = v \\ &\longrightarrow \Gamma, @_a Q && \text{if } u \neq v \end{aligned} \quad (1.17)$$

$$\frac{x \notin \text{dom}(\Gamma) \wedge \Gamma, P \longrightarrow \Gamma', P'}{\Gamma, \mathbf{new} \ x \ \mathbf{in} \ P \longrightarrow \Gamma', \mathbf{new} \ x \ \mathbf{in} \ P'} \quad (1.18)$$

$$\frac{(\Gamma, a \mapsto s), P \longrightarrow (\Gamma', a \mapsto s'), P'}{\Gamma, \mathbf{new} \ a @ s \ \mathbf{in} \ P \longrightarrow \Gamma', \mathbf{new} \ a @ s' \ \mathbf{in} \ P'} \quad (1.19)$$

$$\frac{\Gamma, P \longrightarrow \Gamma', P'}{\Gamma, P \mid Q \longrightarrow \Gamma', P' \mid Q} \quad (1.20)$$

$$\frac{Q \equiv P \wedge \Gamma, P \longrightarrow \Gamma', P' \wedge P' \equiv Q'}{\Gamma, Q \longrightarrow \Gamma', Q'} \quad (1.21)$$

Definition 1.2.7. Reduction in the Nomadic π -calculus

Definition The syntax of the Nomadic π -calculus is summarised in Definition 1.2.6 in terms of processes P, Q, R , channels x, y, z , values u, v , agents a, b and sites s . The Nomadic π -calculus extends the π -calculus with new primitives for agent creation, agent migration, value equality testing and test-and-send to an agent at the current site.

The reduction rules of the Nomadic π -calculus are summarised in Definition 1.2.7. The rules make use of an additional environment Γ to store the locations of mobile agents. The environment is of the form $(a_1 \mapsto s_1, \dots, a_N \mapsto s_N)$, such that each agent a is mapped to a site s . The reduction rules also use a notion of located processes, where $@_a P$ denotes a process P executing inside agent

a , and **new** $a@s$ **in** P denotes a fresh agent a currently at site s :

- (1.12) The execution of **agent** $a = P$ **in** Q spawns a new agent on the current site with body P , and continues executing Q . The new agent has a unique name a , which may be referred to in both P and Q .
- (1.13) The execution of **migrate to** $s \rightarrow P$ results in the whole agent migrating to site s , followed by the execution of process P . Note that these two primitives are often executed in parallel with other processes in the body of an agent.
- (1.14) The low-level calculus provides a single primitive **iflocal** $\langle a \rangle x!v \rightarrow P$ **else** Q for interaction between agents. Executing this primitive in the body of an agent b has two possible outcomes. If a and b are on the same site the message is reliably delivered and process P is executed. Otherwise the message is discarded and Q is executed. This primitive simplifies algorithms involving communication with agents that can migrate at any time and is implementable locally, with minimal overhead.
- (1.17) Value equality testing **if** $u = v$ **then** P **else** Q is useful for expressing distributed algorithms, such as an algorithm for a name server to look up an agent's location.

The only primitive that involves network communication is agent migration, which is used to implement all forms of message passing. Sending a message to channel x of a local agent a is implemented by **iflocal** $\langle a \rangle x!v \rightarrow 0$ **else** 0 . Sending a message to a remote agent is implemented by creating a new agent to deliver the message:

$$\mathbf{agent} \ b = (\mathbf{migrate\ to} \ s \rightarrow (\mathbf{iflocal} \ \langle a \rangle x!v \rightarrow 0 \ \mathbf{else} \ 0)) \ \mathbf{in} \ 0$$

Various high-level primitives can be expressed in terms of this, such as the location independent output to agent a , written $\langle a@? \rangle x!v$.

Example The Nomadic π -calculus can be used to model the following example [83], in which an applet server is created for sending applets to clients on demand:

$$\begin{aligned} & getApplet?*[a, s] \rightarrow \\ & \quad \mathbf{agent} \ b = \\ & \quad \quad \mathbf{migrate\ to} \ s \rightarrow (\langle a@s \rangle ack!b \mid B) \\ & \quad \mathbf{in} \ 0 \end{aligned}$$

The applet server continually listens on the *getApplet* channel for a request consisting of an agent a and the agent's location s . For each request received, the server creates a new agent b , which

moves to site s and sends its name to agent a at this site, on a public acknowledgement channel. If the agent a is present at s then the acknowledgement is delivered, otherwise it is discarded. The example illustrates the main entities represented in the Nomadic π -calculus, namely sites, agents and channels. Note that, unlike the π -calculus, Nomadic π explicitly models the existence of mobile agents as separate entities. It also models the ability of these agents to move around between sites and communicate with each other over a network.

Evaluation The Nomadic Pict project was one of the first to demonstrate the feasibility of specifying a non-trivial mobile application in a process calculus, proving the correctness of the specification and then compiling the specification to executable program code.

From a specification perspective, the Nomadic π -calculus explicitly models the existence of mobile agents and sites, which are essential components in the specification of mobile applications. However, the calculus only considers a two-level architecture of sites and agents, and cannot be used to model a hierarchical network topology. Each agent in Nomadic π is unique, and the name of an agent is restricted on creation. Yet this does not fit well with certain network models, in which agents with the same name reside at different locations, in order to provide a uniform agent interface across multiple locations in the network.

From an implementation perspective, the Nomadic Pict runtime is an extension of the Pict runtime, which is based on an abstract machine that was proved sound with respect to the π -calculus. However, no corresponding abstract machine has yet been formalised for the Nomadic π -calculus.

From a security perspective, the Nomadic Pict project demonstrates how mobile applications can be proved correct using a suitable notion of process equivalence [77]. However, the proofs of such applications are non-trivial, due in part to the complex semantics of the Nomadic π -calculus, which requires an auxiliary environment for tracking the location of processes. The proof of the central server application presented in [77] required the development of a corresponding intermediate language, for which the underlying theory and proof techniques were the subject of an entire PhD thesis [76]. In the general case a new intermediate language needs to be developed for each application, resulting in a significant overhead for proving properties of applications. Another aspect of security in the Nomadic π -calculus is that agents are free to move to any site without restriction. As a result, potentially malicious agents can move to a given site at any time, gaining access to local resources. Such agents can then act as spyware, sending and receiving messages at will and consuming bandwidth and CPU cycles. Additional security mechanisms are therefore needed in order to limit the accessibility of mobile agents to sites. The communication primitives of Nomadic π use a fine-grained semantics, where sending an output to an adjacent agent reduces to a local output inside the agent. In this way, a potentially hostile agent can inject

an unlimited number of output processes into an unsuspecting neighbour. More coarse-grained communication primitives are needed to prevent this form of code insertion.

From an application perspective, one of the main strengths of the Nomadic π -calculus is its close link with mobile applications. The calculus was specifically developed with applications in mind, and a range of non-trivial applications have already been programmed, some of which are presented in [83].

1.2.4 The Ambient Calculus

Background The Ambient calculus was developed to model two distinct aspects of mobility: “*mobile computing*, concerning computation that is carried out in mobile devices, and *mobile computation*, concerning mobile code that moves between devices”, as described in [21]. In addition, the calculus aims “to describe all these aspects of mobility within a single framework that encompasses mobile agents, the *ambients* where agents interact and the mobility of the ambients themselves”.

According to [21], the inspiration for the Ambient calculus “comes from the potential for mobile computation over the World-Wide Web”. The authors argue that “the geographic distribution of the Web naturally calls for mobility of computation, as a way of flexibly managing latency and bandwidth.” They also argue that “the main difficulty with mobile computation on the Web is not in mobility per se, but in the handling of administrative domains. In the early days of the Internet one could rely on a flat name space given by IP addresses... This is no longer the case: firewalls partition the Internet into administrative domains that are isolated from each other except for rigidly controlled pathways.”.

Therefore, the aim of the ambient calculus according to [21] is to be a fundamental model of “locations, of mobility and of authorization to move”. A key feature of the calculus is the hierarchical structure of ambients, which can model “the movement of self-contained nested environments that include data and live computation.” Ultimately, the goal of the calculus is “to make mobile computation scale-up to widely distributed, intermittently connected and well administered computational environments.”

Definition The syntax of the Ambient calculus is summarised in Definition 1.2.8 in terms of processes P, Q and capabilities M, N . The ambient calculus extends the π -calculus with the notion of an *ambient*, which describes a bounded place where computation happens. The boundary determines what is inside or outside an ambient, such that when an ambient moves, everything inside the ambient moves with it. The syntax of processes in the Ambient calculus is defined as follows, where the Null, Parallel Composition and Restriction processes are the same as in the π -calculus:

$P, Q ::=$	0	Null	$M, N ::=$	x	Variable
	$P \mid Q$	Parallel Composition		n	Name
	$\nu n P$	Restriction		$in M$	Enter
	$!P$	Replication		$out M$	Leave
	$a[P]$	Ambient		$open M$	Open
	$M.P$	Capability		ε	Empty
	$(x).P$	Input		$M.N$	Path
	$\langle M \rangle$	Output			

Definition 1.2.8. Syntax of the Ambient Calculus

$$n \boxed{in m.P \mid Q} \mid m \boxed{R} \longrightarrow m \boxed{n \boxed{P \mid Q} \mid R} \quad (1.22)$$

$$m \boxed{n \boxed{out m.P \mid Q} \mid R} \longrightarrow n \boxed{P \mid Q} \mid m \boxed{R} \quad (1.23)$$

$$open m.P \mid m \boxed{Q} \longrightarrow P \mid Q \quad (1.24)$$

$$(x).P \mid \langle M \rangle \longrightarrow P_{\{M/x\}} \quad (1.25)$$

$$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q \quad (1.26)$$

$$P \longrightarrow P' \Rightarrow \nu n P \longrightarrow \nu n P' \quad (1.27)$$

$$P \longrightarrow P' \Rightarrow a \boxed{P} \longrightarrow a \boxed{P'} \quad (1.28)$$

$$Q \equiv P \longrightarrow P' \equiv Q' \Rightarrow Q \longrightarrow Q' \quad (1.29)$$

Definition 1.2.9. Reduction in the Ambient Calculus

Ambient $a \boxed{P}$ executes the process P inside ambient a

Replication $!P$ repeatedly executes process P . This is a more general form of replication than replicated input in the π -calculus.

Capability $M.P$ executes the capability M and then executes the process P .

Capabilities allow an ambient to interact with other ambients, without revealing its true name:

Enter $in m$ tries to move the surrounding ambient into a sibling ambient named m .

Leave $out m$ tries to move the surrounding ambient out of its parent ambient named m .

Open $open m$ tries to dissolve the boundary of a child ambient named m .

Path $M.N$ executes the capability M followed by the capability N . A sequence of capabilities can be used to denote a path to a given ambient, such as $in n.in m$. The empty path is denoted by ε .

The calculus also allows capabilities to be sent and received locally:

Output $\langle M \rangle$ tries to send the capability M

Input $(x).P$ tries to receive a capability and then execute process P , in which the received capability is bound to the variable x .

Although the syntax only allows local communication, courier ambients can be used to send messages across ambient boundaries. For example, an ambient a can send a capability M to a sibling ambient b by creating a courier ambient c , which leaves a , enters b , is opened inside b and then delivers its message locally:

$$a \left[c \left[out\ a.in\ b.\langle M \rangle \right] \mid P \right] \mid b \left[open\ c.(x).Q \right]$$

In the general case, the message can be a name or a capability. However, the intention is that the communication of names will be rather rare, since a name gives significant control over an ambient. Therefore, specific capabilities will usually be sent, such as the ability to enter, leave or open an ambient. For example, ambient a can send the capability $in\ a$, which will allow b to enter a . By definition, the calculus does not allow a name to be deduced from a capability, which provides some protection for the identity of an ambient.

The reduction rules of the Ambient calculus are summarised in Definition 1.2.9:

- (1.22) If an ambient n contains an enter $in\ m.P$ and in parallel there is a sibling ambient m , then n can enter m and execute process P .
- (1.23) If an ambient n contains a leave $out\ m.P$ and is inside a parent ambient m , then n can leave its parent m and execute process P .
- (1.24) If there is an open $open\ m.P$ in parallel with a child ambient m , then the boundary of m can be dissolved.
- (1.25) If there is an output $\langle M \rangle$ in parallel with an input $(x).P$ then the capability M can be communicated and assigned to x in process P .
- (1.26)-(1.26) A reduction can also be performed in parallel with a process Q , inside a restricted name n , inside an ambient a and up to re-arranging of processes, respectively.

Using these reduction rules, the sequence of execution steps for the message delivery example can be computed as follows:

$$\begin{aligned}
& a \boxed{c \text{ out } a.in \ b. \langle M \rangle \mid P} \mid b \boxed{\text{open } c.(x).Q} \\
\longrightarrow & a \boxed{P} \mid c \boxed{\text{in } b. \langle M \rangle} \mid b \boxed{\text{open } c.(x).Q} \\
\longrightarrow & a \boxed{P} \mid b \boxed{\text{open } c.(x).Q \mid c \langle M \rangle} \\
\longrightarrow & a \boxed{P} \mid b \boxed{(x).Q \mid \langle M \rangle} \\
\longrightarrow & a \boxed{P} \mid b \boxed{Q_{\{M/x\}}}
\end{aligned}$$

In particular, if M is an entry capability *in* a , then b can use this capability to enter a . Note that since both names and capabilities can be sent and received, a type system is needed to distinguish the two, in order to prevent meaningless terms such as $n.P$.

Evaluation The Ambient calculus marked a turning point in the evolution of process calculi, because of its ability to model both the physical and logical hierarchy inherent in mobile applications.

From a specification perspective, the calculus models mobile agents, machines, network domains and mobile devices with a unifying ambient paradigm. Communication between ambients is modelled by a combination of ambient mobility and the *open* action. However, in some cases this action can be a security concern, since it completely dissolves the boundary of an ambient, allowing the processes contained in the ambient to merge with the environment. The drawbacks of *open* are a well-documented, yet this action is required in order for two ambients to communicate.

From an implementation perspective, a number of prototype implementations for ambients have been developed. In [17] an informal abstract machine for Ambients is presented, together with a corresponding implementation in Java. In [30] a distributed abstract machine for Ambients is described, based on a formal mapping from the Ambient Calculus to the Distributed Join calculus. An abstract machine for a safe variant of the Ambient calculus is defined in [67].

From a security perspective, a wide range of analysis techniques have been developed for the Ambient calculus. In particular, Ambient Logics [20] seem to be at the right level of abstraction for stating and proving security properties of mobile applications. Such logics seem to allow a more intuitive formulation of security properties than traditional process equivalences [38], while still being amenable to automated analysis [16]. The benefits of using Ambient Logics to reason about the correctness of algorithms have been acknowledged by the authors of the Nomadic π -calculus, although such logics are not directly applicable to Nomadic π . A range of type systems for the ambient calculus have been developed to enforce constraints on the movement of ambients [19, 18, 14]. In addition, capabilities in the ambient calculus allow careful control of the actions that one ambient can perform on another. However, once a capability has been bestowed it cannot

be revoked. For example, if a host ambient sends an entry capability to another guest ambient, this guest can potentially enter the host an infinite number of times. Furthermore, the guest can send the capability to any number of peer ambients who themselves would have unlimited access to the host. Finally, when an ambient enters, exits or opens another ambient, the second ambient undergoes this action and has no control on when the action takes place. This can also be viewed as a potential security risk.

From an application perspective, a number of small example systems have been modelled in the Ambient calculus, but there has been little work on using the calculus to model more substantial mobile applications. This is partly because the calculus itself was not designed for mobile applications specifically, but was developed more as a minimal calculus for mobility.

A number of variants have been defined for the Ambient calculus, in order to overcome some of its limitations. The Safe Ambient calculus [45] defines additional co-actions $\overline{in} n$, $\overline{out} n$, $\overline{open} n$ so that ambients can have more control over when a given action takes place. These co-actions ensure that any action (*in*, *out*, *open*) takes place only if both participants agree. More precise control of ambient interactions has also been obtained by extending these co-actions with additional passwords [47]. The Boxed Ambient calculus [11] avoids the security risks of the *open* action by using more refined communication primitives, which allow communication across ambient boundaries.

1.2.5 The Boxed Ambient Calculus

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 [11] shows how the Ambient calculus can be complemented with finer-grained and more effective mechanisms for ambient interaction. In [12] Boxed Ambients are used to reason about resource access control, and in [27] 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 [46] uses co-capabilities to express explicit permissions for accessing ambients, and the NBA calculus [13] 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.

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, Boxed Ambients have not yet been used to model non-trivial mobile applications. This thesis addresses some of these open issues, and investigates whether a variant of Boxed Ambients can be used for specifying and implementing secure mobile applications.

1.3 Thesis Outline

1.3.1 Thesis Statement

“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.”

1.3.2 Thesis Contribution

This thesis presents a formalism for specifying and implementing secure mobile applications, known as the Channel Ambient System. The thesis builds on recent research in the field of process calculi to develop a new model of computation known as the Channel Ambient calculus, which can be used both to specify mobile applications and to reason about their security properties. The primitives of the calculus were developed with real-world applications in mind, and are designed to be at a level of abstraction suitable for an application programmer. The thesis also bridges a gap between theory and implementation by defining a distributed abstract machine for the Channel Ambient calculus. The abstract machine uses a list semantics, which is close to an implementation language, and a blocking semantics, which leads to an efficient implementation. The machine is proved sound and complete with respect to the underlying calculus. A prototype programming language and runtime are also described, together with an application for tracking the location of migrating agents. The correctness of the machine ensures that the work done in specifying and analysing mobile applications is not lost during their implementation. An outline of each of the main chapters is described below.

1.3.3 The Channel Ambient Calculus

Chapter 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 π -calculus, the Nomadic π -calculus and the Ambient calculus. In many respects, the Channel Ambient 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.

The chapter first describes the assumptions that were made about the nature of mobile applications in order to design the Channel Ambient calculus. The chapter then presents a formal definition of the calculus, based on the above design principles. A corresponding graphical representation is also presented, which can be used to specify execution traces of mobile applications.

The chapter then describes how the calculus can be used to specify mobile applications for networks that support the TCP/IP protocol. The calculus is used to specify a simple mobile application, in which a mobile agent monitors resources on a remote site. Finally, the chapter illustrates how various security mechanisms developed for related calculi can be applied to the Channel Ambient calculus, in order to reason about the security properties of mobile applications.

1.3.4 The Channel Ambient Machine

Chapter 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.

The chapter first describes the design principles of the Channel Ambient Machine, and shows how the machine can be derived from the Channel Ambient calculus. The chapter then presents a formal definition of the machine, based on the above design principles. The machine uses a list syntax, which is close to an implementation language, together with a blocking semantics, which leads to an efficient implementation. A corresponding graphical representation is also presented, which can be used to visualise execution traces of mobile applications. The chapter then describes how the Channel Ambient Machine can be used to execute mobile applications on networks that support the TCP/IP protocol. The machine is used to execute an example application, in which a mobile agent monitors resources on a remote site. Finally, the chapter proves the correctness of the Channel Ambient Machine with respect to the Channel Ambient calculus. The machine is proved both sound and complete with respect to the calculus, and is also proved to be free of runtime errors, deadlocks and livelocks. Soundness ensures that the machine always performs valid execution steps, while completeness ensures that the machine can correctly match all possible execution steps of the calculus.

1.3.5 The Channel Ambient Runtime

Chapter 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 chapter first describes how the Channel Ambient Machine can be used to implement a local runtime, which executes a given calculus process on a single physical device. The chapter then describes how the machine can be used to implement a distributed runtime, which executes a given calculus process over multiple devices in a hierarchical TCP/IP network. Finally, the chapter describes how the distributed runtime can be enhanced in order to improve the efficiency

of process execution, while conserving network bandwidth.

1.3.6 The Channel Ambient Language

Chapter 5 presents a programming language for developing mobile applications, known as the Channel Ambient Language (CAL). The language illustrates how the high-level constructs of the Channel Ambient calculus can be used as programming abstractions for mobile applications. In many respects, the Channel Ambient Language is a product of the previous three chapters, incorporating the main features of the calculus, the abstract machine and the runtime. The Channel Ambient Language and runtime system are available at [52]. 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 chapter first describes how programs in the Channel Ambient Language can be executed by the Channel Ambient Runtime. A number of debugging mechanisms for visualising the execution state of the runtime are also presented, together with a summary of the types of networks in which programs are assumed to execute. The chapter then describes the syntax of programs in the Channel Ambient Language, together with the corresponding rules for program execution. The language extends the calculus with useful programming constructs such as data structures, process definitions and system calls. The language is used to program an example mobile application, in which a mobile agent monitors resources on a remote server. Detailed execution traces for the application are also presented. Finally, the language is used to program an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network.

Chapter 2

The Channel Ambient Calculus

2.1 Introduction

Process calculi were described in Chapter 1 as a promising formalism for specifying and implementing secure mobile applications. In particular, Boxed Ambient calculi were described as well-suited 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 chapter 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 π -calculus, the Nomadic π -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. The chapter is structured as follows.

Section 2.2 describes the assumptions that were made about the nature of mobile applications in order to design the Channel Ambient calculus.

Section 2.3 presents a formal definition of the calculus, based on the above design principles.

A corresponding graphical representation is also presented, which can be used to specify execution traces of mobile applications.

Section 2.4 describes how the Channel Ambient calculus can be used to specify mobile applications for networks that support the TCP/IP protocol.

Section 2.5 uses the calculus to specify an example mobile application, in which a mobile agent monitors resources on a remote site.

Section 2.6 illustrates how various security mechanisms developed for related calculi can be applied to the Channel Ambient calculus, in order to reason about the security properties of mobile applications.

2.2 Design Principles

This section describes the assumptions that were made about the nature of mobile applications in order to design the Channel Ambient calculus.

Subsection 2.2.1 describes the structure and type of components that are assumed to be present in mobile applications.

Subsection 2.2.2 describes how components of mobile applications are assumed to interact with each other.

Subsection 2.2.3 describes how components of mobile applications are assumed to communicate with each other.

Subsection 2.2.4 describes how components of a mobile application are assumed to move relative to each other.

2.2.1 Components of Mobile Applications

The Channel Ambient calculus assumes that mobile applications are made up of named components, arranged in a hierarchy. The components of a mobile application can be machines, agents or modules:

- A machine can be a fixed server or client device, or a mobile device such as a laptop, phone or PDA.
- An agent is a mobile software component that can move between different machines in order to perform various tasks.
- A module is a software component that provides various services. These services are typically invoked using method calls to the module.

The components of a mobile application are named:

- The name or *address* of a machine is used to interact with the machine over a network. Although a given machine can have multiple addresses at different levels, in practice it is the IP address of the machine that is the most widely-used.
- The name or *identity* of an agent is used to interact with the agent, or to authenticate the agent as it moves between machines in a network.
- The name or *identifier* of a module is used to invoke services provided by the module.

The components of a mobile application are arranged in a hierarchy. Figure 2.1 illustrates an example of a hierarchy of components a, \dots, g arranged in a tree structure. The root g of the tree is at the bottom of the hierarchy, and the leaves a, b, c, f are at the top:

- The topology of machines on the Internet is generally hierarchical. For example, Local Area Networks are usually logically contained inside a gateway, which connects them to a Wide Area Network. Figure 2.2 illustrates an example of a Local Area Network of machines m_1, \dots, m_N connected to a Wide Area Network by a gateway machine m_0 . The machines inside the LAN are not directly accessible from the outside, and all communication between the LAN and the WAN needs to take place via the gateway. The gateway can also act as a security barrier or *firewall*, which protects the machines inside the gateway from outside attacks, and regulates access to the outside network. If the gateway disconnects from the WAN, then the machines it contains can still interact with each other, but can no longer interact with machines in the WAN.

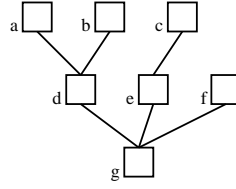


Figure 2.1: Hierarchical Components

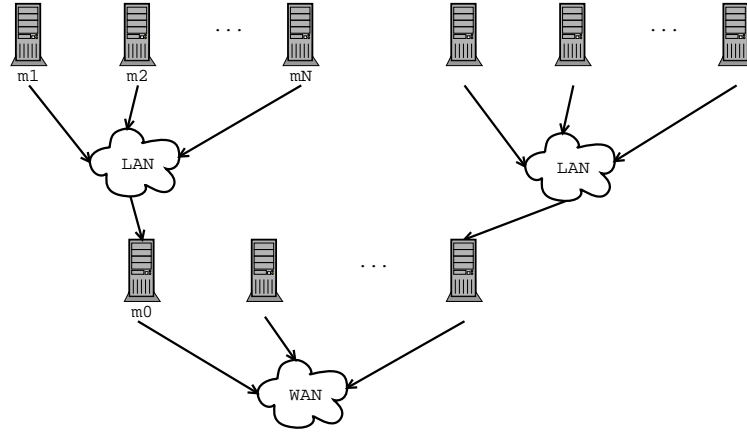


Figure 2.2: Hierarchical Network Topology

- Agents in a mobile application can also form hierarchies. A given agent is logically contained inside a machine, and the agent itself can contain sub-agents and modules. When an agent moves, the sub-agents and modules move with it, and the internal hierarchy of the agent is preserved. In some cases, an agent may wish to dynamically acquire a new component. This can be achieved by moving the component inside the agent, so that it becomes part of the logical contents of the agent.
- Modules can also be hierarchical. A given module can logically contain its own private methods, data and sub-modules, which are not directly accessible from outside the module. Certain large programs such as operating systems have a hierarchy of modules that represents different security levels. Modules at the higher levels cannot interact directly with lower level modules, but need to do so via intermediate hooks and system calls.

2.2.2 Interaction Between Mobile Application Components

The Channel Ambient calculus assumes that only adjacent components can interact directly, that communication and migration are two distinct forms of interaction and that communication and migration are symmetric.

The hierarchical nature of components in mobile applications suggests that only adjacent components should be allowed to interact directly:

- A machine inside a LAN cannot interact directly with a machine outside the LAN, but needs to do so via a gateway. If the gateway disconnects, then the interaction can no longer occur.
- Similarly, a mobile agent cannot directly interact with the components inside another agent, but needs to do so via the containing agent. If the containing agent moves to a new location, then its components are no longer accessible.
- Likewise, a module cannot directly access the private data and methods of another module, but needs to do so via an interface.

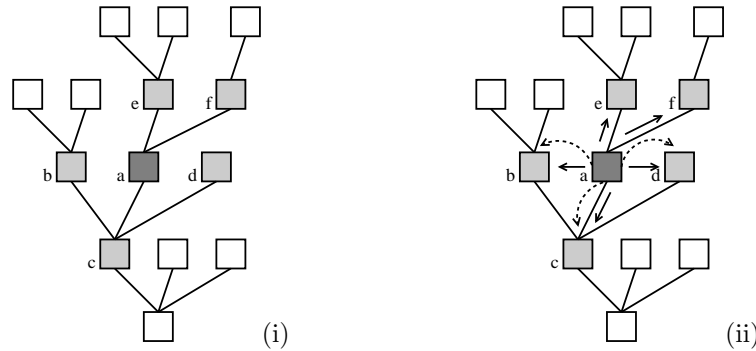


Figure 2.3: Adjacent Components (i) and the Possible Interactions Between Components (ii)

The notion of adjacent components is illustrated in Figure 2.3. According to the figure, a given component in a hierarchy is adjacent to its parent, its siblings and its children. For example, the component a is adjacent to components b, c, d, e and f , where component c is the parent of a , components b and d are its siblings and components e and f are its children. This notion of adjacency places a number of constraints on the way in which components can interact with each other. At best, a component will be able to move into a sibling, move out of its parent, or communicate with its siblings, parent or children. Figure 2.3 illustrates the constraints that adjacency places on the interaction between components. The component a can either move into one of its siblings b, d , move out of its parent, or communicate with adjacent components b, c, d, e, f , where communication and migration are represented by solid and dotted arrows, respectively.

Communication and migration can be regarded as two distinct forms of interaction. From a security perspective, receiving a message involves potentially less risk than receiving an agent containing program code. Therefore, at a high level of abstraction there should be a logical distinction between these two forms of interaction. Some specification models use courier agents

to describe remote communication between machines at a low level of abstraction [21, 83]. In these models, an agent transports a given message from from one machine to another, and then sends the message locally when it reaches its destination. However, even at the lowest implementation level it can be useful to distinguish between communication and migration, since sending a message involves significantly less overhead than sending a migrating agent, which needs to be stopped, moved through the network and re-started, and in many cases will need to be scanned on arrival for potential malicious behaviour. Therefore, it seems logical to distinguish between communication and migration at both high and low levels of abstraction.

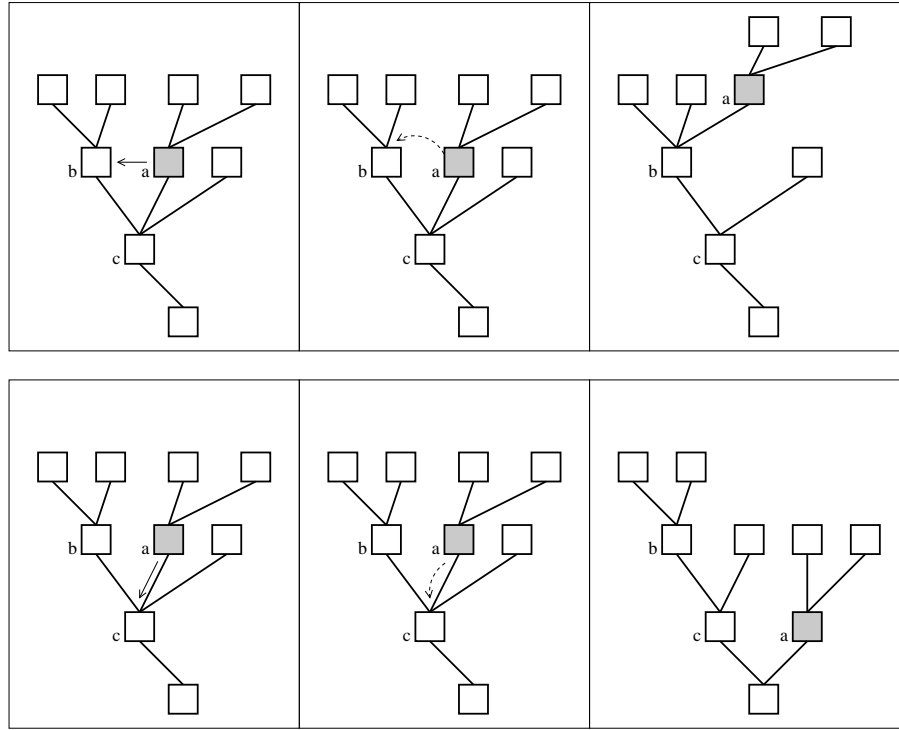


Figure 2.4: Symmetric Communication and Migration

Communication and migration should be symmetric, in the sense that if an agent can try to enter or leave a given component then it should also be able to try to communicate with this component, and vice-versa. Such symmetry is necessary in order for an agent to negotiate permission to enter a sibling or leave its parent. Note that the symmetry only refers to the ability to *attempt* a communication or migration, since a given interaction should only occur if both parties agree. Figure 2.4 illustrates this desired symmetry between communication and migration. In the first sequence of pictures, the component a can first communicate with its sibling b before moving into b . Similarly, in the second sequence the component a can first communicate with its parent c before moving out of c .

2.2.3 Communication Between Mobile Application Components

The Channel Ambient calculus assumes that components communicate over named channels, that peer components can communicate directly, and that a receiver component does not generally know the identity of the sender.

Components communicate over named channels:

- In some of the most widely-used Internet protocols, such as TCP/IP and UDP, components in the network communicate with each other using channels or *ports*. For example, during an ftp transfer a client machine communicates with a server machine on port 21. In parallel, another client may be able to initiate a telnet connection with the server on port 23. In general a given server, identified by its IP address, can provide multiple concurrent services on different ports.
- Similarly, a given module often provides multiple concurrent services, where the name of the service corresponds to the notion of a channel. For example, multiple components can invoke different methods provided by a given module in parallel.

Peer components can communicate directly:

- The TCP/IP protocol allows direct communication between peer machines. Most peer-to-peer applications rely on direct communication between peers, including many well-known content-distribution applications.
- Modules also allow direct communication between peers. In a given application, modules at the same level in a hierarchy can communicate with each other directly. When a new module is downloaded, it is generally accessible to multiple peer components.

The receiver component does not generally know the identity of the sender:

- When a TCP/IP connection is established, the sender specifies the identity (IP address) of the receiver component together with the communication channel (port number). In contrast, the receiver does not specify the IP address of the sender, but only specifies the port number. This communication model allows for a given machine to provide public services for which the identity of the sender is not known beforehand.
- Similarly, when a module is developed the identity of potential client modules is not generally known beforehand, particularly in the case of applications that are continually being upgraded.

Note that although the receiver of a message does not know the identity of the sender in the general case, private channels can be used in order to limit the scope of a given service provided

by a machine or a module. In this sense, a channel acts as a key, which can be publicly available to many users, or only given to a single user, such as a password for a user account.

2.2.4 Migration of Mobile Application Components

The Channel Ambient calculus assumes that components can move autonomously, that components move in and out of each other over channels, and that the receiver of a component does not generally know the origin of the component.

Mobile agents in an application can move autonomously. Autonomy is part of the fundamental definition of a mobile agent, as described in [64]. This does not imply that an agent can move around without constraints, but it does mean that an agent can choose its next destination and has full knowledge of the set of moves it is able to perform. Such autonomous mobility is often referred to as *subjective* mobility. This contrasts with the more passive *objective* mobility, in which an agent does not choose its next destination, but the choice is made entirely by its external environment. The terminology of subjective and objective mobility was first used in [21], in the context of mobile ambients.

Mobile agents are assumed to move in and out of each other over channels. As with communication, channels provide a flexible means of regulating agent migration. A channel is like a key: many agents can use the same key in order to gain access to a machine, or each agent can have its own private key. Not only does an agent need a key to enter a machine, but it also needs a key to leave the machine. In general, requiring a key to leave a location is common practice in many environments. For example, requiring a key to leave a building prevents unauthorised users from departing with equipment or documents that are expected to remain in the building. From a mobile application perspective, an agent can be prevented from leaving a machine without the proper authorisation, in order to preserve the confidentiality of data that should only be accessible from that particular machine. Many computer systems use *personal firewalls* to prevent unauthorised programs or agents from accessing the Internet. This prevents such programs from consuming bandwidth without the consent of the user, and also prevents spyware from sending out covert information about the current state of the user's machine. Thus, it is equally important to regulate the departure of agents from a machine as it is to regulate their arrival. Finally, at an implementation level the TCP/IP protocol requires two machines to interact over a common channel. If agent migration is to be implemented based on such a protocol, migration will also need to take place over channels.

As with communication, the receiver of a component does not generally know the origin of the component. In the case of an agent migrating to a machine using the TCP/IP protocol, the machine is required to accept an agent on a given channel, without knowing the origin of the incoming agent.

2.3 Definition

This section presents the definition of the Channel Ambient calculus, based on the design principles outlined previously. A corresponding graphical representation for the calculus is also presented. The section is structured as follows:

Subsection 2.3.1 presents the syntax of calculus processes.

Subsection 2.3.2 presents the reduction rules of the calculus, which describe how a calculus process can be executed.

Subsection 2.3.3 presents the structural congruence rules of the calculus, which describe what it means for two calculus processes to be equal.

Subsection 2.3.4 presents the substitution rules of the calculus, which describe how one free value can be replaced by another inside a process.

Subsection 2.3.5 defines a number of convenient syntax abbreviations.

2.3.1 Syntax of Calculus Processes

The Channel Ambient calculus uses the notion of an *ambient*, first presented in [21], 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 keeping with the assumptions outlined in the previous section, 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.

The syntax of the Channel Ambient calculus is presented in Definition 2.3.1 in terms of processes P, Q, R , actions α 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 2.3.2.

The processes P, Q, R of the calculus have the following meaning:

Null 0 does nothing and is used to represent the end of a process.

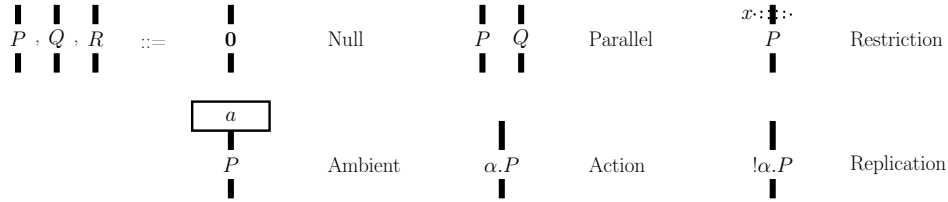
Parallel $P \mid Q$ executes process P in parallel with process Q .

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.

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

Definition 2.3.1. Syntax of CA



Definition 2.3.2. Graphical Syntax of CA

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

Replication $!\alpha.P$ repeatedly tries to perform the action α 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 [42], 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 α 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 $\mathbf{in} a \cdot x$ tries to enter a sibling ambient over channel x .

Leave 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 .

In keeping with the assumptions outlined in the previous section, there is a noticeable symmetry between the actions for communication and migration in the calculus syntax.

2.3.2 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 2.3.3, where $P_{\{v/u\}}$ assigns the value v to the 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 2.3.4:

- (2.1) 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 $x^\dagger(u) . Q$, then the value v can be sent to ambient b along channel x , and assigned to the value u in process Q .
- (2.2) An ambient can send a value to its parent over a channel. If an ambient a contains a parent output $x^\dagger \langle v \rangle . P$, 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 .
- (2.3) 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 $\overline{\text{in}} x . Q$, then a can enter b over channel x .
- (2.4) 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 $\overline{\text{out}} x . Q$, then a can leave its parent over channel x .
- (2.5) 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 .
- (2.6) 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 .
- (2.7) 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 .
- (2.8) 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' .

$$a \boxed{b \cdot x \langle v \rangle . P \mid P'} \mid b \boxed{x^\dagger(u) . Q \mid Q'} \longrightarrow a \boxed{P \mid P'} \mid b \boxed{Q_{\{v/u\}} \mid Q'} \quad (2.1)$$

$$a \boxed{x^\dagger \langle v \rangle . P \mid P'} \mid x(u) . Q \longrightarrow a \boxed{P \mid P'} \mid Q_{\{v/u\}} \quad (2.2)$$

$$a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\text{in } x . Q \mid Q'} \longrightarrow b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \quad (2.3)$$

$$b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \overline{\text{out } x} . Q \mid Q'} \longrightarrow b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \quad (2.4)$$

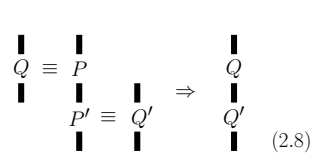
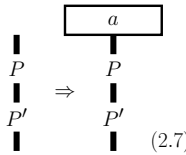
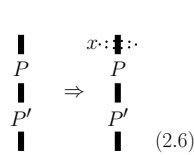
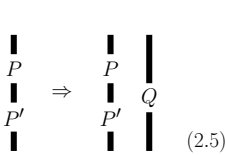
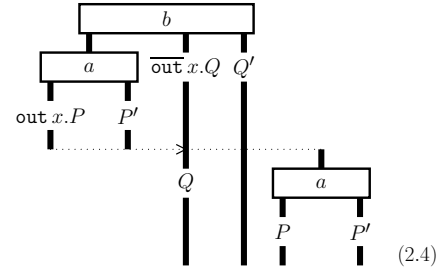
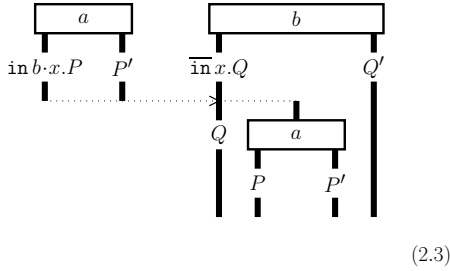
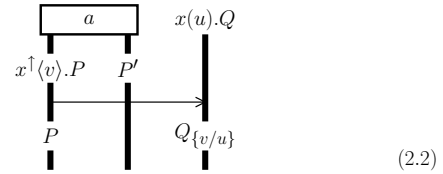
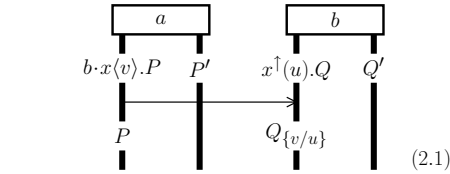
$$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q \quad (2.5)$$

$$P \longrightarrow P' \Rightarrow \nu n P \longrightarrow \nu n P' \quad (2.6)$$

$$P \longrightarrow P' \Rightarrow a \boxed{P} \longrightarrow a \boxed{P'} \quad (2.7)$$

$$Q \equiv P \longrightarrow P' \equiv Q' \Rightarrow Q \longrightarrow Q' \quad (2.8)$$

Definition 2.3.3. Reduction in CA



Definition 2.3.4. Graphical Reduction in CA

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.

2.3.3 Structural Congruence Rules for Equating Calculus Processes

$$\mathbf{0} \mid P \equiv P \quad (2.9)$$

$$P \mid Q \equiv Q \mid P \quad (2.10)$$

$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad (2.11)$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P \equiv \alpha.(P \mid !\alpha.P) \quad (2.12)$$

$$\nu n \mathbf{0} \equiv \mathbf{0} \quad (2.13)$$

$$\nu n \nu m P \equiv \nu m \nu n P \quad (2.14)$$

$$n \notin \text{fn}(Q) \Rightarrow (\nu n P) \mid Q \equiv \nu n (P \mid Q) \quad (2.15)$$

$$a \neq n \Rightarrow a[\nu n P] \equiv \nu n a[P] \quad (2.16)$$

$$\nu a a[\mathbf{0}] \equiv \mathbf{0} \quad (2.17)$$

Definition 2.3.5. Structural Congruence in CA

$$\begin{array}{c} \text{I} \\ \mathbf{0} \end{array} \begin{array}{c} \text{I} \\ P \end{array} \equiv \begin{array}{c} \text{I} \\ P \end{array} \quad (2.9)$$

$$\begin{array}{c} n \cdots \vdots \vdots \\ m \cdots \vdots \vdots \\ \text{I} \\ P \end{array} \equiv \begin{array}{c} m \cdots \vdots \vdots \\ n \cdots \vdots \vdots \\ \text{I} \\ P \end{array} \quad (2.14)$$

$$\begin{array}{c} \text{I} \\ P \end{array} \begin{array}{c} \text{I} \\ Q \end{array} \equiv \begin{array}{c} \text{I} \\ Q \end{array} \begin{array}{c} \text{I} \\ P \end{array} \quad (2.10)$$

$$n \notin \text{fn}(Q) \Rightarrow \begin{array}{c} n \cdots \vdots \vdots \\ \text{I} \\ P \end{array} \begin{array}{c} \text{I} \\ Q \end{array} \equiv \begin{array}{c} n \cdots \vdots \vdots \vdots \vdots \\ \text{I} \\ P \end{array} \begin{array}{c} \text{I} \\ Q \end{array} \quad (2.15)$$

$$\begin{array}{c} \text{I} \\ P \end{array} \begin{array}{c} \text{I} \\ Q \end{array} \begin{array}{c} \text{I} \\ R \end{array} \equiv \begin{array}{c} \text{I} \\ P \end{array} \begin{array}{c} \text{I} \\ Q \end{array} \begin{array}{c} \text{I} \\ R \end{array} \quad (2.11)$$

$$a \neq n \Rightarrow \begin{array}{c} \boxed{a} \\ n \cdots \vdots \vdots \\ \text{I} \\ P \end{array} \equiv \begin{array}{c} n \cdots \vdots \vdots \\ \boxed{a} \\ \text{I} \\ P \end{array} \quad (2.16)$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow \begin{array}{c} \text{I} \\ !\alpha.P \end{array} \equiv \begin{array}{c} \text{I} \\ \alpha.(P \mid !\alpha.P) \end{array} \quad (2.12)$$

$$\begin{array}{c} n \cdots \vdots \vdots \\ \boxed{a} \\ \text{I} \\ \mathbf{0} \end{array} \equiv \begin{array}{c} \text{I} \\ \mathbf{0} \end{array} \quad (2.17)$$

$$\begin{array}{c} n \cdots \vdots \vdots \\ \text{I} \\ \mathbf{0} \end{array} \equiv \begin{array}{c} \text{I} \\ \mathbf{0} \end{array} \quad (2.13)$$

Definition 2.3.6. Graphical 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 2.3.5 and are mostly standard, apart from the rule for replication (2.12). As usual, structural congruence is the least congruence that satisfies the rules in Definition 2.3.5. A corresponding graphical representation of the structural congruence rules is presented in Definition 2.3.6:

- (2.9) - (2.11) Processes are equal up to re-ordering of parallel compositions. The null process can be composed in parallel with a given process, and parallel composition is commutative and associative.
- (2.12) Processes are equal up to expansion of replicated actions. A replicated action $!\alpha.P$ can be expanded to the action α , followed by the process P in parallel with the replicated action $!\alpha.P$. This has the effect of continually executing the action $\alpha.P$. The expansion also ensures that only a single copy of $\alpha.P$ can interact at a time. This is useful from an implementation perspective, and differs from the standard rule for replication, $!\alpha.P \equiv \alpha.P \mid !\alpha.P$, which allows an infinite number of copies of $\alpha.P$ to be created in parallel. Note that the replicated action can only be expanded if the free values of α are not contained in the set of bound values of α . This ensures that channel names are not captured during the expansion of replicated inputs.
- (2.13) - (2.16) Processes are equal up to re-ordering of restricted names. A restricted name can be added to the null process, and successive restricted names can be permuted. The scope of a restricted name n can be extended over a parallel process Q if the name n is not free in Q . The scope of a restricted name n can be extended past an ambient a if n is different from a .
- (2.17) An empty ambient with a restricted name is equal to the null process. This rule is not essential, but is useful for garbage-collecting unused ambients. A similar effect can be achieved by defining an appropriate notion of process equivalence in which $\nu a a[\mathbf{0}] \simeq \mathbf{0}$.

2.3.4 Substitution of Free Values Inside Processes

Substitution in the Channel Ambient calculus is used to replace one free value by another. Definition 2.3.7 describes the application P_σ of a substitution σ to a process P , where σ is a substitution that maps a given value to another (different) value, and v_σ applies the substitution σ 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 . By definition, the expression $\sigma \setminus S$ removes each value in the set S from the domain of σ . If v is not in the domain of σ 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 σ .

The set of free values $\text{fn}(P)$ of a process P in the Channel Ambient calculus is described in Definition 2.3.8. 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

$$\begin{array}{ll}
\mathbf{0}_\sigma & \triangleq \mathbf{0} \\
(P \mid Q)_\sigma & \triangleq P_\sigma \mid Q_\sigma \\
(\nu n P)_\sigma & \triangleq \nu n P_{\sigma \setminus \{n\}} \\
(a \boxed{P})_\sigma & \triangleq a_\sigma \boxed{P_\sigma} \\
(\alpha.P)_\sigma & \triangleq \alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)} \\
(!\alpha.P)_\sigma & \triangleq !(\alpha.P)_\sigma \\
a \cdot x \langle v \rangle_\sigma & \triangleq a_\sigma \cdot x_\sigma \langle v_\sigma \rangle \\
x^\uparrow \langle v \rangle_\sigma & \triangleq x_\sigma^\uparrow \langle v_\sigma \rangle \\
x(u)_\sigma & \triangleq x_\sigma(u) \\
x^\uparrow(u)_\sigma & \triangleq x_\sigma^\uparrow(u) \\
(\text{in } a \cdot x)_\sigma & \triangleq \text{in } a_\sigma \cdot x_\sigma \\
(\text{out } x)_\sigma & \triangleq \text{out } x_\sigma \\
(\overline{\text{in}} x)_\sigma & \triangleq \overline{\text{in}} x_\sigma \\
(\overline{\text{out}} x)_\sigma & \triangleq \overline{\text{out}} x_\sigma
\end{array}$$

Definition 2.3.7. Substitution in CA

$$\begin{array}{ll}
\text{fn}(\mathbf{0}) & \triangleq \emptyset \\
\text{fn}(P \mid Q) & \triangleq \text{fn}(P) \cup \text{fn}(Q) \\
\text{fn}(\nu n P) & \triangleq \text{fn}(P) \setminus \{n\} \\
\text{fn}(a \boxed{P}) & \triangleq \text{fn}(P) \cup \{a\} \\
\text{fn}(\alpha.P) & \triangleq \text{fn}(\alpha) \cup (\text{fn}(P) \setminus \text{bn}(\alpha)) \\
\text{fn}(!\alpha.P) & \triangleq \text{fn}(\alpha) \cup (\text{fn}(P) \setminus \text{bn}(\alpha)) \\
\text{fn}(a \cdot x \langle v \rangle) & \triangleq \{a, x, v\} \\
\text{fn}(x^\uparrow \langle v \rangle) & \triangleq \{x, v\} \\
\text{fn}(x(u)) & \triangleq \{x\} \\
\text{fn}(x^\uparrow(u)) & \triangleq \{x\} \\
\text{fn}(\text{in } a \cdot x) & \triangleq \{a, x\} \\
\text{fn}(\text{out } x) & \triangleq \{x\} \\
\text{fn}(\overline{\text{in}} x) & \triangleq \{x\} \\
\text{fn}(\overline{\text{out}} x) & \triangleq \{x\}
\end{array}$$

Definition 2.3.8. Free Values of CA

α is defined as $\{u\}$ for $\alpha = x(u)$ and $\alpha = x^\uparrow(u)$, and \emptyset otherwise. As usual, processes of the calculus are assumed to be equal up to renaming of bound values.

2.3.5 Syntax Abbreviations for Frequently Used Processes

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

Definition 2.3.9. 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 α 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 2.3.9:

Local Output $x\langle v \rangle.P$ tries to send a value v on channel x locally. This can be encoded by creating a private ambient z , which sends the value v on channel x to the parent and then sends a synchronisation message to the parent on channel z , resulting in the execution of process P .

Child Output $a/x\langle v \rangle.P$ tries to send a value v on channel x to a child ambient a . This can be encoded by creating a private ambient z , which sends the value v on channel x to the sibling ambient a and then sends a synchronisation message to the parent on channel z , resulting in the execution of process P .

Note that the structural congruence rule $\nu a a[\mathbf{0}] \equiv \mathbf{0}$ 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.

2.4 Distributed Model

This section describes how the Channel Ambient calculus can be used to specify mobile applications for networks that support the TCP/IP protocol. The section is structured as follows:

Subsection 2.4.1 describes a number of constraints that can be placed on the syntax of the calculus, in order to more accurately model the properties of TCP/IP networks.

Subsection 2.4.2 describes how the calculus can be used to model the hierarchical infrastructure of networks.

Subsection 2.4.3 describes how the reduction rules of the calculus can be used to model the execution of mobile applications in a network.

Subsection 2.4.4 describes a more flexible network model in which multiple sites can share the same IP address.

2.4.1 Constraining 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 thesis, 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 abstract away from many of the details 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.

$P^g, Q^g ::=$	$\mathbf{0}$	$P^s, Q^s ::=$	$\mathbf{0}$	$\alpha^s ::=$	$s \cdot x \langle v \rangle$
	$ P^g \mid Q^g$		$ P^s \mid Q^s$		$ x^\dagger \langle v \rangle$
	$ \nu n P^g$		$ \nu n P^s$		$ x(u)$
	$ g \boxed{P^g}$		$ g \boxed{P^g}$		$ x^\dagger(u)$
	$ \alpha.P^g$		$ s \boxed{P^s}$		$ \overline{\text{in}} x$
	$!\alpha.P^g$		$ \alpha^s.P^s$		$ \overline{\text{out}} x$
			$!\alpha^s.P^s$		

Definition 2.4.1. Constraints on the Syntax of CA in a Distributed Setting

Definition 2.4.1 describes the constraints that can be placed on the syntax of the Channel Ambient calculus, in order to model the behaviour of sites and agents:

Sites A process P^s 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 P^g 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.

Note that agents can also be used to model mobile devices that move around between sites. In this setting, each site represents a logical domain that is able to accommodate these mobile devices, such as an airport lounge or a conference room. Various other entities may also be present on the network, such as devices with mobile addresses, but these require additional constraints that are beyond the scope of the current model.

2.4.2 Using Sites to Model a Hierarchical Network Topology

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. 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.

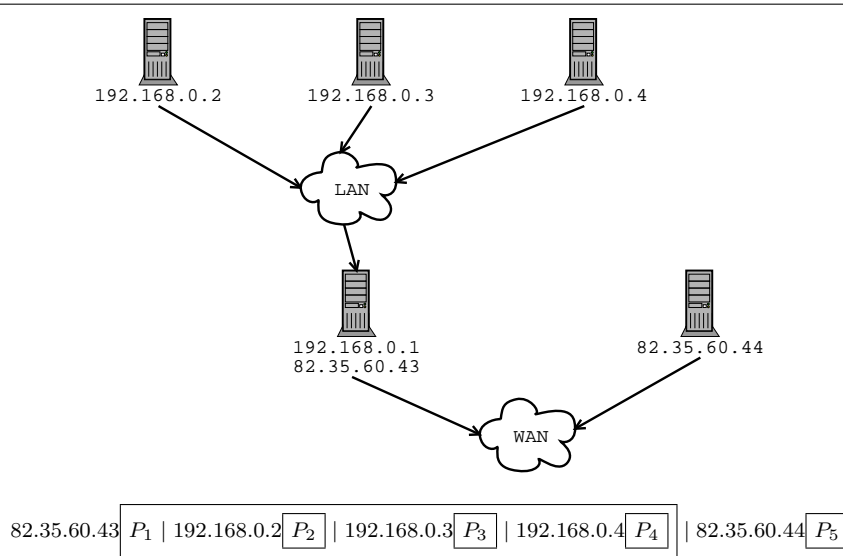


Figure 2.5: Hierarchical TCP/IP Networks and their corresponding calculus representation.

An example of how sites can be used to model the topology of Local and Wide Area Networks is illustrated in Figure 2.5. 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 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.

2.4.3 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. Although the TCP/IP protocol itself is relatively complex, an abstract model of the protocol can be defined by making the following simplifying assumptions:

- During a normal session of the protocol, a given host with address IP_1 sends data to a host with address IP_2 on a port number n .
- If any errors occur during the transmission then the session is aborted.

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:

$$\begin{array}{l} 82.35.60.43 \boxed{!21^\dagger(args).Ftp \mid !23^\dagger(args).Telnet \mid Server} \\ \mid 82.35.60.44 \boxed{82.35.60.43 \cdot 21 \langle v \rangle . P \mid Client} \mid Network \end{array}$$

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.

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 [2, 9].

Table 2.1 describes how the reduction rules of the Channel Ambient calculus can be used to model the execution of mobile applications on TCP/IP networks. The table takes into account the various constraints on the syntax of calculus processes described in the previous section. 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.

Rule	Application in a distributed setting
(2.1)	Ambient a sends value v to site b on port x
(2.2)	Agent a sends value v to parent site on channel x
	Site a sends value v to parent site on port x
(2.3)	Agent a enters site b on port x
(2.4)	Agent a leaves site b on channel x
(2.5)	A process P can execute independently of other processes in the network
(2.6)	A process P can reduce inside a restriction
(2.8)	Processes can execute in any order.
(2.7)	The contents of a site a can execute independently of other sites in the network

Table 2.1: Using the reduction rules of CA in a distributed setting.

Interestingly, the calculus can also be used to model the loss of agents and messages due to network failure. Network failure is distinct from network delay and disconnection in that it results in the loss of data. Note that delay itself is not observable in the calculus, since there is no explicit notion of time and there is no loss of data during a delay. Furthermore, if all the sites in a given network are assumed to have unique IP addresses then network disconnection can be also regarded as a delay, which may be unbounded. This is because, according to the reduction rules of the calculus, a given message or agent that is destined for a disconnected site will wait indefinitely until the site reconnects. If necessary, more precise notions of disconnection can be modelled using site migration. For example, a given site that disconnects from a network can be modelled as a site moving out of the network into some isolated location.

Network failure is typically caused by an error in a TCP/IP session. In the general case a session error can be detected using various protocol signals, and the session can then be restarted. In some instances the error is not detected and any data being transmitted is lost, resulting in a network failure. For example, a failure can occur if there is an error in a TCP/IP session and the error signal itself is also lost. The sender may assume that the message was transmitted successfully and the TCP/IP session will not be restarted, resulting in loss of data.

Communication failure can occur when an ambient tries to send a message to a remote site. The possibility of communication failure to a site s on a channel x can be modelled by placing a replicated input on channel x inside the site s :

$$s \boxed{S \mid !x^\dagger(m).\mathbf{0}}$$

If an ambient tries to send a message to the site s on channel x the communication may be intercepted, resulting in the loss of the message:

$$s \boxed{S \mid x^\dagger(m).Q \mid !x^\dagger(m).\mathbf{0}} \mid a \boxed{A \mid s.x\langle n \rangle.P} \longrightarrow s \boxed{S \mid x^\dagger(m).Q \mid !x^\dagger(m).\mathbf{0} \mid \mathbf{0}} \mid a \boxed{A \mid P}$$

Similarly, migration failure can occur when an agent tries to enter a remote site. The possibility of a migration failure of an agent a can be encoded by adding an enter to a *trash* location inside the agent a :

$$a \boxed{A \mid \text{in trash}.\text{fail}.\mathbf{0}}$$

The *trash* location can be placed in parallel with other sites in the network, such that agents in transit between sites may enter this location, resulting in the loss of the entire agent:

$$\begin{aligned} & \text{trash} \boxed{!\overline{\text{in}} \text{fail}.\mathbf{0}} \mid s \boxed{S \mid \overline{\text{in}} x.Q} \mid a \boxed{A \mid \text{in } s.x.P \mid \text{in trash}.\text{fail}.\mathbf{0}} \\ \longrightarrow & \text{trash} \boxed{!\overline{\text{in}} \text{fail}.\mathbf{0}} \mid a \boxed{A \mid \text{in } s.x.P \mid \mathbf{0} \mid \mathbf{0}} \mid s \boxed{S \mid \overline{\text{in}} x.Q} \end{aligned}$$

Such flexibility in the calculus makes it possible to choose when to model or ignore failures in the specification of mobile applications. In many cases it is convenient to prove the correctness of a specification in the absence of failure, and then add failure to the specification in order to prove a number of safety properties. It may also be useful to assume that certain reliable links never fail, and only use the above encodings on a small number of less reliable channels, such as those that make use of an intermittent wireless connection. Note that the ability to model communication and migration failure in this way is by no means particular to the Channel Ambient calculus, but is still worth bearing in mind when building a model of a mobile application.

2.4.4 Allowing Multiple Sites to Share the Same IP Address

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 device on the network corresponds to a separate site, since a given device is usually assigned a single IP address per network. An alternative, more flexible approach is to map each *socket address* to a site, where a socket address consists of an IP address and a port number. This

allows a given device to contain multiple sites, where the number of sites is limited only by the number of available ports. Arbitrary channel names can then be used to interact with a given site, resulting in a significantly more flexible and dynamic 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.

An example of how multiple sites can share the same IP address is illustrated in Figure 2.6. In this example, the LAN contains two sites that share the same IP address 192.168.0.2, one using port 3000 and the other using port 3001. A site with a given IP address and port number n is written IP : n . Note that the structure of the name is not significant at the calculus level, and is treated as a simple sequence of characters.

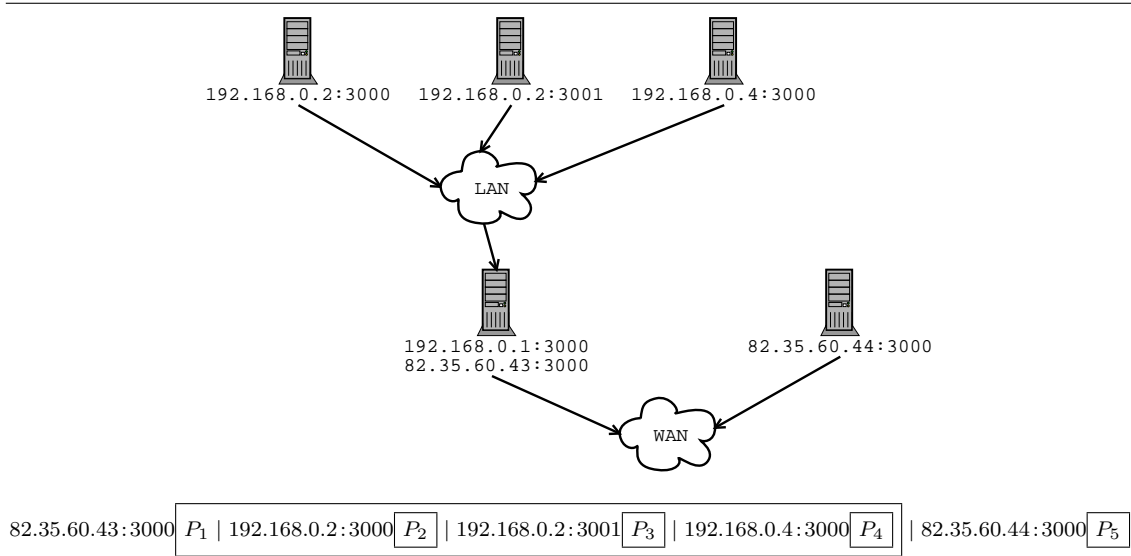


Figure 2.6: Multiplexed TCP/IP Networks and their corresponding calculus representation.

2.5 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. The section is structured as follows:

Subsection 2.5.1 describes various situations in which mobile agents can be used to monitor resources on a remote server.

Subsection 2.5.2 presents an informal specification of a resource monitoring application, using various informal graphical notations

Subsection 2.5.3 presents a formal specification of the application, expressed as a process of the Channel Ambient calculus.

Subsection 2.5.4 uses the reduction rules of the Channel Ambient calculus to formally specify an execution scenario for the application.

2.5.1 Examples of Resource Monitoring Applications

Mobile agents can be used to monitor resources on a remote server in order to enhance application performance. For example, the server can be a stock market trading site, the resource can be a list of stock prices published by the server and the client can be a trader, who wishes to respond to changes in stock prices by buying or selling shares accordingly. Without the use of mobile agent technology, the client is obliged to monitor the stock prices from a remote machine, and respond by sending remote messages to the server. In cases where the server becomes congested, for example during peak trading times, the client may experience a delay in communicating with the server and may not be able to respond quickly enough to price changes. In particular, competing clients that are closer to the server or have a better connection will be able to respond more quickly. To overcome such competition, the client can program an agent with sophisticated trading algorithms, and then send this agent to the server. The agent will be able to respond immediately to changes in stock prices on the server itself, without being affected by network congestion. In addition, the client machine no longer needs to remain connected to the server. This is particularly relevant for lightweight clients with intermittent connection to the server, such as Internet-enabled phones and handheld devices. A trader client could send a mobile agent to the trading site from a handheld device, before disconnecting the device to board a transatlantic flight or an underground train.

Another example in which a mobile agent can be used to monitor resources on a remote server comes from the world of electronic commerce. In this example, the server can be an Internet auction site, the resource can be an item for sale and the client can be a registered user of the site, who wishes to negotiate the purchase of the item. As with the previous example, the client can program an agent with sophisticated bartering algorithms and perhaps a maximum purchase price. The client can then send the agent to the server and subsequently disconnect. In a possible scenario the item for sale could be a piece of sports equipment such as a snowboard, and the client could also wish to program the agent to book a winter sports holiday on successful completion of the purchase. The agent could even be programmed to perform a whole range of tasks such as:

1. Negotiate the purchase of the snowboard
2. Check the weather forecast
3. Reserve an airline ticket to the destination with the best conditions

4. Reserve a suitable hotel at the appropriate destination
5. Prepare a report and email this to the client on completion of the required tasks.

The client can then decide whether to confirm the final purchases after scrutinizing the report. In this sense, a mobile agent can act very much like a Personal Assistant. The client can give a list of instructions to the agent, and then focus on other tasks. From time to time, the client can monitor the performance of the agent, but the bulk of the work is done by the agent itself.

2.5.2 Informal Specification of an Application

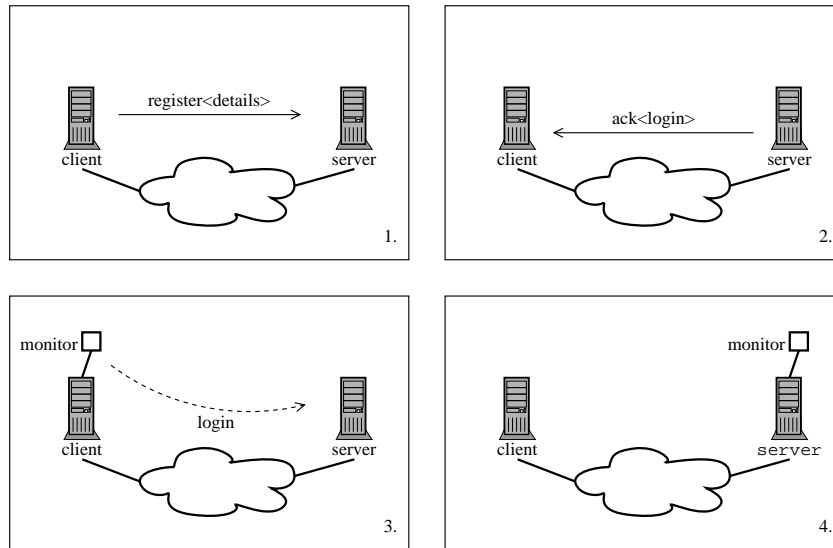


Figure 2.7: Cartoon Scenario

Figure 2.7 uses a sequence of pictures to describe an execution scenario for the resource monitoring application:

1. The client registers with the server by sending its registration details over a registration channel.
2. The server processes the registration request, and sends a login to the client on an acknowledgement channel.
3. The client creates a monitor agent, which moves to the server using its newly acquired login.
4. The agent monitors the resource on the server.

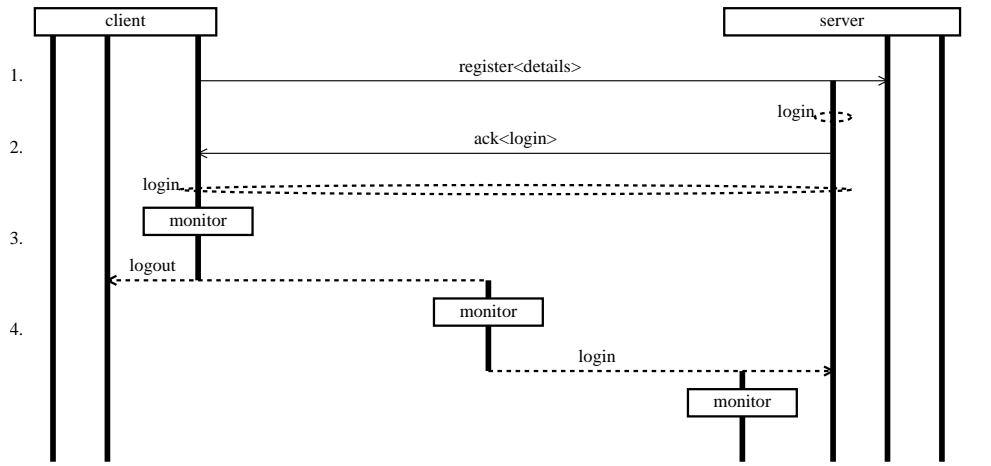


Figure 2.8: Graphical Scenario

Figure 2.8 uses an informal graphical notation to describe a refined version of the previous scenario. The graphical notation is a lightweight version of the graphical Channel Ambient calculus, in which the states of processes are not shown and the interactions between processes are annotated with channel names and values:

1. The client registers with the server by sending its registration details over a registration channel.
2. The server creates a new login channel, and sends the login channel to the client on an acknowledgement channel.
3. The client creates a monitor agent, which leaves on the logout channel.
4. The monitor agent enters the server on the login channel, and monitors the resource on the server.

2.5.3 Formal Calculus Specification

$$\begin{array}{l}
 \boxed{\text{client} \mid \text{server} \cdot \text{register} \langle \text{client}, \text{ack} \rangle . \text{ack}^\dagger(x) . \text{monitor} \mid \text{out } \text{logout} . \text{in } \text{server} . x . P} \\
 \mid \boxed{\text{out } \text{logout} . Q} \mid C \\
 \mid \boxed{\text{server} \mid \text{register}^\dagger(c, k) . \nu \text{login } c . k \langle \text{login} \rangle . \text{in } \text{login} . R} \mid S \mid N
 \end{array}$$

Figure 2.9: Calculus Specification

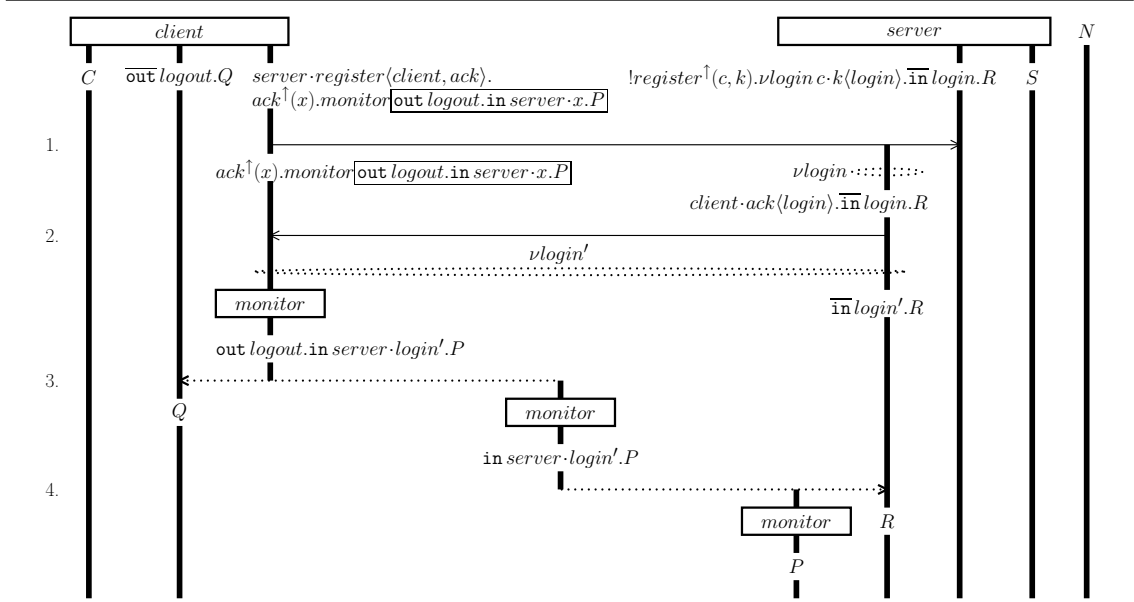
Figure 2.9 presents a formal specification of the resource monitoring application, expressed as a process of the Channel Ambient calculus:

- 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 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.

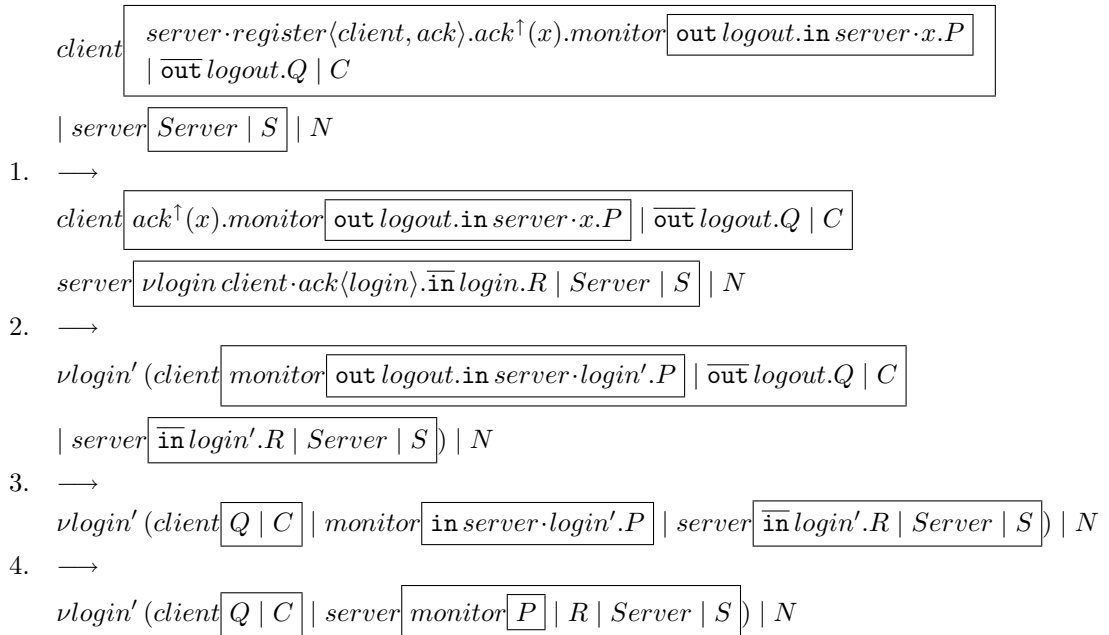
2.5.4 Calculus Execution Scenario

Figure 2.10 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 2.11:

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

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

Figure 2.11: 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 $login' \notin \text{fn}(Q, S, C)$.

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 the following section.

2.6 Security Properties

This section illustrates how various security mechanisms developed for related calculi can be applied to the Channel Ambient calculus, in order to reason about the security properties of mobile applications. Additional proof details are given in Appendix A.1. The section is structured as follows:

Subsection 2.6.1 describes a number of basic safety properties of the Channel Ambient calculus, in order to prevent runtime errors during the execution of mobile applications.

Subsection 2.6.2 describes how a type system for channel communication in the π -calculus can be applied to the Channel Ambient calculus, in order to ensure that values are always sent and received on channels of the correct type.

Subsection 2.6.3 describes how linearity constraints can be placed on the syntax of the calculus, in order to prevent impersonation of ambients.

2.6.1 Proving Safety to Prevent Runtime Errors

Safety ensures that the Channel Ambient calculus always produces a valid process after each execution step. This ensures that the calculus does not produce any runtime errors when executing a

given process, where a runtime error corresponds to a process reaching an undefined state. Calculus execution is defined in terms of structural congruence, substitution and reduction. Therefore, in order to prove the safety of the calculus it is necessary to prove that structural congruence, substitution and reduction are safe.

Structural Safety ensures that the result of applying a structural congruence rule is always a valid calculus process. According to Lemma 2.6.1 (Structural Safety), if a process P is structurally congruent to a process Q then Q is a valid calculus process.

Lemma 2.6.1. (*Structural Safety*) $\forall P. P \in \text{CA} \wedge P \equiv P' \Rightarrow P' \in \text{CA}$

Proof. By induction on Definition 2.3.5 of structural congruence in CA. \square

Substitution Safety ensures that the result of applying a substitution is always a valid calculus process. According to Lemma 2.6.2 (Substitution Safety), if a substitution σ is applied to a process P then the result is a valid calculus process.

Lemma 2.6.2. (*Substitution Safety*) $\forall \sigma, P. P \in \text{CA} \Rightarrow P_\sigma \in \text{CA}$

Proof. By induction on Definition 2.3.7 of substitution in CA. \square

Finally, Reduction Safety ensures that the result of a reduction is always a valid calculus process. According to Theorem 2.6.3 (Reduction Safety), if a process P can reduce to P' then P' is a valid calculus process.

Theorem 2.6.3. (*Reduction Safety*) $\forall P. P \in \text{CA} \wedge P \longrightarrow P' \Rightarrow P' \in \text{CA}$

Proof. By Lemma 2.6.1 (Structural Safety), Lemma 2.6.2 (Substitution Safety) and by induction on Definition 2.3.3 of reduction in CA. \square

2.6.2 Using Channel Types to Ensure Reliable Communication

Interaction in the Channel Ambient calculus takes place over channels. Therefore, various type systems for channel interaction in the π -calculus can also be applied to Channel Ambients. One example is the monomorphic type system for the π -calculus presented in [74], which ensures that values are always sent and received on channels of the correct type. This type system can be applied to the Channel Ambient calculus by defining additional rules for the creation of ambients and for the migration of ambients over channels.

The syntax of types in the Channel Ambient calculus is described in Definition 2.6.4, where a type τ can be an ambient amb , a channel $\langle \tau \rangle$ carrying values of a given type, a migration channel $\langle \text{mv} \rangle$ or the type of a base value δ . The types of the free values in a given process are recorded in a corresponding type context Γ , which consists of a sequence of zero or more bindings

$\tau ::=$	amb	Ambient	$\Gamma ::=$	$v_1 : \tau_1, \dots, v_N : \tau_N$	Type Context
		$\langle \tau \rangle$			Communication Channel
		$\langle mv \rangle$			Migration Channel
		δ			Base Value

Definition 2.6.4. Syntax of Types in CA

$$\begin{aligned}
 (\Gamma, v : \tau)(v) &\triangleq \tau \\
 u \neq v \Rightarrow (\Gamma, u : \tau)(v) &\triangleq \Gamma(v) \\
 ()(v) &\triangleq \text{undefined}
 \end{aligned}$$

Definition 2.6.5. Type Lookup in CA

$$\begin{aligned}
 \Gamma \vdash \mathbf{0} & \quad (2.18) & \frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma, u : \tau \vdash P}{\Gamma \vdash x^\dagger(u).P} & (2.25) \\
 \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} & (2.19) & \frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma, u : \tau \vdash P}{\Gamma \vdash x(u).P} & (2.26) \\
 \frac{\Gamma, n : \tau \vdash P \quad \tau \neq \delta}{\Gamma \vdash \nu n : \tau P} & (2.20) & \frac{\Gamma(a) = \text{amb} \quad \Gamma(x) = \langle mv \rangle \quad \Gamma \vdash P}{\Gamma \vdash \text{in } a.x.P} & (2.27) \\
 \frac{\Gamma(a) = \text{amb} \quad \Gamma \vdash P}{\Gamma \vdash a[\overline{P}]} & (2.21) & \frac{\Gamma(x) = \langle mv \rangle \quad \Gamma \vdash P}{\Gamma \vdash \text{out } x.P} & (2.28) \\
 \frac{\Gamma \vdash \alpha.P}{\Gamma \vdash !\alpha.P} & (2.22) & \frac{\Gamma(x) = \langle mv \rangle \quad \Gamma \vdash P}{\Gamma \vdash \overline{\text{in}} x.P} & (2.29) \\
 \frac{\Gamma(a) = \text{amb} \quad \Gamma(x) = \langle \tau \rangle \quad \Gamma(v) = \tau \quad \Gamma \vdash P}{\Gamma \vdash a.x\langle v \rangle.P} & (2.23) & \frac{\Gamma(x) = \langle mv \rangle \quad \Gamma \vdash P}{\Gamma \vdash \overline{\text{out}} x.P} & (2.30) \\
 \frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma(v) = \tau \quad \Gamma \vdash P}{\Gamma \vdash x^\dagger\langle v \rangle.P} & (2.24)
 \end{aligned}$$

Definition 2.6.6. Typing Rules in CA

$v_1 : \tau_1, \dots, v_N : \tau_N$. By convention, all of the values v_i in a given type context are assumed to be distinct. The expression $\Gamma(v)$ denotes the type associated with the value v in the context Γ , as described in Definition 2.6.5.

The typing rules for the Channel Ambient calculus are described in Definition 2.6.6. The rules are a straightforward extension of the monomorphic type system for the π -calculus presented in [74]. Each rule is of the form $\Gamma \vdash P$, which states that the process P is well-typed in the context Γ . This ensures that the use of free values in the process P is consistent with their type in Γ :

(2.18) The null process $\mathbf{0}$ is always well-typed.

(2.19) A parallel composition $P \mid Q$ is well-typed if the processes P and Q are well-typed.

(2.20) A restriction $\nu n : \tau P$ is well-typed if P is well-typed assuming n is of type τ . Note that

each restriction $\nu n.P$ in the calculus requires an explicit type annotation τ for n , where τ is not a base type.

- (2.21) An ambient $a \boxed{P}$ is well-typed if a is an ambient name and P is well-typed.
- (2.22) A replicated action $!\alpha.P$ is well-typed if the action $\alpha.P$ is well-typed.
- (2.23) A sibling output $a \cdot x \langle v \rangle$ is well-typed if a is an ambient, x is a channel carrying values of type τ and v is of type τ .
- (2.24) A parent output $x^\dagger \langle v \rangle$ is well-typed if x is a channel carrying values of type τ and v is of type τ .
- (2.25) An external input $x^\dagger(u).P$ is well-typed if x is a channel carrying values of type τ and P is well-typed assuming u is of type τ .
- (2.26) An internal input $x(u).P$ is well-typed if x is a channel carrying values of type τ and P is well-typed assuming u is of type τ .
- (2.27) An enter $\text{in } a \cdot x.P$ is well-typed if a is an ambient, x is a migration channel and P is well-typed.
- (2.28) A leave out $x.P$ is well-typed if x is a migration channel and P is well-typed.
- (2.29) An accept $\overline{\text{in}} x.P$ is well-typed if x is a migration channel and P is well-typed.
- (2.30) A release $\overline{\text{out}} x.P$ is well-typed if x is a migration channel and P is well-typed.

The type system satisfies a number of fundamental properties, as described below. Lemma 2.6.7 (Type Weakening) allows a type binding to be added for a value that is unused in a given process. Conversely, Lemma 2.6.8 (Type Strengthening) allows a type binding to be removed for a value that is unused in a given process. Lemma 2.6.9 (Type Substitution) ensures that the type of a process is preserved by substitution of values with the same type, while Lemma 2.6.10 (Type Structure) ensures that the type of a process is preserved by structural congruence. Finally, Lemma 2.6.10 (Subject Reduction) ensures that the type of a process is preserved by reduction. This ensures that well-typed processes remain well-typed after each reduction step, thereby preventing type errors during execution.

Lemma 2.6.7. (*Type Weakening*) $\forall v. \forall P. v \notin \text{fn}(P) \wedge \Gamma \vdash P \Rightarrow \Gamma, v:\tau \vdash P$

Proof. By induction on Definition 2.6.6 of typing in CA. □

Lemma 2.6.8. (*Type Strengthening*) $\forall v. \forall P. v \notin \text{fn}(P) \wedge \Gamma, v:\tau \vdash P \Rightarrow \Gamma \vdash P$

Proof. By induction on Definition 2.6.6 of typing in CA. □

Lemma 2.6.9. (*Type Substitution*) $\forall u \forall v. \forall P. \Gamma(u) = \Gamma(v) \wedge \Gamma \vdash P \Rightarrow \Gamma \vdash P_{\{v/u\}}$

Proof. By induction on Definition 2.6.6 of typing in CA. \square

Lemma 2.6.10. (*Type Structure*)
 1. $\forall P. P \equiv Q \wedge \Gamma \vdash P \Rightarrow \Gamma \vdash Q$
 2. $\forall P. P \equiv Q \wedge \Gamma \vdash Q \Rightarrow \Gamma \vdash P$

Proof. By Lemma 2.6.7 (Type Weakening), Lemma 2.6.8 (Type Strengthening) and by induction on Definition 2.3.5 of structural congruence in CA. \square

Theorem 2.6.11. (*Subject Reduction*) $\forall P. \Gamma \vdash P \wedge P \longrightarrow P' \Rightarrow \Gamma \vdash P'$

Proof. By Lemma 2.6.9 (Type Substitution), Lemma 2.6.8 (Type Strengthening), Lemma 2.6.10 (Type Structure) and by induction on Definition 2.3.3 of reduction in CA. \square

The type system presented in this section is a simple extension of the monomorphic type system presented in [74], which is used as the basis for the core type system in the Pict programming language [58]. Using similar principles, a polymorphic variant of this type system can also be applied to the Channel Ambient calculus. A similar type system is used in the Nomadic Pict programming language, and the benefits of such types for programming mobile applications are highlighted in [83]. Although the typing rules are relatively straightforward, they provide a convenient and powerful mechanism for preventing errors during program execution.

2.6.3 Using Syntactic Constraints to Prevent Ambient Impersonation

The ability to create multiple ambients with the same name can sometimes be seen as a security risk. For example, a message being sent to an ambient a over a channel x can be intercepted by an imposter with the same name a listening on the same channel:

$$b \boxed{a \cdot x \langle n \rangle . Q} \mid a \boxed{x^\uparrow(m) . P} \mid a \boxed{x^\uparrow(m) . Imposter} \longrightarrow b \boxed{Q} \mid a \boxed{x^\uparrow(m) . P} \mid a \boxed{Imposter_{\{n/m\}}}$$

Similarly, an ambient attempting to enter an ambient a over a channel x can become trapped by an imposter with the same name a accepting ambients on the same channel:

$$b \boxed{\text{in } a \cdot x . Q} \mid a \boxed{\text{in } x . P} \mid a \boxed{\text{in } x . Imposter} \longrightarrow a \boxed{\text{in } x . P} \mid a \boxed{b \boxed{Q} \mid Imposter}$$

In the Channel Ambient calculus, these issues can be resolved by using private channels for entering an ambient or sending a message to an ambient. For example, a private communication channel x can be used to prevent an imposter from interfering with the communication between a and b :

$$\nu x (b \boxed{a \cdot x \langle n \rangle . Q} \mid a \boxed{x^\uparrow(m) . P}) \mid a \boxed{x^\uparrow(m) . Imposter} \longrightarrow \nu x (b \boxed{Q} \mid a \boxed{P_{\{n/m\}}}) \mid a \boxed{x^\uparrow(m) . Imposter}$$

Similarly, a private migration channel x can be used to prevent an imposter from interfering with the migration of b into a :

$$\nu x (b \boxed{\text{in } a.x.Q} \mid a \boxed{\text{in } x.P} \mid a \boxed{\text{in } x.\text{Imposter}} \longrightarrow \nu x (a \boxed{b \boxed{Q} \mid P} \mid a \boxed{\text{in } x.\text{Imposter}})$$

However, in practice it can be cumbersome to require every communication or migration to take place over a private channel, particularly since the privacy of these channels may change dynamically over time. Instead, it would be more convenient for an ambient to freely distribute its name and use publicly available channels for communication and migration, without the risk of impersonation.

$$P, Q, R ::= \mathbf{0} \quad \text{Null} \tag{2.31}$$

$$\mid P \mid Q \quad \text{Parallel} \tag{2.32}$$

$$\mid \nu n P \quad \text{Restriction} \tag{2.33}$$

$$\mid a \boxed{P} \quad \text{Ambient} \tag{2.34}$$

$$\mid \alpha.P \quad \text{Action, } \text{fa}(P) \cap \text{bn}(\alpha) = \emptyset \tag{2.35}$$

$$\mid !\alpha.P \quad \text{Replication} \tag{2.36}$$

Definition 2.6.12. Syntax of CA^-

$$\text{fa}(\mathbf{0}) \triangleq \emptyset \tag{2.37}$$

$$\text{fa}(P \mid Q) \triangleq \text{fa}(P) \cup \text{fa}(Q) \tag{2.38}$$

$$\text{fa}(\nu n P) \triangleq \text{fa}(P) \setminus \{n\} \tag{2.39}$$

$$\text{fa}(a \boxed{P}) \triangleq \text{fa}(P) \cup \{a\} \tag{2.40}$$

$$\text{fa}(\alpha.P) \triangleq \text{fa}(P) \setminus \text{bn}(\alpha) \tag{2.41}$$

$$\text{fa}(!\alpha.P) \triangleq \text{fa}(P) \setminus \text{bn}(\alpha) \tag{2.42}$$

Definition 2.6.13. Free Ambients in CA

$$a \boxed{P} \downarrow_a \tag{2.43}$$

$$P \downarrow_a \Rightarrow (P \mid Q) \downarrow_a \tag{2.44}$$

$$P \downarrow_a \Rightarrow (Q \mid P) \downarrow_a \tag{2.45}$$

$$a \neq m \wedge P \downarrow_a \Rightarrow (\nu m P) \downarrow_a \tag{2.46}$$

$$P \downarrow_a \Rightarrow b \boxed{P} \downarrow_a \tag{2.47}$$

Definition 2.6.14. Active Ambients in CA

A simple constraint can be defined on the syntax of the Channel Ambient calculus, in order to prevent the impersonation of ambients. The constraint consists in preventing names that are

received over channels from being used to create new ambients. The syntax of the constrained calculus CA^- is presented in Definition 2.6.12, and relies on the set $fa(P)$ of free ambients in a process P . According to Definition 2.6.13, a name is in the set $fa(P)$ if it is a free name in P that is used to create an ambient. The syntax of CA^- is identical to the syntax of CA , except that for a given action $\alpha.P$, the bound names of α are not in the set of free ambients of P . This prevents a name that is received over a channel from being used to create an ambient. As a result, a given ambient can freely send its name over a channel to a third party, without the risk of impersonation.

A number of basic properties can be proved for the constrained calculus CA^- . In particular, Lemmas 2.6.15, 2.6.16 and 2.6.17 ensure that the calculus is closed under substitution, structural congruence and reduction, respectively.

Lemma 2.6.15. (*Substitution Safety CA^-*) $\forall \sigma, P. P \in CA^- \Rightarrow P_\sigma \in CA^-$

Proof. By induction on Definition 2.3.7 of substitution in CA . □

Lemma 2.6.16. (*Structural Safety CA^-*) $\forall P. P \in CA^- \wedge P \equiv P' \Rightarrow P' \in CA^-$

Proof. By induction on Definition 2.3.5 of structural congruence in CA . □

Lemma 2.6.17. (*Reduction Safety CA^-*) $\forall P. P \in CA^- \wedge P \longrightarrow P' \Rightarrow P' \in CA^-$

Proof. By Lemma 2.6.15 (Substitution Safety CA^-), Lemma 2.6.16 (Structural Safety CA^-) and by induction on Definition 2.3.3 of reduction in CA . □

The desired properties of the constrained calculus CA^- can be made more precise by defining a notion of active ambient $P \downarrow_a$. According to Definition 2.6.14, a process P contains an active ambient a , written $P \downarrow_a$, if P contains an ambient named a inside a restriction, a parallel composition or inside an ambient. In other words, $P \downarrow_a$ if P contains an unguarded ambient named a . Intuitively, the constrained calculus CA^- should ensure that, if a name does not belong to the set of free ambients of a process, then the process can never contain an active ambient with this name. Lemma 2.6.18 (Active Ambients) ensures that the set of free ambients denotes all the active ambients of a process. If a process P contains an active ambient a then a is in the set of free ambients of P . Lemma 2.6.19 (Free Ambient Structure) ensures that the set of free ambients of a process is closed under structural congruence. If a process P is structurally congruent to a process Q then the set of free ambients of P is equal to the set of free ambients of Q . Lemma 2.6.20 (Free Ambient Reduction) ensures that the set of free ambients cannot be augmented by a reduction. If a process P can reduce to P' and z is a free ambient in P' then z must also be a free ambient in P . Lemma 2.6.21 (Free Ambient Transitivity) ensures that if an ambient is not free in a process then it will never be free in subsequent processes. If a is not a free ambient in P and P can reduce to P' in zero or more steps then a is not a free ambient in P' . Finally, Theorem 2.6.22

(Ambient Authenticity) ensures that if a name is not in the set of free ambients of a process then it can never be used to create an ambient. If a is not in the set of free ambients of P and P can reduce to P' in zero or more steps then P cannot contain an ambient named a . This ensures that a process cannot use a name received on input in order to create an ambient.

Lemma 2.6.18. (*Active Ambients*) $\forall P.P \in \text{CA}^- \wedge P \downarrow_a \Rightarrow a \in \text{fa}(P)$

Proof. By induction on Definition 2.6.14 of active ambients in CA. \square

Lemma 2.6.19. (*Free Ambient Structure*) $\forall P.P \in \text{CA}^- \wedge P \equiv Q \Rightarrow \text{fa}(P) = \text{fa}(Q)$

Proof. By induction on Definition 2.3.5 of structural congruence in CA. \square

Lemma 2.6.20. (*Free Ambient Reduction*) $\forall P.P \in \text{CA}^- \wedge z \in \text{fa}(P') \wedge P \longrightarrow P' \Rightarrow z \in \text{fa}(P)$

Proof. By Lemma 2.6.19 (Free Ambient Structure) and by induction on Definition 2.3.3 of reduction in CA. \square

Lemma 2.6.21. (*Free Ambient Transitivity*) $\forall P.P \in \text{CA}^- \wedge a \notin \text{fa}(P) \wedge P \longrightarrow^* P' \Rightarrow a \notin \text{fa}(P')$

Proof. By Lemma 2.6.19 (Free Ambient Structure), Lemma 2.6.20 (Free Ambient Reduction) and by induction on the transitive closure of reduction in CA. \square

Theorem 2.6.22. (*Ambient Authenticity*) $\forall P.P \in \text{CA}^- \wedge a \notin \text{fa}(P) \wedge P \longrightarrow^* P' \Rightarrow P' \not\downarrow_a$

Proof. By Lemma 2.6.18 (Active Ambients) and by Lemma 2.6.21 (Free Ambient Transitivity). \square

The constraints on the syntax of the Channel Ambient calculus can be viewed as a compromise between the capability model of the Ambient calculus and the unique identifiers of the Nomadic π -calculus. In the Ambient calculus, capabilities are used to enter, leave or open an ambient without revealing the ambient's name. Capabilities are not as flexible as channels, since they cannot be revoked and there is no limit on the number of times they can be used. Furthermore, when an ambient receives a capability it has no way of knowing the identity of the ambient to which the capability can be applied. At the other extreme, the Nomadic π -calculus requires all agents to be created with unique, private identifiers. This ensures that no two ambients can have the same name, thereby preventing ambient impersonation. However, in some cases it can be desirable for multiple ambients to have the same name. For example, multiple sites may wish to contain the same service ambient with the same public name, so that agents can invoke the same services on different sites via a uniform interface. The constrained calculus CA^- aims to combine the best of both worlds, by allowing multiple ambients to be created with the same public name, while still allowing ambients with private names to protect their identity.

2.7 Related Work

The Channel Ambient calculus is inspired by previous work on calculi for mobility, including the Ambient calculus, the Nomadic π -calculus and variants of the Boxed Ambient calculus. In many respects, the calculus can be viewed as a variant of Boxed Ambients in which channels are defined as first class entities. The calculus uses guarded replication, which is present in modern variants of Boxed Ambients such as [13], and similar actions to those used in Boxed Ambients [27]. The main differences with Boxed Ambients 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 π -calculus and channel communication is also used in the Seal calculus [23], although sibling seals cannot communicate directly. The use of channels for mobility is inspired by the mechanism of passwords, first introduced in [47] and subsequently adopted in [13]. The main advantage of CA over existing variants of Boxed Ambients is its ability to directly express high-level constructs such as channel-based interaction and sibling communication. These constructs seem to be at a suitable level of abstraction for specifying mobile applications such as the resource monitoring application outlined in Section 2.5. The main advantage of CA over the Nomadic π -calculus is its ability to regulate access to an ambient by means of named channels, and its ability to model computation within nested locations, both of which are lacking in Nomadic π .

In the Safe Ambient calculus [45], 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. For example, the process $m[\overline{in} m.Q_1 \mid Q_2]$ allows an arbitrary ambient to enter the ambient m , but there is no way of directly controlling which particular ambient can enter. This can be undesirable from a security point of view. For example, a host ambient wishing to accept two guests can give an entry capability to each guest. However, one of the guests could send its entry capability to a friend, who could then use this capability to take the place of the second guest. This situation is possible since the host has no way of distinguishing between guests. In contrast, the Channel Ambient calculus uses channels to regulate access to an ambient. This allows the host to create two private channels, one for each guest, and to only accept one guest on each channel. This ensures that no ambient can take the place of a guest without their consent.

One of the main characteristics of the Channel Ambient calculus is that it allows sibling ambients to communicate directly. This is often desirable within a single machine, or between two machines in a local area network. Even over a wide area network, certain protocols such as TCP/IP provide a useful abstraction for synchronous communication. In cases where asynchronous communication is required, such as the UDP protocol, communication between machines can be made asynchronous by requiring messages to be sent via an intermediate *router* ambient. The

Nomadic π -calculus highlights the benefits of allowing synchronous communication between agents on the same site. In [83] the authors of Nomadic π argue that some form of local synchronous communication is fundamental for programming mobile applications. The primitives of Nomadic π allow an agent to try to send a message to another agent on the same site. If the agents are on the same site then the message is delivered, otherwise the message is discarded and an alternative action is executed. 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. In order to avoid such forms of message loss, locking mechanisms need to be put in place to ensure that any undelivered messages are retrieved by an agent before it migrates. Such encodings are non-trivial, and place strict constraints on when an agent can and cannot move. The Nomadic π -calculus overcomes this problem by defining synchronous communication between sibling agents as primitive, and a similar approach is adopted in the Channel Ambient calculus. More generally, it seems natural that if an ambient can enter a sibling or leave its parent then it should also be able to communicate directly with its sibling or parent, in order to request permission to move.

Another characteristic of the Channel Ambient calculus is the use of channels for communication and migration. Basic forms of local and parent-child communication over channels can be encoded in the Boxed Ambient calculus, but more general forms of channel communication do not appear to be readily encodable. The NBA calculus [13] uses additional passwords to allow migration over channels, where the name of the migrating ambient is bound to a variable, resulting in increased expressive power. The Channel Ambient calculus uses channels for both communication and migration, but without the additional variable binders during migration. In general there is a delicate balance between simplicity and expressivity of calculus primitives. The Channel Ambient calculus attempts to remain as simple as possible, while still modelling high-level forms of interaction over channels.

From a security perspective, a range of techniques developed for related calculi can be applied to the Channel Ambient calculus. The use of channels as first class entities allows type systems for channel communication to be readily implemented, such as those defined for the π -calculus and for Nomadic π . The authors of Nomadic π highlight the usefulness of such channel types for detecting program errors. More general analysis techniques such as Ambient Logics can also be readily applied to the Channel Ambient calculus. Ambient Logics were identified by the authors of the Nomadic π -calculus as a promising formalism for proving security properties of mobile applications, although such logics cannot readily be applied to Nomadic π . In general, the Nomadic π -calculus seems to be at a suitable level of abstraction for programming mobile applications, while Ambient calculi are able to model network topology and support more powerful analysis techniques. The

Channel Ambient calculus attempts to combine the best of both worlds, drawing from the many lessons learned in Nomadic π , while still remaining in an Ambient framework.

2.8 Conclusion

This chapter presents a novel calculus for specifying mobile applications, known as the Channel Ambient calculus. The calculus is inspired by previous work on calculi for mobility, and its design is directly influenced by the properties of mobile applications. A graphical representation for the calculus is also presented, which can be used to specify execution traces of applications. The calculus is well-suited to specifying mobile applications for networks that support the TCP/IP protocol. In particular, the use of channels in the calculus bears a close resemblance to the use of ports in TCP/IP network communication. The calculus also satisfies a number of fundamental safety properties, which ensure that mobile applications are free of runtime errors. Various security mechanisms developed for related calculi can be applied to the Channel Ambient calculus, in order to reason about the security properties of mobile applications. These include type systems to ensure reliable channel communication and syntax constraints to ensure agent authenticity. By remaining in the Ambient paradigm, the Channel Ambient calculus can potentially benefit from a wide range of security mechanisms developed for Ambient calculi.

The Channel Ambient calculus has been used to specify a simple mobile application, in which a mobile agent monitors a resource on a remote site. In future, a wider range of mobile applications could be specified, in order to fully evaluate the expressivity of the calculus. A number of extensions to the calculus can also be envisaged, including notions of time and choice. A promising approach to modelling time in process calculi is presented in [7], which describes an elegant notion of time for the π -calculus that can also be readily applied to Ambient calculi. Choice is modelled in the Nomadic π -calculus using a testing semantics, which checks whether or not an agent is present before sending a message. A similar testing semantics could also be defined for the Channel Ambient calculus. Alternatively, a notion of prioritised execution could be defined, based on the notion of priorities for CCS outlined in [57]. A more straightforward extension would be to incorporate a model of non-deterministic choice, as defined in the Bio Ambient calculus [60]. In general, the decision to adopt a particular extension to the calculus will depend largely on the types of mobile applications being modelled, based on experience with a range of applications.

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 model the execution of applications on a range of networks types. For example, wireless networks with mobile devices correspond nicely to 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, [48]

uses the π -calculus to describe a simple protocol for switching mobile phones between base stations.

The theoretical foundations of the Channel Ambient calculus could also be broadened. In particular, a labelled transition system could be defined for the calculus, based on the lts presented in [13] for a variant of Boxed Ambients. The use of channels in the Channel Ambient calculus bears many similarities with the use of passwords in the Safe Ambient calculus with Passwords, and an lts for Channel Ambients could also be based on the lts presented in [47].

It is important to note that the security properties presented in this chapter are at the level of the Channel Ambient calculus only, and do not provide any guarantees at the level of an eventual implementation. In many respects, implementation security can be considered an orthogonal issue, but it is nevertheless an essential one if the calculus is to be of practical significance. Thus, it will be important to prove that the reduction semantics of the calculus is preserved in an implementation, in order to ensure that calculus processes are correctly executed. It will also be important to ensure that typing assumptions hold, that syntax constraints can be enforced and that the privacy of communication channels can be maintained. For example, in the calculus one can model communication between two sites over a private channel, as described in Subsection 2.6.3. This is a useful abstraction if one assumes total privacy of communication. But in cases where malicious attackers are known to be present, specific measures will need to be taken in an implementation to maintain this privacy, such as using encryption to prevent attackers from eavesdropping and observing the channel name.

Perhaps the most interesting area for future work lies in the use of Ambient Logics for reasoning about the security properties of mobile applications. Ambient Logics were presented in [20] as a high-level formalism for describing properties of processes in the Ambient calculus. Such logics are a powerful tool that can potentially express a much broader range of properties than process equivalences. By definition, the Channel Ambient calculus is a variant of the Ambient calculus, in which ambients can interact over named channels. Since Ambient Logics do not depend on the syntax of actions, such logics should be readily applicable to the Channel Ambient calculus. In principle, it should be sufficient to show that satisfaction in the logics is preserved by structural congruence and fresh renaming, using similar techniques to those outlined in [20]. Once the satisfaction relation has been formalised in this way, the full expressive power of Ambient Logics should be directly applicable to the Channel Ambient calculus. The next stage will be to explore the range of properties that can be proved with the logics, perhaps using the examples in this thesis as a starting point.

In the long term, a variant of the graphical representation described in this chapter could perhaps be used to help elaborate the specification of mobile applications. This approach is inspired by ongoing work that uses execution scenarios to aid application development [75]. In this setting, the user draws a series of execution scenarios that characterise the intended behaviour of a concur-

rent system. If sufficient scenarios are given then the program code of the system can be inferred, with additional guidance from the user in discarding unwanted behaviour. A similar process could be applied to the Channel Ambient calculus for the design of mobile applications, where the user draws a collection of scenarios that represent desired execution traces of the application, and the corresponding calculus process is inferred with additional guidance from the user.

Chapter 3

The Channel Ambient Machine

3.1 Introduction

The Channel Ambient calculus was presented in Chapter 2 as a high-level formalism for specifying mobile applications. A given application can first be specified as a process of the Channel Ambient 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 code the application from scratch using a suitable target language. Although this approach offers good flexibility, it also requires considerable overhead when multiple applications are being developed. This is because each new application will require code for agent communication, migration and networking to be re-implemented from scratch. A more efficient approach is to develop an Application Programming Interface (API), which maps the constructs of the calculus to a chosen target language. This allows a calculus specification to be implemented as a series of method calls to the API. Examples of this approach include Java APIs for the Ambient calculus [17] and for a variant of the Boxed Ambient calculus [55]. The main advantage of the approach is that the API can be smoothly integrated into any chosen target language, allowing interoperability with existing code. The main disadvantage, however, is that the constructs of the target language often extend beyond the scope of the calculus specification. Therefore, when method calls to the API are combined with arbitrary program code in the target language, such as code that generates exceptions, the behaviour of the resulting program is unpredictable. As a result, the main benefits of using a calculus to specify mobile applications are lost, since any security properties that hold for the specification will not necessarily hold for its implementation.

An alternative approach for implementing applications specified in the Channel Ambient calculus 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 [8]. 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 chapter 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. The main results of this chapter have been published in [56]. The chapter is structured as follows:

Section 3.2 describes the design principles of the Channel Ambient Machine, and shows how the machine can be derived from the Channel Ambient calculus.

Section 3.3 presents a formal definition of the machine, based on the above design principles.

The machine uses a list syntax, which is close to an implementation language, together with a blocking semantics, which leads to an efficient implementation. A corresponding graphical representation is also presented, which can be used to visualise execution traces of mobile applications.

Section 3.4 describes how the Channel Ambient Machine can be used to execute mobile applications on networks that support the TCP/IP protocol.

Section 3.5 uses the abstract machine to execute an example application, in which a mobile agent monitors resources on a remote site.

Section 3.6 proves the correctness of the Channel Ambient Machine with respect to the Channel Ambient calculus. The machine is proved both sound and complete with respect to the calculus, and is also proved to be free of runtime errors, deadlocks and livelocks.

3.2 Design Principles

This section describes the design principles of the Channel Ambient Machine, and shows how the machine can be derived from the Channel Ambient calculus. The section is structured as follows:

Subsection 3.2.1 evaluates the suitability of using the reduction rules of the calculus to implement a runtime for executing calculus processes. Two main problems are identified, namely reducing a process up to structural congruence and repeatedly reducing a process.

Subsection 3.2.2 describes how the problem of reducing a process up to structural congruence can be overcome by defining a normal form for processes.

Subsection 3.2.3 describes how the problem of repeatedly reducing a process can be overcome by distinguishing between blocked and unblocked actions and ambients.

Subsection 3.2.4 describes how a normal form and a notion of blocking can be used to design an abstract machine, in order to bridge the gap between the reduction rules of the calculus and the program code of a runtime.

3.2.1 Problems with Using Calculus Reduction to Implement a Runtime

The reduction rules of the Channel Ambient calculus are a high-level description of how to execute a calculus process. By definition, a reduction rule of the form $P \longrightarrow P'$ states that the process P can reduce to the process P' by performing a single execution step. A given process P is reduced by trying to match P with the left-hand side of one of the reduction rules. If a match is found, then the reduction is performed and P is modified according to the right hand side of the rule. If no match is found then P cannot reduce and execution terminates. Each reduction corresponds to a single execution step, and a given process is executed by repeatedly reducing the process until no more reductions are possible.

The reduction rules are defined operationally, making them close to the level of abstraction of a programming language. As a result, they can be readily implemented in a suitable target language. This close link with implementation is one of the motivating factors for using a process calculus to specify mobile applications. In spite of these features, however, the reduction rules of the Channel Ambient calculus are still too high-level to be used directly as the basis for implementing a runtime. In principle, the runtime would execute a given process by trying to match it with the left-hand side of one of the reduction rules. Although most of the rules are relatively straightforward to match, the rule that allows reduction up to structural congruence of processes (2.8) is problematic:

$$Q \equiv P \longrightarrow P' \equiv Q' \Rightarrow Q \longrightarrow Q'$$

This rule states that if a process P can reduce to P' then any process that is structurally congruent to P can reduce to any process that is structurally congruent to P' . In order to match a given process Q with the left-hand side of this rule, a runtime needs to check whether there exists a process P that is both structurally congruent to Q and able to reduce. A naive approach is to try to apply each of the structural congruence rules to Q until such a process P is found. However, due to the transitivity of structural congruence there are an infinite number of processes that are structurally congruent to Q . As a result, the runtime could go into an infinite loop without ever finding a process P that is able to reduce. To overcome this problem, the runtime would need to use a suitable evaluation strategy to check, in a finite number of steps, whether Q is structurally congruent to a process P that is able to reduce. Such a strategy is non-trivial and is not defined by the operational semantics of the calculus.

In addition, there is no defined strategy for repeatedly applying the reduction rules in order to perform a sequence of reductions. A naive approach is to systematically attempt all of the reduction rules in turn, at each step. However, the approach is highly inefficient if the runtime needs to perform a sequence of reductions, since only a limited number of reductions are possible at any given step and only a small number of processes are modified by a given reduction rule.

These two problems of reducing a process up to structural congruence and repeatedly reducing a process can both be overcome by refining the definition of the calculus, in order to bridge the gap between the reduction rules of the calculus and the program code of the runtime.

3.2.2 Using a Normal Form to Execute Reductions up to Structural Congruence

In order to reduce a process up to structural congruence, the runtime needs to use a suitable evaluation strategy to check, in a finite number of steps, whether a given process is structurally congruent to a process that is able to reduce. In general, such an evaluation strategy is non-trivial, but the strategy can be greatly simplified by defining a *normal form* for processes. The normal form constrains the syntax of processes to make it easier for the runtime to match a given process with the left-hand side of one of the reduction rules, up to structural congruence. In particular, the normal form can be used to eliminate the need to apply certain structural congruence rules altogether. The normal form can be defined by systematically examining each structural congruence rule in turn, and trying to constrain the syntax of processes so that the rule is no longer required to perform a reduction:

(2.9) The Null process can be composed in parallel with a given process. Since many of the reduction rules explicitly require a parallel process to be present, the syntax of processes can be constrained so that each unguarded process is in parallel with exactly one null process.

- (2.11) Parallel composition is associative. Since none of the reduction rules explicitly require parallel compositions to associate to the left, the syntax of processes can be constrained so that all parallel compositions associate to the right.
- (2.12) A replicated action $!\alpha.P$ can be expanded to the action α , followed by the process P in parallel with the replicated action $!\alpha.P$. Since none of the reduction rules explicitly require a replicated action to be present, the syntax of processes can be constrained so that all unguarded replicated actions are expanded.
- (2.15)-(2.16) The scope of a restricted name can be extended over a parallel composition and outside an ambient. Since none of the reduction rules explicitly contain a restriction inside a parallel composition or an ambient, the syntax of processes can be constrained so that the scope of all unguarded restrictions is extended to the top-level.
- (2.13)-(2.14) A restricted name can be added to the null process and successive restricted names can be permuted. If the syntax of processes is constrained so that the scope of all unguarded restrictions is extended to the top-level, then these rules are no longer required to reduce a process.

The above constraints give rise to a normal form in which each unguarded process is in parallel with exactly one null process, all unguarded parallel compositions associate to the right, all unguarded replicated actions are expanded, and the scope of all unguarded restrictions is extended to the top-level. This normal form is analogous to a list syntax, where a list consists of a sequence of actions and ambients separated by a composition operator that associates to the right, ending with null. The list can also contain a number of top-level restricted names.

By constraining the syntax of processes in this way, only a small number of rules are needed to match a process with the left-hand side of a reduction rule. These matching rules correspond to the structural congruence rules that were not eliminated during the design of the normal form, namely the rule for commutativity of parallel composition (2.10) together with the congruence rules for rearranging processes inside an arbitrary context. Therefore, in order to match a process in normal form with the left-hand side of a reduction rule, a runtime simply needs to change the order of parallel compositions inside a restriction, inside an ambient or inside a parallel composition. This simplified form of matching, subsequently referred to as *selection*, is significantly more efficient than trying to match the left-hand side of a reduction rule using the full definition of structural congruence. Furthermore, the matching will always terminate, since a given parallel composition will always have a finite number of permutations.

3.2.3 Using Blocking to Efficiently Execute Successive Reductions

In order to repeatedly reduce a given process, the runtime needs to use a suitable evaluation strategy to check whether the process matches the left-hand side of a reduction rule. By definition, a reduction rule involves an action and a corresponding co-action. Therefore, in order to match a process with the left-hand side of a reduction rule the runtime needs to find an action, and then look for a corresponding co-action. If a co-action is present then the reduction can be performed, otherwise the runtime needs to try to match a different reduction rule. If none of the reduction rules can be matched then execution terminates, otherwise the runtime continues to reduce the process until no more reductions are possible.

In order to avoid re-checking the entire process after each reduction, the runtime can distinguish between *blocked* and *unblocked* actions and ambients, such that each reduction involves an action and a corresponding blocked co-action. A blocked action represents an action that has already been considered by the runtime, but for which no corresponding blocked co-action has been found. A blocked ambient represents an ambient that does not contain any unguarded unblocked actions or ambients. By labelling an action as blocked, the runtime records the fact that no-corresponding blocked co-action is present, and can therefore avoid looking for such a co-action in subsequent reductions. Similarly, by labelling an ambient as blocked, the runtime records the fact that there are no unblocked actions present inside this ambient, and can therefore avoid looking for such actions inside the ambient in subsequent reductions.

In order to match a process with the left-hand side of one of the modified reduction rules, the runtime needs to find an action and then look for a corresponding blocked co-action. If a blocked co-action is present then the reduction is performed, otherwise the action is blocked and the runtime tries to match a different reduction rule. If all the actions in the process are blocked then execution terminates. Otherwise, the runtime continues reducing the process until no more reductions are possible.

Blocking significantly increases the efficiency of runtime execution, since the runtime can ignore any blocked actions and the entire contents of any blocked ambients when looking for an action to reduce. Blocking also simplifies the condition for termination, since the runtime only needs to check for the presence of an unblocked action to know whether a reduction is possible.

3.2.4 Justification for Defining an Abstract Machine

The reduction rules of the Channel Ambient calculus can be used as the basis for implementing a runtime by introducing a normal form for processes and by distinguishing between blocked and unblocked actions and ambients. However, these modifications would also lower the level of abstraction of the calculus, making it less suitable for specifying mobile applications. An

alternative to modifying the calculus is to define an *abstract machine*, which can be viewed as a more refined version of the calculus that is closer to an implementation. The abstract machine helps to bridge a gap between the reduction rules of the calculus and the program code of the runtime. The syntax of machine terms can be derived from the syntax of calculus processes by converting each process to normal form, and by distinguishing between blocked and unblocked actions and ambients. The reduction rules of the machine can be derived from the reduction rules of the calculus by ensuring that each reduction involves an action and a corresponding blocked co-action. A given machine term can then be executed by matching the term with the left-hand side of one of the reduction rules, using a simplified form of structural congruence.

The abstract machine can be used to execute a given calculus process by first converting the process to a machine term using a suitable encoding function, and then executing the term using the reduction rules of the machine. During execution, it is also necessary to ensure that blocked ambients and actions do not cause a machine term to deadlock. This can be achieved by defining a suitable unblocking function.

3.3 Definition

This section presents a formal definition of the Channel Ambient Machine, based on the design principles outlined previously. The section is structured as follows:

Subsection 3.3.1 presents the syntax of machine terms, where each term represents a corresponding calculus process.

Subsection 3.3.2 presents the construction rules of the machine, which describe how a process can be encoded into a term so that it can be executed by the machine.

Subsection 3.3.3 presents the selection rules of the machine, which describe how a machine term can be re-arranged to match the left-hand side of one of the execution rules. The selection rules are derived from the structural congruence rules of the calculus.

Subsection 3.3.4 describes how a machine term can be unblocked in order to prevent deadlocks during execution.

Subsection 3.4.2 presents the reduction rules of the machine, which describe how a machine term can be executed. The reduction rules are derived from the reduction rules of the calculus.

3.3.1 Syntax of Machine Terms

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 with zero or more restricted names. A list is either empty or contains one or more actions or ambients, which are either blocked or unblocked.

The syntax of machine terms is derived from the syntax of calculus processes by converting each process to a normal form, and by distinguishing between blocked and unblocked actions and ambients. The normal form ensures that each unguarded process is in parallel with exactly one null process, that unguarded parallel compositions associate to the right, that unguarded replicated actions are expanded, and that the scope of unguarded restrictions is extended to the top-level. Unguarded parallel compositions are represented in the machine using the list composition operator $(:)$, which associates to the right. An unguarded null process is represented in the machine using the empty list $[]$, which occurs once at the end of each list. Blocked actions and ambients are represented in the machine by underlining the action or ambient name, and unblocked actions and ambients are unchanged from their calculus representation.

$V ::= \nu n V$	Restriction	(3.1)	$\underline{z} ::= \underline{z}$	Blocked	(3.6)
$\mid A$	List	(3.2)	$\mid z$	Unblocked	(3.7)
$A, B, C ::= []$	Empty	(3.3)			
$\mid \alpha.P :: C$	Action	(3.4)			
$\mid \alpha \boxed{A} :: C$	Ambient	(3.5)			

Definition 3.3.1. Syntax of CAM

$\begin{array}{ c } \hline V \\ \hline \end{array}$	$::=$	$\begin{array}{ c } \hline n :: \vdots \\ \hline V \\ \hline \end{array}$	Restriction	$\begin{array}{ c } \hline A \\ \hline \end{array}$	List	$\begin{array}{ c } \hline \underline{z} \\ \hline \end{array}$	$::=$	$\begin{array}{ c } \hline \underline{z} \\ \hline \end{array}$	Blocked	$\begin{array}{ c } \hline z \\ \hline \end{array}$	U
$\begin{array}{ c } \hline A \\ \hline \end{array}, \begin{array}{ c } \hline B \\ \hline \end{array}, \begin{array}{ c } \hline C \\ \hline \end{array}$	$::=$	$\begin{array}{ c } \hline \\ \hline \end{array}$	Empty	$\begin{array}{ c } \hline \alpha.P \\ \hline \end{array} \begin{array}{ c } \hline C \\ \hline \end{array}$	Action	$\begin{array}{ c } \hline \boxed{a} \\ \hline \begin{array}{ c } \hline A \\ \hline \end{array} \\ \hline \end{array} \begin{array}{ c } \hline C \\ \hline \end{array}$	Ambient				

Definition 3.3.2. Graphical Syntax of CAM

The syntax of the Channel Ambient Machine is presented in Definition 3.3.1:

Restriction $\nu n V$ represents a machine term V with a top-level restricted name n .

List A represents a list with no restricted names.

Empty $[]$ represents an empty list.

Action $\alpha.P::C$ represents an action $\alpha.P$ at the head of a list C , where α is a calculus action and P is a calculus process.

Ambient $a[\boxed{A}]\::C$ represents an ambient a containing a list A , at the head of a list C .

Blocked \underline{z} represents a blocked action or ambient, which is underlined.

Unblocked z represents an unblocked action or ambient, which is unchanged from the calculus representation.

For convenience, $!\alpha.P$ is written as syntactic sugar for an expanded replicated action $\alpha.(P \mid !\alpha.P)$.

A graphical syntax for the Channel Ambient Machine is presented in Definition 3.3.2, based on the graphical syntax of the Channel Ambient calculus presented in Chapter 2. Graphical terms are similar to graphical processes, graphical list composition is similar to graphical parallel composition, and graphical restriction of terms is similar to graphical restriction of processes. In addition, the graphical syntax of the machine distinguishes between blocked and unblocked actions and ambients, by placing a grey box under any actions or ambients that are blocked.

For improved efficiency, the syntax of machine terms can be refined by splitting the list of elements into three sub-lists, consisting of actions, blocked actions and ambients, respectively:

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

This syntax can be refined even further by replacing the list of blocked actions with a map data structure, to enable faster lookup. In general, such optimisations are closer to the level of an implementation, and do not significantly affect the semantics of the abstract machine. A more detailed discussion of machine optimisations is given in Chapter 4.

3.3.2 Using Construction to Encode a Process to a Machine Term

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

Definition 3.3.3. Encoding CA to CAM

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 3.3.3. The construction $P : []$ adds the process P to the empty list $[]$.

$$n \notin \text{fn}(P) \Rightarrow P:(\nu n V) \triangleq \nu n(P:V) \quad (3.8)$$

$$\mathbf{0}:A \triangleq A \quad (3.9)$$

$$(P \mid Q):A \triangleq P:Q:A \quad (3.10)$$

$$n \notin \text{fn}(P:A) \Rightarrow (\nu n P):A \triangleq \nu n(P_{\{n/m\}}:A) \quad (3.11)$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P:A \triangleq \alpha.(P \mid !\alpha.P):A \quad (3.12)$$

$$a[\boxed{P}]:A \triangleq a[\boxed{P:\Box}]:A \quad (3.13)$$

$$\alpha.P:A \triangleq \alpha.P::A \quad (3.14)$$

$$n \notin \text{fn}(a, V) \Rightarrow a[\boxed{\nu n U}]:V \triangleq \nu n(a[\boxed{U}]:V) \quad (3.15)$$

$$n \notin \text{fn}(a, A) \Rightarrow a[\boxed{A}]:\nu n V \triangleq \nu n(a[\boxed{A}]:V) \quad (3.16)$$

$$a[\boxed{A}]:C \triangleq a[\boxed{A}]::C \quad (3.17)$$

Definition 3.3.4. 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 $a[\boxed{U}]:V$ 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.

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. A process is added to a list by discarding the null process, adding each process in a parallel composition separately, expanding replicated actions and extending the scope of any restricted names in the process to the top level. Any processes inside an ambient are first encoded to corresponding terms, which can then be added to the list.

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, and then placing the ambient at the head of the list inside the scope of the restricted names.

The full definition of construction is given in Definition 3.3.4, where $\text{fn}(V)$ is the set of free names in V :

- (3.8)** A process P is added to a term V with a restricted name n by extending the scope of n to the top-level and then adding P to V inside the scope of n , provided n is not free in P . This rule is used to extend the scope of any restricted names in V to the top-level, so that P can be added to the list inside the scope of the restricted names.

- (3.9)** The null process $\mathbf{0}$ is discarded.

- (3.10) The parallel composition $P \mid Q$ is added to a list by first adding the process Q to the list, and then adding the process P to the resulting term.
- (3.11) A process P with a restricted name m is added to a list by replacing the restricted name with a fresh name n , extending the scope of n to the top level and then adding the process $P_{\{n/m\}}$ to the list inside the scope of n .
- (3.12) A replicated action $!\alpha.P$ is added to a list by expanding the replicated action and then adding the resulting action $\alpha.(P \mid !\alpha.P)$ to the list. The expansion assumes that the free values of α are not contained in the set of bound values of α . This ensures that channel names are not captured during the expansion of replicated inputs.
- (3.13) An ambient a with a process P is added to a list by encoding P to a term and then adding the ambient a and the resulting term to the list.
- (3.14) An action $\alpha.P$ is added to a list by placing the action at the front of the list.
- (3.15) An ambient a and a term U with a restricted name n are added to a term V by extending the scope of n to the top level and then adding a and U to V , provided n is not free in a, V . This rule is used to extend the scope of any restricted names in U to the top-level, so that a and a list can be added to V inside the scope of the restricted names.
- (3.16) An ambient a and a list A are added to a term V with a restricted name n by extending the scope of n to the top-level, and adding a and A to V inside the scope of n , provided n is not free in a, A . This rule is used to extend the scope of any restricted names in V to the top-level, so that a and A can be added to the list inside the scope of the restricted names.
- (3.17) An ambient a and a list A are added to a list by placing A inside the ambient a and placing the ambient at the front of the list.

A graphical representation for construction can be defined for the Channel Ambient Machine, based on the graphical representation for composition:

- The construction $P:V$ is represented by placing a vertical bar labelled with the process P next to the graphical term V .
- The construction $a[\overline{U}]:V$ is represented by placing the graphical term U under a box labelled with the ambient name a , next to the graphical term V .

Although the resulting graphical representation for construction closely resembles the graphical representation for composition, any ambiguity can be resolved by looking at the individual elements used in the construction or composition, bearing in mind that the construction $\alpha.P:C$ is equal to the composition $\alpha.P::C$, and the construction $a[\overline{A}]:C$ is equal to the composition $a[\overline{A}]::C$.

3.3.3 Using Selection to Schedule a Machine Term

$$A @ \alpha . P :: A' \succ \alpha . P :: A @ A' \quad (3.18)$$

$$A @ b \boxed{B} :: A' \succ b \boxed{B} :: A @ A' \quad (3.19)$$

$$A \succ A' \Rightarrow q \boxed{A} :: C \succ q \boxed{A'} :: C \quad (3.20)$$

Definition 3.3.5. 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. A list can match itself, an action or ambient inside a list can be brought to the front of the list, and a list can be re-arranged inside an ambient.

The full definition of selection is given in Definition 3.3.5, where the append function $A @ A'$ is used to concatenate the list A with the list A' :

(3.18) An action $\alpha . P$ in a list can be brought to the front of the list.

(3.19) An ambient $q \boxed{A}$ in a list can be brought to the front of the list.

(3.20) A list A can be re-arranged inside an ambient.

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.

3.3.4 Using Unblocking to Prevent Deadlocks During Execution

$$[] \triangleq [] \quad (3.21)$$

$$[\alpha . P :: C] \triangleq \alpha . P :: [C] \quad (3.22)$$

$$[q \boxed{A} :: C] \triangleq q \boxed{A} :: [C] \quad (3.23)$$

Definition 3.3.6. 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 3.3.6:

(3.21) The null list $[]$ is unchanged.

(3.22) An action $\alpha.P$ at the head of a list C is unblocked to $\alpha.P$, and the remaining list C is unblocked.

(3.23) An ambient $a[A]$ at the head of a list C is unchanged, and the remaining list C is unblocked.

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 adjacent 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:

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

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:

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

3.3.5 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. 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. 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.

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 3.3.7, where $A \succ A'$ means that the term A can be re-arranged to match the term A' , and $\lfloor A \rfloor$ unblocks any top-level blocked actions in A , as defined previously:

(3.24) An ambient a can send a value to a sibling over a channel. If a contains a sibling output

$b \cdot x \langle v \rangle . P$, and there is a sibling ambient b with a blocked external input $\underline{x}^\dagger(u).Q$, then the value v can be sent to ambient b over channel x and assigned to the value u in process Q .

(3.25) If there is no corresponding blocked external input then the sibling output is blocked.

(3.26) Similarly, an ambient b can receive a value from a sibling over a channel. If b contains

$$C \succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{P : A} : b \boxed{Q_{\{v/u\}} : B} : C' \quad (3.24)$$

$$C \not\succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \quad (3.25)$$

$$C \succ a \boxed{b \cdot x \langle v \rangle . P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow a \boxed{P : A} : b \boxed{Q_{\{v/u\}} : B} : C' \quad (3.26)$$

$$C \not\succ a \boxed{b \cdot x \langle v \rangle . P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow b \boxed{x^\dagger(u).Q :: B} :: C \quad (3.27)$$

$$C \succ x(u).Q :: C' \Rightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{P : A} : Q_{\{v/u\}} : C' \quad (3.28)$$

$$C \not\succ x(u).Q :: C' \Rightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \quad (3.29)$$

$$C \succ a \boxed{x^\dagger \langle v \rangle . P :: A} :: C' \Rightarrow x(u).Q :: C \longrightarrow a \boxed{P : A} : Q_{\{v/u\}} : C' \quad (3.30)$$

$$C \not\succ a \boxed{x^\dagger \langle v \rangle . P :: A} :: C' \Rightarrow x(u).Q :: C \longrightarrow x(u).Q :: C \quad (3.31)$$

$$C \succ b \boxed{\text{in } x.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow b \boxed{Q : a \boxed{P : [A]} : B} : C' \quad (3.32)$$

$$C \not\succ b \boxed{\text{in } x.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \quad (3.33)$$

$$C \succ a \boxed{\text{in } b \cdot x.P :: A} :: C' \Rightarrow b \boxed{\text{in } x.Q :: B} :: C \longrightarrow b \boxed{Q : a \boxed{P : [A]} : B} : C' \quad (3.34)$$

$$C \not\succ a \boxed{\text{in } b \cdot x.P :: A} :: C' \Rightarrow b \boxed{\text{in } x.Q :: B} :: C \longrightarrow b \boxed{\text{in } x.Q :: B} :: C \quad (3.35)$$

$$B \succ \overline{\text{out}} x.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{Q : B'} : a \boxed{P : [A]} : C \quad (3.36)$$

$$B \not\succ \overline{\text{out}} x.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \quad (3.37)$$

$$B \succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\overline{\text{out}} x.Q :: B} :: C \longrightarrow b \boxed{Q : B'} : a \boxed{P : [A]} : C \quad (3.38)$$

$$B \not\succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\overline{\text{out}} x.Q :: B} :: C \longrightarrow b \boxed{\overline{\text{out}} x.Q :: B} :: C \quad (3.39)$$

$$V \longrightarrow V' \Rightarrow \nu n V \longrightarrow \nu n V' \quad (3.40)$$

$$B \succ A \wedge A \longrightarrow V' \Rightarrow B \longrightarrow V' \quad (3.41)$$

$$A \longrightarrow V' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{V'} : C \quad (3.42)$$

$$A \not\succ \alpha.P :: A' \wedge A \not\succ b \boxed{B} :: A' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{A} :: C \quad (3.43)$$

Definition 3.3.7. Reduction in CAM

an external input $x^\dagger(u).Q$, and there is a sibling ambient a with a blocked sibling output $b \cdot x \langle v \rangle . P$, then the value v can be received over channel x and assigned to the value u in process Q .

- (3.27) If there is no corresponding blocked sibling output then the external input is blocked.
- (3.28) An ambient a can send a value to its parent over a channel. If a contains a parent output $x^\uparrow\langle v \rangle.P$, and in parallel with a there is a blocked internal input $\underline{x(u)}.Q$, then the value v can be sent over channel x and assigned to the value u in process Q .
- (3.29) If there is no corresponding blocked internal input then the parent output is blocked.
- (3.30) Similarly, a parent ambient can receive a value from a child over a channel. If there is an internal input $x(u).Q$, and in parallel there is an ambient a with a blocked parent output $\underline{x^\uparrow\langle v \rangle}.P$, then the value v can be received over channel x and assigned to the value u in process Q .
- (3.31) If there is no corresponding blocked parent output then the internal input is blocked.
- (3.32) An ambient a can enter a sibling over a channel. If a contains an enter $\text{in } b.x.P$, and there is a sibling ambient b with a blocked accept $\overline{\text{in } x}.Q$, then a can enter b over channel x .
- (3.33) If there is no corresponding blocked accept then the enter is blocked.
- (3.34) Similarly, an ambient b can accept a sibling over a channel. If b contains an accept $\overline{\text{in } x}.Q$, and there is a sibling ambient a with a blocked enter $\text{in } b.x.P$, then b can accept a over channel x .
- (3.35) If there is no corresponding blocked enter then the accept is blocked.
- (3.36) An ambient a can leave its parent over a channel. If a contains a leave $\text{out } x.P$, and there is a parent ambient with a blocked release $\overline{\text{out } x}.Q$, then a can leave its parent over channel x .
- (3.37) If there is no corresponding blocked release then the leave is blocked.
- (3.38) Similarly, an ambient b can release a child over a channel. If b contains a release $\overline{\text{out } x}.Q$, and there is a child ambient with a blocked leave $\text{out } x.P$, then b can release child a over channel x .
- (3.39) If there is no corresponding blocked leave then the release is blocked.
- (3.40) A reduction can occur inside a restriction. If a term V can reduce to V' then the reduction can also take place if V has a restricted name n .
- (3.41) Matching terms can perform the same reduction. If a list A can reduce to V' and B matches A then B can reduce to V' .

(3.42) A reduction can occur inside an ambient. If a list A can reduce to V' then the reduction can also take place if A is inside an ambient a .

(3.43) If an ambient does not contain any unblocked actions or ambients then the ambient is blocked.

A graphical representation for reduction can be defined for the Channel Ambient Machine, based on the graphical representation of reduction in the Channel Ambient calculus. Like the calculus, time proceeds vertically downward, the interaction between terms is represented using solid arrows for communication and dotted arrows for migration, and each vertical bar is labelled with the current state of the term. In addition, the graphical machine reduction rules ensure that each interaction takes place between an action and a corresponding blocked co-action.

3.4 Distributed Execution

This section describes how the Channel Ambient Machine can be used to model the execution of mobile applications on networks that support the TCP/IP protocol. The section is structured as follows:

Subsection 3.4.1 summarises the constraints that can be placed on the syntax of Channel Ambient calculus in a distributed setting, and shows how these constraints are preserved when a calculus process is encoded to a machine term.

Subsection 3.4.2 describes how the reduction rules of the Channel Ambient Machine can be used to model the execution of mobile applications on TCP/IP networks with broadcast.

Subsection 3.4.3 describes a number of constraints that 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.

3.4.1 Summary of Distributed Constraints for the Channel Ambient Calculus

In Chapter 2, a number of constraints were placed on the syntax of the Channel Ambient calculus in a distributed setting, in order to model the properties of networks that support the TCP/IP protocol. These constraints are used 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. The constraints on the syntax of agents and sites are summarised as follows:

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

Sites A process inside a site 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 is constrained so that it cannot contain an enter or a leave. This reflects the assumption that sites have a fixed network address.

By definition, a process P of the Channel Ambient calculus is encoded to a machine term using an encoding function $\llbracket P \rrbracket$, which adds the process to an empty list using a construction operator. According to the definition of construction, any constraints on the syntax of agents or sites will be preserved in the corresponding machine term. In a distributed setting, the machine term will generally consist of a list of sites with a number of top-level names:

$$\nu n_1 \dots \nu n_N s_1 \boxed{S_1} :: \dots :: s_M \boxed{S_M} :: []$$

The top-level names represent the names of all the channels and ambients in the network, and each site $s_i \boxed{S_i}$ represents a machine with IP address s_i and contents S_i . It is also possible to add a layer of multiplexing by using s_i to represent an IP address and port number, so that multiple sites can share the same IP address.

3.4.2 Modelling Execution on TCP/IP Networks With Broadcast

The Channel Ambient Machine executes a given term in a non-deterministic sequence of steps, according to the reduction rules of the machine. At first glance, this sequential execution model seems unrealistic, since multiple reductions in a given network appear to occur simultaneously. However, by considering a sufficiently small interval of time it can be shown that no two independent events will happen at exactly the same instant. Therefore, a non-deterministic sequential execution model can be considered an acceptable representation of execution in a distributed setting.

Table 3.1 describes how 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 table takes into account the various constraints on the syntax of machine terms described in the previous section. The main rules in the table are explained below:

(3.24) 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

Rule	Application in a distributed setting
(3.24)	Ambient a sends value v to site b on port x
(3.25)	Ambient a blocks output $b \cdot x \langle v \rangle$ to site b
(3.26)	Site b receives value v from ambient a on channel x (broadcast)
(3.27)	Site b blocks external input $x^\dagger(u)$ on port x
(3.28)	Agent a sends value v to parent site on channel x
	Site a sends value v to parent site on port x
(3.29)	Agent a blocks output $x^\dagger \langle v \rangle$ to parent site
	Site a blocks output $x^\dagger \langle v \rangle$ to parent site
(3.30)	Parent site receives value v from child agent a on channel x
	Parent site receives value v from child site a on port x (broadcast)
(3.31)	Site blocks input $x(u)$ from a child agent on channel x
	Site blocks input $x(u)$ from a child site on port x
(3.32)	Agent a enters site b on port x
(3.33)	Agent a blocks enter $\text{in } b \cdot x$ to site b on channel x
(3.34)	Site b accepts agent a on port x (broadcast)
(3.35)	Site b blocks accept $\overline{\text{in}} x$ on port x
(3.36)	Agent a leaves site b on channel x
(3.37)	Agent a blocks leave $\text{out } x$ from parent site b
(3.38)	Site b releases agent a on channel x
(3.39)	Site b blocks release $\overline{\text{out}} x$ of an agent on channel x
(3.40)	A term V can reduce inside a restriction
(3.41)	A list A can reduce up to re-ordering of its contents
(3.42)	The contents of a site a can be executed independently of other sites
(3.43)	Site a can block if it does not contain any unblocked actions or ambients

Table 3.1: Using the Reduction Rules of CAM in TCP/IP Networks.

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 .

(3.28) 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.

(3.32) 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.

(3.41) 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.

(3.42) 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 (3.26), (3.30) and (3.34) require additional support for network broadcast. In particular, rule (3.26) 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 (3.30) and the rule that allows an agent to be accepted from a remote site (3.34).

3.4.3 Modelling Execution on TCP/IP Networks Without Broadcast

Rule	Constraint
(3.25)	b is an agent g
(3.29)	a is an agent g
(3.33)	b is an agent g

Definition 3.4.1. Constraining the Reduction Rules of CAM for Networks Without Broadcast.

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.

Definition 3.4.1 describes the constraints that can be placed on the reduction rules of the Channel Ambient Machine:

- (3.25) Constraining b to be an agent ensures that a sibling output $b \cdot x \langle v \rangle$ will not block if b is a site. This ensures that a site b will never interact with a blocked sibling output according to (3.26).
- (3.29) Constraining a to be an agent ensures that a parent output $x^\dagger \langle v \rangle$ inside an ambient a will not block if a is a site. This ensures that a parent site will never interact with a blocked parent output according to (3.30).
- (3.33) Constraining b to be an agent ensures that an enter $\text{in } b \cdot x$ will not block if b is a site. This ensures that a site b will never interact with a blocked enter according to (3.34).

3.5 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. The section is structured as follows:

Subsection 3.5.1 summarises the calculus specification of the application and shows how it can be encoded to a corresponding machine term.

Subsection 3.5.2 describes a possible sequence of reductions that can occur when the encoded term is executed by the Channel Ambient Machine.

Subsection 3.5.3 highlights the main features of the Channel Ambient Machine that are illustrated by the example.

3.5.1 Encoding the Application Specification to a Machine Term

Figure 3.1 contains a formal specification of the example application described in Chapter 2, in which a mobile agent monitors resources on a remote site. The specification is 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 the registration request has been sent, the client waits for a login

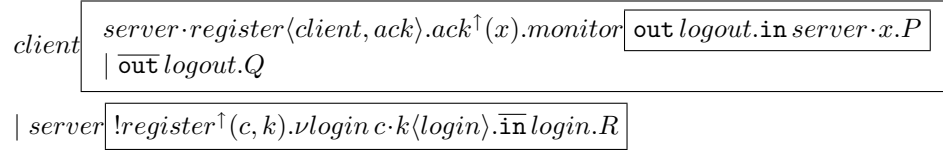


Figure 3.1: Application Specification

channel x on the acknowledgement channel. After the login channel has been received, 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 *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 .
-

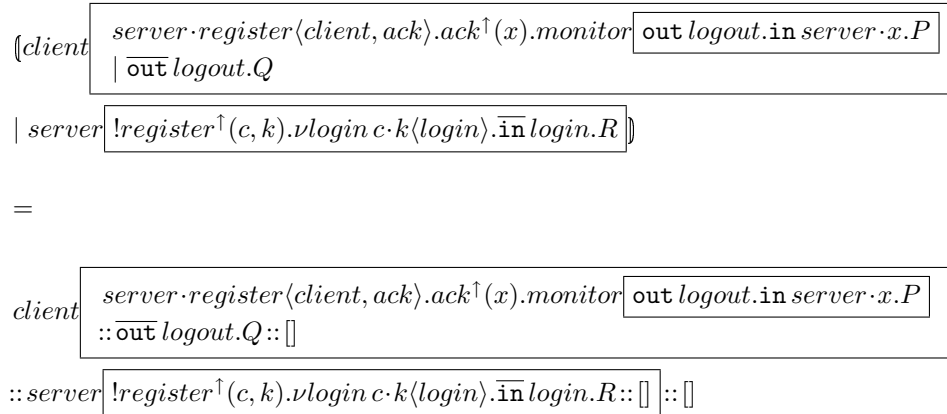


Figure 3.2: Application Encoding

Figure 3.2 shows how the calculus specification can be encoded to a corresponding machine term, using the encoding function $\llbracket P \rrbracket = P \vdash \square$

- 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.

3.5.2 Graphical Execution Scenario for the Application

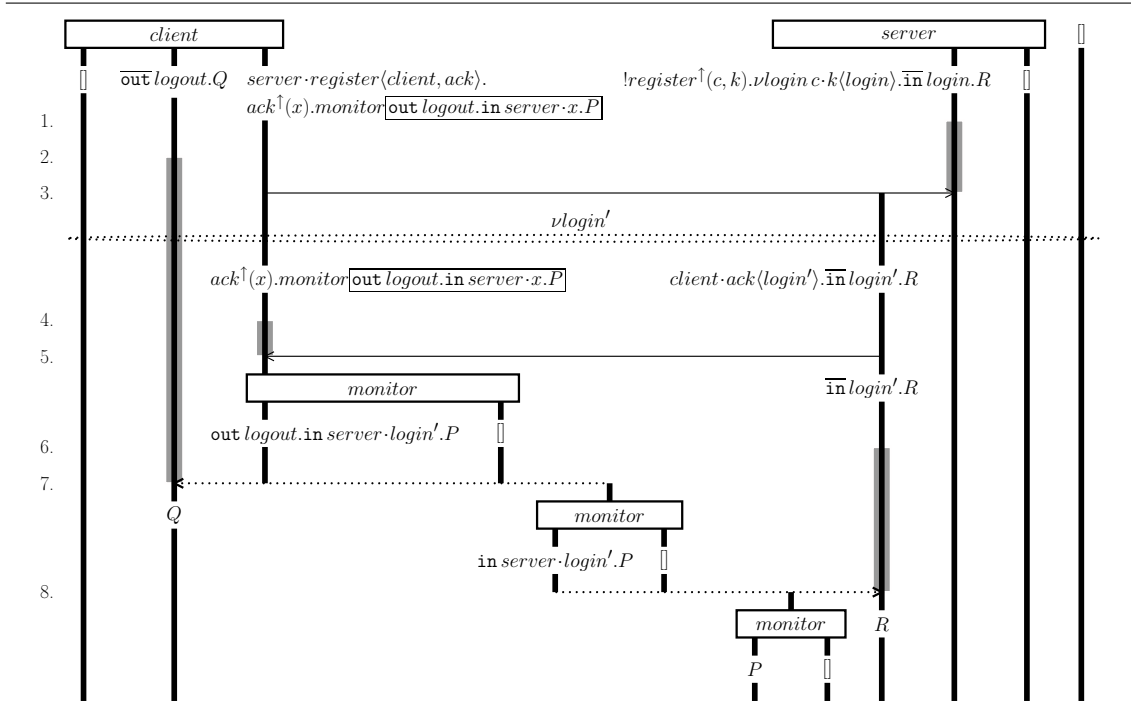


Figure 3.3: Graphical Machine Scenario, where $login' \notin \text{fn}(P, Q, R)$

Once the calculus process has been encoded to a corresponding machine term, it can then be executed by the machine. Figure 3.3 uses the graphical representation of the Channel Ambient Machine to describe an execution scenario for the encoded term. For convenience, the figure assumes that a list of the form $\alpha.P :: \dots :: \alpha'.P' :: []$ can also be written from right to left as $[] :: \alpha'.P' :: \dots :: \alpha.P$. Each reduction step in the figure is numbered, and the corresponding explanation for each number is given below:

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.

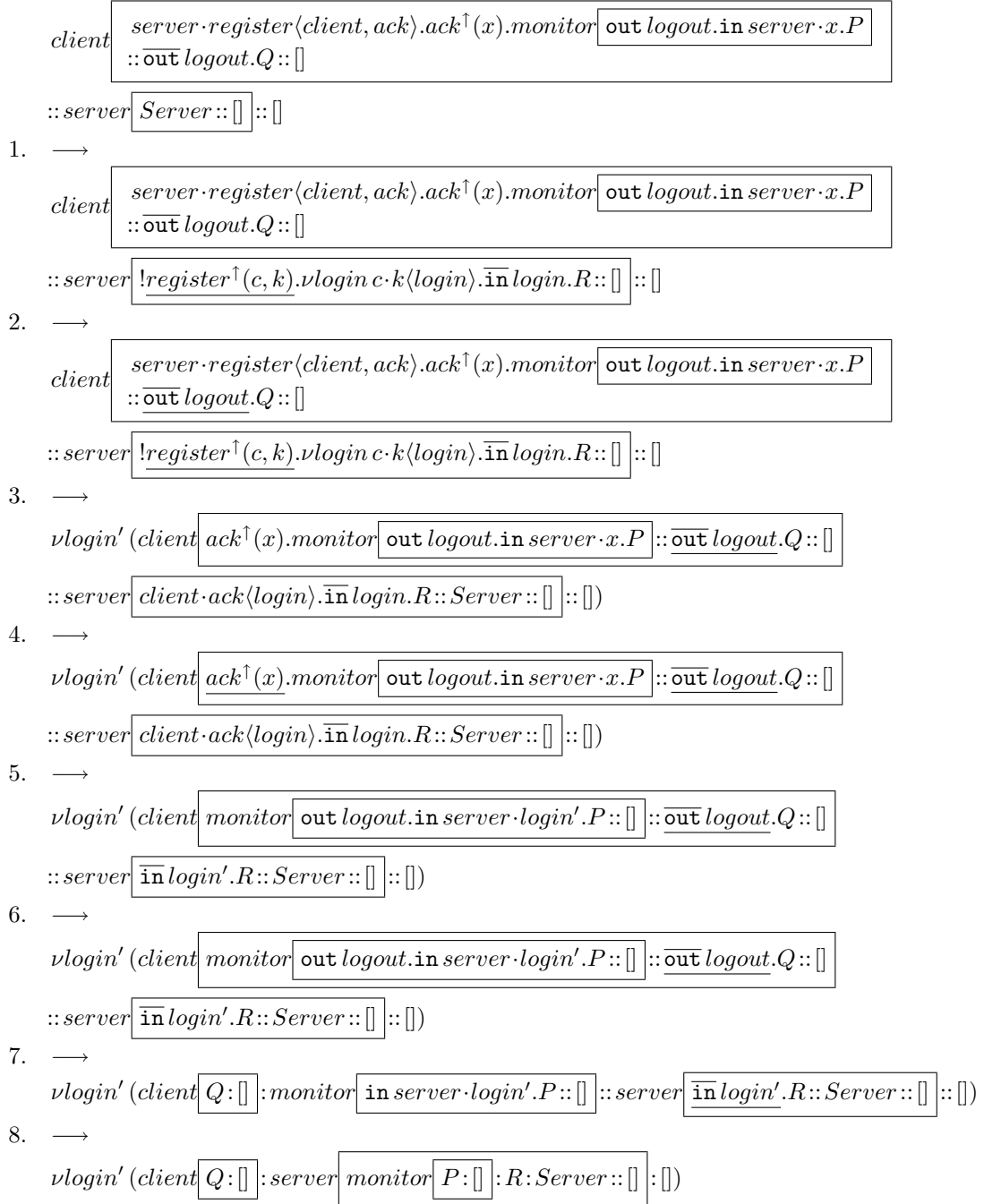
Figure 3.4 uses the textual syntax of the Channel Ambient Machine to describe the graphical scenario presented in Figure 3.3. By comparison, the graphical syntax is more compact than the textual syntax, since only those processes that are modified by a given reduction need to be re-written. The graphical syntax also gives a clearer representation of how the structure of the machine term evolves over time.

3.5.3 Using the Application to Evaluate the Channel Ambient Machine

The above execution scenario for the resource monitoring application illustrates the main features of the Channel Ambient Machine:

- Initially, all of the actions in a machine term are unblocked. Then, as execution progresses, various actions block non-deterministically, waiting for corresponding co-actions. For example, the external input on the register channel blocks inside the server, waiting for a corresponding sibling output from a client (Step 1). Similarly, the release on the logout channel blocks inside the client, waiting for an agent in the client to perform a corresponding leave (Step 2).
- Each reduction step involves an action and a corresponding blocked co-action. For example, the sibling output to the server on the register channel interacts with the blocked external input inside the server (Step 3).

$$Server \triangleq !register^\uparrow(c, k). \nu login\ c \cdot k \langle login \rangle. \overline{in}\ login. R$$

Figure 3.4: Machine Scenario, where $login' \notin \text{fn}(P, Q, R)$

- When a new channel is created, a globally unique channel name is used and the scope of the channel is extended to the top-level. For example, when the server receives a registration request from a client, it creates a globally unique *login'* channel (Step 5).

The example also illustrates how the Channel Ambient Machine can be used in a distributed setting:

- Each site is executed on a different physical machine, and the agents inside a site can be executed independently of other sites. For example, the client and server sites are executed on two different machines, and the *monitor* agent inside the server is executed independently of the client.
- The actions inside two different sites can synchronise over a network. For example, the sibling output inside the client can synchronise with the blocked external input in the server, using the protocols of the underlying network.
- By definition, a sibling output or enter to a site will never block. For example, the client continues trying to send a message to the server on the *register* channel without blocking (Step 3). In general, it is assumed that the server is launched before the client, and therefore the replicated input on the server will have time to block before the client attempts to send a message. However, the server may also be disconnected or may no longer be accepting requests on the register channel. In these cases, the client will continue polling the server until the registration request is sent. In order to avoid causing a denial of service attack, the interval between polls is increased exponentially over time.
- Since a sibling output or enter to a site will never block, a corresponding co-action needs to block in order for a reduction to occur. For example, the replicated input on the register channel of the server needs to block before the client can send a registration request to the server (Step 1). Similarly, the input on the acknowledgement channel of the client needs to block before the server can send an acknowledgement to the client (Step 4). In addition, the accept on the login channel of the server needs to block before the monitor agent can enter the server (Step 6). In a typical execution scenario, the network delay will be sufficiently large relative to the execution speed of the client and server for the relevant co-actions to block. If not, each remote action will continue polling until a corresponding co-action becomes blocked.

3.6 Correctness

This section proves the correctness of the Channel Ambient Machine with respect to the Channel Ambient calculus. Additional proof details are given in Appendix A.2. The section is structured as follows:

Subsection 3.6.1 proves the safety of the machine, to ensure that it does not produce runtime errors.

Subsection 3.6.2 proves the soundness of the machine, to ensure that it always performs valid execution steps.

Subsection 3.6.3 proves the completeness of the machine, to ensure that it can correctly match all possible execution steps of the calculus.

Subsection 3.6.4 proves the liveness of the machine, to ensure that it does not deadlock.

Subsection 3.6.5 proves the termination of the machine, to ensure that it does not livelock.

3.6.1 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. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove the safety of the machine it is necessary to prove that construction, selection, unblocking and reduction are safe.

Lemma 3.6.1 (Construction Safety) ensures that the result of a construction is always a valid machine term. The lemma states that if a process P is added to a term V then the result is a valid machine term. Similarly, if an ambient a with a term U is added to a term V then the result is a valid machine term.

Lemma 3.6.1. (*Construction Safety*) $\forall P. \forall V. P \in \text{CA} \wedge V \in \text{CAM} \Rightarrow P : V \in \text{CAM}$
 $\forall U. \forall V. U, V \in \text{CAM} \Rightarrow a[\overline{U}]:V \in \text{CAM}$

Proof. By induction on Definition 3.3.4 of construction in CAM. □

Lemma 3.6.2 (Selection Safety) ensures that the result of a selection is always a valid machine term. The lemma states that if the machine selects a term A' from a term A then A' is a valid machine term.

Lemma 3.6.2. (*Selection Safety*) $\forall A. A \in \text{CAM} \wedge A \succ A' \Rightarrow A' \in \text{CAM}$

Proof. By induction on Definition 3.3.5 of selection in CAM. □

Lemma 3.6.3 (Unblocking Safety) ensures that the result of an unblocking is always a valid machine term. The lemma states that if the machine unblocks a term V then the result is a valid machine term.

Lemma 3.6.3. (*Unblocking Safety*) $\forall V.V \in \text{CAM} \Rightarrow [V] \in \text{CAM}$

Proof. By induction on Definition 3.3.6 of unblocking in CAM. \square

Finally, Theorem 3.6.4 (Reduction Safety) ensures that the result of a reduction is always a valid machine term. The theorem states that if the machine reduces a term V to V' then V' is a valid machine term.

Theorem 3.6.4. (*Reduction Safety*) $\forall V.V \in \text{CAM} \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}$

Proof. By Lemma 3.6.1 (Construction Safety), Lemma 3.6.2 (Selection Safety), Lemma 3.6.3 (Unblocking Safety) and by induction on Definition 3.3.7 of reduction in CAM. \square

3.6.2 Proving Soundness to Ensure Valid Execution Steps

$$[\nu n V] \triangleq \nu n [V] \quad (3.44)$$

$$[\Box] \triangleq \mathbf{0} \quad (3.45)$$

$$[\alpha.P :: C] \triangleq \alpha.P \mid [C] \quad (3.46)$$

$$[a \Box A :: C] \triangleq a \Box [A] \mid [C] \quad (3.47)$$

Definition 3.6.5. Decoding CAM to CA

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* $[V]$, which maps a given machine term V to a corresponding calculus process. 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. According to Definition 3.6.5:

(3.44) A term V with a private name n maps to the decoding of V with private name n .

(3.45) The null list \Box maps to the null process $\mathbf{0}$.

(3.46) An action $\alpha.P$ at the head of a list C maps to $\alpha.P$ in parallel with the decoding of C .

(3.47) An ambient a with term A at the head of a list C maps to ambient a containing the decoding of A , in parallel with the decoding of C .

Lemma 3.6.6 (Decoding Soundness) ensures that decoding always produces a valid calculus process.

Lemma 3.6.6. (*Decoding Soundness*) $\forall V.V \in \text{CAM} \Rightarrow \llbracket V \rrbracket \in \text{CA}$

Proof. By induction on Definition 3.6.5 of decoding in CAM. \square

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. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove the soundness of the machine it is necessary to prove that construction, selection, unblocking and reduction are sound.

Lemma 3.6.7 (Construction Soundness) ensures that construction in the machine corresponds to parallel composition in the calculus, up to structural congruence. The lemma states that if a process P is added to a term V then the decoding of the resulting term is structurally congruent to P in parallel with the decoding of V . Similarly, if an ambient a with a term U is added to a term V then the decoding of the resulting term is structurally congruent to ambient a containing the decoding of U , in parallel with the decoding of V .

Lemma 3.6.7. (*Construction Soundness*) $\forall P.V.P \in \text{CA} \wedge V \in \text{CAM} \Rightarrow \llbracket P:V \rrbracket \equiv P \mid \llbracket V \rrbracket$
 $\forall U.V.U, V \in \text{CAM} \Rightarrow \llbracket a[U]:V \rrbracket \equiv a[\llbracket U \rrbracket] \mid \llbracket V \rrbracket$

Proof. By Lemma 3.6.6 and by induction on Definition 3.3.4 of construction in CAM. \square

Lemma 3.6.8 (Selection Soundness) ensures that selection in the machine corresponds to structural congruence in the calculus. The lemma states that if the machine selects a term A' from a term A then the decoding of A' is structurally congruent to the decoding of A .

Lemma 3.6.8. (*Selection Soundness*) $\forall A.A \in \text{CAM} \wedge A \succ A' \Rightarrow \llbracket A \rrbracket \equiv \llbracket A' \rrbracket$

Proof. By induction on Definition 3.3.5 of selection in CAM. \square

Lemma 3.6.9 (Unblocking Soundness) ensures that unblocking in the machine corresponds to equality in the calculus. The lemma states that if the machine unblocks a term V then the decoding of the result is equal to the decoding of V .

Lemma 3.6.9. (*Unblocking Soundness*) $\forall V.V \in \text{CAM} \Rightarrow \llbracket [V] \rrbracket = \llbracket V \rrbracket$

Proof. By induction on Definition 3.3.6 of unblocking in CAM. \square

Theorem 3.6.10 (Reduction Soundness) ensures that reduction in the machine corresponds to at most one reduction in the calculus. The theorem 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 3.6.10. (*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 Lemma 3.6.6 (Decoding Soundness), Lemma 3.6.8 (Selection Soundness), Lemma 3.6.9 (Unblocking Soundness), Lemma 3.6.7 (Construction Soundness) and by induction on Definition 3.3.7 of reduction in CAM. \square

3.6.3 Proving Completeness to Ensure Accurate Execution

$$n \notin \text{fn}(V) \Rightarrow \nu n V \equiv V \quad (3.48)$$

$$\nu n \nu m V \equiv \nu m \nu n V \quad (3.49)$$

$$A @ B @ C \equiv B @ A @ C \quad (3.50)$$

$$V \equiv V' \Rightarrow \nu n V \equiv \nu n V' \quad (3.51)$$

$$A \equiv A' \Rightarrow q[\boxed{A}] :: C \equiv q[\boxed{A'}] :: C \quad (3.52)$$

$$P \equiv P' \Rightarrow \alpha.P :: C \equiv \alpha.P' :: C \quad (3.53)$$

$$C \equiv C' \Rightarrow A @ C \equiv A @ C' \quad (3.54)$$

Definition 3.6.11. Structural Congruence in CAM

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. These re-orderings can also take place inside a restriction, inside an ambient or inside the tail of a list. According to Definition 3.6.11:

(3.48) Unused restricted names can be discarded.

(3.49) Successive restricted names can be permuted.

(3.50) Successive segments of a list can be permuted.

(3.51) A term can be re-arranged inside a restriction.

(3.52) A list can be re-arranged inside an ambient.

(3.53) A process can be re-arranged inside an action.

(3.54) The tail of a list can be re-arranged.

As usual, structural congruence is the least relation that is reflexive, symmetric and transitive, and that satisfies the rules in Definition 3.6.11. Lemma 3.6.12 (Structural Correctness) ensures that structural congruence always produces a valid machine term.

Lemma 3.6.12. (*Structural Correctness*) $\forall V.V \in \text{CAM} \wedge V \equiv V' \Rightarrow V' \in \text{CAM}$

Proof. By induction on Definition 3.6.11 of structural congruence in CAM. \square

Lemma 3.6.13 (Structural Reduction) ensures that structurally congruent terms can perform corresponding reductions. This property needs to be proved explicitly for the machine, since structural congruence is not used in the definition of reduction. The omission is deliberate, and avoids the need to examine all possible re-orderings of a term in order to perform a reduction, thereby significantly increasing the efficiency of the machine. The theorem states that if the machine can reduce a term V to V' then it can reduce any term that is structurally congruent to V to a term that is structurally congruent to V' .

Lemma 3.6.13. (*Structural Reduction*) $\forall V.V \in \text{CAM} \wedge U \equiv V \wedge V \longrightarrow V' \Rightarrow U \longrightarrow \equiv V'$

Proof. By induction on Definition 3.6.11 of structural congruence in CAM. \square

Once a structural congruence relation has been defined in this way, it is possible to prove the completeness of the machine. Calculus execution is defined in terms of structural congruence and reduction. Therefore, in order to prove the completeness of the machine it is necessary to prove that structural congruence and reduction are complete.

Lemma 3.6.14 (Structural Completeness) ensures that if two calculus processes are structurally congruent then their encodings are structurally congruent. The lemma states that if a given process P is structurally congruent to process Q then the encoding of P is structurally congruent to the encoding of Q .

Lemma 3.6.14. (*Structural Completeness*) $P \equiv Q \Rightarrow \llbracket P \rrbracket \equiv \llbracket Q \rrbracket$

Proof. By induction on Definition 2.3.5 of structural congruence in CA. \square

Theorem 3.6.15 (Reduction Completeness) ensures that if a process can perform a reduction in the calculus then its encoding can perform a corresponding sequence of reductions in the machine, up to structural congruence. The theorem 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 3.6.15. (*Reduction Completeness*) $\forall P.P \in \text{CA} \wedge P \longrightarrow P' \Rightarrow \llbracket P \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \rrbracket$

Proof. By Lemma 3.6.13, Lemma 3.6.14 and by induction on Definition 2.3.3 of reduction in CA. \square

3.6.4 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 CAM^\vee and is defined by placing constraints on the set of machine terms CAM . According to Definition 3.6.16, a deadlock-free term is a deadlock-free list with zero or more restrictions, where a deadlock-free list is one of the following:

(3.57) An empty list.

(3.58) An action $\alpha.P$ at the head of a deadlock-free list C , such that if $\alpha.P$ is a blocked internal input on channel x then C does not contain an ambient with a blocked parent output on x .

(3.59) An ambient a with a deadlock-free list A , at the head of a deadlock-free list C , such that if a is blocked then A does not contain an unblocked action or ambient, and if A contains a blocked action then C does not contain a corresponding blocked co-action.

$$V ::= \nu n V \quad \text{Restriction} \quad (3.55)$$

$$\mid A \quad \text{List} \quad (3.56)$$

$$A, B, C ::= [] \quad \text{Empty} \quad (3.57)$$

$$\mid \alpha.P :: C \quad \text{Action, } \alpha.P = \underline{x(m)}.P \Rightarrow C \not\vdash q \boxed{x^\uparrow \langle n \rangle . P :: A} :: C' \quad (3.58)$$

$$\mid q \boxed{A} :: C \quad \text{Ambient, } q = \underline{a} \Rightarrow (A \not\vdash \alpha.P :: A' \wedge A \not\vdash b \boxed{B} :: A') \quad (3.59)$$

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

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

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

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

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

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

Definition 3.6.16. Syntax of Deadlock-Free Terms CAM^\vee

Lemma 3.6.17 (Deadlock-Free Subset) ensures that deadlock-free terms are a subset of machine terms. The lemma states that if a given term V is deadlock-free then it is a valid machine term.

Lemma 3.6.17. (*Deadlock-Free Subset*) $\forall V. V \in \text{CAM}^\vee \Rightarrow V \in \text{CAM}$

Proof. By induction on Definition 3.6.16 of deadlock-free terms 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. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove that the machine is deadlock-free it is necessary to prove that construction, selection, unblocking and reduction are deadlock-free.

Lemma 3.6.18 (Deadlock-Free Construction) ensures that construction cannot cause a term to deadlock. The lemma states that if a process P is added to a term V then the result is deadlock-free. Similarly, if an ambient a with unblocked contents \boxed{A} is added to a term V then the

result is deadlock-free. Note that a direct consequence of this lemma is that $\llbracket P \rrbracket \in \text{CAM}^\vee$, since $\llbracket P \rrbracket = P : []$.

Lemma 3.6.18. (*Deadlock-Free Construction*)

$$\forall P. \forall V. P \in \text{CA} \wedge V \in \text{CAM}^\vee \Rightarrow P : V \in \text{CAM}^\vee$$

$$\forall A. \forall V. A \in \text{CAM}^\vee \wedge V \in \text{CAM}^\vee \Rightarrow a[\![A]\!]:V \in \text{CAM}^\vee$$

Proof. By induction on Definition 3.3.4 of construction in CAM. \square

Lemma 3.6.19 (Deadlock-Free Selection) ensures that selection cannot cause a term to deadlock. The lemma states that if the machine selects a term A' from a deadlock-free term A then A' is deadlock-free.

Lemma 3.6.19. (*Deadlock-Free Selection*) $\forall A. A \in \text{CAM}^\vee \wedge A \succ A' \Rightarrow A' \in \text{CAM}^\vee$

Proof. By Definition 3.6.16 of deadlock-free terms CAM^\vee . \square

Lemma 3.6.20 (Deadlock-Free Unblocking) ensures that unblocking cannot cause a term to deadlock. The lemma states that if a deadlock-free term V is unblocked then the result is deadlock-free.

Lemma 3.6.20. (*Deadlock-Free Unblocking*) $\forall V. V \in \text{CAM}^\vee \Rightarrow \llbracket V \rrbracket \in \text{CAM}^\vee$

Proof. By induction on Definition 3.3.6 of unblocking in CAM. \square

Finally, Lemma 3.6.21 (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 3.6.21. (*Deadlock-Free Reduction*) $\forall V. V \in \text{CAM}^\vee \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}^\vee$

Proof. By Lemma 3.6.18 (Deadlock-Free Construction), Lemma 3.6.19 (Deadlock-Free Selection), Lemma 3.6.20 (Deadlock-Free Unblocking) and by induction on Definition 3.3.7 of reduction in CAM. \square

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 3.6.22 (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 3.6.22. (*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 3.3.7 of reduction in CAM. \square

Lemma 3.6.23 (Decoding Encoding) ensures that a process is structurally congruent to the decoding of its encoding.

Lemma 3.6.23. (*Decoding Encoding*) $\forall P. P \in \text{CA} \Rightarrow \llbracket \llbracket P \rrbracket \rrbracket \equiv P$

Proof. Follows from Lemma 3.6.7. \square

Lemma 3.6.24 (Encoding Decoding) ensures that a term and the encoding of its decoding both have the same decoding. Note that by Definition 3.6.5, if two terms have the same decoding then they are equal up to blocking of actions and ambients.

Lemma 3.6.24. (*Encoding Decoding*) $\forall V. V \in \text{CAM} \Rightarrow \llbracket \llbracket \llbracket V \rrbracket \rrbracket \rrbracket = \llbracket V \rrbracket$

Proof. By induction on Definition 3.6.5 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 3.6.25 (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 3.6.25. (*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 3.6.23 (Decoding Encoding), Lemma 3.6.24 (Encoding Decoding), Lemma 3.6.22 (Decoding Reduction) and by Theorem 3.6.15 (Reduction Completeness). \square

3.6.5 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 3.6.26 (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 3.6.26. (*Termination*) $\forall V \in \text{CAM}. \llbracket V \rrbracket \not\longrightarrow \Rightarrow \exists V'. V \longrightarrow^* V' \wedge V' \not\longrightarrow$

Proof. The reduction rules of the machine can be divided into two types of rules: blocking rules $V \longrightarrow_b V'$ and interactive rules $V \longrightarrow_i V'$. From the proof of Theorem 3.6.10 it can be shown

that:

$$\begin{aligned} V \longrightarrow_i V' &\Rightarrow \llbracket V \rrbracket \longrightarrow \llbracket V' \rrbracket \\ V \longrightarrow_b V' &\Rightarrow \llbracket V \rrbracket \equiv \llbracket V' \rrbracket \end{aligned}$$

Therefore, if $\llbracket V \rrbracket \not\rightarrow$ then either $V \not\rightarrow$ or $V \longrightarrow_b V'$. Furthermore, if $V \longrightarrow_b V'$ 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.

3.7 Related Work

The Channel Ambient Machine is inspired by the Pict abstract machine [74], which is used as a basis for implementing the asynchronous π -calculus. Like the Pict machine, CAM uses a list syntax to represent the parallel composition of processes. 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 π -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 [17] an informal abstract machine for Ambients is presented, which has not been proved sound or complete. In [30] 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 ab-

abstract machine in [30] 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 [66] 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.

In principle, a variety of properties could be used to define the correctness of the Channel Ambient Machine. The properties used in this thesis are based on the properties for the Pict machine described in [74], together with the properties for the Ambient machine described in [17]. In [74], the Pict machine was proved to produce valid executions, which is analogous to the soundness theorem of Subsection 3.6.2. The authors explicitly state that completeness does *not* hold, since the machine imposes a deterministic scheduling algorithm. Instead, they prove a liveness property for the machine, which states that if a well-formed machine term cannot reduce then there are no possible reductions in the corresponding calculus process. The notion of well-formedness prevents the machine from being in certain states that are known to deadlock. A corresponding notion of well-formedness is used to prove liveness for the Channel Ambient Machine. For the Ambient machine in [17], a number of desirable correctness properties are described, although none of them are formally proved. The properties are soundness, completeness and liveness, and are analogous to the main correctness properties of the Channel Ambient Machine. The author argues that although completeness is not desirable in an implementation, the abstract machine on which an implementation is based should be complete with respect to the calculus. This offers maximum flexibility for an implementation, particularly with regard to the choice of scheduling algorithms. For the mapping from the Ambient calculus to the Distributed Join calculus in [30], a notion of *barbed coupled simulation* is used to prove the correctness of a distributed execution algorithm for Ambients. Then, a notion of *hybrid barbed bisimulation* is used to prove the correctness of the translation into the Join calculus. The particular choice of equivalences is directly influenced by the nature of the algorithm and its corresponding translation.

3.8 Conclusion

This chapter presents an abstract machine for the Channel Ambient calculus, known as the Channel Ambient Machine. 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. The Channel Ambient Machine is derived from the Channel Ambient calculus by defining a list syntax, which is close to an implementation language, together with a blocking

semantics, which leads to an efficient implementation. A corresponding graphical representation for the Channel Ambient Machine is also presented, which can be used to visualise execution traces of the machine for both debugging and tutorial purposes. The machine is also well-suited to modelling the execution of mobile applications on networks that support the TCP/IP protocol. Finally, the Channel Ambient machine is proved both sound and complete with respect to the Channel Ambient calculus, and is also proved to be free of runtime errors, deadlocks and livelocks. Although the proofs are non-trivial, they only need to be done once in order to ensure that any application specified in the calculus will be correctly executed by the machine. This ensures that the work done in specifying and verifying mobile applications is not lost during their implementation.

In future, the design principles that were used to define an abstract machine for the Channel Ambient calculus could also applied a range of existing calculi, including the π -calculus [48], the Ambient calculus [21], the Safe Ambient calculus [45] and variants of the Boxed Ambient calculus [11, 12, 27, 13, 46]. Already, these principles have been used to develop an abstract machine for the stochastic π -calculus with mixed choice, in order to simulate models of biological processes [54]. Similar principles could also be used to develop an abstract machine for the Seal calculus [22], which shares a number of common features with the Channel Ambient calculus, including the use of bounded locations (seals) that cannot be opened and the use of channels for communicating between locations.

In future, the correctness of the Channel Ambient Machine could also be formulated in terms of equivalences. More precisely, it should be possible to prove full abstraction with respect to observational congruence, in the sense of [28]. First, a suitable notion of observation can be defined based on standard formulations such as [13], in which an ambient is observed if it can accept another ambient. The corresponding formulation in the Channel Ambient calculus would be $P \downarrow_a$ if and only if $P \equiv \nu \tilde{z} (a \boxed{\text{in } x.P} \mid A \mid C)$ for $\{b, x\} \cap \{\tilde{z}\} = \emptyset$. A similar formulation of observation could also be used for the Channel Ambient Machine. It should then be possible to show that

$$\begin{aligned} P \approx_{\text{CA}} Q &\iff (P) \approx_{\text{CAM}} (Q) \\ A \approx_{\text{CAM}} B &\iff \llbracket A \rrbracket \approx_{\text{CA}} \llbracket B \rrbracket \end{aligned}$$

where \approx_{CA} and \approx_{CAM} denote observational congruence in the calculus and the abstract machine, respectively. Such a strong result should be possible, due to the straightforward nature of the encodings from calculus to machine and vice-versa. When encoding a calculus process to a machine term, the process is simply re-arranged in a way that is compatible with structural congruence. During execution, a notion of blocking is used to tag ambients and actions that are waiting to execute, but this is merely a housekeeping mechanism. Conversely, when decoding a machine term

to a calculus process, the tags are removed and the constructs of the machine are simply replaced by their equivalent constructs in the calculus. Based on the completeness and liveness theorems in Section 3.6, it should be possible to prove the preservation of observables during encoding and decoding, and ultimately the preservation of observational congruence. An alternative approach to defining observation congruence is the Honda-Yoshida method [41], which gives greater flexibility in the choice of observable. For example, the approach is used in [80] to show how different formulations of barbs lead to the same barbed congruence for Safe Ambients.

The Channel Ambient Machine was used to model the execution of mobile applications on networks that support the TCP/IP protocol. In principle, the machine could also be used to model the execution of applications on a range of networks types. For example, wireless networks with mobile devices correspond nicely to the synchronisation primitives of the machine, particularly since broadcasting in such networks can be achieved with minimal overhead. Indeed, wireless networks were one of the first network types to be targeted by process calculi. The abstract machine presented in this chapter could provide further insight into the deployment and execution of mobile applications on such networks. In future, similar principles could also be used to define an execution model for applications that support mobile IP, using the TCP/IP protocol version 6.

Although the proofs outlined in this chapter are not immediate, they do appear sufficiently direct as to be readily automated. In future, the proofs could perhaps be mechanised using a standard proof assistant such as HOL [39]. Such an approach was already used to develop mechanised proofs for the semantics of the UDP calculus, as described in [82].

Chapter 4

The Channel Ambient Runtime

4.1 Introduction

The Channel Ambient calculus was presented in Chapter 2 as a high-level formalism for specifying mobile applications. The Channel Ambient Machine was then presented in Chapter 3 as a formal specification of a runtime for executing calculus processes. The machine was proved both sound and complete with respect to the calculus, and was shown to be free of runtime errors, deadlocks and livelocks. Once a runtime has been formally specified in this way, the next stage is to implement the specification in a suitable target language. It can be argued that functional programming languages are well-suited to such an implementation:

1. Firstly, the recursive nature of functional languages is ideal for implementing the recursive definitions of the Channel Ambient Machine, allowing an almost direct mapping from abstract machine to functional program code. In particular, the data structures of the machine map nicely to functional data structures such as lists and datatypes. This close correspondence between runtime specification and implementation helps to limit the introduction of programming errors.
2. Secondly, using a functional language has a number of inherent benefits, including concise code that is easier to debug, the detection of most errors during typechecking rather than at runtime, and the relative ease with which complex data structures can be defined and manipulated. Such benefits are particularly apparent when developing runtime systems for programming languages.

This chapter 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, using the OCaml programming language [44]. The chapter is structured as follows:

Section 4.2 describes how the Channel Ambient Machine can be used to implement a local runtime, which executes a given calculus process on a single physical device.

Section 4.3 describes how the Channel Ambient Machine can be used to implement a distributed runtime, which executes a given calculus process over multiple devices in a hierarchical TCP/IP network.

Section 4.4 describes how the distributed runtime can be enhanced in order to improve the efficiency of process execution, while conserving network bandwidth.

4.2 Local Runtime Implementation

This section describes how the Channel Ambient Machine can be used to implement a local runtime, which executes a given calculus process on a single physical device. The section is structured as follows:

Subsection 4.2.1 describes the architecture of the local runtime, and shows how the main components are used to execute a process of the Channel Ambient calculus.

Subsection 4.2.2 describes how runtime terms can be implemented as functional data structures.

Subsection 4.2.3 describes the implementation of a substitution function for binding a value to a variable inside a calculus process. This is used to bind newly created channels and values received over channels during execution.

Subsection 4.2.4 describes the implementation of a construction function for adding a process to a runtime term. This is used to add a newly created process to a term during execution. It is also used to encode a process into a corresponding term so that it can be executed by the runtime.

Subsection 4.2.5 describes the implementation of a selection function for choosing a given ambient or action from inside a runtime term. This is used to schedule the next action or ambient to be executed by the runtime and to find a pair of actions that can interact during runtime execution.

Subsection 4.2.6 describes the implementation of an unblocking function for unblocking the contents of an ambient. This is used to re-bind an ambient to its new environment when it moves to a new location

Subsection 4.2.7 describes the implementation of a reduction function for executing a runtime term.

4.2.1 Architecture of the Local Runtime

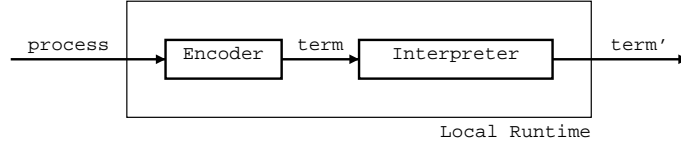


Figure 4.1: Architecture of the Local Runtime

The architecture of the local runtime is described in Figure 4.1. The main components of the runtime are an *encoder* and an *interpreter*, and a given process of the Channel Ambient calculus is executed by the local runtime as follows:

1. First, the process is encoded to a corresponding runtime term by the encoder.
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. Execution continues until no more reductions are possible.

```

let run (p:process) = interpret (encode p)

let rec interpret (t:term) = match reduce t with
  [] -> t
  | t' -> interpret t'
  
```

Definition 4.2.1. Top-Level Program Code for the Local Runtime

The top-level program code for the local runtime is presented in Definition 4.2.1:

run p executes the process *p* on a local runtime. The function first encodes *p* into a runtime term using the *encode* function and then interprets the resulting term using the *interpret* function.

encode p encodes the process *p* into a corresponding runtime term.

interpret t repeatedly reduces the term *t* with the help of the *reduce* function.

reduce t tries to reduce the term t in order to perform a single execution step. If a reduction is possible then the function returns an updated term t' , otherwise the function returns an empty term.

4.2.2 Implementing Runtime Terms as Functional Data Structures

In order for a given process to be executed, it must first be encoded into a suitable runtime term. The definition of runtime terms is summarised in Definition 4.2.2, together with the corresponding implementation. By definition, a runtime term V is a list of actions or ambients, which are either *blocked* or *unblocked*. Actions or ambients that are active and ready to execute are labelled as unblocked, whereas actions or ambients that are waiting on external conditions to continue executing are labelled as blocked. A given term can also contain a number of top-level private values $\nu n_1 \dots \nu n_N$. However, the privacy of these top-level values does not need to be implemented explicitly, since a given runtime will implicitly have its own private address space for storing and manipulating values. As a result, a runtime term can be implemented simply as a list of elements, where each element can be either an ambient or an action, and each ambient or action can be either blocked or unblocked. Note that the actions α and processes P are identical to the actions and processes of the calculus. Values a, x, m, n are implemented as functional data types, where separate constructors are used to distinguish between different types of values, including ambient and channel identifiers, tuples, constants, etc.

4.2.3 Implementing Substitution to Bind Values to Variables

Substitution is used to bind a value to a variable inside a process. This is needed to bind newly created channels and values received over channels during execution. The definition of substitution for actions and processes is summarised in Definition 4.2.3, together with the corresponding implementation. By definition, P_σ denotes the application of a substitution σ to a process P , where σ is a substitution that maps a given value to another (different) value, and v_σ applies the substitution σ to the value v . The expression $\sigma \setminus S$ removes each value in the set S from the domain of σ . If v is not in the domain of σ then $v_\sigma = v$.

- A substitution σ is implemented as a list of value pairs, where each pair (n, m) assigns the value n to a corresponding value m .
- The application v_σ of a substitution σ to a value v is implemented by the function *vbind*. If the substitution s contains the pair (n, v) then *vbind* s v returns the value n , otherwise it returns the initial value v .
- The removal $\sigma \setminus l$ of a set of values l from a substitution σ is implemented by the function

type process =	$P, Q, R ::=$	0	Null
Null		$P \mid Q$	Parallel
Parallel of process*process		$\nu n P$	Restriction
Restrict of value*process		$a \boxed{P}$	Ambient
Ambient of value*process		$\alpha.P$	Action
Action of action*process		$!\alpha.P$	Replication
Replicate of action*process			
type action =	$\alpha ::=$	$a \cdot x \langle n \rangle$	Sibling Output
Sibling of value*value*value		$x^\dagger \langle n \rangle$	Parent Output
Parent of value*value		$x(m)$	Internal Input
External of value*pattern		$x^\dagger(m)$	External Input
Internal of value*pattern		$\text{in } a \cdot x$	Enter
Enter of value*value		$\text{out } x$	Leave
Leave of value		$\overline{\text{in}} x$	Accept
Accept of value		$\overline{\text{out}} x$	Release
Release of value			
type term = element list	$V ::=$	$\nu n V$	Restriction
		A	List
type element =	$A, B, C ::=$	$[]$	Empty
Act of action*state*process		$\alpha.P :: C$	Action
Amb of value*state*(element list)		$a \boxed{A} :: C$	Ambient
type state =	$z ::=$	\underline{z}	Blocked
Blocked		z	Unblocked
Ok			

Definition 4.2.2. Implementation of Runtime Terms

remove. For each value v in a list l , if a substitution s contains the pair (n, v) then *remove* l s returns s without the pair (n, v) .

- The set of bound values $\text{bn}(\alpha)$ of an action α is implemented by the function *bound*, where *bound* k returns the list of values in k that are bound by an input.
- The application α_σ of a substitution σ to an action α is implemented by the function *kbind*, where *kbind* s k applies the substitution s to each of the values in k .
- Similarly, the application P_σ of a substitution σ to a process P is implemented by the function *bind*, where *bind* s P applies the substitution s to each of the processes, actions or values in P . Since the application of s to P can result in the removal of pairs of values from s , a gain in efficiency can be obtained by checking whether s is empty before attempting to

let rec bind (s:bindings) (p:process) =	$\mathbf{0}_\sigma \triangleq \mathbf{0}$
if s=[] then p else match p with	$(P \mid Q)_\sigma \triangleq P_\sigma \mid Q_\sigma$
Null -> Null	$(\nu n P)_\sigma \triangleq \nu n P_{\sigma \setminus \{n\}}$
Parallel(p,q) ->	$a \boxed{P}_\sigma \triangleq a_\sigma \boxed{P_\sigma}$
Parallel(bind s p, bind s q)	$(\alpha.P)_\sigma \triangleq \alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)}$
Restrict(n,p) ->	$!(\alpha.P)_\sigma \triangleq !(\alpha.P)_\sigma$
Restrict(n,bind (remove [n] s) p)	
Ambient(a,p) ->	
Ambient(vbind s a,bind s p)	
Action(k,p) ->	
let s' = (remove (bound k) s)	
in Action(kbind s k,bind s' p)	
Replicate(k,p) ->	
let s' = (remove (bound k) s)	
in Replicate(kbind s k,bind s' p)	
let kbind (s:bindings) (k:action) =	
match k with	$a \cdot x \langle n \rangle_\sigma \triangleq a_\sigma \cdot x_\sigma \langle n_\sigma \rangle$
Internal(x,m) -> Internal(vbind s x,m)	$x^\uparrow \langle n \rangle_\sigma \triangleq x_\sigma^\uparrow \langle n_\sigma \rangle$
External(x,m) -> External(vbind s x,m)	$x(m)_\sigma \triangleq x_\sigma(m)$
Parent(x,n) -> Parent(vbind s x,vbind s n)	$x^\uparrow(m)_\sigma \triangleq x_\sigma^\uparrow(m)$
Sibling(a,x,n) ->	$(\text{in } a \cdot x)_\sigma \triangleq \text{in } a_\sigma \cdot x_\sigma$
Sibling(vbind s a,vbind s x,vbind s n)	$(\text{out } x)_\sigma \triangleq \text{out } x_\sigma$
Enter(a,x) -> Enter(vbind s a,vbind s x)	$(\overline{\text{in } x})_\sigma \triangleq \overline{\text{in } x}_\sigma$
Leave(x) -> Leave(vbind s x)	$(\overline{\text{out } x})_\sigma \triangleq \overline{\text{out } x}_\sigma$
Accept(x) -> Accept(vbind s x)	
Release(x) -> Release(vbind s x)	

Definition 4.2.3. Implementation of Substitution

apply s to P .

By convention, the application of a substitution σ to a restriction $\nu n P$ assumes that the bound value n is not in the set of substituted values given by the range of σ . Similarly, the application of σ to an action α assumes that none of the bound values $\text{bn}(\alpha)$ of α are in the set of substituted values. These assumptions are enforced by using a functional datatype to syntactically distinguish between the bound values of a process (also called *variables*) and other types of values.

4.2.4 Implementing Construction to Encode a Process to a Runtime Term

Construction is used to encode a process into a corresponding term so that it can be executed by the runtime. Construction is also used to add a newly created process to a term during execution. A given process P is encoded by adding it to an empty term using the construction operator $(:)$. The definition of construction is summarised in Definition 4.2.4, together with the corresponding

let rec (*) (p:process) (t:term) =	$n \notin \text{fn}(P) \Rightarrow P:(\nu n V) \triangleq \nu n (P:V)$
match p with	$\mathbf{0}:A \triangleq A$
Null -> t	$(P \mid Q):A \triangleq P:Q:A$
Parallel(p,q) ->	$n \notin \text{fn}(P:A) \Rightarrow (\nu m P):A \triangleq \nu n (P_{\{n/m\}}:A)$
p*(q*t)	$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P:A \triangleq \alpha.(P \mid !\alpha.P)::A$
Restrict(m,p) ->	$a[\boxed{P}]:A \triangleq a[\boxed{P:\Box}]:A$
(bind [fresh m,m] p)*t	$\alpha.P:A \triangleq \alpha.P::A$
Replicate(k,p') ->	$n \notin \text{fn}(a,V) \Rightarrow a[\boxed{\nu n U}]:V \triangleq \nu n (a[\boxed{U}]:V)$
Act(k,Ok,Parallel(p',p))::t	$n \notin \text{fn}(a,A) \Rightarrow a[\boxed{A}]:\nu n V \triangleq \nu n (a[\boxed{A}]:V)$
Ambient(a,p) ->	$a[\boxed{A}]:C \triangleq a[\boxed{A}]::C$
Amb(a,Ok,p*[])::t	
Action(k,p) ->	
Act(k,Ok,p)::t	
	$\llbracket P \rrbracket \triangleq P:\Box$

let encode (p:process) = p*[]

Definition 4.2.4. Implementation of Construction

implementation. The construction $P:V$ adds the process P to the term V , and the construction $a[\boxed{U}]:V$ adds the ambient a and the term U to the term 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. 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, and then placing the ambient at the head of the list inside the scope of the restricted names. Construction is implemented by the function $(*)$, where $p*t$ adds the process p to the term t . An infix operator syntax $(*)$ is used instead of a prefix function in order to improve code readability.

When a replicated action $!\alpha.P$ is expanded, it is assumed that the free values of α are not in the set of bound values of α . This ensures that channel names are not captured during the expansion of a replicated input. Since the implementation uses separate constructors for bound and free values, these two sets of values cannot overlap, thereby allowing replications to be safely expanded.

When a process P or an ambient $a[\boxed{A}]$ is added to a term $\nu n V$, the value n is assumed not to be in the set of free values of P or of (a,A) , respectively. Similarly, when an ambient a with a term $\nu n U$ is added to a term V , the value n is assumed not to be in the set of free values of (a,V) . These assumptions can only hold if, whenever a restriction $\nu m P$ is added to a term A , the bound value m is replaced with a value n that is globally unique. This ensures that the scope of n can always be extended to the top-level. Since top-level restricted names implicitly become part of the private address space of the runtime, the rules for extending the scope of a restricted value in a term do not need to be implemented explicitly.

The generation of globally unique values is implemented by the function *fresh*, where *fresh m* returns a globally unique value based on the initial value *m*. This is achieved by adding a unique time stamp to *m*, together with a globally unique address. A separate constructor is used to distinguish between restricted values and free values, in order to prevent restricted values from being guessed outside the intended scope of the restriction.

4.2.5 Implementing Selection to Schedule a Runtime Term

Selection is used to choose a given ambient or action from inside a term so that it can be executed by the runtime. This is needed to schedule the next action or ambient to be executed, and to check whether a given action can interact with a corresponding co-action. The definition of selection is summarised in Definition 4.2.5 together with the corresponding implementation, where @ denotes the standard list append function. The selection $A \succ A'$ re-arranges the runtime term *A* to match the term *A'*. The matching can be made more or less precise by modifying the term *A'* accordingly. For example, $A \succ \alpha.P :: A'$ selects an arbitrary action $\alpha.P$ from *A*, whereas $A \succ a \cdot x\langle n \rangle.P :: A'$ selects a sibling output $a \cdot x\langle n \rangle.P$ from *A*. Depending on the context in which the selection is made, the values *a*, *x* and *n* can refer to arbitrary values, or to a specific ambient *a*, channel *x* and value *n*. This varying precision of matching can be implemented by defining multiple selection functions, one for each type pattern to be matched. Four main types of pattern can be identified, each requiring its own selection function:

select_element t selects an arbitrary unblocked action or ambient from the term *t*. This is used to select the next action or ambient to be executed.

select_action k t selects a specific blocked action *k* from the term *t*. This makes use of a function *matches k k0*, which checks whether the action *k* matches the action *k0*. The matching is weaker than a standard equality test, since it does not distinguish between values that are sent or received over channels.

select_child a k t selects an ambient *a* containing a blocked action *k* from the term *t*.

select_any_child k t selects any ambient containing a blocked action *k* from the term *t*.

Note that the selection $B \succ A@A'$ is used to split a given term *B* into two parts *A* and *A'*. In practice, this is implemented by using an accumulator *A* and recursing down the structure of *B* until a term that matches *A'* is found. The accumulator *A* is initialised to empty [] and elements are appended to *A* as the recursion progresses. A gain in efficiency can be achieved by adding elements to the *front* of *A* during the recursion, building the accumulator in reverse order, and then using the more efficient tail-recursive function *rev_append* to append *A'* to the reverse of *A*.

$$\begin{aligned}
A @ \alpha . P :: A' &\succ \alpha . P :: A @ A' \\
A @ b \boxed{B} :: A' &\succ b \boxed{B} :: A @ A' \\
A \succ A' &\Rightarrow a \boxed{A} :: C \succ a \boxed{A'} :: C
\end{aligned}$$

```

let select_element (t:term) =
  let rec select (t:term) (t':term) = match t' with
    [] -> []
  | Act(k,Ok,p)::t' -> Act(k,Ok,p)::t@t'
  | Amb(a,Ok,tA)::t' -> Amb(a,Ok,tA)::t@t'
  | e::t' -> select (t@[e]) t'
  in select [] t

let select_action (k0:action) (t:term) =
  let rec select (t:term) (t':term) = match t' with
    [] -> []
  | Act(k,Blocked,p)::t' ->
    if (matches k k0) then Act(k,Blocked,p)::t@t'
    else select (t@[Act(k,Blocked,p)]) t'
  | e::t' -> select (t@[e]) t'
  in select [] t

let select_child (a0:value) (k0:action) (t:term) =
  let rec select (t:term) (t':term) = match t' with
    [] -> []
  | Amb(a,z,tA)::t' ->
    if a = a0 then match select_action k0 tA with
      Act(k,Blocked,q)::tA -> Amb(a,z,Act(k,Blocked,q)::tA)::t@t'
    | _ -> select (t@[Amb(a,z,tA)]) t'
    else select (t@[Amb(a,z,tA)]) t'
  | e::t' -> select (t@[e]) t'
  in select [] t

let select_any_child (k0:action) (t:term) =
  let rec select (t:term) (t':term) = match t' with
    [] -> []
  | Amb(a,z,tA)::t' -> ( match select_action k0 tA with
    Act(k,Blocked,q)::tA -> Amb(a,z,Act(k,Blocked,q)::tA)::t@t'
  | _ -> select (t@[Amb(a,z,tA)]) t'
  )
  | e::t' -> select (t@[e]) t'
  in select [] t

```

Definition 4.2.5. Implementation of Selection

Selection in the Channel Ambient Machine is non-deterministic, which means that the elements of a list can be selected in any order. In contrast, the implementation of selection in Definition 4.2.5 is deterministic, since it selects the first available element that matches the selection criteria. Based on the non-determinism of selection in the abstract machine, a variety of

alternative implementations of selection are possible, some of which are discussed in Section 4.4.

4.2.6 Implementing Unblocking to Prevent Deadlocks During Execution

<pre> let rec unblock (t:term) = match t with [] -> [] Act(k,z,p)::tC -> Act(k,Ok,p)::(unblock tC) Amb(a,z,tA)::tC -> Amb(a,z,tA)::(unblock tC) </pre>	$ \begin{aligned} [] &\triangleq [] \\ [\alpha.P::C] &\triangleq \alpha.P::[C] \\ [q\boxed{A}::C] &\triangleq q\boxed{A}::[C] \end{aligned} $
---	---

Definition 4.2.6. Implementation of Unblocking

Unblocking 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 surroundings. The definition of unblocking is summarised in Definition 4.2.6, together with the corresponding implementation. The unblocking function $[A]$ unblocks all of the top-level actions in a given list A . Nested ambients inside A remain unchanged, since they cannot interact directly with the new environment.

4.2.7 Implementing Reduction to Execute a Runtime Term

Reduction is used to execute a term of the Channel Ambient Runtime. By definition, a runtime term is a list of actions and ambients with a number of top-level restricted names:

$$\nu n \dots \nu n' \alpha.P :: \dots :: \alpha'.P' :: q\boxed{A} :: \dots :: q'\boxed{A'} :: []$$

The actions and ambients in the list can be either blocked or unblocked, and each ambient contains its own internal list. This gives rise to a tree structure, in which the leaves of the tree are actions and the nodes are ambients. The runtime executes a given list by selecting an arbitrary unblocked element in the list. If an action $\alpha.P$ is selected then the runtime looks for a corresponding blocked co-action. If a suitable blocked co-action is found then an interaction can occur, otherwise the action $\alpha.P$ is blocked to $\underline{\alpha}.P$. If an ambient $a\boxed{A}$ is selected then the runtime tries to recursively execute the list A by selecting an arbitrary unblocked element in A . If no unblocked elements are present then the ambient is blocked to $\underline{a}\boxed{A}$. The runtime continues performing reductions in this way until all the elements in the term are blocked, at which point execution terminates.

$$\begin{aligned}
C \succ b \boxed{x^\dagger(m).Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{b \cdot x \langle n \rangle . P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P :: A} : b \boxed{Q_{\{n/m\}} :: B} :: C'} :: D \\
C \not\succ b \boxed{x^\dagger(m).Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{b \cdot x \langle n \rangle . P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{b \cdot x \langle n \rangle . P :: A} :: C} :: D \\
C \succ b \boxed{a \cdot x \langle n \rangle . Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{x^\dagger(m).P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P_{\{n/m\}} :: A} : b \boxed{Q :: B} :: C'} :: D \\
C \not\succ b \boxed{a \cdot x \langle n \rangle . Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{x^\dagger(m).P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{x^\dagger(m).P :: A} :: C} :: D
\end{aligned}$$

$$\begin{aligned}
C \succ x(m).Q :: C' &\Rightarrow c \boxed{a \boxed{x^\dagger \langle n \rangle . P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P :: A} : Q_{\{n/m\}} :: C'} :: D \\
C \not\succ x(m).Q :: C' &\Rightarrow c \boxed{a \boxed{x^\dagger \langle n \rangle . P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{x^\dagger \langle n \rangle . P :: A} :: C} :: D \\
A \succ b \boxed{x^\dagger \langle n \rangle . Q :: B} :: A' &\Rightarrow c \boxed{a \boxed{x(m).P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P_{\{n/m\}} : b \boxed{Q :: B} :: A'} :: C} :: D \\
C \not\succ b \boxed{x^\dagger \langle n \rangle . Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{x(m).P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{x(m).P :: A} :: C} :: D
\end{aligned}$$

$$\begin{aligned}
C \succ b \boxed{\overline{\text{in}} x.Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{\text{in } b \cdot x.P :: A} :: C} :: D \longrightarrow c \boxed{b \boxed{Q : a \boxed{P : [A]} :: B} :: C'} :: D \\
C \not\succ b \boxed{\overline{\text{in}} x.Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{\text{in } b \cdot x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{\text{in } b \cdot x.P :: A} :: C} :: D \\
C \succ b \boxed{\text{in } a \cdot x.Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{\overline{\text{in}} x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P : b \boxed{Q : [B]} :: A} :: C'} :: D \\
C \not\succ b \boxed{\text{in } a \cdot x.Q :: B} :: C' &\Rightarrow c \boxed{a \boxed{\overline{\text{in}} x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{\overline{\text{in}} x.P :: A} :: C} :: D
\end{aligned}$$

$$\begin{aligned}
C \succ \overline{\text{out}} x.Q :: C' &\Rightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D \longrightarrow c \boxed{Q : C' : a \boxed{P : [A]} :: D} \\
C \not\succ \overline{\text{out}} x.Q :: C' &\Rightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D \\
A \succ b \boxed{\text{out } x.Q :: B} :: A' &\Rightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{P : A' : b \boxed{Q : [B]} :: C} :: D} \\
A \not\succ b \boxed{\text{out } x.Q :: B} :: A' &\Rightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D \longrightarrow c \boxed{a \boxed{\text{out } x.P :: A} :: C} :: D
\end{aligned}$$

$$a \boxed{b \boxed{B} :: A} :: C \longrightarrow V' \Rightarrow c \boxed{a \boxed{b \boxed{B} :: A} :: C} :: D \longrightarrow c \boxed{V'} :: D$$

$$A \not\succ \alpha.P :: A' \wedge A \not\succ b \boxed{B} :: A' \Rightarrow c \boxed{a \boxed{A} :: C} :: D \longrightarrow c \boxed{a \boxed{A} :: C} :: D$$

$$B \succ A \wedge A \longrightarrow V' \Rightarrow B \longrightarrow V'$$

$$V \longrightarrow V' \Rightarrow \nu n V \longrightarrow \nu n V'$$

Definition 4.2.7. Uniform Reduction

The definition of reduction is summarised in Definition 4.2.7. Each reduction rule is of the form $V \longrightarrow V'$, which states that the term V can evolve to the term V' during a single execution step. The reduction rules are based on the rules presented in Chapter 3, with the additional constraint that the left hand side most rules is of the form:

$$c \boxed{a \boxed{A} :: C} :: D$$

This corresponds to a *uniform* representation of the reduction rules, which contains the maximal context needed to reduce an arbitrary unblocked action or ambient inside the term A . For each of the possible unblocked actions α that can be selected from A there exists a reduction rule of the form:

$$c \boxed{a \boxed{\alpha.P :: A'} :: C} :: D$$

For example, if α is an internal input $x(m).P$ then the runtime can look for a corresponding output inside A before deciding whether to block the input or to receive a value. Similarly, if α is a leave out $x.P$ then the runtime can look for a corresponding release inside C before deciding whether to block the leave or to let the ambient a leave its parent. The runtime is also able to reduce an arbitrary unblocked ambient b that can be selected from A :

$$c \boxed{a \boxed{b \boxed{B} :: A'} :: C} :: D$$

If the ambient contains an unblocked action or ambient then a reduction can occur recursively, otherwise the ambient will block. Therefore, the uniform reduction rules allow the runtime to perform a reduction simply by selecting an arbitrary unblocked element inside A . This gives rise to a simplified execution algorithm for reducing runtime terms.

An important property of uniform reduction is that it should preserve the correctness of the Channel Ambient Machine. This can be ensured by defining a uniform syntax for runtime terms, which requires a term to be of the form:

$$\nu \tilde{z} c \boxed{a_1 \boxed{A_1} :: \dots :: a_N \boxed{A_N} :: []} :: [] \quad (4.1)$$

For all runtime terms of this form it can easily be shown that reduction and uniform reduction coincide. It can also be shown that the set of uniform runtime terms, denoted by $|\text{CAM}|$, is preserved by reduction. According to Proposition 4.2.8, if A is a uniform runtime term and the runtime reduces A to A' , then the resulting term A' is also a uniform runtime term.

Proposition 4.2.8. (*Uniform Reduction*) $A \in |\text{CAM}| \wedge A \longrightarrow A' \Rightarrow A' \in |\text{CAM}|$

Proof. By straightforward induction on the definition of reduction in CAM. □

```

let reduce_action (k:action) (p:process) a tA c tC tD =
  let tD':term = Amb(c,Ok,Amb(a,Ok,Act(k,Blocked,p)::tA)::tC)::tD
  in match k with
    Sibling(b,x,n) -> ( match select_child b (External(x,m0)) tC with
      Amb(b,_,Act(External(x,m),Blocked,q)::tB)::tC ->
        Amb(c,Ok,Amb(a,Ok,p*tA)::Amb(b,Ok,(bind [n,m] q)*tB)::tC)::tD
      | _ -> tD' )
    | External(x,m) -> ( match select_any_child (Sibling(a,x,v0)) tC with
      Amb(b,_,Act(Sibling(a,x,v),Blocked,q)::tB)::tC ->
        Amb(c,Ok,Amb(a,Ok,(bind [v,m] p)*tA)::Amb(b,Ok,q*tB)::tC)::tD
      | _ -> tD' )
    | Parent(x,v) -> ( match select_action (Internal(x,m0)) tC with
      Act(Internal(x,m),Blocked,q)::tC ->
        Amb(c,Ok,Amb(a,Ok,p*tA)::((bind [v,m] q)*tC))::tD
      | _ -> tD' )
    | Internal(x,m) -> ( match select_any_child (Parent(x,v0)) tA with
      Amb(b,_,Act(Parent(x,v),Blocked,q)::tB)::tA ->
        Amb(c,Ok,Amb(a,Ok,(bind [v,m] p)*(Amb(b,Ok,q*tB)::tA))::tC)::tD
      | _ -> tD' )
    | Enter(b,x) -> ( match select_child b (Accept(x)) tC with
      Amb(b,_,Act(Accept(x),Blocked,q)::tB)::tC ->
        Amb(c,Ok,Amb(b,Ok,q*(Amb(a,Ok,p*(unblock tA))::tB))::tC)::tD
      | _ -> tD' )
    | Accept(x) -> ( match select_any_child (Enter(a,x)) tC with
      Amb(b,_,Act(Enter(a,x),Blocked,q)::tB)::tC ->
        Amb(c,Ok,Amb(a,Ok,p*(Amb(b,Ok,q*(unblock tB))::tA))::tC)::tD
      | _ -> tD' )
    | Leave(x) -> ( match select_action (Release(x)) tC with
      Act(Release(x),Blocked,q)::tC ->
        Amb(c,Ok,q*tC)::Amb(a,Ok,p*(unblock tA))::tD
      | _ -> tD' )
    | Release(x) -> ( match select_any_child (Leave(x)) tA with
      Amb(b,_,Act(Leave(x),Blocked,q)::tB)::tA ->
        Amb(c,Ok,Amb(a,Ok,p*tA)::Amb(b,Ok,q*(unblock tB))::tC)::tD
      | _ -> tD' )

let rec reduce_ambient a tA c tC tD = match select_element tA with
  Act(k,Ok,p)::tA -> reduce_action k p a tA c tC tD
  | Amb(b,Ok,tB)::tA -> Amb(c,Ok,reduce_ambient b tB a tA tC)::tD
  | _ -> Amb(c,Ok,Amb(a,Blocked,tA)::tC)::tD

let reduce (t:term) = match t with
  Amb(c,Ok,tC)::[] -> ( match select_element tC with
    Amb(a,Ok,tA)::tC -> reduce_ambient a tA c tC []
    | _ -> [] )
  | _ -> []

```

Definition 4.2.9. Implementation of Uniform Reduction

The corresponding implementation of reduction is summarised in Definition 4.2.9. The implementation is derived almost directly from the reduction rules in Definition 4.2.7. Note that

since top-level restricted values are stored in the private address space of the runtime, the rule for reducing a term inside a restricted value does not need to be implemented explicitly:

reduce_action $k\ p\ a\ tA\ c\ tC\ tD$ tries to reduce an unblocked action k inside a term of the form $c\ a\ [k.p::tA]::tC::tD$, according to the first 16 reduction rules of Definition 4.2.7. For each action k an appropriate selection function is used to try to find a corresponding blocked co-action inside the term. If a blocked co-action is present then an interaction can occur, otherwise the action k is blocked. Note that the order in which the different actions are matched is not significant, since all of the matchings are distinct.

reduce_ambient $a\ tA\ c\ tC\ tD$ tries to reduce an unblocked ambient a inside a term of the form $c\ a\ [tA]::tC::tD$. First, the *select_element* function is used to select an arbitrary unblocked action or ambient from tA . If an unblocked ambient is chosen then the *reduce_ambient* function is called recursively. If an unblocked action is chosen then the *reduce_action* function is called. If no unblocked actions or ambients are present then the ambient a is blocked.

reduce t tries to reduce a term t that is assumed to be uniform, according to (4.1). First, the *select_element* function is used to choose an arbitrary unblocked ambient in c . The *reduce_ambient* function is then used to reduce the chosen ambient. If no unblocked ambients are present inside c then the *reduce* function returns an empty term.

The implementation of the *reduce_action* function can be illustrated using the reduction rules for sibling output:

$$\begin{aligned} C \succ b\ [x^\uparrow(m).Q::B]::C' &\Rightarrow c\ a\ [b \cdot x\langle n \rangle.P::A]::C::D \longrightarrow c\ a\ [P::A]::b\ [Q_{\{n/m\}}::B]::C'::D \\ C \not\succ b\ [x^\uparrow(m).Q::B]::C' &\Rightarrow c\ a\ [b \cdot x\langle n \rangle.P::A]::C::D \longrightarrow c\ a\ [b \cdot x\langle n \rangle.P::A]::C::D \end{aligned}$$

The first rule allows a sibling output to interact with a blocked external input. If no blocked external input is present then the second rule blocks the sibling output. The corresponding program code is derived almost directly from these two rules, using a single case for sibling output:

```
Sibling(b,x,n) -> ( match select_child b (External(x,m0)) tC with
  Amb(b,_,Act(External(x,m),Blocked,q)::tB)::tC ->
    Amb(c,Ok,Amb(a,Ok,p*tA)::Amb(b,Ok,(bind [n,m] q)*tB)::tC)::tD
  | _ -> Amb(c,Ok,Amb(a,Ok,Act(Sibling(b,x,n),Blocked,p)::tA)::tC)::tD )
```

The cases for each of the remaining actions are defined in a similar manner, as described in Definition 4.2.9.

4.3 Distributed Runtime Implementation

This section describes how the Channel Ambient Machine can be used to implement a distributed runtime, which executes a given calculus process over multiple devices in a hierarchical TCP/IP network. The section is structured as follows:

Subsection 4.3.1 describes how runtime terms can be implemented in a distributed setting.

Subsection 4.3.2 describes the architecture of the distributed runtime, and shows how the main components are used to execute a process of the Channel Ambient calculus in a hierarchical TCP/IP network.

Subsection 4.3.3 describes the implementation of a function for executing a runtime term in a distributed setting.

Subsection 4.3.4 describes the implementation of a daemon to manage the interactions between runtimes in a distributed setting.

4.3.1 Implementing Runtime Terms in a Distributed Setting

The Channel Ambient Runtime is assumed to execute on networks that support the TCP/IP protocol. According to the Channel Ambient Machine presented in Chapter 3, the syntax of runtime terms is constrained in a distributed setting 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 a socket address consisting of an IP address and a port number, while the name of an agent corresponds to a simple identifier.

In a hierarchical network topology, sites can be logically contained inside other sites to form Local Area Networks. A given site can also act as a *gateway* between two networks: the local network it contains and the global network in which it is contained. As a result, a gateway site is usually assigned two network addresses, one for the local network and one for the global network. The global address corresponds to main address of the site, whereas the local address is merely an implementation mechanism for routing messages or agents to a default parent site.

An example of how sites can be used to model the topology of Local and Wide Area Networks is illustrated in Figure 4.2. In this example, each site executes on a separate machine using port number 3000. The sites 192.168.0.2:3000 - 92.168.0.4:3000 are part of a Local Area Network inside a gateway site with local address 192.168.0.1:3000 and global address 82.35.60.43:3000. The ambients inside the LAN can 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 can send messages

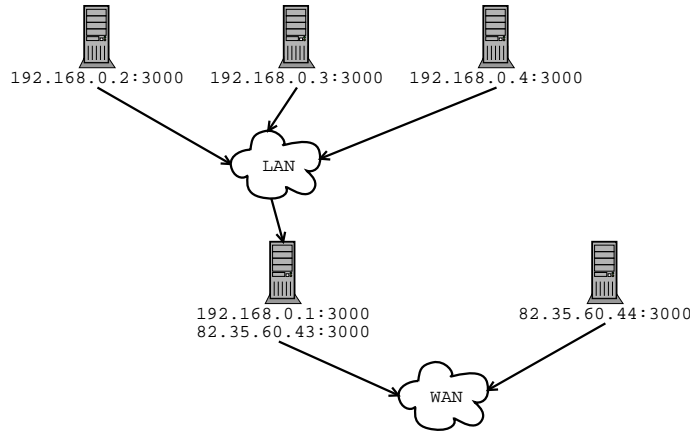


Figure 4.2: Hierarchical Network Topology

or agents to the global address of the gateway. In this way, the local and global addresses allow the gateway to distinguish between local and global interactions.

A runtime term containing multiple sites is executed in a distributed setting 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 the of the distributed runtimes. As with the local runtime, they 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:

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

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.

4.3.2 Architecture of the Distributed Runtime

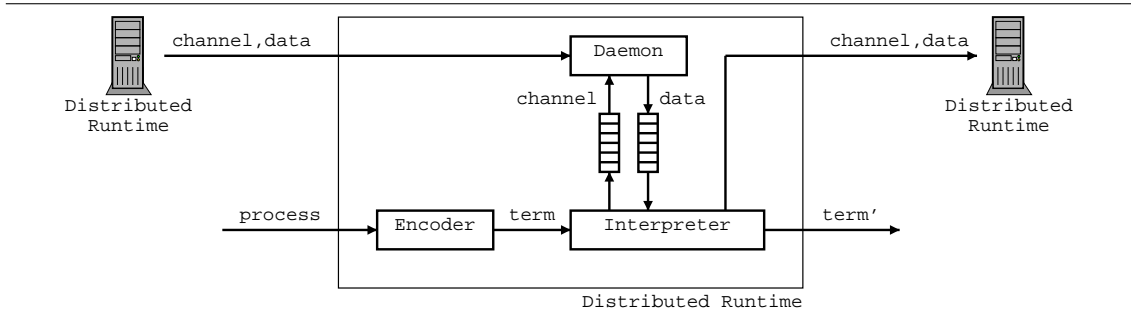


Figure 4.3: Architecture of the Distributed Runtime

The architecture of the distributed runtime is described in Figure 4.3. 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.

```

let run (m:value) (s:value) (l:value) (p:process) =
  let t:term = Amb(m,Ok,Amb(s,Ok,encode p)::[]):[] in
  let ds:data buffer = empty in
  let cs:data buffer = empty in
  let z1:thread = Thread.create interpret (ds,cs,t) in
  let z2:thread = Thread.create daemon (ds,cs,s,l)
  in Thread.join z1; Thread.join z2

let rec interpret ((ds:data buffer),(cs:channel buffer),(t:term)) =
  let rec interpret (t:term) = match reduce cs t with
    [] -> let l:data list = await_data ds
          in interpret (add_data l t)
    | t' -> match poll_data ds with
          Some(l) -> interpret (add_data l t')
          | None -> interpret t'
  in interpret t

```

Definition 4.3.1. Top-Level Program Code for the Distributed Runtime

The top-level program code for the distributed runtime is presented in Definition 4.3.1:

$run\ m\ s\ l\ p$ executes the process p on a distributed runtime with address s , local address l and parent runtime with local address m . The arguments are used to construct a runtime term of the form $m\ s\ \boxed{s\ p::[]}::[]$ with the help of the *encode* function. An empty data buffer ds is then initialised, together with an empty channel buffer cs . The buffers can be accessed using synchronous communication channels that are part of the buffer data structure. A parallel

interpret thread is then created to execute the runtime term, together with a parallel *daemon* thread to accept data from distributed runtimes.

encode p is unchanged from the local runtime.

interpret (ds, cs, t) repeatedly reduces the term t with the help of the *reduce* function. After each reduction the data buffer ds is polled for incoming data, which is added to the runtime term. If no reductions are possible then the interpreter goes into a blocked state, waiting for an interrupt from the data buffer to signal the arrival of new data.

reduce $cs\ t$ tries to reduce the term t in order to perform a single execution step. If a reduction is possible then an updated term t' is returned, otherwise the an empty term is returned. During a reduction, a given channel can be added to the channel buffer cs in order to notify the daemon to accept data on this channel.

daemon (ds, cs, s, l) executes a daemon to receive data from distributed runtimes. The addresses s and l are used to accept data from external and internal runtimes, respectively. The channel buffer cs stores a list of channels on which data can be accepted. Any data received by the daemon is forwarded to the interpreter via the data buffer ds .

4.3.3 Implementing Reduction in a Distributed Setting

In Chapter 3, the reduction rules of the Channel Ambient Machine are constrained in a distributed setting to prevent a sibling output, parent output or enter to a site from blocking. These constraints are necessary because the corresponding external input, internal input and accept actions inside a site do not specify a given ambient with which to interact. As a result, they can potentially interact with an arbitrary ambient in the network. Such interactions can be readily implemented in a local area network by broadcasting to all the ambients in the network, but do not scale to wide area networks with potentially large numbers of ambients. One way to resolve the issue is to prevent remote actions to a site from blocking, as described in Chapter 3. This ensures that the corresponding co-actions will block systematically, allowing the remote actions to continue polling until an interaction occurs. Since these remote actions are always addressed to a specific site in the network, no broadcasting is necessary. Note that in some cases an interaction may never occur and the remote action will continue polling indefinitely. In these cases the interval between polling can be increased exponentially over time, in order to avoid causing a potential denial of service attack.

The interaction between remote runtimes can be further simplified by treating all parent outputs inside a site as system calls. This is achieved by only allowing a parent output inside a site on a limited number of *system* channels. For example, a parent output $print^\dagger\langle s \rangle$ can be used to print

the value s on a user console. Similarly, a parent output $\text{parent}^\dagger\langle x, n \rangle$ can be used to send a fresh agent $z \boxed{x^\dagger\langle n \rangle}$ to the parent, in order to perform an output locally inside the parent. The main advantage of this simplification is that local inputs inside a runtime do not need to synchronise over a network. Instead, all parent outputs are intercepted inside by the runtime as system calls, which are either processed locally or sent to the parent runtime as lightweight agents.

The implementation of reduction in a distributed setting is summarised in Definition 4.3.2. 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.

When executing a given term of the form $c \boxed{a \boxed{\alpha.P :: A} :: C} :: D$, the runtime distinguishes between local and remote actions as follows. If c is an agent then the runtime performs a local reduction. If c is a site then the runtime performs a local or remote reduction, depending on the action α :

- If α is a sibling output to a remote site b then the runtime tries to send a sibling output to b using the function *remote_sibling*.
- If α is an enter to a remote site b then the runtime tries to move the agent a inside b using the function *remote_enter*.
- If α is a parent output inside a site then the runtime tries to treat this as a system call using the function *system_call*.
- If α is an external input or accept on a channel x inside a site then the runtime sends the channel x to the channel buffer using the function *block_remote*. This notifies the daemon to receive data on channel x , where the type of data received depends on the type of the channel.
- If α is not one of the above actions then the runtime performs a local reduction using the function *reduce_action*, as defined in the local runtime.

remote_action cs k p a tA c tC tD tries to reduce an unblocked action k inside a term of the form

$c \boxed{a \boxed{k.p :: tA} :: tC} :: tD$, according to the reduction rules of the Channel Ambient Machine. If

```

let remote_action (cs:channel buffer) (k:action) (p:process) a tA c tC tD =
  let tD':term = Amb(c,Ok,Amb(a,Ok,Act(k,Blocked,p)::tA)::tC)::tD in
  let tD'':term = Amb(c,Ok,Amb(a,Ok,Act(k,Ok,p)::tA)::tC)::tD in
  let t =
    if site c
    then match k with
      Sibling(b,x,n) ->
        if site b then
          if remote_sibling b x n
          then Amb(c,Ok,Amb(a,Ok,p*tA)::tC)::tD
          else tD''
        else []
      | External(x,m) -> if site a then (block_remote x cs; tD') else []
      | Enter(b,x) ->
        if site b then
          if remote_enter b x a tA
          then Amb(c,Ok,tC)::tD
          else tD''
        else []
      | Accept(x) -> if site a then (block_remote x cs; tD') else []
      | Parent(x,n) ->
        if site a then
          if system_call x n a tA
          then Amb(c,Ok,Amb(a,Ok,p*tA)::tC)::tD
          else tD'
        else []
      | _ -> []
    else []
  in if t=[] then reduce_action k p a tA c tC tD else t

let rec reduce_ambient (cs:channel buffer) a tA c tC tD =
  match select_element tA with
    Act(k,Ok,p)::tA -> remote_action cs k p a tA c tC tD
  | Amb(b,Ok,tB)::tA -> Amb(c,Ok,reduce_ambient cs b tB a tA tC)::tD
  | _ -> Amb(c,Ok,Amb(a,Blocked,tA)::tC)::tD

let reduce (cs:channel buffer) (t:term) = match t with
  Amb(c,Ok,tC)::[] -> (
    match select_element tC with
      Amb(a,Ok,tA)::tC -> proxy_ambient cs a tA c tC []
    | _ -> [] )
  | _ -> []

```

Definition 4.3.2. Distributed Reduction

c is a site then a can potentially interact with an ambient inside a remote runtime, depending on the action k .

$reduce_action\ k\ p\ a\ tA\ c\ tC\ tD$ is unchanged from the local runtime.

site a returns *true* if the ambient *a* is a site, and *false* otherwise.

remote_sibling b x n tries to send the value *n* on channel *x* to a remote site *b*. The value *true* is returned if the send succeeds, and *false* otherwise.

remote_enter b x a tA tries to move the agent *a* containing term *tA* over channel *x* to a remote site *b*. The value *true* is returned if the move succeeds, and *false* otherwise.

block_remote x cs adds the channel *x* to the channel buffer *cs*. Depending on the type of *x*, this allows the runtime to accept an agent on channel *x* or receive a message on channel *x* from a remote runtime.

system_call x n tA executes a system call *x* with arguments *n* inside a site with name *a* and contents *tA*. The system call is an arbitrary function that, for security reasons, can only be executed by the top-level site. The value *true* is returned if the system call succeeds, and *false* otherwise. System calls include I/O functions and network outputs to the parent site.

reduce_ambient cs a tA c tC tD is unchanged from the local runtime, except that the channel buffer *cs* is passed as an additional argument and the *remote_action* function is used instead of *reduce_action*.

reduce cs t is unchanged from the local runtime, except that the channel buffer *cs* is passed as an additional argument and the *proxy_ambient* function is used instead of *reduce_ambient*.

One aspect of the implementation that is not described explicitly in the reduction rules is the case where a mobile agent leaves a given site. The simplest way to handle this case is to systematically send the agent over the network to the parent site. In practice, however, this approach quickly leads to a centralised bottleneck, since any time an agent leaves a site on its way another site, it will automatically be sent via the parent. One way to decentralise the implementation is to keep the agent inside a proxy of the parent for as long as possible, and only move it to the real parent when necessary. By definition of the reduction rules, a given agent will only need to be moved from a proxy to the parent if it wants to interact directly with the parent or with a sibling agent. It does not need to leave the proxy if it only wants to interact with a child agent or a remote site.

proxy_ambient cs a tA c tC tD moves the ambient $a \boxed{tA}$ to the parent runtime as appropriate.

The simplest way to implement this function is to systematically move all agents to the parent runtime. A more sophisticated approach is described in Definition 4.3.3, in which an ambient *a* is only moved if it is an agent trying to perform a parent output, an external input, a sibling output to an agent or an enter to an agent. Otherwise, the ambient *a* is executed as usual, with the *reduce_ambient* function.

```

let proxy_ambient (cs:channel buffer) a tA c tC tD =
  let proxy (k:action) tA =
    if site a then false
    else match k with
      | Sibling(b,x,n) -> if (site b) then false else send_agent c a tA
      | Internal(x,m) -> false
      | Enter(b,x) -> if (site b) then false else send_agent c a tA
      | Release(x) -> false
      | _ -> send_agent c a tA
  in match select_element tA with
    | Act(k,Ok,p)::tA ->
      if proxy k (Act(k,Ok,p)::tA)
      then Amb(c,Ok,tC)::[]
      else remote_action cs k p a tA c tC tD
    | Amb(b,Ok,tB)::tA -> Amb(c,Ok,reduce_ambient cs b tB a tA tC)::tD
    | _ -> Amb(c,Ok,Amb(a,Blocked,tA)::tC)::tD

```

Definition 4.3.3. Distributed Reduction with proxy Sites

4.3.4 Implementing a Daemon to Manage Network Interactions

The distributed runtime uses a daemon to receive data from remote runtimes over a given channel. This data can be either a message or a mobile agent, depending on the type of the channel. If sites are implemented as IP addresses and channels as port numbers then the interaction between remote runtimes is implemented using TCP/IP directly. In practice, however, it is often useful to implement sites as socket addresses consisting of an IP address and port number, and to implement channels as arbitrary names. This gives increased program flexibility by allowing multiple sites to execute on the same IP address. As a result, a thin layer of multiplexing needs to be built on top of TCP/IP, in order to allow a given socket address to exchange data on multiple named channels.

Figure 4.4 describes an execution scenario for a network session between a runtime B and a remote runtime A , which makes use of a form of TCP/IP multiplexing:

1. The interpreter of the runtime B sends the channel x to the channel buffer during execution, in order to notify the daemon to accept data on this channel.
2. The interpreter of a remote runtime A connects to the daemon using the IP address and port number of the daemon.
3. The remote runtime sends the channel x to the daemon, requesting permission to send data on this channel.
4. The daemon checks that the channel x is present in the channel buffer, and then removes x from the buffer.

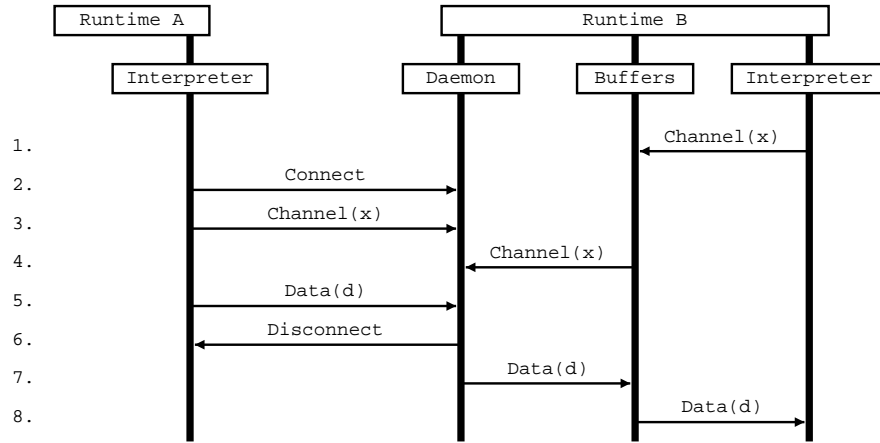


Figure 4.4: Protocol for Interaction between Sibling Runtimes

5. The daemon receives data from the remote runtime, and then checks that this data matches the type of channel x .
6. The daemon disconnects the remote runtime after receiving the data.
7. The daemon sends the received data to the interpreter via the data buffer.
8. The interpreter periodically checks the data buffer during execution, in order to receive any incoming data.

The above scenario represents a typical interaction session between two runtimes over a network. In general, the session can fail at a number of points. If the channel x is not present in the data buffer Step 4, or if the data received is not compatible with the type of channel x Step 5 then the daemon will abort the session. If the remote interpreter detects that the session has been aborted it will re-attempt the connection at a later date. Conversely, if the daemon detects that the session has been aborted it will replace any channels that were removed from the channel buffer. If the interpreter does not receive an error before the end of the session then it will assume that the data was sent successfully. However, there is also a chance that the interpreter may not detect a session failure, in which case the data is lost. Such failures can be modelled explicitly in the calculus, as described in Chapter 2.

4.4 Enhanced Runtime Implementation

This section describes how the distributed runtime can be enhanced in order to improve the efficiency of process execution, while conserving network bandwidth.

Subsection 4.4.1 describes how runtime terms can be implemented using a map data structure instead of a list, in order to improve the efficiency of selection.

Subsection 4.4.2 describes how a deterministic selection algorithm can be used in order to improve the efficiency of runtime execution, while ensuring some notion of fairness.

Subsection 4.4.3 describes how network masks can be used to conserve bandwidth, by preventing spurious attempts to interact with inaccessible network addresses.

Subsection 4.4.4 describes how modules can be used to structure the program code of the runtime, by grouping the functions for each of the main runtime data structures into a separate module.

4.4.1 Using a Map Data Structure to Improve Selection

By definition, a runtime term is a list of elements with a number of top-level restricted names, where the elements of the list are actions and ambients, and each element can be either blocked or unblocked. In order to perform a reduction, the runtime first tries to select an arbitrary unblocked action from the list, and then tries to select a corresponding blocked co-action. Four main selection functions are used, one for selecting an arbitrary unblocked action from the list and three for selecting a corresponding blocked co-action. In the basic runtime, these selection functions are implemented by linearly traversing the list of elements until a suitable action is found. Interestingly, a significant gain in efficiency can be obtained by sorting the list into three separate sub-lists, containing unblocked actions, blocked actions and ambients, respectively. This gives rise to a list of the form:

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

By grouping elements of similar type in this way, the selection functions can be optimised so that they only search the sub-list corresponding to the type of element being selected. This allows a given selection function to avoid searching through the entire list when looking for an element of a particular type. The sorting can be achieved by re-arranging the contents of the list according to the structural congruence rules of the runtime, as described in Chapter 3. These rules allow the elements of a list to be re-arranged by an arbitrary number of permutations, while still preserving the correctness of the runtime.

For improved efficiency, the list of blocked actions can be further sorted into sub-lists of blocked actions of similar type. At the finest level, a separate sub-list can be defined for each type of blocked action that is waiting to interact with a given ambient on a given channel. For example, a separate sub-list can be defined for blocked enter actions to a given ambient a over a given

channel x . Similarly, a separate sub-list can be defined for blocked leave, accept or release actions on a given channel x . A similar approach can be used for input and output actions, by ignoring the values being sent or received. For example, a separate sub-list can be defined for blocked sibling output actions to a given ambient a over a given channel x , ignoring the value v being sent. Similarly, a separate sub-list can be defined for blocked parent output, external input or internal input actions on a given channel x .

These sub-lists can be stored in a map data structure, indexed by the type of the action. A given blocked action can be selected by first looking up the corresponding sub-list for this action, and then searching through the sub-list in a linear fashion. The comparison function for looking up a given blocked action is slightly weaker than a standard equality test, since it does not compare values that are being sent or received. Map data structures typically use balanced binary trees that give a logarithmic lookup on average, as opposed to a linear lookup for lists. Since the lookup of a blocked action is required at each reduction step, a substantial gain in efficiency can be achieved by using maps. The gain in efficiency is even more apparent when trying to select a blocked action that is not present. If a list structure is used then the entire list needs to be traversed, whereas for a map structure only the sub-list for the particular action needs to be traversed. The current version of the Channel Ambient Runtime uses a map structure in this way to improve runtime efficiency.

4.4.2 Using Deterministic Selection to Improve Efficiency

Selection in the Channel Ambient Machine is non-deterministic by definition, which means that elements in a machine term can be selected in any order. Since the machine has been proved both sound and complete with respect to the Channel Ambient calculus, the correctness of the runtime will be preserved regardless of the order in which selection is implemented. This flexibility in the implementation provides numerous opportunities for runtime optimisation.

Perhaps the simplest way to optimise selection is to choose the first element in the list that satisfies the selection criteria. However, this approach can allow the runtime to get stuck in an infinite loop by repeatedly selecting the same elements. In general, an implementation of selection should satisfy some notion of fairness, to ensure that certain elements of the list are not repeatedly selected in favour of other elements. Recent work on defining fairness [81] unifies a number of contemporary definitions, which generally require a particular choice to be sufficiently often taken if sufficiently often possible. In the case of the Channel Ambient Runtime, one way to ensure fairness is to randomly select elements from the list with equal probability. This can be achieved by first shuffling the elements of the list and then selecting the first element. However, shuffling requires considerable overhead and is less efficient than most deterministic selection algorithms. A possible compromise is to select the elements of a list in a round-robin fashion, on a first-come,

first-serve basis.

The Channel Ambient Runtime distinguishes between four different selection functions, one for selecting an arbitrary unblocked action from a list and three others for selecting a given blocked action. Unblocked actions can be selected in a round-robin fashion by making sure they are selected in order of appearance. This can be achieved as follows. Given a sorted list of the form

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

the runtime selects the first action $\alpha_1.P_1$. According to the reduction rules, this action will either block to $\underline{\alpha_1}.P_1$ or interact with a suitable blocked co-action to become P_1 . If the action is blocked then it is placed at the back of the sub-list of blocked actions. Otherwise, the resulting process P_1 is added to the back of the list. Once all the actions $\alpha_1, \dots, \alpha_i$ have been selected, the lists inside the child ambients a_1, \dots, a_N are then executed, recursively. Once a given child ambient has been executed, the ambient is placed at the back of the list to ensure a cyclic, round-robin selection algorithm. Note that there is a slight variability in the selection algorithm when an ambient moves to a new location. In this case the ambient is placed at the back of the list, awaiting its turn to be executed. This form of round-robin scheduling is relatively straightforward to implement for the local runtime. For the distributed runtime, the inherent non-determinism of network interactions should be sufficient to ensure a suitable level of fairness over multiple runtimes in a network. Note that the remaining functions for selecting a blocked action can be implemented by simply choosing the first available blocked action in the list. This is because, if the order in which actions are selected is fair, then the order in which they are blocked will also be fair, and no additional re-ordering is necessary.

4.4.3 Using Network Masks to Conserve Bandwidth

The Channel Ambient Runtime uses socket communication in order to interact with remote runtimes over a network. These interactions take place when an ambient inside one runtime wishes to interact with a site inside a remote runtime. In order for an interaction to occur, the ambient must first specify the network address of a target site by performing either an enter or a sibling output to the site. In the general case the ambient can specify an arbitrary site, which may or may not be accessible in the current network. In order to conserve bandwidth, the Channel Ambient Runtime could use network masks to check whether a given site is accessible, before attempting to interact with the site. This can be achieved by annotating each site with a network mask indicating the range of IP addresses that the site is able to contain. When an ambient wishes to interact with a remote site, the runtime first checks the network mask of its parent to ensure that the address of the remote site is within the range of accessible addresses. Since the address of the parent is

stored locally, no network interaction is necessary to perform this check.

Network masks are a widely-used mechanism for configuring Local Area Networks. By definition, if a site s_1 with address IP_1 wants to send a message directly to a site s_2 with address IP_2 , the communication can only take place if s_2 is on the same local network, i.e. if

$$IP_1 \wedge IP_0 = IP_2 \wedge IP_0$$

Where \wedge is the bitwise AND between two IP addresses, and IP_0 is the network mask. Thus, network masks are a simple means of checking whether two sites are on the same local network, without having to perform any network communication. If the check succeeds, the communication can take place directly. Otherwise, the communication needs to take place via the respective gateways of the two sites. An example of a typical network mask is 255.255.255.0, which defines a network of 255 sites. Suppose a site on this network has an IP address 192.168.0.2, and is trying to send a message to a site with IP address 192.168.0.253. The communication can take place directly, since the logical AND between the IP address and the network mask is the same for both IP addresses. By definition, an ordinary site does not logically contain any other sites, and has the network mask 255.255.255.255. In contrast, a gateway site can logically contain one or more sites and can potentially have any network mask apart from a full sequence of ones. In practice, however, the binary form of most network masks typically consists of a sequence of ones, followed by a sequence of zeros. Based on this principle, a network mask can be represented using an integer between 0 and 32 instead of a full IP address, as defined by RFC 1519 [31]. The integer represents the number of initial bits that need to be identical in order for two IP addresses to be on the same network. For example, if the number is 24 then the first 24 bits of the two IP addresses need to be the same. This means that the 8 remaining bits can differ, resulting in a network of up to 255 IP addresses. As such, the number 24 is an abbreviation of the IP mask 255.255.255.0. If no network mask is specified, then the site is assumed to be an ordinary host, and a mask of 32 is used by default.

The Channel Ambient Runtime could make use of network masks to conserve bandwidth, by only trying to interact with a remote runtime if its network address is within the range of accessible addresses. If the address is not accessible then the corresponding action can be blocked, since according to the network configuration it can never occur. In general there will be little variability in the network mask, which could be initialised when the runtime is launched. Where necessary, the mask could be modified during execution by the runtime administrator.

4.4.4 Using Modules to Structure the Runtime Code

The code of the Channel Ambient Runtime can be structured by defining a separate module for each of the runtime components. The modules of the runtime are summarised in Figure 4.5, together with the dependencies between these modules. The dependencies were generated automatically from the runtime code using the dependency generation tool `ocamlldot` [44], which produces a directed graph that can be visualised using the DOT layout engine [33]. The main modules of the Channel Ambient Runtime are summarised below:

Runtime defines the top-level execution loop for the runtime, together with the initialisation of the remaining runtime components.

Io defines the main input and output functions, together with the networking functions that use socket libraries.

Lexer converts a source file into a stream of tokens, according to the syntax of the Channel Ambient calculus.

Parser converts a stream of tokens into a runtime term, according to the syntax of the Channel Ambient Machine. This module also performs typechecking of processes, to ensure that values can only be sent and received on channels of the correct type.

Interpreter defines the main function for executing a runtime term, based on the reduction rules of the Channel Ambient Machine.

Ambient defines the terms of the runtime together with operations on these terms, including selection, construction and unblocking. For convenience, each runtime term is packaged as a separate ambient, consisting of a term and a corresponding ambient name. This ambient data structure is also used for sending ambients over a network during a migration.

Process defines the processes of the Channel Ambient Runtime, together with operations on these processes such as substitution.

The full set of modules can be viewed by downloading the runtime implementation, available from [52]. Although the modules are defined in using a functional module system, they can be given an object-oriented flavour by defining an abstract type t inside each module, together with the corresponding functions for performing operations on this type. In particular, each module defines a `to_string` function and a `to_html` function for the type t . This is used for debugging purposes to display the full execution state of the runtime in both plain text and html.

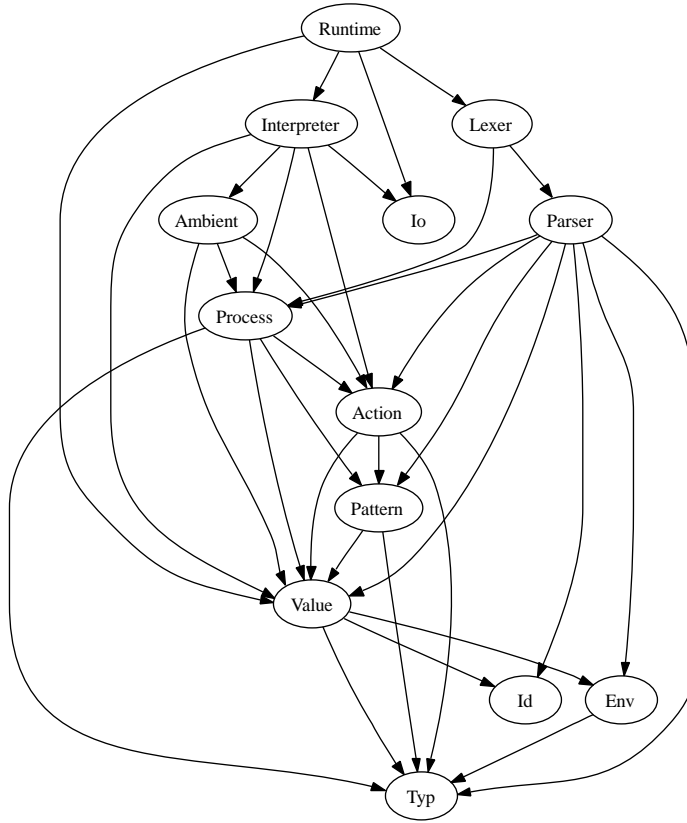


Figure 4.5: Modular Implementation

4.5 Related Work

The Pict project demonstrated how a runtime for the π -calculus could be implemented based on solid theoretical foundations. The Pict runtime [74] is implemented based on an abstract machine for the π -calculus and a detailed compilation of the abstract machine to C code, resulting in the efficient execution of concurrent processes. Each Pict program is executed by a separate instance of the Pict runtime, which is first compiled to C using the Pict compiler and then compiled to machine code on a given architecture using a standard C compiler. In principle, similar techniques could be applied to the Channel Ambient Runtime, in order to compile Channel Ambient programs to C code and then to machine code. Although such compilation would result in the efficient execution of programs on a single machine, it is not suitable for moving programs between machines in a distributed environment. Instead, modern distributed runtimes typically rely on an interpreter to execute platform-independent code. This high-level code can be executed by multiple runtimes on different machine architectures, and can readily be moved between runtimes over a network. Such portability is vital in a language for mobile applications, and is one of the main motivations

for using an interpreter instead of a compiler. As with the Pict runtime, the Channel Ambient Runtime uses a map data structure to increase the efficiency of selecting a given blocked process. The round-robin scheduling algorithm used in the Channel Ambient Runtime is also inspired by scheduling in Pict. Unlike the Pict runtime, the Channel Ambient Runtime is based on an abstract machine that is both sound and complete, giving greater flexibility in the implementation of scheduling, while still preserving the correctness of the runtime. In addition, the Pict runtime executes on a single machine, whereas the Channel Ambient Runtime is distributed across multiple machines. This presents additional challenges for migrating agents between runtimes, re-binding an agent to a new environment after a migration, and providing support for restricted names across multiple runtimes.

The Nomadic Pict runtime [84] 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 Chapter 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. A publish-subscribe name server is used to facilitate the interaction between programs launched separately on different runtimes. In contrast, the Channel Ambient runtime allows applications launched on separate runtimes to interact with each other directly, using public agent names and channels. This allows a given application to be stopped, upgraded, and then restarted independently of other applications, without the need to re-publish values on a name server.

The JoCaml runtime [26] is implemented based on a high-level abstract machine for the Join calculus [28] and a compilation of the calculus to a suitable target language [43]. This research was the foundation for the JoCaml language and runtime system, which is used to execute concurrent, distributed and mobile applications. If Ambient calculi are to be used as the basis for writing such applications, a similar level of research is needed for the implementation of ambients. In [83] a number of comparisons are made between the JoCaml and Nomadic Pict runtimes. Similar comparisons can also be made between the JoCaml runtime and the Channel Ambient Runtime. 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 to deliver messages to migrating agents, to atomically migrate entire subtrees of locations, and to prevent race conditions between migrating locations. These algorithms can be difficult to implement efficiently, especially in wide-area networks. An example of one such algorithm is described in

Chapter 5, which presents an application for tracking the location of agents as they move between sites in a network. 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.

Over the years, there has been substantial theoretical research on the Ambient calculus and its many variants, yet there has been comparatively little research on its implementation. Preliminary research on the implementation of Ambients is presented in [17], which describes a non-distributed implementation of the basic operations of the Ambient calculus. The implementation uses shared-memory concurrent programming technology such as threads, mutexes and conditions, in the form provided by the Java language. The paper describes an implementation of the reduction relation for ambients, such that each process P is executed by a separate concurrent thread. A number of locking mechanisms are used in order to implement synchronisation between ambients. For large numbers of threads this can result in significant overhead, particularly for high-level languages such as Java where context switching between threads is relatively expensive. In contrast, the Channel Ambient Runtime is based on a lower-level abstract machine, which describes a detailed scheduling and execution algorithm for multi-threading. This detailed abstract machine gives more control over the implementation, both for optimisation and for stronger guarantees of program behaviour. More importantly, such control is crucial for being able to stop a thread so that it can be migrated over a network and restarted on a different runtime. Direct control of thread migration is not currently supported in most languages, including Java.

More advanced research on the implementation of ambients is described in [67], which presents an abstract machine for the Safe Ambient calculus (PAN), together with a brief outline of a runtime implementation in Java. Each ambient is implemented as a separate thread, which is executed by a separate instance of the runtime. For large numbers of ambients, this requires many instances of the runtime to execute concurrently, which may result in significant overhead. The ambients are implemented in a flat topology, where each ambient has a link to its parent. In contrast, the Channel Ambient runtime allows nested execution of ambients, where a single runtime is used to execute an entire tree of ambients, and a separate runtime is only needed for ambients that have a network address. When an ambient moves from one location to another all of its sub-ambients move with it. In contrast, physical mobility in PAN only takes place when an ambient is opened. Forwarders are used to maintain a path of logical moves so that the contents of an ambient can be physically moved once the ambient is opened. As a result, the contents of an ambient may go through a long chain of forwarders before reaching its target destination. Since ambients in PAN have a flat topology, each ambient needs to act as a server on the network. A network address is assigned to each ambient, together with an additional identifier to allow

multiple ambients to execute on the same address. In contrast, the Channel Ambient Runtime only assigns an address to those ambients that correspond to addressable sites on the network. Ambients that do not have an address correspond to mobile software agents or modules. In this setting, an agent a on a given site can interact directly with a sibling agent b on the same site using the agent's name. If b is on a remote site then a direct interaction is not possible, and a needs to either move to the remote site and perform a local interaction with b , or send a message to the remote site indirectly, which will be forwarded to b at the site's discretion. Note that PAN also separates the logical and the physical distribution of ambients, where the logical distribution is given by the tree structure of the ambient syntax and the physical distribution is given by the association of a location to each ambient. As a result, each ambient needs to act as a server on the network, since it can potentially be reached by any other ambient in the network. In contrast, the Channel Ambient Runtime assumes that the physical structure is constrained by the logical ambient hierarchy. This makes it possible to directly model the topology of hierarchical networks using the ambient paradigm, resulting in a simplified implementation.

4.6 Conclusion

This chapter 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. A local runtime is first implemented by mapping the Channel Ambient Machine to simple function definitions. The local implementation highlights the close correspondence between the abstract machine and functional program code. A distributed runtime is then implemented by adding a thin layer of multiplexing on top of the TCP/IP protocol, and mapping the Channel Ambient Machine to a combination of function definitions and calls to the TCP/IP socket library. The distributed implementation highlights the close correspondence between the abstract machine and the main features of the TCP/IP protocol. A number of enhancements are then made to the distributed runtime in order to improve the efficiency and security of process execution, while reducing network congestion.

The Pict runtime demonstrated the efficiency of using a list structure and a notion of blocking to implement synchronisation between concurrent processes. The Channel Ambient Runtime uses similar underlying principles, and in future it will be important to quantify the efficiency of the runtime so that it can be compared with related runtimes for mobile agents. The completeness of the Channel Ambient Machine also enables a number of optimisations for the scheduling of processes in the runtime implementation. At present, a limited number of optimisations have been considered, such as using a round-robin scheduling algorithm. In future, more sophisticated scheduling algorithms could be developed, which also take into account the particular properties of

the applications being executed. These optimisations will also depend on the various constraints on the execution of processes. More sophisticated constraints on fairness can be imposed, such as a maximum length of time for which a given process can be scheduled. This could be implemented by limiting the number of successive reduction steps that a given process is allowed to perform before a different process is scheduled. Such optimisations and guarantees are possible because the runtime has precise control over the scheduling of processes.

The Channel Ambient Runtime is an interpreter for a high-level language, in the style of Java or .Net. The use of an interpreter provides a uniform interface so that agents can migrate seamlessly between runtimes on different machine architectures. Migration of agents between runtimes is implemented by sending serialised data structures over a network. These data structures are unserialised on arrival and cast into an agent data structure. The casting will only be successful if the data structure is of the correct type, which provides some level of quality control on the data received from remote runtimes. Instead of transmitting serialised data structures directly, modern runtimes typically transmit intermediate bytecode, which provides a more standardised format that is not dependent on the internal data structures of the runtime. In future, the Channel Ambient Runtime could also make use of a suitable bytecode format, in order to maintain compatibility between different versions of the runtime.

The runtime implementation presented in this chapter could also be applied to other variants of Boxed Ambients, by defining an abstract machine for these variants using the approach described in Chapter 3. Taking this a step further, it should be possible to define a single runtime that takes the semantics of different variants of Boxed Ambients as a parameter. Such a runtime would be useful for experimenting with different programming abstractions for mobile applications, within a single runtime environment.

The implementation techniques described in this chapter have also been used to implement a programming language and runtime system for the stochastic π -calculus with mixed choice, in order to simulate models of biological systems [54]. In future, similar techniques could be used to implement a simulator for a stochastic variant of the Ambient calculus, such as [60].

The Channel Ambient Runtime is implemented in the OCaml programming language. The high-level nature of the abstract machine means that the runtime could also be implemented in a range of other languages, including Haskell, Java or C++. As languages for distributed computing evolve and new languages are proposed, the Channel Ambient Runtime could be re-implemented to take advantage of new developments in language design. For example, the Acute language [70] contains various high-level features designed specifically for distributed programming, including type-safe serialisation and unserialisation of arbitrary values, and stronger guarantees of type safety across an entire distributed system. This would simplify some aspects of the runtime implementation, while providing stronger guarantees in cases where multiple versions of the runtime

co-exist on the same network.

This chapter gives a detailed description of the implementation of the Channel Ambient Runtime, based on the Channel Ambient Machine. One reason for such detail is to highlight the close correspondence between the abstract machine and functional program code. Such correspondence is not a coincidence, but is the end result of a long process of refinement of both the Channel Ambient Machine and its corresponding runtime implementation. In future, it may also be possible to prove a more formal correspondence between abstract machine and program code, perhaps with the help of standard theorem provers such as HOL [39]. This could be used to define a provably correct series of transformations from the high-level calculus right down to the executable runtime code, giving greater guarantees about the security of runtime execution.

A number of improvements could also be made to the distributed aspects of the Channel Ambient Runtime. The runtime uses a daemon to receive data from remote runtimes, which is then forwarded to the interpreter. In future, the architecture of the daemon could be formally specified, in order to prove a more direct correspondence with the Channel Ambient Machine. Finally, the security of the network protocols used by the distributed runtime could also be enhanced, by defining a mapping from communication on private channels to encrypted communication on public channels, using similar techniques to those described in [1]. In particular, it should be possible to implement runtime communication on top of a Secure Socket Layer (SSL) protocol. This would help prevent eavesdropping on communication between runtimes, in order to protect the scope of private names.

Chapter 5

The Channel Ambient Language

5.1 Introduction

The Channel Ambient calculus was presented in Chapter 2 as a high-level formalism for specifying mobile applications. The Channel Ambient Machine was then presented in Chapter 3 as a formal specification of a runtime for executing calculus processes, and a corresponding runtime implementation was presented in Chapter 4. This chapter presents a programming language for mobile applications, known as the Channel Ambient Language (CAL). The language illustrates how the high-level constructs of the Channel Ambient calculus can be used as programming abstractions for mobile applications. In many respects, the Channel Ambient Language is a product of the previous three chapters, incorporating the main features of the calculus, the abstract machine and the runtime. The Channel Ambient Language and runtime system are available at [52]. 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 chapter is structured as follows:

Section 5.2 describes how programs in the Channel Ambient Language can be executed by the Channel Ambient Runtime. A number of debugging mechanisms for visualising the execution state of the runtime are also presented, together with a brief description of the networks in which programs are assumed to execute.

Section 5.3 describes the syntax of programs in the Channel Ambient Language, together with the corresponding rules for program execution. 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.

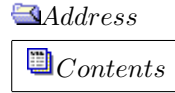
Section 5.4 uses the Channel Ambient Language to program an example mobile application, in which a mobile agent monitors resources on a remote server. Detailed execution traces for the application are also presented.

Section 5.5 uses the Channel Ambient Language to program an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network.

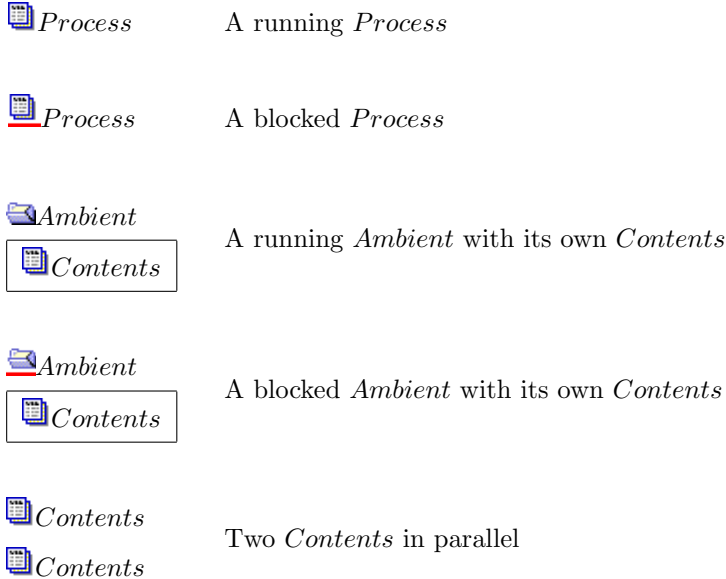
5.2 Runtime Execution

5.2.1 Execution State

The Channel Ambient Language uses the notion of an *ambient* to program the main components of a mobile application, including hardware sites, mobile agents and modules. The runtime for the language is an extended version of the Channel Ambient Runtime presented in Chapter 4. In general, each runtime executes a single site with a given network address. The state of a runtime with a given *Address* and *Contents* is represented as:



The *Contents* of the runtime can either be empty or it can be one of the following:



In addition, the following notation is used to represent an ambient that is either running or blocked:



When the runtime executes a given program, its internal state is modified with each execution step. This transformation of state is described using rules of the form:

$$\boxed{\text{Contents}}$$

$$\longrightarrow \boxed{\text{Contents}'}$$

Note that if Contents_1 can evolve to $\text{Contents}'_1$ then this transformation can also take place inside Ambient_1 :

$$\boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}_1} \\ \text{Contents}_2 \end{array}} \longrightarrow \boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}'_1} \\ \text{Contents}_2 \end{array}}$$

If an Ambient_1 does not contain any unblocked actions or ambients then Ambient_1 is blocked:

$$\boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}_1} \\ \text{Contents}_2 \end{array}} \longrightarrow \boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}_1} \\ \text{Contents}_2 \end{array}}$$

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.

A number of formatting rules for the runtime are also defined, of the form:

$$\boxed{\text{Contents}} = \boxed{\text{Contents}'}$$

These rules describe how the internal state of the runtime is expanded so that an execution step can take place. As with the execution rules, if Contents_1 can be expanded to $\text{Contents}'_1$ then this transformation can also take place inside Ambient_1 :

$$\boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}_1} \\ \text{Contents}_2 \end{array}} = \boxed{\begin{array}{c} \text{Ambient}_1 \\ \boxed{\text{Contents}'_1} \\ \text{Contents}_2 \end{array}}$$

Finally, the contents of an ambient can be unblocked during execution. This is represented by an unblocking function $\lfloor \text{Contents} \rfloor$, which unblocks any top-level blocked actions inside Contents :

$$\boxed{\lfloor \text{Contents} \rfloor}$$

As explained in Chapter 3, unblocking allows an ambient to re-bind to its environment after a migration, by giving any blocked actions in the ambient the chance to interact with their new location. Note that only top-level actions need to be unblocked, since actions inside nested ambients cannot interact directly with the new environment.

5.2.2 Networking

Applications written in the Channel Ambient Language are assumed to execute on networks that support the widely-used TCP/IP version 4 protocol. Based on the properties of TCP/IP networks, a distinction is made between two types of ambients: *sites* and *agents*. Sites represent physical machines that are assumed have a fixed network address, while agents represent software programs that can move in and out of sites and other agents. A site name is an *Address* of the form *IP:N*, which represents the IP address and port number of a machine on the network, while an agent name is a simple string. In a hierarchical network topology, sites can be logically contained inside other sites to form Local Area Networks, as described in Chapter 4.

5.2.3 Debugging

The Channel Ambient Runtime can be used to execute a file *source.ca* by typing the following command in a console:

```
cam.exe source.ca
```

Alternatively, a graphical file navigation tool can be configured so that *cam.exe* is used by default to open all *.ca* files. Each source file is executed in a separate directory containing the following files:


source.ca source file to be executed by the runtime


state.html current state of the runtime

start.html initial state of the runtime



log log of outputs made by the runtime


_thread.gif  icon to display the state of a parallel thread in the runtime

_open.gif  icon to display the state of an ambient in the runtime

_closed.gif  icon to hide the state of an ambient in the runtime

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 , and each currently executing ambient is enclosed in a box with an *open* folder icon  next to the ambient's name. 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. The state of the ambient can be revealed again by clicking on this icon.

Any statements that are printed on the console are also recorded in a *log* file.

5.3 Language Definition

5.3.1 Programs

The syntax of *Programs* is summarised below, where optional elements are enclosed in braces as *{Optional}*. The full syntax of the language is presented in Appendix B:

$Program ::=$	$Name \{Address\} [Process]$	Local Site
	$Address \{Address\} [Process]$	Network Site
	$Address [Address \{Address\} [Process]]$	Nested Site

Local Site $Name_1 \{Address_1\} [Process_1]$

Executes $Process_1$ inside a site with name $Name_1$ and local address $Address_1$. The local address allows the site to logically contain a Local Area Network of child sites, which interact with their parent using $Address_1$. If no $Address_1$ is specified then the site cannot logically contain any child sites.

Network Site $Address_2 \{Address_1\} [Process_1]$

Executes $Process_1$ inside a site with local address $Address_1$ and global address $Address_2$. The global address allows the site to interact with peer sites on the global network using $Address_2$.

Nested Site $Address_3 [Address_2 \{Address_1\} [Process_1]]$

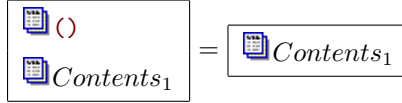
Executes $Process_1$ inside a site with local address $Address_1$, global address $Address_2$, and parent address $Address_3$. The parent address allows the site to interact with its parent anonymously, using $Address_3$ by default.

5.3.2 Processes

$Process ::= ()$	Null
$ Process \mid Process$	Parallel Composition
$ \text{new } Name : Type \ Process$	Restriction
$ Agent[Process]$	Agent
$ Action\{ ; \ Process\}$	Action
$!Action\{ ; \ Process\}$	Replicated Action
$ \text{if } Value\{ \rightarrow Value\} \ \text{then } Process \ \{\text{else } Process\}$	Pattern Matching
$ \text{type } Name = Type \ Process$	Type Definition
$ \text{let } Pattern = Value \ \text{in } Process$	Value Definition
$ \text{let } Macro(\{Pattern\}) = Process \ \text{in } Process$	Macro Definition
$ Macro<\{Value\}>$	Macro Instantiation
$ (Process)$	Nested Process

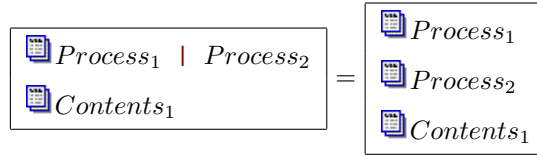
Null $()$

Represents the end of a process:



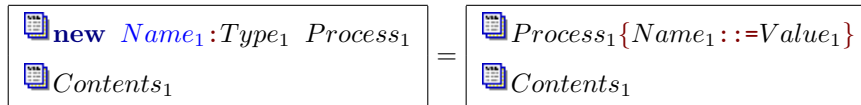
Parallel Composition $Process_1 \mid Process_2$

Creates two processes $Process_1$ and $Process_2$, which execute in parallel:



Restriction $\text{new } Name_1 : Type_1 \ Process_1$

Creates a globally unique value $Value_1$ of type $Type_1$. The value is assigned to $Name_1$ in $Process_1$, written $Process_1\{Name_1 ::= Value_1\}$, which then executes:



Agent $Agent_1[Process_1]$

Creates an agent with name $Agent_1$ that executes $Process_1$:

$$\boxed{\begin{array}{l} \text{Agent}_1[\text{Process}_1] \\ \text{Contents}_1 \end{array}} = \boxed{\begin{array}{l} \text{Agent}_1 \\ \boxed{\begin{array}{l} \text{Process}_1 \\ \text{Contents}_1 \end{array}} \end{array}}$$

Action $\text{Action}_1\{; \text{Process}_1\}$

Tries to perform Action_1 and then execute Process_1 . If no Process_1 is specified then the null process $()$ is used by default. The behaviour of the various actions is described in Subsection 5.3.3.

Replicated Action $!\text{Action}_1\{; \text{Process}_1\}$

Tries to perform Action_1 and then execute Process_1 in parallel with the original replicated action. This has the effect of repeatedly executing Action_1 followed by Process_1 :

$$\boxed{\begin{array}{l} !\text{Action}_1; \text{Process}_1 \\ \text{Contents}_1 \end{array}} = \boxed{\begin{array}{l} \text{Action}_1; (\text{Process}_1 \mid !\text{Action}_1; \text{Process}_1) \\ \text{Contents}_1 \end{array}}$$

If no Process_1 is specified then the null process $()$ is used by default. For convenience, the replicated action can also be represented in its original unexpanded form $!\text{Action}_1; \text{Process}_1$ without ambiguity.

Pattern Matching $\text{if } \text{Value}_1 \{-> \text{Value}_2\} \text{ then } \text{Process}_1 \{ \text{else } \text{Process}_2 \}$

Tries to match Value_1 with Value_2 and then execute Process_1 . If there is a match then Process_1 executes, where each value in Value_1 is assigned to a corresponding variable in Value_2 , written $\text{Process}_1\{\text{Value}_1 \rightarrow \text{Value}_2\}$:

$$\boxed{\begin{array}{l} \text{if } \text{Value}_1 \rightarrow \text{Value}_2 \text{ then } \text{Process}_1 \\ \text{else } \text{Process}_2 \\ \text{Contents}_1 \end{array}} = \boxed{\begin{array}{l} \text{Process}_1\{\text{Value}_1 \rightarrow \text{Value}_2\} \\ \text{Contents}_1 \end{array}}$$

This requires Value_1 and Value_2 to have compatible patterns. If there is no match then Process_2 executes:

$$\boxed{\begin{array}{l} \text{if } \text{Value}_1 \rightarrow \text{Value}_2 \text{ then } \text{Process}_1 \\ \text{else } \text{Process}_2 \\ \text{Contents}_1 \end{array}} = \boxed{\begin{array}{l} \text{Process}_2 \\ \text{Contents}_1 \end{array}}$$

If no Process_2 is specified then the null process $()$ is used by default. If no Value_2 is specified then the value **true** is used by default.

Type Definition $\text{type } \text{Name}_1 = \text{Type}_1 \text{ Process}_1$

Is short for Process_1 , in which Name_1 is substituted with Type_1 .

Value Definition $\text{let } Pattern_1 = Value_1 \text{ in } Process_1$

Is short for $Process_1$, in which $Pattern_1$ is substituted with $Value_1$.

Macro Definition $\text{let } Macro_1(Pattern_1) = Process_1 \text{ in } Process_2$

Is short for $Process_2$, in which $Macro_1\langle Value_1 \rangle$ is substituted with $Process_1$. For each macro substitution, the parameter $Pattern_1$ is instantiated with $Value_1$ in $Process_1$.

Note that *Type*, *Value* and *Macro* substitutions are performed during parsing, before any program code is executed.

5.3.3 Actions

$Action ::= Ambient.Channel\langle\{Value\}\rangle$	Sibling Output
$Ambient/Channel\langle\{Value\}\rangle$	Child Output
$Channel^{\sim}(\{Pattern\})$	External Input
$Channel^{\sim}\langle\{Value\}\rangle$	Parent Output
$Channel\langle\{Value\}\rangle$	Local Output
$Channel(\{Pattern\})$	Internal Input
$\text{in } Ambient.Channel$	Enter
$-\text{in } Channel$	Accept
$\text{out } Channel$	Leave
$-\text{out } Channel$	Release

By definition, an *Ambient* can be either a mobile *Agent* or a remote *Site*. Based on the properties of TCP/IP networks outlined in Chapter 2, only a subset of actions are allowed inside a given site:

$Action_{Site} ::= Site.Channel\langle\{Value\}\rangle$	Site Output
$Ambient/Channel\langle\{Value\}\rangle$	Child Output
$Channel^{\sim}(\{Pattern\})$	External Input
$System^{\sim}\langle\{Value\}\rangle$	System Output
$Channel\langle\{Value\}\rangle$	Local Output
$Channel(\{Pattern\})$	Internal Input
$-\text{in } Channel$	Accept
$-\text{out } Channel$	Release

Sites are constrained so that they 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, sites are constrained so that they can only contain a parent output on a limited set of *System* channels. This simplifies the synchronisation between nested

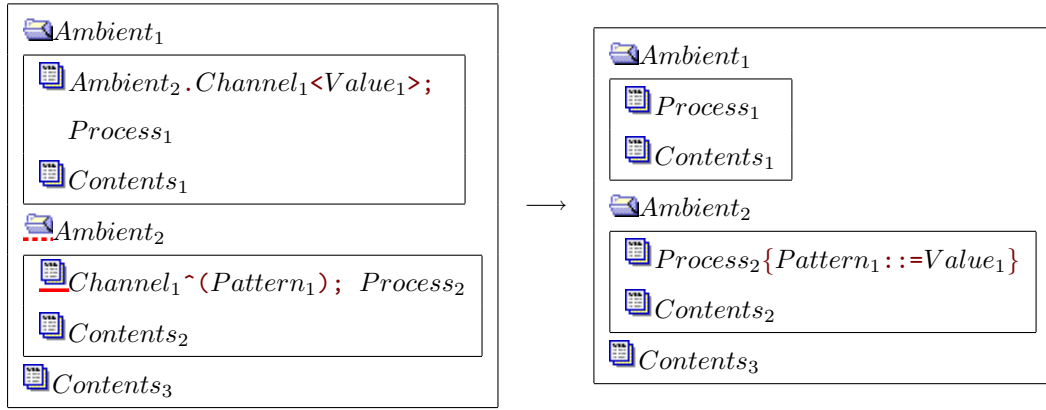
sites in a network. Finally, sites are constrained so that they cannot contain an enter or a leave. This reflects the assumption that sites have a fixed network address.

Sibling Output $Ambient_2.Channel_1<Value_1>; Process_1$

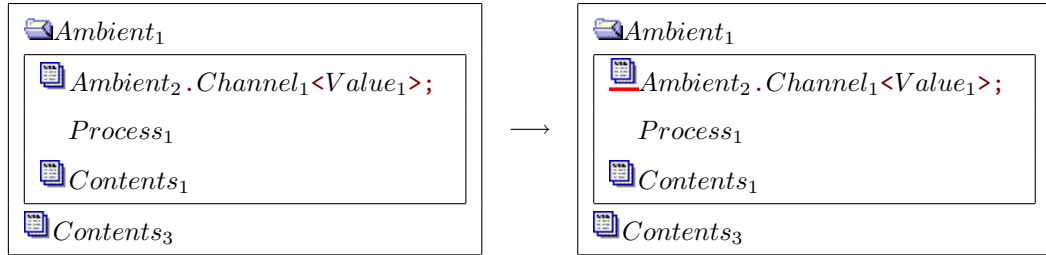
Tries to send $Value_1$ on $Channel_1$ to a sibling $Ambient_2$, and then execute $Process_1$. If this process executes inside $Ambient_1$, and there is a sibling $Ambient_2$ with a blocked External Input process

$$Channel_1^{\wedge}(Pattern_1); Process_2$$

then $Value_1$ is sent along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Ambient_1$ executes in parallel with $Ambient_2$, $Process_1$ executes inside $Ambient_1$ and $Process_2$ executes inside $Ambient_2$. If $Ambient_2$ is a remote *Site* then $Value_1$ is sent over the network using TCP/IP:



If there is no sibling $Ambient_2$ with a corresponding blocked External Input process, and $Ambient_2$ is a mobile *Agent*, then the Sibling Output process is blocked:



If $Ambient_2$ is a remote *Site* then the Sibling Output process remains active and is re-attempted after a specified delay. The delay is increased exponentially after each attempt, in order to avoid causing a denial of service attack.

Child Output $Ambient_1/Channel_1\langle Value_1 \rangle$

Tries to send $Value_1$ on $Channel_1$ to child $Ambient_1$. This is short for a corresponding sibling output process inside a private $Ambient_2$:

$$Ambient_2[Ambient_1.Channel_1\langle Value_1 \rangle]$$

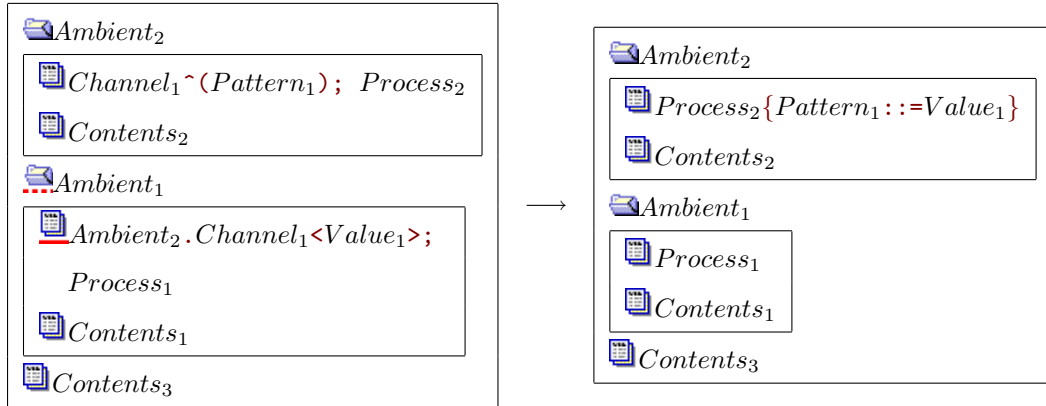
Once the output has been performed the resulting empty $Ambient_2$ can be garbage-collected immediately.

External Input $Channel_1 \sim (Pattern_1); Process_2$

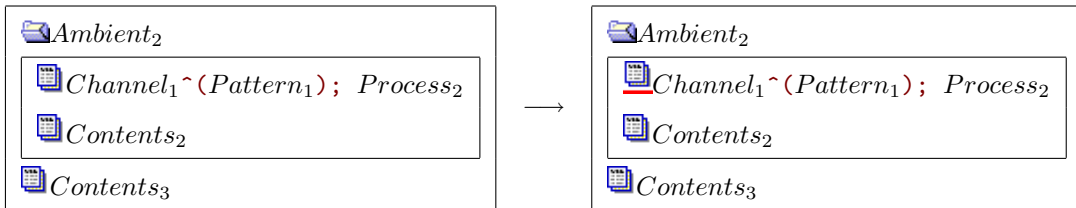
Tries to receive a value on $Channel_1$ from a sibling ambient, and then assign it to $Pattern_1$ in $Process_2$. If this process executes inside $Ambient_2$, and there is a sibling $Ambient_1$ with a blocked Sibling Output process

$$Ambient_2.Channel_1\langle Value_1 \rangle; Process_1$$

then $Value_1$ is received along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Ambient_2$ executes in parallel with $Ambient_1$, $Process_2$ executes inside $Ambient_2$ and $Process_1$ executes inside $Ambient_1$:



If there is no sibling $Ambient_1$ with a corresponding blocked Sibling Output process then the External Input process is blocked:



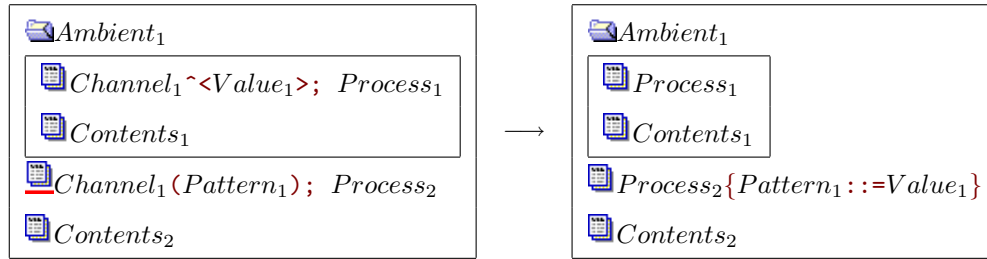
Note that external inputs inside a site will block systematically, since by definition corresponding sibling outputs to a site are never blocked.

Parent Output $Channel_1 \hat{<Value_1>; Process_1}$

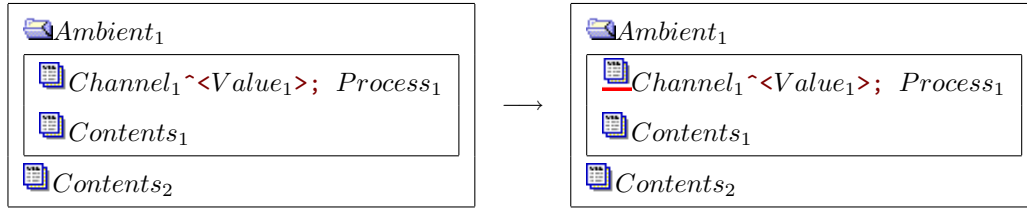
Tries to send $Value_1$ on $Channel_1$ to the parent ambient, and then execute $Process_1$. If this process executes inside $Ambient_1$, and in parallel there is a blocked Internal Input process

$$Channel_1(Pattern_1); Process_2$$

then $Value_1$ is sent along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Process_1$ executes inside $Ambient_1$, and $Process_2$ executes in parallel with $Ambient_1$:



If there is no parallel blocked Internal Input process then the Parent Output process is blocked:



In order to simplify the interaction between nested sites in a network, parent outputs inside a site are only allowed on a limited set of *System* channels. These parent outputs correspond to system calls to the parent site. By definition, a given runtime executes a single site, with a link to its parent site. The top-level process inside the site is specified by the user when the runtime is launched, 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. 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.

Conceptually, the parent site is assumed to contain a number of replicated input processes on a collection of private *System* channels, each of which provides a given service:

print(s:string) Prints the string s on the console of the runtime.

`println(s:string)` Prints the string `s` followed by a newline character on the console of the runtime.

`delay(duration:int,ack:<void>)` Sends a void message to the runtime on the `ack` channel after a delay of `duration` seconds.

`run(program:string,arguments:string)` Runs the given executable `program` with the given string `arguments`. The program must be installed on the system in order to be executed.

`read(file:string,result:<string>)` Reads the given `file` and sends its string contents to the runtime on the `result` channel.

`write(file:string,contents:string)` Writes the string `contents` to the given `file`. If no such file exists, then a new file is created.

`route(s:site,x:<'a>,v:'a)` Sends the value `v` on channel `x` to site `s`. This allows a site to communicate asynchronously with a sibling site via its parent, as an alternative to using a direct sibling output.

`parent(x:<'a>,v:'a)` Sends the value `v` on channel `x` inside the parent site. This allows a site to communicate asynchronously with its parent site.

Local Output $Channel_1 \langle Value_1 \rangle$

Tries to send $Value_1$ on $Channel_1$ locally. This is short for a corresponding parent output process inside a private $Ambient_1$.

$$Ambient_1 [Channel_1 \hat{\ } \langle Value_1 \rangle]$$

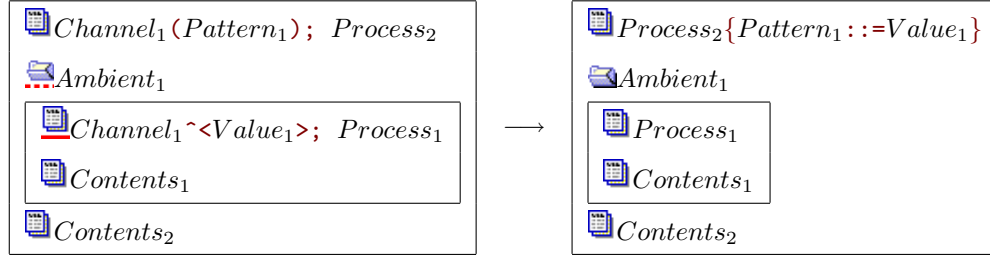
Once the output has been performed the resulting empty $Ambient_1$ can be garbage-collected immediately.

Internal Input $Channel_1 (Pattern_1); Process_2$

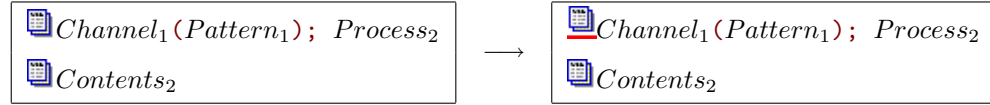
Tries receive a value on $Channel_1$ from a child ambient, and then assign it to $Pattern_1$ in $Process_2$. If there is a child $Ambient_1$ with a blocked Parent Output process

$$Channel_1 \hat{\ } \langle Value_1 \rangle; Process_1$$

then $Value_1$ is received along $Channel_1$ and assigned to $Pattern_1$ in $Process_2$. Afterwards, $Process_2$ executes in parallel with $Ambient_1$ and $Process_1$ executes inside $Ambient_1$:



If there is no child $Ambient_1$ with a corresponding blocked Parent Output process then the Internal Input process is blocked:

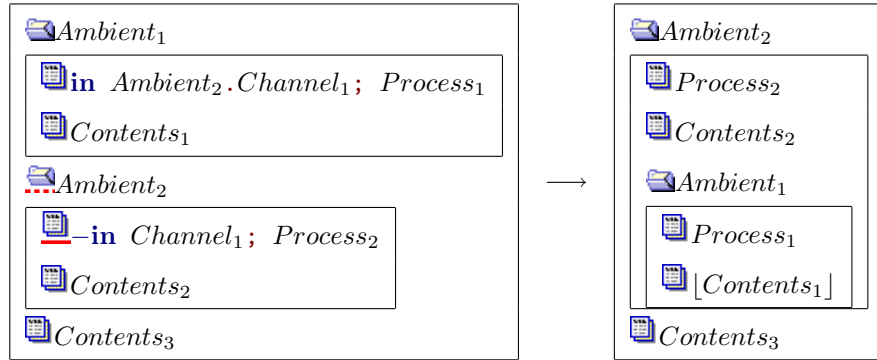


Enter **in** $Ambient_2.Channel_1; Process_1$

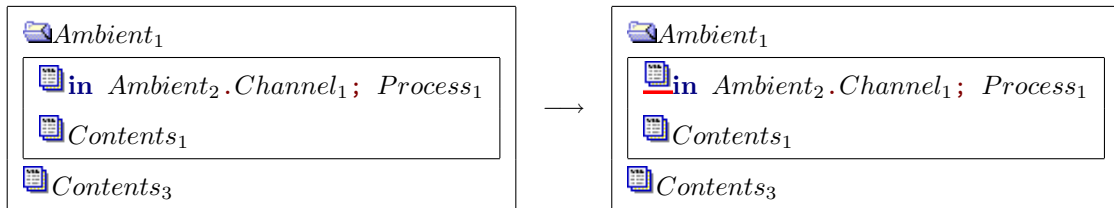
Tries to move the enclosing ambient on $Channel_1$ inside a sibling $Ambient_2$, and then execute $Process_1$. If this process executes inside $Ambient_1$, and there is a sibling $Ambient_2$ with a blocked Accept process

in $Channel_1; Process_2$

then $Ambient_1$ enters $Ambient_2$ on $Channel_1$. Afterwards, $Ambient_1$ executes inside $Ambient_2$, $Process_1$ executes inside $Ambient_1$ with $Contents_1$ unblocked, and $Process_2$ executes inside $Ambient_2$. If $Ambient_2$ is a remote *Site* then $Ambient_1$ is sent over the network using TCP/IP:



If there is no sibling $Ambient_2$ with a corresponding blocked Accept process, and $Ambient_2$ is a mobile *Agent*, then the Enter process is blocked:



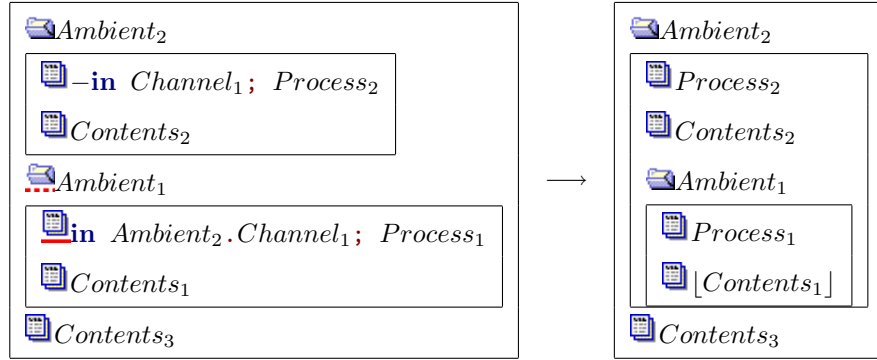
If $Ambient_2$ is a remote *Site* then the Enter process remains active and is re-attempted after a specified delay. The delay is increased exponentially after each attempt, in order to avoid causing a denial of service attack.

Accept $\text{--in } Channel_1; Process_2$

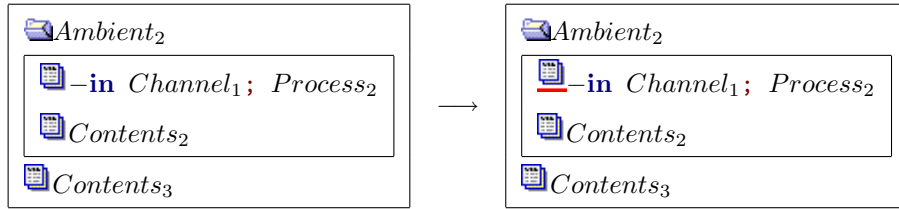
Tries to allow a sibling ambient to enter on $Channel_1$, and then execute $Process_1$. If this process executes inside $Ambient_2$, and there is a sibling $Ambient_1$ with a blocked Enter process

$\text{in } Ambient_2.Channel_1; Process_1$

then $Ambient_1$ enters $Ambient_2$ on $Channel_1$. Afterwards, $Ambient_1$ executes inside $Ambient_2$, $Process_2$ executes inside $Ambient_2$ and $Process_1$ executes inside $Ambient_1$, with $Contents_1$ unblocked:



If there is no sibling $Ambient_1$ with a corresponding blocked Enter process then the Accept process is blocked:



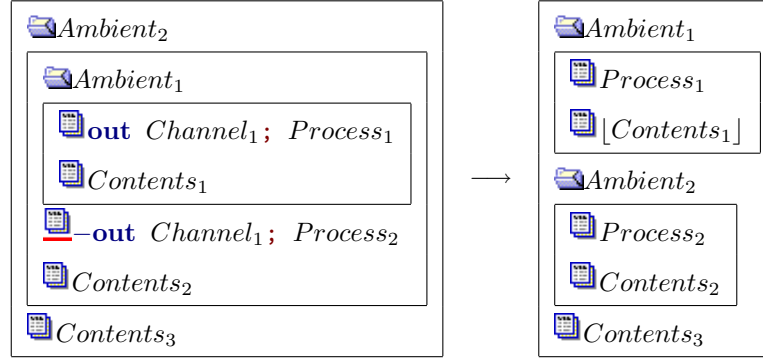
Note that accepts inside a site will block systematically, since by definition corresponding enters to a site are never blocked.

Leave $\text{out } Channel_1; Process_1$

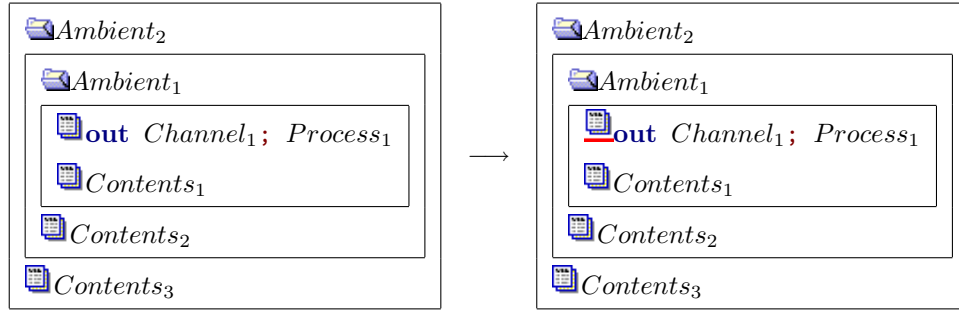
Tries to leave the parent ambient on $Channel_1$ and then execute $Process_1$. If this process executes inside $Ambient_1$ and there is a parent $Ambient_2$ with a blocked Release process

$\text{--out } Channel_1; Process_2$

then $Ambient_1$ can leave its parent $Ambient_2$. Afterwards, $Ambient_1$ executes in parallel with $Ambient_2$, $Process_1$ executes inside $Ambient_1$ with $Contents_1$ unblocked, and $Process_2$ executes inside $Ambient_2$:



If there is no parent ambient with a corresponding blocked Release process then the leave process is blocked:

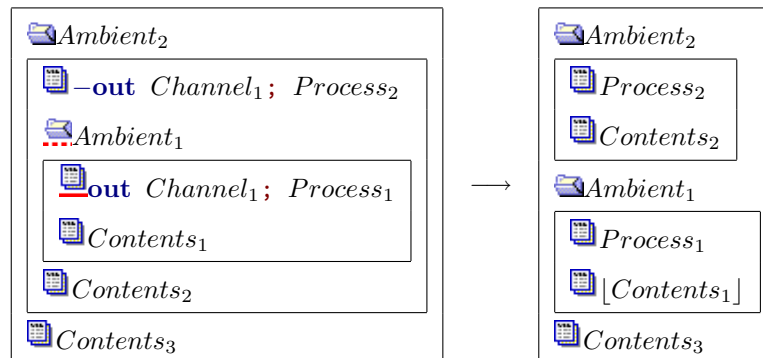


Release $\text{--out Channel}_1; \text{Process}_2$

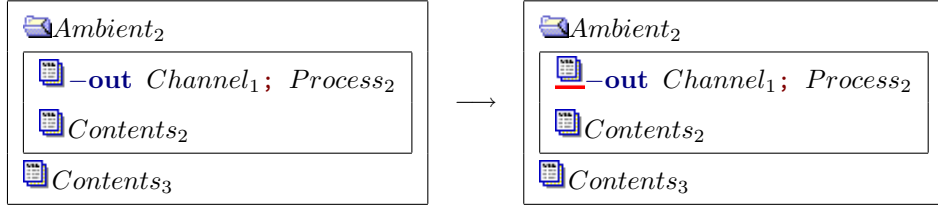
Tries to release a child ambient on Channel_1 and then execute Process_2 . If this process executes inside Ambient_2 and there is a child Ambient_1 with a blocked Leave process

$\text{out Channel}_1; \text{Process}_1$

then Ambient_2 can release Ambient_1 . Afterwards, Ambient_2 executes in parallel with Ambient_1 , Process_2 executes inside Ambient_2 and Process_1 executes inside Ambient_1 with Contents_1 unblocked:



If there is no child Ambient_1 with a corresponding blocked Leave process then the Release process is blocked:



5.3.4 Patterns

$Pattern ::=$	$-$	Wildcard Pattern
	$()$	Void Pattern
	$Name\{ :Type\}$	Name Pattern
	$Pattern, Pattern$	Tuple Pattern
	$(Pattern)$	Nested Pattern

Substitution A *Pattern* can be substituted with a *Value* inside a given *Process*, written

$$Process_1\{Pattern_1 := Value_1\}$$

The definition of substitution is based on the definition outlined in Chapter 2, given that names are bound by Restriction, Input and Macro Definitions. Each unbound name from $Pattern_1$ is substituted with the corresponding value from $Value_1$ inside $Process_1$.

5.3.5 Types

$Type ::=$	void	Void Type
	string	String Type
	int	Integer Type
	float	Float Type
	char	Character Type
	bool	Boolean Type
	site	Site Type
	agent	Agent Type
	migrate	Migration Type
	$\langle \{Type\} \rangle$	Channel Type
	$Name$	Type Variable
	$'Name$	Polymorphic Type
	sub $Type$	Sub Type
	$Type, Type$	Tuple Type
	$Upper$	Type Constant
	$Upper\ Type$	Type Constructor
	$Type \mid Type$	Data Type
	$Type\ list$	List Type
	rec $Name.Type$	Recursive Type
	$(Type)$	Nested Type

Typechecking Before the runtime executes a given program, it checks to see whether the program is well-typed. The type system for The Channel Ambient Language is based on the type system for the Channel Ambient calculus described in Chapter 2. A corresponding type-checking algorithm has been implemented based on the typing rules for the calculus.

5.3.6 Values

<i>Value</i> ::=	()	Void Value
	<i>String</i>	String Value
	<i>Integer</i>	Integer Value
	<i>Float</i>	Float Value
	<i>Character</i>	Character Value
	true	Boolean True
	false	Boolean False
	<i>Address</i>	Site Value
	<i>Name</i> : agent	Agent Value
	<i>Name</i> :<{ <i>Type</i> }>	Channel Value
	<i>Value</i> , <i>Value</i>	Tuple Value
	<i>Upper</i>	Data Constant
	<i>Upper Value</i>	Data Constructor
	[]	Empty List
	<i>Value</i> :: <i>Value</i>	Value List
	<i>Name</i>	Variable
	<i>Value</i> + <i>Value</i>	Addition
	<i>Value</i> - <i>Value</i>	Subtraction
	<i>Value</i> * <i>Value</i>	Multiplication
	<i>Value</i> / <i>Value</i>	Division
	<i>Value</i> = <i>Value</i>	Equal
	<i>Value</i> <> <i>Value</i>	Different
	<i>Value</i> < <i>Value</i>	Less Than
	<i>Value</i> > <i>Value</i>	Greater Than
	<i>Value</i> <= <i>Value</i>	Less Than or Equal
	<i>Value</i> >= <i>Value</i>	Greater Than or Equal
	- <i>Value</i>	Negation
	show <i>Value</i>	String Representation
	(<i>Value</i>)	Nested Value

Basic Values A value can be a *String*, *Integer*, *Float*, or *Character* value, the void value **()**, boolean **true** or boolean **false**. In addition, a *Value* can be a Channel, Agent, or Site. A channel value consists of a *Name* followed by a colon and the *Type* of values that the channel is capable of sending or receiving, enclosed in angle brackets. An agent value consists of a *Name* followed by a colon and the keyword **agent**. A site value consists of an IP *Address* followed by an *Integer*

port number, enclosed in parentheses.

Compound Values A value can also be a tuple of values, where parentheses are used to define nested tuples. In addition, a value can be a data constant, which is simply an *Upper* case name, or a data constructor, which is an *Upper* case name followed by a value. A value can also be a list, which can either be empty `[]` or of the form $Value_1 :: Value_2$, where $Value_1$ is the first element of the list and $Value_2$ is the remainder of the list. Note that all values in a list must be of the same type.

Expressions A value can also be an expression, which is eagerly evaluated by the runtime. The most basic expression is a variable *Name*, which can be substituted with a *Value* during program execution. An expression can also consist of a prefix operator followed by a *Value* argument or an infix operator between two *Value* arguments.

The prefix operator `show` $Value_1$ converts $Value_1$ to a string value. By definition, every value has a string representation, although for certain values this may simply be the empty string. The prefix operator `-` $Value$ is defined in Table 5.1.

Infix operators take two arguments of any type, provided both types are the same. The comparison operators (`=`, `<`, `>`, `>=`, `<=`) return a result of boolean type. They rely on an ordering to compare both arguments. The arithmetic operators (`+`, `-`, `*`, `/`) return a result of the same type as their arguments. Table 5.1 describes the behaviour of the operators for each corresponding type of arguments. The `_` symbol means that the behaviour of the operator is unspecified, although the result will always be of the correct type.

Type	+	-	*	/	- (prefix)	= ,<>,>,>=,<=
String	Concatenate	-	-	-	-	Lexicographic Order
Integer	Add	Subtract	Multiply	Divide	Minus	Integer Order
Float	Add	Subtract	Multiply	Divide	Minus	Float Order
Character	-	-	-	-	-	ASCII Code Order
Boolean	Or	-	And	-	Not	Lexicographic Order
List	Append	-	-	-	-	Order of Elements
Data	-	-	-	-	-	Lexicographic Order
Other	-	-	-	-	-	-

Table 5.1: Operator Definitions

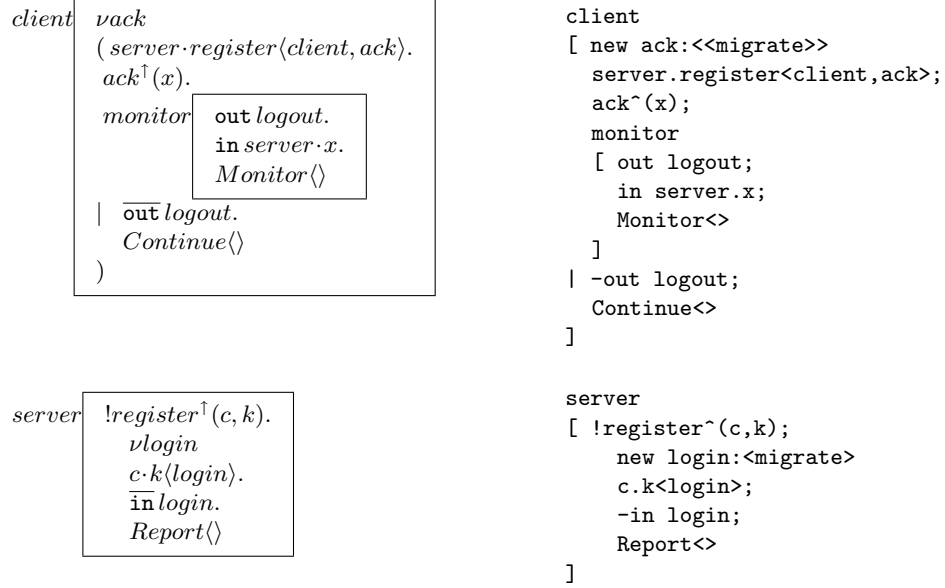


Figure 5.1: Specification and Implementation of the Resource Monitoring Application

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

Figure 5.2: Program code for the Resource Monitoring Application

5.4 Resource Monitoring Application

5.4.1 Program Code

The Channel Ambient language can be used to program a *client* site, which sends a mobile agent over a network to monitor resources on a remote *server* site. The specification of this application

is presented in Figure 5.1, together with an outline of the corresponding implementation.

The server runs on port 3001 of IP address 192.168.0.2 and 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 *Report* process, which forwards data about the resource on the server to the agent. The code for the server is given in the right column of Figure 5.2.

The client runs on port 3000 of IP address 192.168.0.3 and 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 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 *Monitor* process, which monitors data about the resource on the server. In parallel, the client tries to release an agent on the *logout* channel and then execute the *Continue* process. The code for the client is given in the left column of Figure 5.2.

In this example the *Report*, *Monitor* and *Continue* processes are defined as simple outputs. In general they can be defined as arbitrarily complex processes.

5.4.2 Initial State

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 *ack₁* is an abbreviation for a private channel name. All occurrences of the name *ack* are substituted with the *ack₁* inside the client:

192.168.0.3:3000

```

192.168.0.2:3001.register<192.168.0.3:3000,ack1>;
  ack1^(x);
  monitor
  [ out logout;
    in 192.168.0.2:3001.x;
    print<"monitor">
  ]
-out logout;
print<"continue">

```

192.168.0.2:3001

```

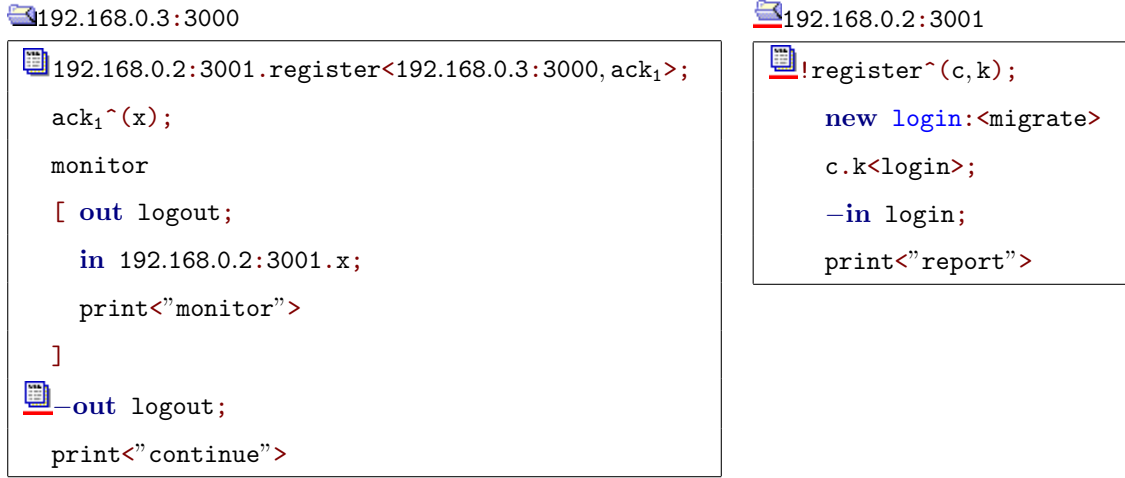
!register^(c,k);
  new login:<migrate>
  c.k<login>;
-in login;
print<"report">

```

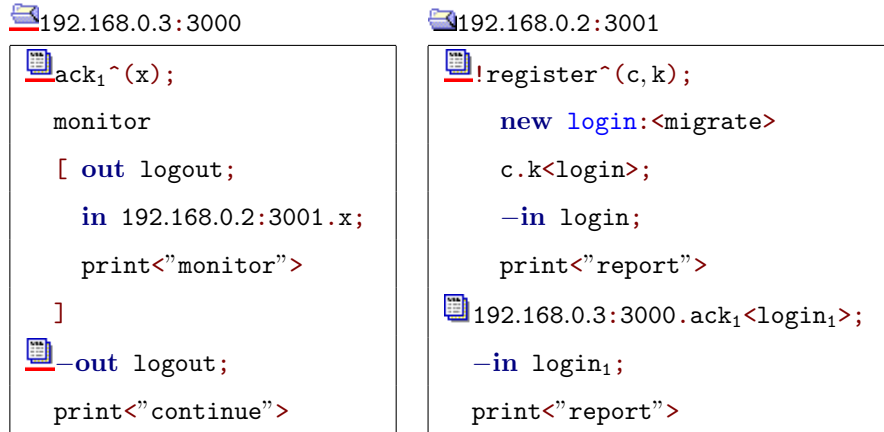
The remainder of this section describes how the internal state of the client and server runtimes can evolve during program execution. These states correspond to HTML output that is automatically generated by the runtimes after each execution step.

5.4.3 Execution

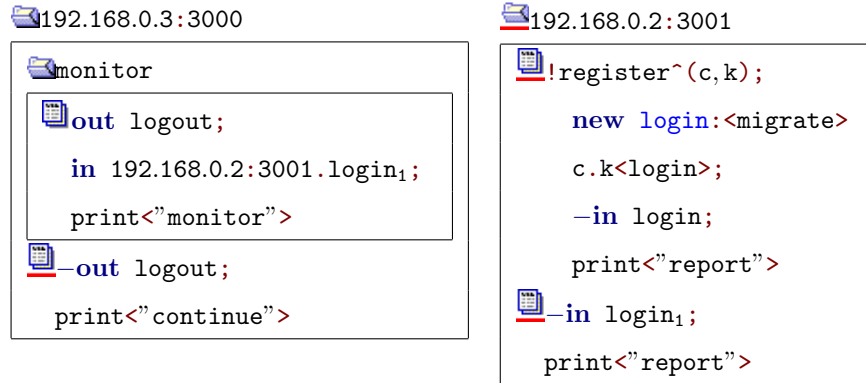
The server blocks a replicated external input on the *register* channel, waiting to receive a value on this channel. The client then blocks a release on the *logout* channel, waiting to release an agent on this channel.



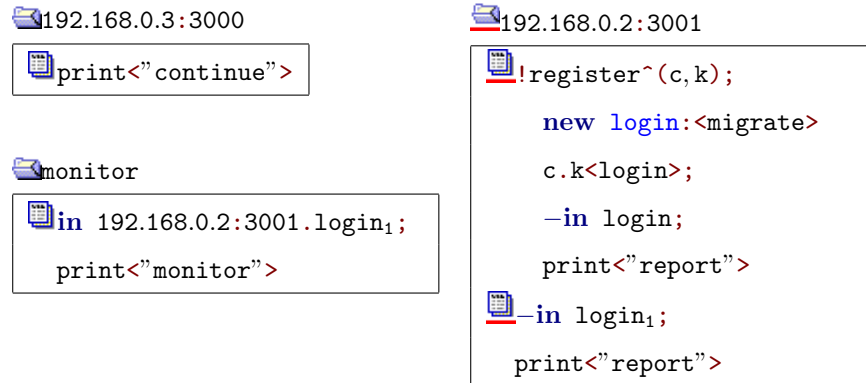
The client sends its name and an acknowledgement channel to the server on the register channel. The client then blocks an external input on the acknowledgement channel, waiting to receive a value on this channel.



After creating a globally unique *login₁* channel, the server sends this channel to the client on the acknowledgement channel. The server then blocks an accept on the login channel, waiting to accept an agent on this channel.

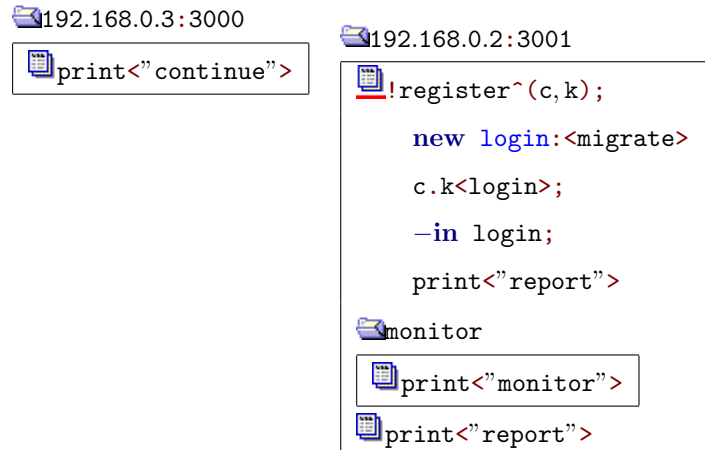


After the client creates a *monitor* agent, the monitor leaves the client on the logout channel, and the client continues executing.



5.4.4 Final State

The monitor agent enters the server on the login channel and then 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 [52], along with debugging information about the initial and final states of the runtime.

$Server(s_i) \triangleq$ $!register(client, ack).$ $\nu tracker \nu send \nu move \nu deliver \nu lock$ $(tracker \boxed{Tracker\langle client, send, move, deliver, lock \rangle}$ $ client/ack\langle tracker, send, move, deliver, lock \rangle$ $ tracker/lock\langle s_i \rangle)$	<pre>let Server(home:site) = !register(client:agent,ack:<details>); 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>)</pre>
$Tracker(client, send, move, deliver, lock) \triangleq$ $(!send^\uparrow(x, m).lock^\uparrow(s).$ $forward^\uparrow\langle s, client, deliver, (s, x, m) \rangle$ $!move^\uparrow(s').s'^\uparrow().lock^\uparrow(s)forward^\uparrow\langle s, client, lock, s' \rangle)$	<pre>let Tracker(client:agent, move:<site>, deliver:<site,<'a>,'a>, lock:<site>) = (!send^(x:<'a>,m:'a); lock^(s:site); forward^<s,client,deliver,(s,x,m)> !move^(s1:site); s1^<>; lock^(s:site); forward^<s,client,lock,s1>)</pre>
$Site() \triangleq$ $(!child^\uparrow(a, x, m).a/x\langle m \rangle$ $!fwd(s, a, x, m).s \cdot child\langle a, x, m \rangle$ $!\overline{in} \ login !\overline{out} \ logout$ $!s_0() \dots !s_N())$	<pre>let Site()= (!child^(a:agent,x:<'a>,m:'a); a/x<m> !forward(s:site,a:agent,x:<'a>,m:'a); s.child<a,x,m> !-in login !-out logout)</pre>
$Client(home, tracker, deliver, lock) \triangleq$ $(!lock^\uparrow(s).out \ logout.in \ s \ login.$ $forward^\uparrow\langle home, tracker, lock, s \rangle.moved\langle s \rangle$ $!deliver^\uparrow(s, x, m).$ $forward^\uparrow\langle home, tracker, lock, s \rangle.x\langle m \rangle)$	<pre>let Client(home:site, tracker:agent, deliver:<site,<'a>,'a>, lock:<site>) = (!lock^(s:site); out logout; in s.login; forward^<home,tracker,lock,s>; moved<s> !deliver^(s:site,x:<'a>,m:'a sub); forward^<home,tracker,lock,s>; x<m>)</pre>

Figure 5.3: Agent Tracker Specification and Implementation

5.5 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 π -calculus [83] and the Nomadic Pict programming language [84]. 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 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 [64]. 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 below.

This section describes a decentralised version of the agent tracker algorithm given in [77]. 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 5.3. 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 5.4 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 Chapter 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 5.5 describes a scenario in which a tracker agent sends a message to its client. A message

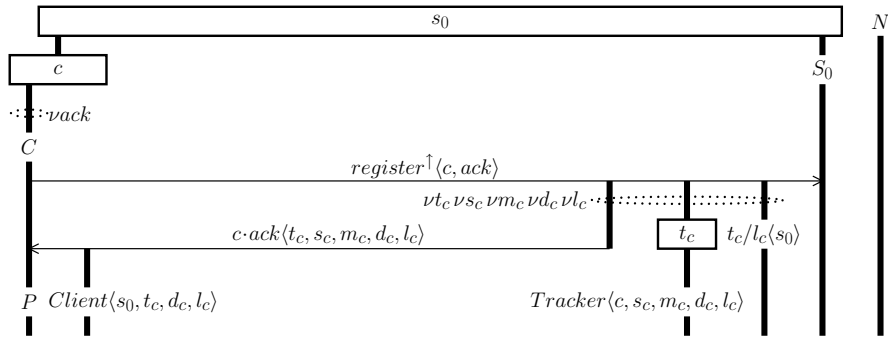


Figure 5.4: Tracker Registration

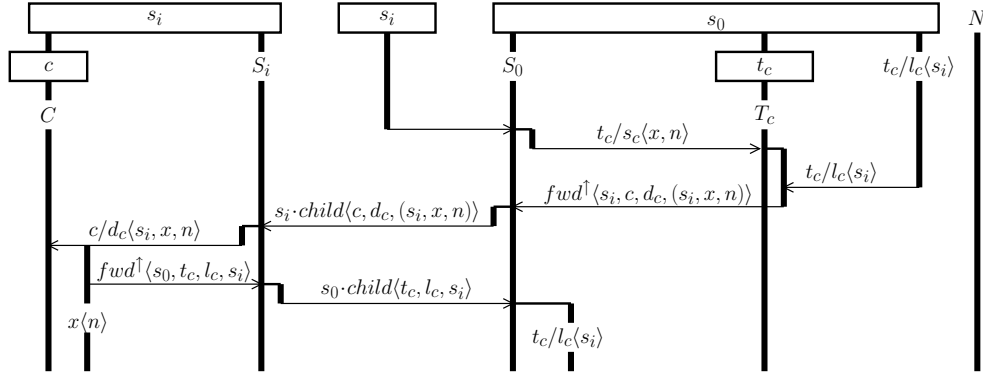


Figure 5.5: Tracker Delivery

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 presented below. 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 5.6. 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.

Program Code for the Agent Tracker Application

```
192.168.0.3:3011
[ let child = child:<agent,<'a>,'a sub> in
  let forward = forward:<site,agent,<'a>,'a sub> in
  let login = login:<migrate> in
  let logout = logout:<migrate> in
  ( !child^(a,x,m); a/x<m>
    | !forward(s,a,x,m); route^<s,child,(a,x,m)>
    | !-in login
    | !-out logout
    | !println(s); println<s>; print<s>
```

```

)
]
192.168.0.2:3010
[ type details = agent,<site>,<site,<'a>,'a sub>,<site>
  let s0 = 192.168.0.2:3010 in
  let s1 = 192.168.0.3:3011 in
  let s2 = 192.168.0.4:3012 in
  let s3 = 192.168.0.5:3013 in
  let child = child:<agent,<'a>,'a sub> in
  let forward = forward:<site,agent,<'a>,'a sub> in
  let login = login:<migrate> in
  let logout = logout:<migrate> in
  let register = register:<agent,<details>> in
  let moved = moved:<site> in
  let send = send:<<'a>,'a sub> in
  let Trusted()=
  ( !child^(a:agent,x:<'a>,m:'a sub); a/x<m>
  | !forward(s:site,a:agent,x:<'a>,m:'a sub);
    println<"sending " + show m + " on " + show x + " to site " + show s>;
    route^<s,child,(a,x,m)>
  | !-in login
  | !-out logout
  | !println(s:string); println<s>; print<s>
  )
  in
  let Client(home:site, client:agent, tracker:agent, move:<site>,
    deliver:<site,<'a>,'a sub>, lock:<site>) =
  ( !lock^(s:site);
    out logout;
    in s.login;
    forward^<home,tracker,lock,s>;
    moved<s>
  | !deliver^(s:site,x:<'a>,m:'a sub);
    forward^<home,tracker,lock,s>;
    println<show client + " received message from " + show m>;
    x<m>
  )
  in
  let Server(home:site) =
  ( !register(client:agent,ack:<details>);
    new tracker:agent
    new move:<site>
    new deliver:<site,<'a>,'a sub>
    new lock:<site>
    ( tracker
      [ !send^(x:<'a>,m:'a sub); lock^(s:site);
        forward^<s,client,deliver,(s,x,m)>
      | !move^(s1:site); s1^<>;
        lock^(s:site); forward^<s,client,lock,s1>
      ]
      | client/ack<tracker,move,deliver,lock>
      | tracker/lock<home>
    )
  | !s0() | !s1() | !s2() | !s3()
  )
  in
  let C(c1:agent,c2:agent,c3:agent,s2:site,s3:site,x1:<agent>,x2:<agent>,x3:<agent>) =
  new ack:<details>
  register^<c1,ack>;
  ack^(tracker:agent,move:<site>,deliver:<site,<'a>,'a sub>,lock:<site>);
  ( Client<s0,c1,tracker,move,deliver,lock>
  | c2.x1<tracker>
  | c3.x1<tracker>
  | x2^(t2:agent);
    x3^(t3:agent);
    ( tracker.move<s2>;
      moved(s2:site);

```

```

        println<show c1 + " was here">;
        forward<s0,t2,send,(x1,c1)>;
        forward<s0,tracker,move,s3>;
        moved(s3:site);
        forward<s0,t3,send,(x1,c1)>
      )
    )
  in
  let alice = alice:agent in
  let bob = bob:agent in
  let chris = chris:agent in
  let from_alice = from_alice:<agent> in
  let from_bob = from_bob:<agent> in
  let from_chris = from_chris:<agent> in
  ( Trusted<>
  | Server<s0>
  | alice[C<alice,bob,chris,s2,s3,from_alice,from_bob,from_chris>]
  | bob[C<bob,chris,alice,s3,s1,from_bob,from_chris,from_alice>]
  | chris[C<chris,alice,bob,s1,s2,from_chris,from_alice,from_bob>]
  )
]

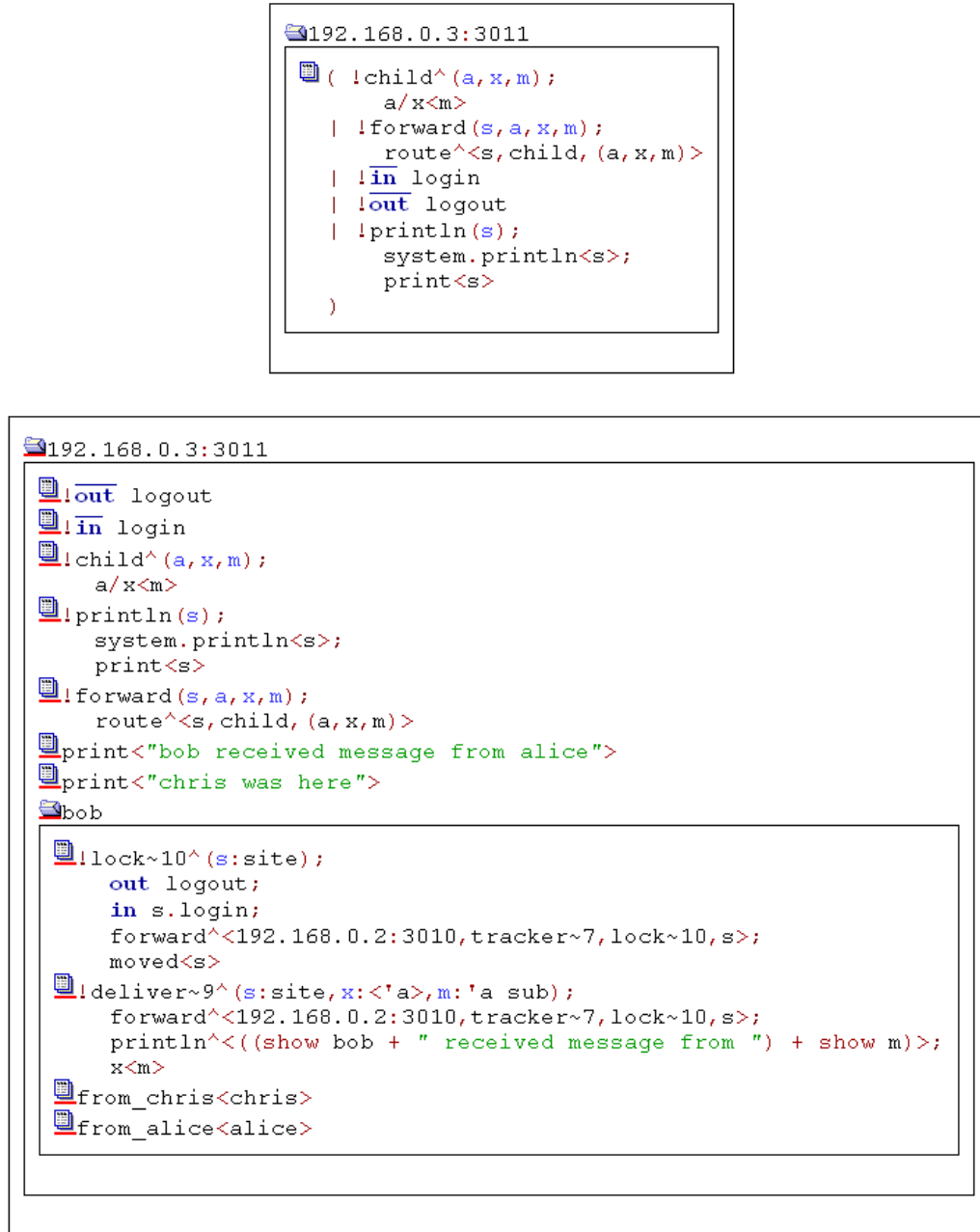
```

5.6 Related Work

The Channel Ambient Language is inspired by the Pict language [58], which uses the asynchronous π -calculus as the basis for programming concurrent applications. Pict also demonstrates how many high-level programming constructs, including functions, procedure calls and libraries for concurrency, can be encoded into a core π -calculus language. The Nomadic Pict language [84] is an extension of Pict with constructs for mobile programming. Nomadic Pict demonstrates how languages for mobile programming can be based on a formal model and used to program communication infrastructures for mobile agents. In [83] 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 [77]. 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.

The JoCaml project [26] extends the OCaml programming language with constructs for concurrent, distributed and mobile agent programming. The extensions are based on the Distributed Join calculus [29], which contains abstractions for mobile agents. The Join Language separates the logical structure of ambients from the physical structure of the network. In contrast, the Channel Ambient Language uses ambients to directly model the hierarchical topology of the network. This approach makes a number of assumptions about the network topology during application specification, resulting in a simplified implementation.

Probably the first and to date certainly the most widely used distributed functional program-

Figure 5.6: Initial and final Runtime states for site s_1 at address 192.168.0.3:3011

ming language is the untyped language Erlang [5]. Erlang was designed to be used within a distributed environment [4]. Data is exchanged between processes using asynchronous message passing over global channels. A process can spawn new processes to run on a different system, but the language is not designed for code mobility. More recently there have been a number of new distributed functional languages. The Acute language [70] extends an OCaml core to support

distributed programming. The details of how to do the distribution are left to the programmer but the system is powerful enough to enable threads of execution to be moved. Acute is interested in issues arising from deployment and development as well as execution, and addresses problems of type-safe versioning and rebinding. The language Alice [63] has been designed to support typed *open* programming, where blocks of code (as well as data) may be transferred between sites. Alice is an extension to SML. It provides features for concurrency and a higher-order extension to the SML module system that enhances modularity. Type-safe marshaling is enabled by introducing dynamically typed modules called packages.

Java [6] was the first major language designed for distributed programming. It is based on untyped serialisation and remote method invocation. Using reflection, other components can exploit type information. However, reflection is expensive and invites abuse, and code serialisation in Java needs to be programmed explicitly. No structural type checks are performed when a class is loaded, and method calls may cause a `NoSuchMethodError`. There are several Java-based agent programming systems that support strong mobility. These include Sumatra [3], Ara [51] and Nomads [72] which are implemented by modifying or rewriting the Java virtual machine, which must then be deployed at each site. Other agent programming systems avoid this disadvantage through code instrumentation. These include WASP [32], an extended version of Aglets [35], Java $\delta\pi$ [55], and any system which could be based on the thread migration techniques developed in [73] and [65].

5.7 Conclusion

This chapter presents a programming language for developing mobile applications, known as the Channel Ambient Language. The language illustrates how the high-level constructs of the Channel Ambient calculus can be used as programming abstractions for mobile applications. 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. A number of debugging mechanisms for mobile applications are presented, including the ability to visualise the current execution state of an application in real time. The Channel Ambient Language extends the calculus with useful programming constructs such as data structures, process definitions and system calls. The language is used to program an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network. This application was the subject of extensive research in the Nomadic Pict project, and was one of the first non-trivial mobile application to be proved correct using techniques developed for process calculi. Some of the benefits of programming this application in the Channel Ambient Language include the ability to specify access control mechanisms in a hierarchical network structure and

the use of a runtime system based on a provably correct abstract machine.

The Channel Ambient Language is merely a prototype, and a number of extensions to the language can be envisaged. One such extension is to provide a complete embedding of a functional language. The Pict language demonstrated how high-level encodings of functions, procedures and data structures could be readily embedded in a π -calculus language. This embedding also extends to Nomadic Pict. Since the Channel Ambient calculus is an extension of the π -calculus, a full embedding of a functional language can be incorporated into the Channel Ambient language in a similar way.

Other extensions to the language include a module system, libraries and system calls, which are standard features of functional languages. One particular extension that should be relatively straightforward is the addition of a thread library, since the language already provides powerful abstractions for multi-threading. Extensions for distributed communication and migration should also be relatively straightforward, since the language already incorporates high-level abstractions for sending messages and moving agents over a network. These high-level abstractions map directly to the TCP/IP socket layer and simplify the task of writing distributed applications. In future, it may also be useful to give the programmer more control over the underlying network settings. This would allow an experienced programmer to fine-tune the settings for each TCP/IP session, including the number of connection attempts, the retry delay, the number of pending accepts on a socket etc., while still remaining within the high-level abstractions of the Channel Ambient Language.

In future, it would also be useful to identify those circumstances in which the language abstractions are not adequate. This could help drive the development of novel abstractions for mobile programming, resulting in improvements to the underlying calculus. In general, there is a close link between the Channel Ambient Language presented here and the Channel Ambient calculus presented in Chapter 2. Therefore, suggested extensions to the calculus such as time and choice would directly influence the design of the language, and any new language constructs would in turn influence the design of the calculus. With each extension, care must be taken to ensure that both the language and the calculus are consistent, and that any formal reasoning done at the calculus level still holds in the language. The differences between language and calculus are often subtle. Apart from the obvious syntactic differences, the language also contains a number of high-level constructs and primitives that can be encoded in the core calculus. Broadly speaking, the language can therefore be regarded as the calculus plus syntactic sugar. This correspondence should be preserved with each new extension to the language or the calculus.

As far as modularity is concerned, mobile agents in the Channel Ambient Language can be viewed as a form of dynamic modules. They have an interface, which corresponds to the replicated services that can be accessed from outside the agent, they can move around, which enables them

to be downloaded on demand, they can be nested, which provides some form of privacy, and they allow for namespace separation, since different agents can contain replicated services with the same name. In future, the relationship between mobile agents and functional modules in the ML sense is certainly worth exploring. There are also a number of possible language extensions that are specific to distributed programming, such as module versioning and type-safe serialisation of program code. Many of these issues are being addressed in modern distributed extensions to functional languages, such as Acute [70].

A number of extensions could also be made to the runtime environment of the Channel Ambient Language. A clickable windows interface with menus and buttons would certainly be helpful. From a debugging perspective, in addition to generating the current execution state of the runtime it would be useful to generate a trace history, based on the graphical representation of the runtime state presented in Chapter 3. This trace history would be reminiscent of Message Sequence Charts, which represent a sequence of interactions as horizontal arrows between concurrent components and are ideal for visualising a particular run of an application. By generating such traces automatically, a complete graphical history of the execution of an application can be maintained. This could help identify the sequence of events leading to unexpected application behaviour.

More experience is also needed in using the Channel Ambient Language to develop mobile applications. The application presented in this chapter is based on the home server application developed in Nomadic Pict [83]. A range of other applications developed in Nomadic Pict could also be programmed [84], in order to further evaluate the benefits of using a language with nested agents and access control mechanisms.

Although the Channel Ambient Language is still in the early stages of development, it can already be used to rapidly develop prototypes of mobile applications and to test these prototypes in a distributed setting. These prototypes can be analysed using standard security mechanisms suggested in Chapter 2, in order to detect errors in the specification. Once the prototypes satisfy the necessary security requirements, they can be used as a basis for developing the final application in any chosen language, with the help of an appropriate Application Programming Interface. Of course, the real benefits will come from being able to specify the application in the Channel Ambient calculus, perform the analysis, and then execute the application in a provably correct runtime environment.

Chapter 6

Conclusion

6.1 Thesis Summary

This thesis presents a formalism for specifying and implementing secure mobile applications, known as the Channel Ambient System. The system consists of a calculus for specifying mobile applications and reasoning about their security properties, a programming language for implementing these applications and a runtime execution environment based on a provably correct abstract machine. A running example is used throughout the thesis to validate the various components of the Channel Ambient System. The example describes how a mobile agent can be used to monitor resources on a remote site, in order to overcome the limitations of network delay and disconnection. The main contributions of each chapter are summarised below.

Chapter 2 presents a novel calculus for specifying mobile applications, known as the Channel Ambient calculus. The calculus is inspired by previous work on calculi for mobility, and its design is directly influenced by the properties of mobile applications. A graphical representation for the calculus is also presented, which can be used to specify execution traces of applications. The calculus is well-suited to specifying mobile applications for networks that support the TCP/IP protocol. In particular, the use of channels in the calculus bears a close resemblance to the use of ports in TCP/IP network communication. The calculus also satisfies a number of fundamental safety properties, which ensure that mobile applications are free of runtime errors. Various security mechanisms developed for related calculi can be applied to the Channel Ambient calculus, in order to reason about the security properties of mobile applications. These include type systems to ensure reliable channel communication and syntax constraints to ensure agent authenticity. By remaining in the Ambient paradigm, the Channel Ambient calculus can potentially benefit from a wide range of security mechanisms developed for Ambient calculi.

Chapter 3 presents an abstract machine for the Channel Ambient calculus, known as the Chan-

nel Ambient Machine. 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. The Channel Ambient Machine is derived from the Channel Ambient calculus by defining a list syntax, which is close to an implementation language, together with a blocking semantics, which leads to an efficient implementation. A corresponding graphical representation for the Channel Ambient Machine is also presented, which can be used to visualise execution traces of applications for both debugging and tutorial purposes. The machine is also well-suited to modelling the execution of mobile applications on networks that support the TCP/IP protocol. Finally, the Channel Ambient machine is proved both sound and complete with respect to the Channel Ambient calculus, and is also proved to be free of runtime errors, deadlocks and livelocks. Although the proofs are non-trivial, they only need to be done once in order to ensure that any application specified in the calculus will be correctly executed by the machine. This ensures that the work done in specifying and verifying mobile applications is not lost during their implementation.

Chapter 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. A local runtime is first implemented by mapping the Channel Ambient Machine to simple function definitions. The local implementation highlights the close correspondence between the abstract machine and functional program code. A distributed runtime is then implemented by adding a thin layer of multiplexing on top of the TCP/IP protocol, and mapping the Channel Ambient Machine to a combination of function definitions and calls to the TCP/IP socket library. The distributed implementation highlights the close correspondence between the abstract machine and the main features of the TCP/IP protocol. A number of enhancements are then made to the distributed runtime in order to improve the efficiency and security of process execution, while reducing network congestion.

Chapter 5 presents a programming language for developing mobile applications, known as the Channel Ambient Language. The language illustrates how the high-level constructs of the Channel Ambient calculus can be used as programming abstractions for writing mobile applications. 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. A number of debugging mechanisms for mobile applications are presented, including the ability to visualise the current execution state of an application in real time. The Channel Ambient Language extends the calculus with useful programming constructs such as data structures, process definitions and system calls. The language is used to program an agent tracker application, which keeps track of the location of registered client agents as they move between trusted sites in a network. This application was the subject of extensive research in the Nomadic Pict project, and was one of the first non-trivial mobile applications to be proved correct using techniques developed for process

calculi. Some of the benefits of programming this application in the Channel Ambient Language include the ability to specify access control mechanisms in a hierarchical network structure and the use of a runtime system based on a provably correct abstract machine.

6.2 Future Work

Mobile Applications The Channel Ambient System has been used to specify and implement a simple resource monitoring application, together with a more complex agent tracker application for information retrieval. These two examples rely on distributed algorithms previously presented in [83], 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. It is only by experimenting with a wide variety of algorithms and applications that a realistic evaluation of the Channel Ambient System can be achieved. This will provide invaluable feedback on the most appropriate ways to modify and extend the system.

The design of the Channel Ambient System was heavily influenced by the properties of mobile applications. Throughout the development of the system, there was also continuous interplay between the calculus, the abstract machine, the runtime and the language. In some cases, a new calculus primitive was proposed to model a particular aspect of an application, but was then rejected in favour of an alternative primitive that enabled a more elegant abstract machine or a simplified runtime. In other cases, attempts to simplify the correctness proofs of the abstract machine resulted in new ideas for the design of the calculus. Future extensions to the Channel Ambient System will need to take into account all the components of the system in a similar fashion, in order to correct inconsistencies at an early stage.

Security of Mobile Applications 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 [20]. In future, these logics could be used to verify a number of security properties for the simple resource monitoring application described in Chapter 2, and also for the more complex agent tracker application described in Chapter 5. The latter application is based on a distributed algorithm that was previously proved correct in [76], using a novel form of process equivalence. The proofs were non-trivial, and required the development of a new intermediate language tailored to the particular algorithm. Furthermore, a new intermediate language needs to be developed to prove the correctness of each new distributed algorithm, resulting in significant overhead. It is hoped that Ambient Logics will provide a more general framework not

only for proving the correctness of algorithms, but also for reasoning about a wide range safety and security properties.

The security of mobile applications can also be improved by enhancing the security of their runtime environment. One way to achieve this is via the automation of correctness proofs, to ensure that the runtime implementation is correct with respect to its specification. In future, the correctness proof of the runtime specification with respect to the calculus could be automated with the help of a standard proof assistant such as HOL. Taking this a step further, a correctness proof of the runtime code with respect to the runtime specification could also be automated, resulting in a provably correct series of transformations from the high-level calculus right down to the executable runtime code.

In general, there has been a significant amount of research on security mechanisms for Ambient calculi. Although only a fragment of this research has been applied to the Channel Ambient calculus, much more could be applied in future. Rather than adding to the vast literature on security mechanisms at the calculus level, this thesis focuses instead on ensuring that, if a given security properties holds for the calculus specification of an application, it will also hold for its implementation.

Implementing Mobile Applications From an implementation perspective, a number of improvements can be made to both the Channel Ambient Runtime and the Channel Ambient Language, as discussed in Chapters 4 and 5, respectively. 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. At present, a limited number of optimisations have been considered, such as using a round-robin scheduling algorithm. In future, more sophisticated scheduling algorithms could be developed, which also take into account the particular properties of the applications being executed. At present, migration of agents between runtimes is implemented by sending serialised data structures over a network. In future, the runtime could be extended so that mobile agents are transmitted using a suitable bytecode format, in order to maintain compatibility between runtimes regardless of the internal format of runtime data structures. The runtime implementation presented in this thesis could also be applied to other variants of Boxed Ambients. Taking this a step further, it should be possible to define a single runtime that takes the semantics of different variants of Boxed Ambients as a parameter. Such a runtime would be useful for experimenting with different programming abstractions for mobile applications within a single runtime system. The Channel Ambient Runtime is currently implemented in the OCaml programming language, but the high-level nature of the abstract machine means that it could also be implemented in a range of languages, such as Haskell, Java or C++. As languages for distributed computing evolve and new languages are proposed, the Channel Ambient Runtime

could be re-implemented to take advantage of new developments in language design.

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 π -calculus language. Since the Channel Ambient calculus is an extension of the π -calculus, a full embedding of a functional language can be incorporated into the Channel Ambient language in a similar way. Other extensions to the language include a module system, additional libraries and system calls, most of which are standard features of functional languages. One particular extension that should be relatively straightforward is the addition of a thread library, since the language already provides powerful abstractions for multi-threading. Extensions for distributed communication and migration should also be relatively straightforward, since the language already incorporates high-level abstractions for sending messages and moving agents over a network. These high-level abstractions map directly to the TCP/IP socket layer and greatly simplify the task of writing distributed applications. Although the Channel Ambient Language is still in the early stages of development, it can already be used to rapidly develop prototypes of mobile applications and test these prototypes in a distributed setting. The prototypes can be analysed using standard security mechanisms described in Chapter 2, and then used as a model for developing the final application in any chosen language. Of course, the real benefits will come from being able to specify the application in the Channel Ambient calculus, perform the analysis and then execute the final application in a provably correct runtime environment. More importantly, the compositional nature of the calculus will allow parts of the application to be specified in a modular fashion, so that the application can be built component by component, with the knowledge that application security will be preserved with each new extension.

Specifying Mobile Applications From a specification perspective, the Channel Ambient calculus appears to be at a suitable level of abstraction for specifying mobile applications such as resource monitoring and tracking the location of mobile agents. A number of extensions to the calculus can also be envisaged, including adding a notion of time and non-deterministic choice, as discussed in Chapter 2. A promising approach to modelling time in the π -calculus is presented in [7], which can also be readily applied to Ambient calculi. A possible extension for modelling non-deterministic choice is described in the Bio Ambient calculus [60]. The precise nature of these extensions will depend largely on the nature of the applications to be specified. This will require more experience in using the Channel Ambient calculus to specify a range of applications. 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 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, [48] uses the π -calculus to describe a simple protocol for switching mobile phones between base stations.

Stochastic Simulation Somewhat unexpectedly, the research presented in this thesis can also be applied to the stochastic simulation of mobile applications. Such simulations are typically used to evaluate the performance of distributed algorithms, or to detect potential bottlenecks within a range of networks topologies. They can also be used to detect flaws in distributed algorithms, which may only become apparent after multiple simulation runs. An example of how process calculi can be applied to the stochastic simulation of concurrent applications is the PEPA Workbench [37], which is used to study both behavioural and performance properties of applications.

In future, the Channel Ambient Machine presented in this thesis could be used to implement a stochastic simulator for mobile applications. The machine is a formal specification of a runtime for executing calculus processes, which has been proved correct with respect to the Channel Ambient calculus. The processes of the calculus can be made stochastic by adding reaction rates to each channel, as in the stochastic variant of the Ambient calculus presented in [60]. An abstract machine for a stochastic simulator can then be defined by augmenting the Channel Ambient Machine with an appropriate stochastic scheduling algorithm [36]. Using these principles, an abstract machine has already been defined for the stochastic π -calculus with mixed choice [54], known as the Stochastic Pi Machine. The abstract machine was proved correct with respect to the stochastic π -calculus, and was used to implement a stochastic simulator for biological processes [53]. The existence of an abstract machine provided increased flexibility in the choice of implementation language, resulting in a significant gain in efficiency over a previously existing simulator [59]. The correctness of the abstract machine also helped to provide greater confidence in the simulation results. In future, similar principles could be used to develop a stochastic simulator for the Channel Ambient calculus, in order to perform stochastic simulations of mobile applications.

Ambient Programming In many respects, this thesis investigates to what extent a variant of the Ambient calculus can be used for specifying and implementing secure mobile applications. In particular, the thesis 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 π -calculus, for which the underlying theory [76] and implementation [83] were developed in concert, in order to produce the beginnings of a programming language and runtime system for developing real mobile applications [84]. 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 [21], which states: “On this foundation, we can envision new programming methodologies, programming libraries and programming languages for global computation”.

Appendix A

Proofs

The proofs in this appendix make use of the notations described in Figure A.1, where implication has the lowest precedence among the symbols used.

$\stackrel{\text{IH}}{\Rightarrow}$	\triangleq	By Induction Hypothesis
$\stackrel{\text{A.1.3}}{\Rightarrow}$	\triangleq	By Definition A.1.3
$\stackrel{\text{A.2.10}}{\Rightarrow}$	\triangleq	By Lemma A.2.10
$\stackrel{(\text{A.1})}{\Rightarrow}$	\triangleq	By Rule (A.1)
$[\text{A.1}]$	\triangleq	Case for Rule (A.1)

Figure A.1: Proof Notations

A.1 Security of the Channel Ambient Calculus

The Channel Ambient calculus is summarised in Definitions A.1.1 - A.1.3. The remainder of this appendix outlines the proof of security properties of the Channel Ambient calculus (CA), as described in Chapter 2. Note that for proofs by induction on the definition of structural congruence in CA, the cases for the symmetric rules are omitted, since they are analogous to the cases for rules (A.15)-(A.22). The cases for the reflexive, transitive and congruence rules are also omitted, since they are straightforward.

$P, Q, R ::=$	$\mathbf{0}$	Null	(A.1)	$\alpha ::=$	$a \cdot x \langle v \rangle$	Sibling Output	(A.7)
$ $	$P \mid Q$	Parallel	(A.2)	$ $	$x^\dagger \langle v \rangle$	Parent Output	(A.8)
$ $	$\nu n P$	Restriction	(A.3)	$ $	$x(u)$	Internal Input	(A.9)
$ $	$a \boxed{P}$	Ambient	(A.4)	$ $	$x^\dagger(u)$	External Input	(A.10)
$ $	$\alpha.P$	Action	(A.5)	$ $	$\text{in } a \cdot x$	Enter	(A.11)
$ $	$! \alpha.P$	Replication	(A.6)	$ $	$\text{out } x$	Leave	(A.12)
				$ $	$\overline{\text{in}} x$	Accept	(A.13)
				$ $	$\overline{\text{out}} x$	Release	(A.14)

Definition A.1.1. Syntax of CA

	$\mathbf{0} P$	\equiv	P	(A.15)
	$P Q$	\equiv	$Q P$	(A.16)
	$P (Q R)$	\equiv	$(P Q) R$	(A.17)
	$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow ! \alpha.P$	\equiv	$\alpha.(P ! \alpha.P)$	(A.18)
	$\nu n \mathbf{0}$	\equiv	$\mathbf{0}$	(A.19)
	$\nu n \nu m P$	\equiv	$\nu m \nu n P$	(A.20)
	$n \notin \text{fn}(Q) \Rightarrow (\nu n P) Q$	\equiv	$\nu n (P Q)$	(A.21)
	$a \neq n \Rightarrow a \boxed{\nu n P}$	\equiv	$\nu n a \boxed{P}$	(A.22)
	$P \equiv P' \Rightarrow Q P$	\equiv	$Q P'$	(A.23)
	$P \equiv P' \Rightarrow \nu n P$	\equiv	$\nu n P'$	(A.24)
	$P \equiv P' \Rightarrow a \boxed{P}$	\equiv	$a \boxed{P'}$	(A.25)
	$P \equiv P' \Rightarrow ! \alpha.P$	\equiv	$! \alpha.P'$	(A.26)
	$P \equiv P' \Rightarrow \alpha.P$	\equiv	$\alpha.P'$	(A.27)
	P	\equiv	P	(A.28)
	$P \equiv Q \Rightarrow Q$	\equiv	P	(A.29)
	$P \equiv Q \wedge Q \equiv R \Rightarrow P$	\equiv	R	(A.30)

Definition A.1.2. Structural Congruence in CA

$$a \boxed{b \cdot x \langle v \rangle . P | P'} | b \boxed{x^\dagger \langle u \rangle . Q | Q'} \longrightarrow a \boxed{P | P'} | b \boxed{Q_{\{v/u\}} | Q'} \quad (\text{A.31})$$

$$a \boxed{x^\dagger \langle v \rangle . P | P'} | x(u) . Q \longrightarrow a \boxed{P | P'} | Q_{\{v/u\}} \quad (\text{A.32})$$

$$a \boxed{\text{in } b \cdot x . P | P'} | b \boxed{\overline{\text{in}} x . Q | Q'} \longrightarrow b \boxed{Q | Q'} | a \boxed{P | P'} \quad (\text{A.33})$$

$$b \boxed{a \boxed{\text{out } x . P | P'} | \overline{\text{out}} x . Q | Q'} \longrightarrow b \boxed{Q | Q'} | a \boxed{P | P'} \quad (\text{A.34})$$

$$P \longrightarrow P' \Rightarrow P | Q \longrightarrow P' | Q \quad (\text{A.35})$$

$$P \longrightarrow P' \Rightarrow \nu n P \longrightarrow \nu n P' \quad (\text{A.36})$$

$$P \longrightarrow P' \Rightarrow a \boxed{P} \longrightarrow a \boxed{P'} \quad (\text{A.37})$$

$$Q \equiv P \longrightarrow P' \equiv Q' \Rightarrow Q \longrightarrow Q' \quad (\text{A.38})$$

Definition A.1.3. Reduction in CA

$$\mathbf{0}_\sigma \triangleq \mathbf{0} \quad (\text{A.39})$$

$$(P \mid Q)_\sigma \triangleq P_\sigma \mid Q_\sigma \quad (\text{A.40})$$

$$(\nu n P)_\sigma \triangleq \nu n P_{\sigma \setminus \{n\}} \quad (\text{A.41})$$

$$(a \boxed{P})_\sigma \triangleq a_\sigma \boxed{P_\sigma} \quad (\text{A.42})$$

$$(\alpha.P)_\sigma \triangleq \alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)} \quad (\text{A.43})$$

$$(!\alpha.P)_\sigma \triangleq !\alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)} \quad (\text{A.44})$$

$$a \cdot x \langle v \rangle_\sigma \triangleq a_\sigma \cdot x_\sigma \langle v_\sigma \rangle \quad (\text{A.45})$$

$$x^\dagger \langle v \rangle_\sigma \triangleq x_\sigma^\dagger \langle v_\sigma \rangle \quad (\text{A.46})$$

$$x(u)_\sigma \triangleq x_\sigma(u) \quad (\text{A.47})$$

$$x^\dagger(u)_\sigma \triangleq x_\sigma^\dagger(u) \quad (\text{A.48})$$

$$(\text{in } a \cdot x)_\sigma \triangleq \text{in } a_\sigma \cdot x_\sigma \quad (\text{A.49})$$

$$(\text{out } x)_\sigma \triangleq \text{out } x_\sigma \quad (\text{A.50})$$

$$(\overline{\text{in}} x)_\sigma \triangleq \overline{\text{in}} x_\sigma \quad (\text{A.51})$$

$$(\overline{\text{out}} x)_\sigma \triangleq \overline{\text{out}} x_\sigma \quad (\text{A.52})$$

Definition A.1.4. Substitution in CA

$$\text{fn}(\mathbf{0}) \triangleq \emptyset$$

$$\text{fn}(P \mid Q) \triangleq \text{fn}(P) \cup \text{fn}(Q)$$

$$\text{fn}(\nu n P) \triangleq \text{fn}(P) \setminus \{n\}$$

$$\text{fn}(a \boxed{P}) \triangleq \text{fn}(P) \cup \{a\}$$

$$\text{fn}(\alpha.P) \triangleq \text{fn}(\alpha) \cup (\text{fn}(P) \setminus \text{bn}(\alpha))$$

$$\text{fn}(!\alpha.P) \triangleq \text{fn}(\alpha) \cup (\text{fn}(P) \setminus \text{bn}(\alpha))$$

$$\text{fn}(a \cdot x \langle v \rangle) \triangleq \{a, x, v\}$$

$$\text{fn}(x^\dagger \langle v \rangle) \triangleq \{x, v\}$$

$$\text{fn}(x(u)) \triangleq \{x\}$$

$$\text{fn}(x^\dagger(u)) \triangleq \{x\}$$

$$\text{fn}(\text{in } a \cdot x) \triangleq \{a, x\}$$

$$\text{fn}(\text{out } x) \triangleq \{x\}$$

$$\text{fn}(\overline{\text{in}} x) \triangleq \{x\}$$

$$\text{fn}(\overline{\text{out}} x) \triangleq \{x\}$$

Definition A.1.5. Free Values of CA

$$\Gamma \vdash \mathbf{0} \quad (\text{A.53})$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \quad (\text{A.54})$$

$$\frac{\Gamma, n:\tau \vdash P \quad \tau \neq \delta}{\Gamma \vdash \nu n:\tau P} \quad (\text{A.55})$$

$$\frac{\Gamma(a) = \text{amb} \quad \Gamma \vdash P}{\Gamma \vdash a \boxed{P}} \quad (\text{A.56})$$

$$\frac{\Gamma \vdash \alpha.P}{\Gamma \vdash !\alpha.P} \quad (\text{A.57})$$

$$\frac{\Gamma(a) = \text{amb} \quad \Gamma(x) = \langle \tau \rangle \quad \Gamma(v) = \tau \quad \Gamma \vdash P}{\Gamma \vdash a \cdot x \langle v \rangle.P} \quad (\text{A.58})$$

$$\frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma(v) = \tau \quad \Gamma \vdash P}{\Gamma \vdash x^\dagger \langle v \rangle.P} \quad (\text{A.59})$$

$$\frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma, u:\tau \vdash P}{\Gamma \vdash x^\dagger(u).P} \quad (\text{A.60})$$

$$\frac{\Gamma(x) = \langle \tau \rangle \quad \Gamma, u:\tau \vdash P}{\Gamma \vdash x(u).P} \quad (\text{A.61})$$

$$\frac{\Gamma(a) = \text{amb} \quad \Gamma(x) = \langle \text{mv} \rangle \quad \Gamma \vdash P}{\Gamma \vdash \text{in } a \cdot x.P} \quad (\text{A.62})$$

$$\frac{\Gamma(x) = \langle \text{mv} \rangle \quad \Gamma \vdash P}{\Gamma \vdash \text{out } x.P} \quad (\text{A.63})$$

$$\frac{\Gamma(x) = \langle \text{mv} \rangle \quad \Gamma \vdash P}{\Gamma \vdash \overline{\text{in}} x.P} \quad (\text{A.64})$$

$$\frac{\Gamma(x) = \langle \text{mv} \rangle \quad \Gamma \vdash P}{\Gamma \vdash \overline{\text{out}} x.P} \quad (\text{A.65})$$

Definition A.1.6. Typing Rules in CA

$\tau ::=$	amb	Ambient	$\Gamma ::=$	$v_1 : \tau_1, \dots, v_N : \tau_N$	Type Context
		$\langle \tau \rangle$			Communication Channel
		$\langle mv \rangle$			Migration Channel
		δ			Base Value

Definition A.1.7. Types in CA

$$\begin{aligned}
 (\Gamma, v : \tau)(v) &\triangleq \tau \\
 u \neq v \Rightarrow (\Gamma, u : \tau)(v) &\triangleq \Gamma(v) \\
 ()(v) &\triangleq \text{undefined}
 \end{aligned}$$

Definition A.1.8. Type Lookup in CA

$$\text{fa}(\mathbf{0}) \triangleq \emptyset \quad (\text{A.66})$$

$$\text{fa}(P \mid Q) \triangleq \text{fa}(P) \cup \text{fa}(Q) \quad (\text{A.67})$$

$$\text{fa}(\nu n P) \triangleq \text{fa}(P) \setminus \{n\} \quad (\text{A.68})$$

$$\text{fa}(a \boxed{P}) \triangleq \text{fa}(P) \cup \{a\} \quad (\text{A.69})$$

$$\text{fa}(\alpha.P) \triangleq \text{fa}(P) \setminus \text{bn}(\alpha) \quad (\text{A.70})$$

$$\text{fa}(!\alpha.P) \triangleq \text{fa}(P) \setminus \text{bn}(\alpha) \quad (\text{A.71})$$

Definition A.1.9. Free Ambients in CA

$$a \boxed{P} \downarrow_a \quad (\text{A.72})$$

$$P \downarrow_a \Rightarrow (P \mid Q) \downarrow_a \quad (\text{A.73})$$

$$P \downarrow_a \Rightarrow (Q \mid P) \downarrow_a \quad (\text{A.74})$$

$$a \neq m \wedge P \downarrow_a \Rightarrow (\nu m P) \downarrow_a \quad (\text{A.75})$$

$$P \downarrow_a \Rightarrow b \boxed{P} \downarrow_a \quad (\text{A.76})$$

Definition A.1.10. Active Ambients in CA

$$P, Q, R ::= \mathbf{0} \quad \text{Null} \quad (\text{A.77})$$

$$\mid P \mid Q \quad \text{Parallel} \quad (\text{A.78})$$

$$\mid \nu n P \quad \text{Restriction} \quad (\text{A.79})$$

$$\mid a \boxed{P} \quad \text{Ambient} \quad (\text{A.80})$$

$$\mid \alpha.P \quad \text{Action, } \text{fa}(P) \cap \text{bn}(\alpha) = \emptyset \quad (\text{A.81})$$

$$\mid !\alpha.P \quad \text{Replication} \quad (\text{A.82})$$

Definition A.1.11. Syntax of CA^-

A.1.1 Safety

Safety ensures that the calculus always produces a valid process after each execution step. This ensures that the calculus does not produce any runtime errors when executing a given process, where a runtime error corresponds to a process reaching an undefined state. Calculus execution is defined in terms of substitution, structural congruence and reduction. Therefore, in order to prove the safety of the calculus it is necessary to prove that substitution, structural congruence and reduction are safe.

Substitution Safety ensures that the result of applying a substitution is always a valid calculus process. According to Lemma A.1.12 (Substitution Safety), if a substitution σ is applied to a process P then the result is a valid calculus process.

Structural Safety ensures that the result of applying a structural congruence rule is always a valid calculus process. According to Lemma A.1.13 (Structural Safety), if a process P is structurally congruent to a process Q then Q is a valid calculus process.

Finally, Reduction Safety ensures that the result of a reduction is always a valid calculus process. According to Theorem A.1.14 (Reduction Safety), if a process P can reduce to P' then P' is a valid calculus process.

Lemma A.1.12. (*Substitution Safety*) $\forall \sigma, P. P \in \text{CA} \Rightarrow P_\sigma \in \text{CA}$

Proof. By induction on Definition A.1.4 of substitution in CA:

$$\begin{array}{llll}
\mathbf{0} \in \text{CA} & \xRightarrow{(\text{A.39})} & \mathbf{0}_\sigma \in \text{CA} \\
P \mid Q \in \text{CA} & \xRightarrow{(\text{A.2})} P, Q \in \text{CA} \xRightarrow{\text{IH}} P_\sigma, Q_\sigma \in \text{CA} \xRightarrow{(\text{A.2})} P_\sigma \mid Q_\sigma \in \text{CA} \xRightarrow{(\text{A.40})} (P \mid Q)_\sigma \in \text{CA} \\
\nu n P \in \text{CA} & \xRightarrow{(\text{A.3})} P \in \text{CA} \xRightarrow{\text{IH}} P_{\sigma \setminus \{n\}} \in \text{CA} \xRightarrow{(\text{A.3})} \nu n P_{\sigma \setminus \{n\}} \in \text{CA} \xRightarrow{(\text{A.41})} (\nu n P)_\sigma \in \text{CA} \\
a[\overline{P}] \in \text{CA} & \xRightarrow{(\text{A.4})} P \in \text{CA} \xRightarrow{\text{IH}} P_\sigma \in \text{CA} \xRightarrow{(\text{A.4})} a_\sigma[\overline{P_\sigma}] \in \text{CA} \xRightarrow{(\text{A.42})} (a[\overline{P}])_\sigma \in \text{CA} \\
\alpha.P \in \text{CA} & \xRightarrow{(\text{A.5})} P \in \text{CA} \xRightarrow{\text{IH}} P_{\sigma \setminus \text{bn}(\alpha)} \in \text{CA} \xRightarrow{(\text{A.5})} \alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)} \in \text{CA} \xRightarrow{(\text{A.43})} (\alpha.P)_\sigma \in \text{CA} \\
!\alpha.P \in \text{CA} & \xRightarrow{(\text{A.6})} P \in \text{CA} \xRightarrow{\text{IH}} P_{\sigma \setminus \text{bn}(\alpha)} \in \text{CA} \xRightarrow{(\text{A.6})} !\alpha_\sigma.P_{\sigma \setminus \text{bn}(\alpha)} \in \text{CA} \xRightarrow{(\text{A.44})} (!\alpha.P)_\sigma \in \text{CA}
\end{array}$$

□

Lemma A.1.13. (*Structural Safety*) $\forall P. P \in \text{CA} \wedge P \equiv P' \Rightarrow P' \in \text{CA}$

Proof. By induction on Definition A.1.2 of structural congruence in CA:

$$\begin{array}{lll}
\mathbf{0} \mid P \in \text{CA} & \xRightarrow{(A.2)} & P \in \text{CA} \\
P \mid Q \in \text{CA} & \xRightarrow{(A.2)} P, Q \in \text{CA} \xRightarrow{(A.2)} & Q \mid P \in \text{CA} \\
P \mid (Q \mid R) \in \text{CA} & \xRightarrow{(A.2)} P, Q, R \in \text{CA} \xRightarrow{(A.2)} & (P \mid Q) \mid R \in \text{CA} \\
\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \wedge !\alpha.P \in \text{CA} & \xRightarrow{(A.6)} P \in \text{CA} \xRightarrow{(A.2)} P \mid !\alpha.P \in \text{CA} \xRightarrow{(A.5)} & \alpha.(P \mid !\alpha.P) \in \text{CA} \\
\nu n \mathbf{0} \in \text{CA} & \xRightarrow{(A.1)} & \mathbf{0} \in \text{CA} \\
\nu n \nu m P \in \text{CA} & \xRightarrow{(A.3)} P \in \text{CA} \xRightarrow{(A.3)} & \nu m \nu n P \in \text{CA} \\
n \notin \text{fn}(Q) \wedge (\nu n P) \mid Q \in \text{CA} & \xRightarrow{(A.3, A.2)} P, Q \in \text{CA} \xRightarrow{(A.2)} P \mid Q \in \text{CA} \xRightarrow{(A.3)} & \nu n (P \mid Q) \in \text{CA} \\
a \neq n \wedge a \boxed{\nu n P} \in \text{CA} & \xRightarrow{(A.3, A.4)} P \in \text{CA} \xRightarrow{(A.4)} a \boxed{P} \in \text{CA} \xRightarrow{(A.3)} & \nu n a \boxed{P} \in \text{CA}
\end{array}$$

□

Theorem A.1.14. (*Reduction Safety*) $\forall P. P \in \text{CA} \wedge P \longrightarrow P' \Rightarrow P' \in \text{CA}$

Proof. By Lemma A.1.13 (Structural Safety), Lemma A.1.12 (Substitution Safety) and by induction on Definition A.1.3 of reduction in CA:

$$\begin{array}{lll}
P \longrightarrow P' \wedge P \mid Q \in \text{CA} & \xRightarrow{(A.2)} P, Q \in \text{CA} \xRightarrow{\text{IH}} P' \in \text{CA} \xRightarrow{(A.2)} & P' \mid Q \in \text{CA} \\
P \longrightarrow P' \wedge \nu n P \in \text{CA} & \xRightarrow{(A.3)} P \in \text{CA} \xRightarrow{\text{IH}} P' \in \text{CA} \xRightarrow{(A.3)} & \nu n P' \in \text{CA} \\
Q \equiv P \longrightarrow P' \equiv Q' \wedge Q \in \text{CA} & \xRightarrow{A.1.13} P \in \text{CA} \xRightarrow{\text{IH}} P' \in \text{CA} \xRightarrow{A.1.13} & Q' \in \text{CA} \\
P \longrightarrow P' \wedge a \boxed{P} \in \text{CA} & \xRightarrow{(A.4)} P \in \text{CA} \xRightarrow{\text{IH}} P' \in \text{CA} \xRightarrow{(A.4)} & a \boxed{P'} \in \text{CA} \\
\\
a \boxed{b \cdot x \langle v \rangle . P \mid P'} \mid b \boxed{x^\dagger(u) . Q \mid Q'} \in \text{CA} & \xRightarrow{A.1.1} P, P', Q, Q' \in \text{CA} \xRightarrow{A.1.12} Q_{\{v/u\}} \in \text{CA} \\
& \xRightarrow{A.4, A.2} a \boxed{P \mid P'} \mid b \boxed{Q_{\{v/u\}} \mid Q'} \in \text{CA} \\
a \boxed{x^\dagger \langle v \rangle . P \mid P'} \mid x(u) . Q \in \text{CA} & \xRightarrow{A.1.1} P, P', Q \in \text{CA} \xRightarrow{A.1.12} Q_{\{v/u\}} \in \text{CA} \\
& \xRightarrow{A.4, A.2} a \boxed{P \mid P'} \mid Q_{\{v/u\}} \in \text{CA} \\
a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\overline{\text{in } x} . Q \mid Q'} \in \text{CA} & \xRightarrow{A.1.1} P, P', Q, Q' \in \text{CA} \\
& \xRightarrow{A.4, A.2} b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \in \text{CA} \\
b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \overline{\text{out } x} . Q \mid Q'} \in \text{CA} & \xRightarrow{A.1.1} P, P', Q, Q' \in \text{CA} \\
& \xRightarrow{A.4, A.2} b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \in \text{CA}
\end{array}$$

□

A.1.2 Typing

Lemma A.1.15 (Type Weakening) allows a type binding to be added for a value that is unused in a given process. Conversely, Lemma A.1.16 (Type Strengthening) allows a type binding to be removed for a value that is unused in a given process. Lemma A.1.17 (Type Substitution) ensures that the type of a process is preserved by substitution of values with the same type, while Lemma A.1.18 (Type Structure) ensures that the type of a process is preserved by structural congruence. Finally, Theorem A.1.19 (Subject Reduction) ensures that the type of a process is preserved by reduction. This ensures that well-typed processes remain well-typed after each reduction step.

Lemma A.1.15. (*Type Weakening*) $\forall x. \forall P. x \notin \text{fn}(P) \wedge \Gamma \vdash P \Rightarrow \Gamma, x:\tau \vdash P$

Proof. By induction on Definition A.1.6 of typing in CA. \square

Lemma A.1.16. (*Type Strengthening*) $\forall x. \forall P. x \notin \text{fn}(P) \wedge \Gamma, x:\tau \vdash P \Rightarrow \Gamma \vdash P$

Proof. By induction on Definition A.1.6 of typing in CA. \square

Lemma A.1.17. (*Type Substitution*) $\forall u. \forall v. \forall P. \Gamma(u) = \Gamma(v) \wedge \Gamma \vdash P \Rightarrow \Gamma \vdash P_{\{v/u\}}$

Proof. By induction on Definition A.1.6 of typing in CA. \square

Lemma A.1.18. (*Type Structure*)
 1. $\forall P. P \equiv Q \wedge \Gamma \vdash P \Rightarrow \Gamma \vdash Q$
 2. $\forall P. P \equiv Q \wedge \Gamma \vdash Q \Rightarrow \Gamma \vdash P$

Proof. By Lemma A.1.15 (Type Weakening), Lemma A.1.16 (Type Strengthening) and by induction on Definition A.1.2 of structural congruence in CA, where $\Gamma \vdash P, Q$ is short for $\Gamma \vdash P \wedge \Gamma \vdash Q$. Cases for part (2) that are analogous to those of part (1) are omitted. The case for symmetry follows from the induction hypothesis. The remaining cases for the reflexive, transitive and congruence rules are straightforward and are omitted as usual:

$$\begin{array}{lll}
 \Gamma \vdash \mathbf{0} \mid P & \stackrel{(A.54)}{\Rightarrow} \Gamma \vdash \mathbf{0} \wedge \Gamma \vdash P \Rightarrow & \Gamma \vdash P \\
 \Gamma \vdash P \mid Q & \stackrel{(A.54)}{\Rightarrow} \Gamma \vdash P \wedge \Gamma \vdash Q \Rightarrow \Gamma \vdash Q \wedge \Gamma \vdash P \stackrel{(A.54)}{\Rightarrow} & \Gamma \vdash Q \mid P \\
 \Gamma \vdash P \mid (Q \mid R) & \stackrel{(A.54)}{\Rightarrow} \Gamma \vdash P \wedge \Gamma \vdash Q, R \Rightarrow \Gamma \vdash P, Q \wedge \Gamma \vdash R \stackrel{(A.54)}{\Rightarrow} & \Gamma \vdash (P \mid Q) \mid R \\
 \text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \wedge & & \\
 \Gamma \vdash !\alpha. P & \stackrel{(A.57)}{\Rightarrow} \Gamma \vdash \alpha. P \stackrel{(A.58-A.65)}{\Rightarrow} \Gamma \vdash P \stackrel{(A.58-A.65, A.54)}{\Rightarrow} & \Gamma \vdash \alpha. (P \mid !\alpha. P) \\
 \Gamma \vdash \nu n:\tau \mathbf{0} & \stackrel{(A.55)}{\Rightarrow} \Gamma, n:\tau \vdash \mathbf{0} \stackrel{A.1,16}{\Rightarrow} & \Gamma \vdash \mathbf{0} \\
 \Gamma \vdash \mathbf{0} & \stackrel{A.1,15}{\Rightarrow} \Gamma, n:\tau \vdash \mathbf{0} \stackrel{(A.55)}{\Rightarrow} & \Gamma \vdash \nu n:\tau \mathbf{0} \\
 \Gamma \vdash \nu n:\tau \nu m:\tau' P & \stackrel{(A.55)}{\Rightarrow} \Gamma, n:\tau, m:\tau' \vdash P \Rightarrow \Gamma, m:\tau', n:\tau \vdash P \stackrel{(A.55)}{\Rightarrow} & \Gamma \vdash \nu m:\tau' \nu n:\tau P \\
 n \notin \text{fn}(Q) \wedge & & \\
 \Gamma \vdash (\nu n:\tau P) \mid Q & \stackrel{(A.55, A.54)}{\Rightarrow} \Gamma, n:\tau \vdash P \wedge \Gamma \vdash Q \stackrel{A.1,15}{\Rightarrow} \Gamma, n:\tau \vdash P, Q \stackrel{(A.55, A.54)}{\Rightarrow} & \Gamma \vdash \nu n:\tau (P \mid Q) \\
 n \notin \text{fn}(Q) \wedge & & \\
 \Gamma \vdash \nu n (P \mid Q) & \stackrel{(A.55, A.54)}{\Rightarrow} \Gamma, n:\tau \vdash P, Q \stackrel{A.1,16}{\Rightarrow} \Gamma, n:\tau \vdash P \wedge \Gamma \vdash Q \stackrel{(A.55, A.54)}{\Rightarrow} & \Gamma \vdash (\nu n:\tau P) \mid Q \\
 a \neq n \wedge \Gamma \vdash a \boxed{\nu n:\tau P} & \stackrel{(A.55, A.56)}{\Rightarrow} \Gamma(a) = \text{amb} \wedge \Gamma, n:\tau \vdash P \stackrel{(A.55, A.56)}{\Rightarrow} & \Gamma \vdash \nu n:\tau a \boxed{P}
 \end{array}$$

\square

Theorem A.1.19. (*Subject Reduction*) $\forall P. \Gamma \vdash P \wedge P \longrightarrow P' \Rightarrow \Gamma \vdash P'$

Proof. By Lemma A.1.17 (Type Substitution), Lemma A.1.16 (Type Strengthening), Lemma A.1.18 (Type Structure) and by induction on Definition A.1.3 of reduction in CA, where $\Gamma \vdash P, Q$ is short for $\Gamma \vdash P \wedge \Gamma \vdash Q$:

$$\begin{array}{ll}
P \longrightarrow P' \wedge \Gamma \vdash P \mid Q & \xRightarrow{(A.54)} \Gamma \vdash P \wedge \Gamma \vdash Q \xRightarrow{\text{IH}} \Gamma \vdash P' \xRightarrow{(A.54)} \Gamma \vdash P' \mid Q \\
P \longrightarrow P' \wedge \Gamma \vdash \nu n : \tau P & \xRightarrow{(A.55)} \Gamma, n : \tau \vdash P \xRightarrow{\text{IH}} \Gamma, n : \tau \vdash P' \xRightarrow{(A.55)} \Gamma \vdash \nu n P' \\
Q \equiv P \longrightarrow P' \equiv Q' \wedge \Gamma \vdash Q & \xRightarrow{A.1.18} \Gamma \vdash P \xRightarrow{\text{IH}} \Gamma \vdash P' \xRightarrow{A.1.18} \Gamma \vdash Q' \\
P \longrightarrow P' \wedge \Gamma \vdash a \boxed{P} & \xRightarrow{(A.56)} \Gamma, a : \text{amb} \vdash P \xRightarrow{\text{IH}} \Gamma, a : \text{amb} \vdash P' \xRightarrow{(A.56)} \Gamma \vdash a \boxed{P'}
\end{array}$$

$$\begin{array}{ll}
\Gamma \vdash a \boxed{b \cdot x \langle v \rangle . P \mid P'} \mid b \boxed{x^\uparrow(u) . Q \mid Q'} & \xRightarrow{A.1.6} \Gamma(x) = \langle \tau \rangle \wedge \Gamma(v) = \tau \wedge \Gamma(a) = \text{amb} \wedge \Gamma(b) = \text{amb} \\
& \wedge \Gamma \vdash P, P', Q' \wedge \Gamma, u : \tau \vdash Q \\
& \xRightarrow{A.1.17} \Gamma, u : \tau \vdash Q_{\{v/u\}} \\
& \xRightarrow{A.1.16} \Gamma \vdash Q_{\{v/u\}} \\
& \xRightarrow{(A.56; A.54)} \Gamma \vdash a \boxed{P \mid P'} \mid b \boxed{Q_{\{v/u\}} \mid Q'} \\
\Gamma \vdash a \boxed{x^\uparrow(v) . P \mid P'} \mid x(u) . Q & \xRightarrow{A.1.6} \Gamma(x) = \langle \tau \rangle \wedge \Gamma(v) = \tau \wedge \Gamma(a) = \text{amb} \wedge \\
& \Gamma \vdash P, P' \wedge \Gamma, u : \tau \vdash Q \\
& \xRightarrow{A.1.17} \Gamma, u : \tau \vdash Q_{\{v/u\}} \\
& \xRightarrow{A.1.16} \Gamma \vdash Q_{\{v/u\}} \\
& \xRightarrow{(A.56; A.54)} \Gamma \vdash a \boxed{P \mid P'} \mid Q_{\{v/u\}} \\
\Gamma \vdash a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\overline{\text{in}} x . Q \mid Q'} & \xRightarrow{A.1.6} \Gamma(x) = \langle \text{mv} \rangle \wedge \Gamma(a) = \text{amb} \wedge \Gamma(b) = \text{amb} \wedge \\
& \Gamma \vdash P, P', Q, Q' \\
& \xRightarrow{(A.56; A.54)} \Gamma \vdash b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \\
\Gamma \vdash b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \overline{\text{out}} x . Q \mid Q'} & \xRightarrow{A.1.6} \Gamma(x) = \langle \text{mv} \rangle \wedge \Gamma(a) = \text{amb} \wedge \Gamma(b) = \text{amb} \wedge \\
& \Gamma \vdash P, P', Q, Q' \\
& \xRightarrow{(A.56; A.54)} \Gamma \vdash b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'}
\end{array}$$

□

A.1.3 Authenticity

Lemmas A.1.20, A.1.21 and A.1.22 ensure that the calculus CA^- is closed under substitution, structural congruence and reduction, respectively. Lemma A.1.23 (Active Ambients) ensures that the set of free ambients denotes all the active ambients of a process. If a process P contains an active ambient a then a is in the set of free ambients of P . Lemma A.1.24 (Free Ambient Structure) ensures that the set of free ambients of a process is closed under structural congruence. If a process P is structurally congruent to a process Q then the set of free ambients of P is equal to the set of free ambients of Q . Lemma A.1.25 (Free Ambient Reduction) ensures that the set of free ambients cannot be augmented by a reduction. If a process P can reduce to P' and z is a free ambient in P' then z must also be a free ambient in P . Lemma A.1.26 (Free Ambient Transitivity) ensures that if an ambient is not free in a process then it will never be free in subsequent processes. If a is not a free ambient in P and P can reduce to P' in zero or more steps then a is not a free ambient in P' . Finally, Theorem A.1.27 (Ambient Authenticity) ensures that if a name is not in the set of free ambients of a process then it can never be used to create an ambient. If a is not in the set of free ambients of P and P can reduce to P' in zero or more steps then P cannot contain an ambient named a .

Lemma A.1.20. (*Substitution Safety CA-*) $\forall \sigma, P.P \in \text{CA}^- \Rightarrow P_\sigma \in \text{CA}^-$

Proof. By induction on Definition A.1.4 of substitution in CA. The proof is analogous to that of Lemma A.1.12 (Substitution Safety). \square

Lemma A.1.21. (*Structural Safety CA-*) $\forall P.P \in \text{CA}^- \wedge P \equiv P' \Rightarrow P' \in \text{CA}^-$

Proof. By induction on Definition A.1.2 of structural congruence in CA. The proof is analogous to that of Lemma A.1.13 (Structural Safety). \square

Lemma A.1.22. (*Reduction Safety CA-*) $\forall P.P \in \text{CA}^- \wedge P \longrightarrow P' \Rightarrow P' \in \text{CA}^-$

Proof. By Lemma A.1.20, Lemma A.1.21 and by induction on Definition A.1.3 of reduction in CA. The proof is analogous to that of Theorem A.1.14 (Reduction Safety). \square

Lemma A.1.23. (*Active Ambients*) $\forall P.P \in \text{CA}^- \wedge P \downarrow_a \Rightarrow a \in \text{fa}(P)$

Proof. By induction on Definition A.1.10 of active ambients in CA:

$$\begin{aligned}
a[\boxed{P}] \downarrow_a &\Rightarrow a \in (\text{fa}(P) \cup \{a\}) \xRightarrow{(\text{A.69})} a \in \text{fa}(a[\boxed{P}]) \\
P \downarrow_a \wedge (P \mid Q) \downarrow_a &\xRightarrow{\text{IH}} a \in \text{fa}(P) \Rightarrow a \in (\text{fa}(P) \cup \text{fa}(Q)) \xRightarrow{(\text{A.67})} a \in \text{fa}(P \mid Q) \\
P \downarrow_a \wedge (Q \mid P) \downarrow_a &\xRightarrow{\text{IH}} a \in \text{fa}(P) \Rightarrow a \in (\text{fa}(Q) \cup \text{fa}(P)) \xRightarrow{(\text{A.67})} a \in \text{fa}(Q \mid P) \\
a \neq m \wedge P \downarrow_a \wedge (\nu m P) \downarrow_a &\xRightarrow{\text{IH}} a \in \text{fa}(P) \Rightarrow a \in (\text{fa}(P) \setminus \{m\}) \xRightarrow{(\text{A.68})} a \in \text{fa}(\nu m P) \\
P \downarrow_a \wedge b[\boxed{P}] \downarrow_a &\xRightarrow{\text{IH}} a \in \text{fa}(P) \Rightarrow a \in (\text{fa}(P) \cup \{b\}) \xRightarrow{(\text{A.69})} a \in \text{fa}(b[\boxed{P}])
\end{aligned}$$

\square

Lemma A.1.24. (*Free Ambient Structure*) $\forall P.P \in \text{CA}^- \wedge P \equiv Q \Rightarrow \text{fa}(P) = \text{fa}(Q)$

Proof. By induction on Definition A.1.2 of structural congruence in CA:

$$\begin{aligned}
\mathbf{0} \mid P \in \text{CA} &\Rightarrow \text{fa}(P) \cup \emptyset = \text{fa}(P) \\
&\xRightarrow{(\text{A.67}, \text{A.66})} \text{fa}(\mathbf{0} \mid P) = \text{fa}(P) \\
P \mid Q \in \text{CA} &\Rightarrow \text{fa}(P) \cup \text{fa}(Q) = \text{fa}(Q) \cup \text{fa}(P) \\
&\xRightarrow{(\text{A.67})} \text{fa}(P \mid Q) = \text{fa}(Q \mid P) \\
P \mid (Q \mid R) \in \text{CA} &\Rightarrow \text{fa}(P) \cup (\text{fa}(Q) \cup \text{fa}(R)) = (\text{fa}(P) \cup \text{fa}(Q)) \cup \text{fa}(R) \\
&\xRightarrow{(\text{A.67})} \text{fa}(P \mid (Q \mid R)) = \text{fa}((P \mid Q) \mid R) \\
\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \wedge !\alpha.P \in \text{CA} &\Rightarrow \text{fa}(P) \setminus \text{bn}(\alpha) = (\text{fa}(P) \cup (\text{fa}(P) \setminus \text{bn}(\alpha))) \setminus \text{bn}(\alpha) \\
&\xRightarrow{(\text{A.70}, \text{A.71}, \text{A.67})} \text{fa}(!\alpha.P) = \text{fa}(\alpha.(P \mid !\alpha.P)) \\
\nu n \mathbf{0} \in \text{CA} &\Rightarrow \emptyset \setminus \{n\} = \emptyset \\
&\xRightarrow{(\text{A.68}, \text{A.66})} \text{fa}(\nu n \mathbf{0}) = \text{fa}(\mathbf{0}) \\
\nu n \nu m P \in \text{CA} &\Rightarrow (\text{fa}(P) \setminus \{m\}) \setminus \{n\} = (\text{fa}(P) \setminus \{n\}) \setminus \{m\} \\
&\xRightarrow{(\text{A.68})} \text{fa}(\nu n \nu m P) = \text{fa}(\nu m \nu n P) \\
n \notin \text{fn}(Q) \wedge (\nu n P) \mid Q \in \text{CA} &\Rightarrow (\text{fa}(P) \setminus \{n\}) \cup \text{fa}(Q) = (\text{fa}(P) \cup \text{fa}(Q)) \setminus \{n\} \\
&\xRightarrow{(\text{A.68}, \text{A.67})} \text{fa}((\nu n P) \mid Q) = \text{fa}(\nu n (P \mid Q)) \\
a \neq n \wedge a[\boxed{\nu n P}] \in \text{CA} &\Rightarrow \{a\} \cup (\text{fa}(P) \setminus \{n\}) = (\{a\} \cup \text{fa}(P)) \setminus \{n\} \\
&\xRightarrow{(\text{A.69}, \text{A.68})} \text{fa}(a[\boxed{\nu n P}]) = \text{fa}(\nu n a[\boxed{P}])
\end{aligned}$$

\square

Lemma A.1.25. (*Free Ambient Reduction*) $\forall P.P \in \text{CA}^- \wedge z \in \text{fa}(P') \wedge P \longrightarrow P' \Rightarrow z \in \text{fa}(P)$

Proof. By Lemma A.1.24 (Free Ambient Structure) and by induction on Definition A.1.3 of reduction in CA:

$$\begin{aligned}
P \longrightarrow P' \wedge z \in \text{fa}(P' \mid Q) &\stackrel{(\text{A.67})}{\Rightarrow} z \in \text{fa}(P') \cup \text{fa}(Q) \stackrel{\text{IH}}{\Rightarrow} z \in \text{fa}(P) \cup \text{fa}(Q) \stackrel{(\text{A.67})}{\Rightarrow} z \in \text{fa}(P \mid Q) \\
P \longrightarrow P' \wedge z \in \text{fa}(\nu n P') &\stackrel{(\text{A.68})}{\Rightarrow} z \in \text{fa}(P') \setminus \{n\} \stackrel{\text{IH}}{\Rightarrow} z \in \text{fa}(P) \setminus \{n\} \stackrel{(\text{A.68})}{\Rightarrow} z \in \text{fa}(\nu n P) \\
Q \equiv P \longrightarrow P' \equiv Q' \wedge z \in \text{fa}(Q') &\stackrel{\text{A.1.24}}{\Rightarrow} z \in \text{fa}(P') \stackrel{\text{IH}}{\Rightarrow} z \in \text{fa}(P) \stackrel{\text{A.1.24}}{\Rightarrow} z \in \text{fa}(Q) \\
P \longrightarrow P' \wedge z \in \text{fa}(a \boxed{P'}) &\stackrel{(\text{A.69})}{\Rightarrow} z \in \text{fa}(P') \cup \{a\} \stackrel{\text{IH}}{\Rightarrow} z \in \text{fa}(P) \cup \{a\} \stackrel{(\text{A.69})}{\Rightarrow} z \in \text{fa}(a \boxed{P})
\end{aligned}$$

$$\begin{aligned}
&\{u\} \cap \text{fa}(Q) = \emptyset \wedge \\
z \in \text{fa}(a \boxed{P \mid P'} \mid b \boxed{Q_{\{v/u\}} \mid Q'}) &\stackrel{(\text{A.69}, \text{A.67})}{\Rightarrow} z \in \{a\} \cup \{b\} \cup \text{fa}(P, P', Q') \cup \text{fa}(Q_{\{v/u\}}) \\
&\Rightarrow z \in \{a\} \cup \{b\} \cup \text{fa}(P, P', Q') \cup (\text{fa}(Q) \setminus \{u\}) \\
&\stackrel{(\text{A.69}, \text{A.67}, \text{A.70})}{\Rightarrow} z \in \text{fa}(a \boxed{b \cdot x \langle v \rangle . P \mid P'} \mid b \boxed{x^\dagger(u) . Q \mid Q'}) \\
&\{u\} \cap \text{fa}(Q) = \emptyset \wedge \\
z \in \text{fa}(a \boxed{P \mid P'} \mid Q_{\{v/u\}}) &\stackrel{(\text{A.69}, \text{A.67})}{\Rightarrow} z \in \{a\} \cup \text{fa}(P, P') \cup \text{fa}(Q_{\{v/u\}}) \\
&\Rightarrow z \in \{a\} \cup \text{fa}(P, P') \cup (\text{fa}(Q) \setminus \{u\}) \\
&\stackrel{(\text{A.69}, \text{A.67}, \text{A.70})}{\Rightarrow} z \in \text{fa}(a \boxed{x^\dagger \langle v \rangle . P \mid P'} \mid x(u) . Q) \\
z \in \text{fa}(b \boxed{Q \mid Q' \mid a \boxed{P \mid P'}}) &\stackrel{(\text{A.69}, \text{A.67})}{\Rightarrow} z \in \{a\} \cup \{b\} \cup \text{fa}(P, P', Q, Q') \\
&\stackrel{(\text{A.69}, \text{A.67}, \text{A.70})}{\Rightarrow} z \in \text{fa}(a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\text{in } x . Q \mid Q'}) \\
z \in \text{fa}(b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'}) &\stackrel{(\text{A.69}, \text{A.67})}{\Rightarrow} z \in \{a\} \cup \{b\} \cup \text{fa}(P, P', Q, Q') \\
&\stackrel{(\text{A.69}, \text{A.67}, \text{A.70})}{\Rightarrow} z \in \text{fa}(b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \overline{\text{out } x} . Q \mid Q'})
\end{aligned}$$

□

Lemma A.1.26. (*Free Ambient Transitivity*) $\forall P.P \in \text{CA}^- \wedge a \notin \text{fa}(P) \wedge P \longrightarrow^* P' \Rightarrow a \notin \text{fa}(P')$

Proof. By Lemma A.1.24 (Free Ambient Structure), Lemma A.1.25 (Free Ambient Reduction) and by induction on the transitive closure of reduction in CA:

$$\begin{aligned}
a \notin \text{fa}(P) \wedge P \equiv P' &\stackrel{\text{A.1.24}}{\Rightarrow} \text{fa}(P) = \text{fa}(P') \Rightarrow a \notin \text{fa}(P') \\
a \notin \text{fa}(P) \wedge P \longrightarrow P' \wedge P' \longrightarrow^* P'' &\stackrel{\text{A.1.25}}{\Rightarrow} a \notin \text{fa}(P') \stackrel{\text{IH}}{\Rightarrow} a \notin \text{fa}(P'')
\end{aligned}$$

□

Theorem A.1.27. (*Ambient Authenticity*) $\forall P.P \in \text{CA}^- \wedge a \notin \text{fa}(P) \wedge P \longrightarrow^* P' \Rightarrow P' \not\ll_a$

Proof. By Lemma A.1.23 (Active Ambients) and by Lemma A.1.26 (Free Ambient Transitivity):

$$P \in \text{CA}^- \wedge a \notin \text{fa}(P) \wedge P \longrightarrow^* P' \stackrel{\text{A.1.26}}{\Rightarrow} a \notin \text{fa}(P') \stackrel{\text{A.1.23}}{\Rightarrow} P' \not\ll_a$$

□

A.2 Correctness of the Channel Ambient Machine

$V ::= \nu n V$	Restriction	(A.83)	$z ::= \underline{z}$	Blocked	(A.88)
$\mid A$	List	(A.84)	$\mid z$	Unblocked	(A.89)
$A, B, C ::= []$	Empty	(A.85)			
$\mid \alpha.P :: C$	Action	(A.86)			
$\mid q[A] :: C$	Ambient	(A.87)			

Definition A.2.1. Syntax of CAM

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

Definition A.2.2. Encoding CA to CAM

$$n \notin \text{fn}(P) \Rightarrow P : (\nu n V) \triangleq \nu n (P : V) \quad (\text{A.90})$$

$$0 : A \triangleq A \quad (\text{A.91})$$

$$(P \mid Q) : A \triangleq P : Q : A \quad (\text{A.92})$$

$$n \notin \text{fn}(P : A) \Rightarrow (\nu m P) : A \triangleq \nu n (P_{\{n/m\}} : A) \quad (\text{A.93})$$

$$\text{fn}(\alpha) \not\subseteq \text{bn}(\alpha) \Rightarrow !\alpha.P : A \triangleq \alpha.(P \mid !\alpha.P) : A \quad (\text{A.94})$$

$$a[P] : A \triangleq a[P : []] : A \quad (\text{A.95})$$

$$\alpha.P : A \triangleq \alpha.P :: A \quad (\text{A.96})$$

$$n \notin \text{fn}(a, V) \Rightarrow a[\nu n U] : V \triangleq \nu n (a[U] : V) \quad (\text{A.97})$$

$$n \notin \text{fn}(a, A) \Rightarrow a[A] : \nu n V \triangleq \nu n (a[A] : V) \quad (\text{A.98})$$

$$q[A] : C \triangleq q[A] :: C \quad (\text{A.99})$$

Definition A.2.3. Construction in CAM

$$A @ \alpha.P :: A' \succ \alpha.P :: A @ A' \quad (\text{A.100})$$

$$A @ b[B] :: A' \succ b[B] :: A @ A' \quad (\text{A.101})$$

$$A \succ A' \Rightarrow q[A] :: C \succ q[A'] :: C \quad (\text{A.102})$$

Definition A.2.4. Selection in CAM

$$[] \triangleq [] \quad (\text{A.103})$$

$$[\alpha.P :: C] \triangleq \alpha.P :: [C] \quad (\text{A.104})$$

$$[q[A] :: C] \triangleq q[A] :: [C] \quad (\text{A.105})$$

Definition A.2.5. Unblocking in CAM

$$C \succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{P : A} : b \boxed{Q_{\{v/u\}} : B} : C' \quad (\text{A.106})$$

$$C \not\succ b \boxed{x^\dagger(u).Q :: B} :: C' \Rightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{b \cdot x \langle v \rangle . P :: A} :: C \quad (\text{A.107})$$

$$C \succ a \boxed{b \cdot x \langle v \rangle . P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow a \boxed{P : A} : b \boxed{Q_{\{v/u\}} : B} : C' \quad (\text{A.108})$$

$$C \not\succ a \boxed{b \cdot x \langle v \rangle . P :: A} :: C' \Rightarrow b \boxed{x^\dagger(u).Q :: B} :: C \longrightarrow b \boxed{x^\dagger(u).Q :: B} :: C \quad (\text{A.109})$$

$$C \succ \underline{x(u).Q} :: C' \Rightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{P : A} : Q_{\{v/u\}} : C' \quad (\text{A.110})$$

$$C \not\succ \underline{x(u).Q} :: C' \Rightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \longrightarrow a \boxed{x^\dagger \langle v \rangle . P :: A} :: C \quad (\text{A.111})$$

$$C \succ a \boxed{x^\dagger \langle v \rangle . P :: A} :: C' \Rightarrow \underline{x(u).Q} :: C \longrightarrow a \boxed{P : A} : Q_{\{v/u\}} : C' \quad (\text{A.112})$$

$$C \not\succ a \boxed{x^\dagger \langle v \rangle . P :: A} :: C' \Rightarrow \underline{x(u).Q} :: C \longrightarrow \underline{x(u).Q} :: C \quad (\text{A.113})$$

$$C \succ b \boxed{\underline{\text{in } x}.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow b \boxed{Q : a \boxed{P : [A]} : B} : C' \quad (\text{A.114})$$

$$C \not\succ b \boxed{\underline{\text{in } x}.Q :: B} :: C' \Rightarrow a \boxed{\text{in } b \cdot x.P :: A} :: C \longrightarrow a \boxed{\underline{\text{in } b \cdot x}.P :: A} :: C \quad (\text{A.115})$$

$$C \succ a \boxed{\underline{\text{in } b \cdot x}.P :: A} :: C' \Rightarrow b \boxed{\underline{\text{in } x}.Q :: B} :: C \longrightarrow b \boxed{Q : a \boxed{P : [A]} : B} : C' \quad (\text{A.116})$$

$$C \not\succ a \boxed{\underline{\text{in } b \cdot x}.P :: A} :: C' \Rightarrow b \boxed{\underline{\text{in } x}.Q :: B} :: C \longrightarrow b \boxed{\underline{\text{in } x}.Q :: B} :: C \quad (\text{A.117})$$

$$B \succ \overline{\text{out } x}.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{Q : B'} : a \boxed{P : [A]} : C \quad (\text{A.118})$$

$$B \not\succ \overline{\text{out } x}.Q :: B' \Rightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \longrightarrow b \boxed{a \boxed{\text{out } x.P :: A} :: B} :: C \quad (\text{A.119})$$

$$B \succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\overline{\text{out } x}.Q :: B} :: C \longrightarrow b \boxed{Q : B'} : a \boxed{P : [A]} : C \quad (\text{A.120})$$

$$B \not\succ a \boxed{\text{out } x.P :: A} :: B' \Rightarrow b \boxed{\overline{\text{out } x}.Q :: B} :: C \longrightarrow b \boxed{\overline{\text{out } x}.Q :: B} :: C \quad (\text{A.121})$$

$$V \longrightarrow V' \Rightarrow \nu n V \longrightarrow \nu n V' \quad (\text{A.122})$$

$$B \succ A \wedge A \longrightarrow V' \Rightarrow B \longrightarrow V' \quad (\text{A.123})$$

$$A \longrightarrow V' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{V'} : C \quad (\text{A.124})$$

$$A \not\succ \alpha.P :: A' \wedge A \not\succ b \boxed{B} :: A' \Rightarrow a \boxed{A} :: C \longrightarrow a \boxed{A} :: C \quad (\text{A.125})$$

Definition A.2.6. Reduction in CAM

$$\llbracket \nu n V \rrbracket \triangleq \nu n \llbracket V \rrbracket \quad (\text{A.126})$$

$$\llbracket [] \rrbracket \triangleq \mathbf{0} \quad (\text{A.127})$$

$$\llbracket \alpha.P :: C \rrbracket \triangleq \alpha.P \mid \llbracket C \rrbracket \quad (\text{A.128})$$

$$\llbracket a[A] :: C \rrbracket \triangleq a[\llbracket A \rrbracket] \mid \llbracket C \rrbracket \quad (\text{A.129})$$

Definition A.2.7. Decoding CAM to CA

$$n \notin \text{fn}(V) \Rightarrow \nu n V \equiv V \quad (\text{A.130})$$

$$\nu n \nu m V \equiv \nu m \nu n V \quad (\text{A.131})$$

$$A @ B @ C \equiv B @ A @ C \quad (\text{A.132})$$

$$V \equiv V' \Rightarrow \nu n V \equiv \nu n V' \quad (\text{A.133})$$

$$A \equiv A' \Rightarrow a[A] :: C \equiv a[A'] :: C \quad (\text{A.134})$$

$$P \equiv P' \Rightarrow \alpha.P :: C \equiv \alpha.P' :: C \quad (\text{A.135})$$

$$C \equiv C' \Rightarrow A @ C \equiv A @ C' \quad (\text{A.136})$$

$$V \equiv V \quad (\text{A.137})$$

$$V \equiv U \Rightarrow U \equiv V \quad (\text{A.138})$$

$$V \equiv V' \wedge V' \equiv U \Rightarrow V \equiv U \quad (\text{A.139})$$

Definition A.2.8. Structural Congruence in CAM

$$V ::= \nu n V \quad \text{Restriction} \quad (\text{A.140})$$

$$\mid A \quad \text{List} \quad (\text{A.141})$$

$$A, B, C ::= [] \quad \text{Empty} \quad (\text{A.142})$$

$$\mid \alpha.P :: C \quad \text{Action, } \alpha.P = \underline{x(m)}.P \Rightarrow C \not\vdash a[\underline{x^\uparrow\langle n \rangle}.P :: A] :: C' \quad (\text{A.143})$$

$$\mid a[A] :: C \quad \text{Ambient, } a = \underline{a} \Rightarrow (A \not\vdash \alpha.P :: A' \wedge A \not\vdash b[\underline{B}] :: A') \quad (\text{A.144})$$

$$A \succ \overline{\text{out}} x.Q :: A' \Rightarrow A' \not\vdash b[\underline{\text{out } x.P :: B}] :: A''$$

$$A \succ b \cdot \underline{x\langle n \rangle}.P :: A' \Rightarrow C \not\vdash b[\underline{x^\uparrow\langle m \rangle}.Q :: B] :: C'$$

$$A \succ \underline{x^\uparrow\langle m \rangle}.P :: A' \Rightarrow C \not\vdash b[\underline{a \cdot \underline{x\langle n \rangle}.Q :: B}] :: C'$$

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

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

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

Definition A.2.9. Syntax of Deadlock-Free Terms CAM'

The Channel Ambient Machine is summarised in Definitions A.2.1 - A.2.9. The remainder of this appendix outlines the proof of correctness of the Channel Ambient Machine (CAM) with respect to the Channel Ambient calculus (CA), as described in Chapter 3. Note that for proofs by induction on the definition of reduction in CAM, the cases for external input (A.108), internal input (A.112), accept (A.116) and release (A.120) are omitted, since they are analogous to the cases for sibling output (A.106), parent output (A.110), enter (A.114) and leave (A.118), respectively. The cases for blocking (A.107), (A.109), (A.111), (A.113), (A.115), (A.117), (A.119), (A.121), (A.125) are also omitted, since they are straightforward. For proofs by induction on the definition of structural congruence in CAM, the cases for the symmetric rules are omitted, since they are analogous to the cases for rules (A.130)-(A.136). The cases for the reflexive and transitive rules are also omitted, since they are straightforward. For proofs by induction on the definition of structural congruence in CA, the cases for the symmetric rules are omitted, since they are analogous to the cases for rules (A.15)-(A.22). The cases for the reflexive, transitive and congruence rules (A.26)-(A.27) are also omitted, since they are straightforward.

A.2.1 Safety

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. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove the safety of the machine it is necessary to prove that construction, selection, unblocking and reduction are safe.

Lemma A.2.10 (Construction Safety) ensures that the result of a construction is always a valid machine term. The lemma states that if a process P is added to a term V then the result is a valid machine term. Similarly, if an ambient a with a term U is added to a term V then the result is a valid machine term.

Lemma A.2.11 (Selection Safety) ensures that the result of a selection is always a valid machine term. The lemma states that if the machine selects a term A' from a term A then A' is a valid machine term.

Lemma A.2.12 (Unblocking Safety) ensures that the result of an unblocking is always a valid machine term. The lemma states that if the machine unblocks a term V then the result is a valid machine term.

Finally, Theorem A.2.13 (Reduction Safety) ensures that the result of a reduction is always a valid machine term. The theorem states that if the machine reduces a term V to V' then V' is a valid machine term.

Lemma A.2.10. (*Construction Safety*) $\forall P. \forall V. P \in \text{CA} \wedge V \in \text{CAM} \Rightarrow P : V \in \text{CAM}$
 $\forall U. \forall V. U, V \in \text{CAM} \Rightarrow a[\overline{U}] : V \in \text{CAM}$

Proof. By induction on Definition A.2.3 of construction in CAM:

$$\begin{array}{lll}
P \in \text{CA} \wedge \nu n V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{\text{IH}} P : V \in \text{CAM} \xRightarrow{(A.83)} & \nu n (P : V) \in \text{CAM} \\
& \xRightarrow{(A.90)} & P : (\nu n V) \in \text{CAM} \\
\mathbf{0} \in \text{CA} \wedge A \in \text{CAM} & \Rightarrow A \in \text{CAM} \xRightarrow{(A.91)} & \mathbf{0} : A \in \text{CAM} \\
(P \mid Q) \in \text{CA} \wedge A \in \text{CAM} & \xRightarrow{(A.2)} P, Q \in \text{CA} \xRightarrow{\text{IH}} Q : A \in \text{CAM} \xRightarrow{\text{IH}} & P : Q : A \in \text{CAM} \\
& \xRightarrow{(A.92)} & (P \mid Q) : A \in \text{CAM} \\
(\nu m P) \in \text{CA} \wedge A \in \text{CAM} & \xRightarrow{(A.3)} P_{\{n/m\}} \in \text{CA} \xRightarrow{\text{IH}} P_{\{n/m\}} : A \in \text{CAM} \xRightarrow{(A.83)} & \nu n (P_{\{n/m\}} : A) \in \text{CAM} \\
& \xRightarrow{(A.93)} & (\nu m P) : A \in \text{CAM} \\
!\alpha. P \in \text{CA} \wedge A \in \text{CAM} & \xRightarrow{(A.6)} \alpha.(P \mid !\alpha. P) \in \text{CA} \xRightarrow{\text{IH}} & \alpha.(P \mid !\alpha. P) : A \in \text{CAM} \\
& \xRightarrow{(A.94)} & !\alpha. P :: A \in \text{CAM} \\
a[\overline{P}] \in \text{CA} \wedge A \in \text{CAM} & \xRightarrow{(A.4)} P \in \text{CA} \xRightarrow{\text{IH}} P : \square \in \text{CAM} \xRightarrow{\text{IH}} & a[\overline{P : \square}] : A \in \text{CAM} \\
& \xRightarrow{(A.95)} & a[\overline{P}] : A \in \text{CAM} \\
\alpha. P \in \text{CA} \wedge A \in \text{CAM} & \xRightarrow{(A.86)} \alpha. P :: A \in \text{CAM} \xRightarrow{(A.96)} & \alpha. P : A \in \text{CAM} \\
A \in \text{CAM} \wedge \nu n V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{\text{IH}} a[\overline{A}] : V \in \text{CAM} \xRightarrow{(A.83)} & \nu n (a[\overline{A}] : V) \in \text{CAM} \\
& \xRightarrow{(A.98)} & a[\overline{\nu n U}] : V \in \text{CAM} \\
\nu n U \in \text{CAM} \wedge V \in \text{CAM} & \xRightarrow{(A.83)} U \in \text{CAM} \xRightarrow{\text{IH}} a[\overline{U}] : V \in \text{CAM} \xRightarrow{(A.83)} & \nu n (a[\overline{U}] : V) \in \text{CAM} \\
& \xRightarrow{(A.97)} & a[\overline{A}] : \nu n V \in \text{CAM} \\
A \in \text{CAM} \wedge C \in \text{CAM} & \xRightarrow{(A.87)} a[\overline{A}] :: C \in \text{CAM} \xRightarrow{(A.99)} & a[\overline{A}] : C \in \text{CAM}
\end{array}$$

□

Lemma A.2.11. (*Selection Safety*) $\forall A. A \in \text{CAM} \wedge A \succ A' \Rightarrow A' \in \text{CAM}$

Proof. By induction on Definition A.2.4 of selection in CAM:

$$\begin{array}{lll}
A @_{\alpha}. P :: A' \in \text{CAM} & \Rightarrow A @ A' \in \text{CAM} \xRightarrow{(A.86)} & \alpha. P :: A @ A' \in \text{CAM} \\
A @ b[\overline{B}] :: A' \in \text{CAM} & \Rightarrow A @ A' \in \text{CAM} \xRightarrow{(A.87)} & b[\overline{B}] :: A @ A' \in \text{CAM} \\
A \succ A' \wedge a[\overline{A}] :: C \in \text{CAM} & \xRightarrow{\text{IH}} A' \in \text{CAM} \xRightarrow{(A.87)} C \in \text{CAM} \xRightarrow{(A.87)} & a[\overline{A'}] :: C \in \text{CAM}
\end{array}$$

□

Lemma A.2.12. (*Unblocking Safety*) $\forall V. V \in \text{CAM} \Rightarrow \lfloor V \rfloor \in \text{CAM}$

Proof. By induction on Definition A.2.5 of unblocking in CAM:

$$\begin{array}{lll}
 \boxed{} \in \text{CAM} & \xRightarrow{(A.103)} & \lfloor \boxed{} \rfloor \in \text{CAM} \\
 \alpha.P :: C \in \text{CAM} & \xRightarrow{(A.86)} C \in \text{CAM} \xRightarrow{\text{IH}} \lfloor C \rfloor \in \text{CAM} \xRightarrow{(A.86)} & \alpha.P :: \lfloor C \rfloor \in \text{CAM} \\
 & \xRightarrow{(A.104)} & \lfloor \alpha.P :: C \rfloor \in \text{CAM} \\
 a\boxed{A} :: C \in \text{CAM} & \xRightarrow{(A.87)} A, C \in \text{CAM} \xRightarrow{\text{IH}} \lfloor C \rfloor \in \text{CAM} \xRightarrow{(A.87)} & a\boxed{A} :: \lfloor C \rfloor \in \text{CAM} \\
 & \xRightarrow{(A.105)} & \lfloor a\boxed{A} :: C \rfloor \in \text{CAM}
 \end{array}$$

□

Theorem A.2.13. (*Reduction Safety*) $\forall V. V \in \text{CAM} \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}$

Proof. By Lemma A.2.10 (Construction Safety), Lemma A.2.11 (Selection Safety), Lemma A.2.12 (Unblocking Safety) and by induction on Definition A.2.6 of reduction in CAM:

$$\begin{array}{lll}
 V \longrightarrow V' \wedge \nu n V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{\text{IH}} V' \in \text{CAM} \xRightarrow{(A.83)} & \nu n V' \in \text{CAM} \\
 B \succ A \wedge A \longrightarrow V' \wedge B \in \text{CAM} & \xRightarrow{A.2.11} A \in \text{CAM} \xRightarrow{\text{IH}} & V' \in \text{CAM} \\
 A \longrightarrow V' \wedge a\boxed{A} :: C \in \text{CAM} & \xRightarrow{(A.87)} A, C \in \text{CAM} \xRightarrow{\text{IH}} V' \in \text{CAM} \xRightarrow{A.2.10} & a\boxed{V'} :: C \in \text{CAM}
 \end{array}$$

$$\begin{array}{lll}
 C \succ b\boxed{x^\uparrow(m).Q :: B} :: C' \wedge a\boxed{b \cdot x\langle n \rangle . P :: A} :: C \in \text{CAM} & \xRightarrow{A.2.1} & A, C \in \text{CAM} \wedge P \in \text{CA} \\
 & \xRightarrow{A.2.11} & B, C' \in \text{CAM} \wedge Q \in \text{CA} \\
 & \xRightarrow{A.2.10} & a\boxed{P:A} : b\boxed{Q_{\{n/m\}}:B} :: C' \in \text{CAM} \\
 C \succ x(m).Q :: C' \wedge a\boxed{x^\uparrow\langle n \rangle . P :: A} :: C \in \text{CAM} & \xRightarrow{A.2.1} & A, C \in \text{CAM} \wedge P \in \text{CA} \\
 & \xRightarrow{A.2.11} & C' \in \text{CAM} \wedge Q \in \text{CA} \\
 & \xRightarrow{A.2.10} & a\boxed{P:A} : Q_{\{n/m\}} : C' \in \text{CAM} \\
 C \succ b\boxed{\text{in } x.Q :: B} :: C' \wedge a\boxed{\text{in } b \cdot x.P :: A} :: C \in \text{CAM} & \xRightarrow{A.2.1} & A, C \in \text{CAM} \wedge P \in \text{CA} \\
 & \xRightarrow{A.2.11} & B, C' \in \text{CAM} \wedge Q \in \text{CA} \\
 & \xRightarrow{A.2.12} & \lfloor A \rfloor \in \text{CAM} \\
 & \xRightarrow{A.2.10} & b\boxed{Q : a\boxed{P:\lfloor A \rfloor} : B} :: C' \in \text{CAM} \\
 B \succ \text{out } x.Q :: B' \wedge b\boxed{a\boxed{\text{out } x.P :: A} :: B} :: C \in \text{CAM} & \xRightarrow{A.2.1} & A, B, C \in \text{CAM} \wedge P \in \text{CA} \\
 & \xRightarrow{A.2.11} & B' \in \text{CAM} \wedge Q \in \text{CA} \\
 & \xRightarrow{A.2.12} & \lfloor A \rfloor \in \text{CAM} \\
 & \xRightarrow{A.2.10} & b\boxed{Q:B'} : a\boxed{P:\lfloor A \rfloor} :: C \in \text{CAM}
 \end{array}$$

□

A.2.2 Soundness

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.

Lemma A.2.14 (Decoding Soundness) ensures that decoding always produces a valid calculus process.

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. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove the soundness of the machine it is necessary to prove that construction, selection, unblocking and reduction are sound.

Lemma A.2.15 (Selection Soundness) ensures that selection in the machine corresponds to structural congruence in the calculus. The lemma states that if the machine selects a term A' from a term A then the decoding of A' is structurally congruent to the decoding of A .

Lemma A.2.16 (Unblocking Soundness) ensures that unblocking in the machine corresponds to equality in the calculus. The lemma states that if the machine unblocks a term V then the decoding of the result is equal to the decoding of V .

Lemma A.2.17 (Construction Soundness) ensures that construction in the machine corresponds to parallel composition in the calculus, up to structural congruence. The lemma states that if a process P is added to a term V then the decoding of the resulting term is structurally congruent to P in parallel with the decoding of V . Similarly, if an ambient a with a term U is added to a term V then the decoding of the resulting term is structurally congruent to ambient a containing the decoding of U , in parallel with the decoding of V .

Theorem A.2.18 (Reduction Soundness) ensures that reduction in the machine corresponds to at most one reduction in the calculus. The theorem 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.

Lemma A.2.14. (*Decoding Soundness*) $\forall V. V \in \text{CAM} \Rightarrow \llbracket V \rrbracket \in \text{CA}$

Proof. By induction on Definition A.2.7 of decoding in CAM:

$$\begin{array}{lll}
\nu n V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{\text{IH}} \llbracket V \rrbracket \in \text{CA} \xRightarrow{(A.3)} \nu n \llbracket V \rrbracket \in \text{CA} \xRightarrow{(A.126)} \llbracket \nu n V \rrbracket \in \text{CA} \\
[] \in \text{CAM} & \xRightarrow{(A.1)} \mathbf{0} \in \text{CA} \xRightarrow{(A.127)} \llbracket [] \rrbracket \in \text{CA} \\
\alpha.P :: C \in \text{CAM} & \xRightarrow{(A.86)} \alpha.P \in \text{CA} \wedge C \in \text{CAM} \xRightarrow{\text{IH}} \llbracket C \rrbracket \in \text{CA} \xRightarrow{(A.2)} \alpha.P \mid \llbracket C \rrbracket \in \text{CA} \\
& \xRightarrow{(A.128)} \llbracket \alpha.P :: C \rrbracket \in \text{CA} \\
a[\boxed{A}] :: C \in \text{CAM} & \xRightarrow{(A.87)} A, C \in \text{CAM} \xRightarrow{\text{IH}} \llbracket A \rrbracket, \llbracket C \rrbracket \in \text{CA} \xRightarrow{(A.4;A.2)} a[\llbracket A \rrbracket] \mid \llbracket C \rrbracket \in \text{CA} \\
& \xRightarrow{(A.129)} \llbracket a[\boxed{A}] :: C \rrbracket \in \text{CA}
\end{array}$$

□

Lemma A.2.15. (*Selection Soundness*) $\forall A. A \in \text{CAM} \wedge A \succ A' \Rightarrow \llbracket A \rrbracket \equiv \llbracket A' \rrbracket$

Proof. By induction on Definition A.2.4 of selection in CAM:

$$\begin{array}{lll}
[A.100] & A @ \alpha.P :: A' \in \text{CAM} & \xRightarrow{(A.16;A.23)} \llbracket A \rrbracket \mid b[\llbracket B \rrbracket] \mid \llbracket A' \rrbracket \equiv b[\llbracket B \rrbracket] \mid \llbracket A \rrbracket \mid \llbracket A' \rrbracket \\
& \xRightarrow{(A.86)} A, A' \in \text{CAM} \wedge \alpha.P \in \text{CA} & \xRightarrow{(A.129)} \llbracket A \rrbracket \mid \llbracket b[\boxed{B}] :: A' \rrbracket \equiv \llbracket b[\boxed{B}] :: A \rrbracket \mid \llbracket A' \rrbracket \\
& \xRightarrow{A.2.14} \llbracket A \rrbracket, \llbracket A' \rrbracket \in \text{CAM} & \Rightarrow \llbracket A @ b[\boxed{B}] :: A' \rrbracket \equiv \llbracket b[\boxed{B}] :: A @ A' \rrbracket \\
[A.16;A.23] & \xRightarrow{} \llbracket A \rrbracket \mid \alpha.P \mid \llbracket A' \rrbracket \equiv \alpha.P \mid \llbracket A \rrbracket \mid \llbracket A' \rrbracket \\
& \xRightarrow{(A.128)} \llbracket A \rrbracket \mid \llbracket \alpha.P :: A' \rrbracket \equiv \llbracket \alpha.P :: A \rrbracket \mid \llbracket A' \rrbracket & [A.102] \quad A \succ A' \wedge a[\boxed{A}] :: C \in \text{CAM} \\
& \Rightarrow \llbracket A @ \alpha.P :: A' \rrbracket \equiv \llbracket \alpha.P :: A @ A' \rrbracket & \xRightarrow{(A.87)} A \in \text{CAM} \xRightarrow{\text{IH}} \llbracket A \rrbracket \equiv \llbracket A' \rrbracket \\
& & \xRightarrow{(A.23;A.25)} a[\llbracket A \rrbracket] \mid \llbracket C \rrbracket \equiv a[\llbracket A' \rrbracket] \mid \llbracket C \rrbracket \\
[A.101] & A @ b[\boxed{B}] :: A' \in \text{CAM} & \xRightarrow{(A.129)} \llbracket a[\boxed{A}] :: C \rrbracket \equiv \llbracket a[\boxed{A'}] :: C \rrbracket \\
& \xRightarrow{(A.86)} A, A', B \in \text{CAM} \\
& \xRightarrow{A.2.14} \llbracket A \rrbracket, \llbracket A' \rrbracket, \llbracket B \rrbracket \in \text{CAM}
\end{array}$$

□

Lemma A.2.16. (*Unblocking Soundness*) $\forall V. V \in \text{CAM} \Rightarrow \llbracket \llbracket V \rrbracket \rrbracket = \llbracket V \rrbracket$

Proof. By induction on Definition A.2.5 of unblocking in CAM:

$$\begin{array}{lll}
[A.103] & [] \in \text{CAM} & \xRightarrow{(A.128;A.104)} \llbracket \llbracket \alpha.P :: C \rrbracket \rrbracket = \llbracket \alpha.P :: C \rrbracket \\
& \Rightarrow \mathbf{0} = \mathbf{0} \\
(A.127;A.103) & \xRightarrow{} \llbracket \llbracket [] \rrbracket \rrbracket = \llbracket [] \rrbracket \\
[A.104] & \alpha.P :: C \in \text{CAM} & [A.105] \quad a[\boxed{A}] :: C \in \text{CAM} \\
& \xRightarrow{(A.86)} C \in \text{CAM} \xRightarrow{\text{IH}} \llbracket \llbracket C \rrbracket \rrbracket = \llbracket C \rrbracket & \xRightarrow{(A.87)} C \in \text{CAM} \xRightarrow{\text{IH}} \llbracket \llbracket C \rrbracket \rrbracket = \llbracket C \rrbracket \\
& \xRightarrow{(A.2)} \alpha.P \mid \llbracket \llbracket C \rrbracket \rrbracket = \alpha.P \mid \llbracket C \rrbracket & \xRightarrow{(A.2)} a[\llbracket A \rrbracket] \mid \llbracket \llbracket C \rrbracket \rrbracket = a[\llbracket A \rrbracket] \mid \llbracket C \rrbracket \\
& & \xRightarrow{(A.129;A.105)} \llbracket \llbracket a[\boxed{A}] :: C \rrbracket \rrbracket = \llbracket a[\boxed{A}] :: C \rrbracket
\end{array}$$

□

Lemma A.2.17. (*Construction Soundness*)

$$\begin{aligned} \forall P. \forall V. P \in \text{CA} \wedge V \in \text{CAM} &\Rightarrow \llbracket P : V \rrbracket \equiv P \mid \llbracket V \rrbracket \\ \forall U. \forall V. U, V \in \text{CAM} &\Rightarrow \llbracket a[U] : V \rrbracket \equiv a[\llbracket U \rrbracket] \mid \llbracket V \rrbracket \end{aligned}$$

Proof. By Lemma A.2.14 and by induction on Definition A.2.3 of construction in CAM:

$$\begin{aligned} & \begin{array}{l} \text{[A.90]} \quad n \notin \text{fn}(P) \wedge P \in \text{CA} \wedge \nu n V \in \text{CAM} \\ \xRightarrow{\text{(A.83)}} V \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket V \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.21)}} \nu n (P \mid \llbracket V \rrbracket) \equiv P \mid (\nu n \llbracket V \rrbracket) \\ \xRightarrow{\text{IH}} \nu n \llbracket P : V \rrbracket \equiv P \mid (\nu n \llbracket V \rrbracket) \\ \xRightarrow{\text{(A.126)}} \llbracket \nu n (P : V) \rrbracket \equiv P \mid (\nu n \llbracket V \rrbracket) \\ \xRightarrow{\text{(A.90)}} \llbracket P : (\nu n V) \rrbracket \equiv P \mid \llbracket \nu n V \rrbracket \end{array} & \begin{array}{l} \text{[A.95]} \quad a[P] \in \text{CA} \wedge C \in \text{CAM} \\ \xRightarrow{\text{(A.4)}} P \in \text{CA} \xRightarrow{\text{A.2.14}} \llbracket C \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.15;A.25)}} a[P \mid \mathbf{0}] \mid \llbracket C \rrbracket \equiv a[P] \mid \llbracket C \rrbracket \\ \xRightarrow{\text{IH}} a[\llbracket P : [] \rrbracket] \mid \llbracket C \rrbracket \equiv a[P] \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.7}} \llbracket a[P : []] : C \rrbracket \equiv a[P] \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.95)}} \llbracket a[P] : C \rrbracket \equiv a[P] \mid \llbracket C \rrbracket \end{array} \\ \\ & \begin{array}{l} \text{[A.91]} \quad \mathbf{0} \in \text{CA} \wedge C \in \text{CAM} \\ \xRightarrow{\text{A.2.14}} \llbracket C \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.15)}} \llbracket C \rrbracket \equiv \mathbf{0} \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.91)}} \llbracket \mathbf{0} : C \rrbracket \equiv \mathbf{0} \mid \llbracket C \rrbracket \end{array} & \begin{array}{l} \text{[A.96]} \quad \alpha.P \in \text{CA} \wedge A \in \text{CAM} \\ \xRightarrow{\text{(A.128)}} \llbracket \alpha.P : A \rrbracket \equiv \alpha.P \mid \llbracket A \rrbracket \\ \xRightarrow{\text{(A.96)}} \llbracket \alpha.P : A \rrbracket \equiv \alpha.P \mid \llbracket A \rrbracket \end{array} \\ \\ & \begin{array}{l} \text{[A.92]} \quad (P \mid Q) \in \text{CA} \wedge C \in \text{CAM} \\ \xRightarrow{\text{A.2.14}} \llbracket C \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.17)}} P \mid (Q \mid \llbracket C \rrbracket) \equiv (P \mid Q) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{IH}} P \mid \llbracket Q : C \rrbracket \equiv (P \mid Q) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{IH}} \llbracket P : Q : C \rrbracket \equiv (P \mid Q) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.92)}} \llbracket (P \mid Q) : C \rrbracket \equiv (P \mid Q) \mid \llbracket C \rrbracket \end{array} & \begin{array}{l} n \notin \text{fn}(a, V) \wedge \\ \text{[A.97]} \quad \nu n U \in \text{CAM} \wedge V \in \text{CAM} \\ \xRightarrow{\text{(A.83)}} U \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket U \rrbracket, \llbracket V \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.22;A.21)}} \nu n (a[\llbracket U \rrbracket] \mid \llbracket V \rrbracket) \equiv a[\llbracket \nu n U \rrbracket] \mid \llbracket V \rrbracket \\ \xRightarrow{\text{IH}} \nu n \llbracket a[U] : V \rrbracket \equiv a[\llbracket \nu n U \rrbracket] \mid \llbracket V \rrbracket \\ \xRightarrow{\text{(A.126)}} \llbracket \nu n (a[U] : V) \rrbracket \equiv a[\llbracket \nu n U \rrbracket] \mid \llbracket V \rrbracket \\ \xRightarrow{\text{(A.97)}} \llbracket a[\nu n U] : V \rrbracket \equiv a[\llbracket \nu n U \rrbracket] \mid \llbracket V \rrbracket \end{array} \\ \\ & \begin{array}{l} n \notin \text{fn}(\nu m P, C) \wedge \\ \text{[A.93]} \quad \nu m P \in \text{CA} \wedge C \in \text{CAM} \\ \xRightarrow{\text{(A.3)}} P \in \text{CA} \Rightarrow P_{\{n/m\}} \in \text{CA} \xRightarrow{\text{A.2.14}} \llbracket C \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.21)}} \nu n (P_{\{n/m\}} \mid \llbracket C \rrbracket) \equiv (\nu m P) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{IH}} \nu n \llbracket P_{\{n/m\}} : C \rrbracket \equiv (\nu m P) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.7}} \llbracket \nu n (P_{\{n/m\}} : C) \rrbracket \equiv (\nu m P) \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.93)}} \llbracket (\nu m P) : C \rrbracket \equiv (\nu m P) \mid \llbracket C \rrbracket \end{array} & \begin{array}{l} n \notin \text{fn}(a, V) \wedge \\ \text{[A.98]} \quad A \in \text{CAM} \wedge \nu n V \in \text{CAM} \\ \xRightarrow{\text{(A.83)}} V \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket A \rrbracket, \llbracket V \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.21)}} \nu n (a[\llbracket A \rrbracket] \mid \llbracket V \rrbracket) \equiv a[\llbracket A \rrbracket] \mid \llbracket \nu n V \rrbracket \\ \xRightarrow{\text{IH}} \nu n \llbracket a[A] : V \rrbracket \equiv a[\llbracket A \rrbracket] \mid \llbracket \nu n V \rrbracket \\ \xRightarrow{\text{(A.126)}} \llbracket \nu n (a[A] : V) \rrbracket \equiv a[\llbracket A \rrbracket] \mid \llbracket \nu n V \rrbracket \\ \xRightarrow{\text{(A.98)}} \llbracket a[A] : \nu n V \rrbracket \equiv a[\llbracket A \rrbracket] \mid \llbracket \nu n V \rrbracket \end{array} \\ \\ & \begin{array}{l} \text{[A.94]} \quad !\alpha.P \in \text{CA} \wedge C \in \text{CAM} \\ \xRightarrow{\text{(A.6)}} \alpha.(P \mid !\alpha.P) \in \text{CA} \xRightarrow{\text{A.2.14}} \llbracket C \rrbracket \in \text{CA} \\ \xRightarrow{\text{(A.18;A.35)}} \alpha.(P \mid !\alpha.P) \mid \llbracket C \rrbracket \equiv !\alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{\text{IH}} \llbracket \alpha.(P \mid !\alpha.P) : C \rrbracket \equiv !\alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.94)}} \llbracket !\alpha.P : C \rrbracket \equiv !\alpha.P \mid \llbracket C \rrbracket \end{array} & \begin{array}{l} \text{[A.99]} \quad A \in \text{CAM} \wedge C \in \text{CAM} \\ \xRightarrow{\text{(A.129)}} \llbracket a[A] : C \rrbracket \equiv a[A] \mid \llbracket C \rrbracket \\ \xRightarrow{\text{(A.99)}} \llbracket a[A] : C \rrbracket \equiv a[A] \mid \llbracket C \rrbracket \end{array} \end{aligned}$$

□

Theorem A.2.18. (*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 Lemma A.2.14 (Decoding Soundness), Lemma A.2.15 (Selection Soundness), Lemma A.2.16 (Unblocking Soundness), Lemma A.2.17 (Construction Soundness) and by induction on Definition A.2.6 of reduction in CAM, where $P \dashrightarrow P'$ is short for $(P \longrightarrow P') \vee (P \equiv P')$:

$$\begin{array}{ll}
\begin{array}{l}
\text{[A.123]} \quad B \succ A \wedge A \longrightarrow V' \wedge B \in \text{CAM} \\
\text{A.2.11} \quad \xRightarrow{\text{IH}} A \in \text{CAM} \xRightarrow{\text{IH}} \llbracket A \rrbracket \dashrightarrow \llbracket V' \rrbracket \\
\text{A.2.15} \quad \xRightarrow{\text{IH}} \llbracket B \rrbracket \equiv \llbracket A \rrbracket \\
\text{(A.38)} \quad \xRightarrow{\text{IH}} \llbracket B \rrbracket \dashrightarrow \llbracket V' \rrbracket
\end{array}
&
\begin{array}{l}
\text{(A.36)} \quad \nu n \llbracket V \rrbracket \dashrightarrow \nu n \llbracket V' \rrbracket \\
\text{A.2.7} \quad \llbracket \nu n V \rrbracket \dashrightarrow \llbracket \nu n V' \rrbracket
\end{array} \\
\begin{array}{l}
\text{[A.122]} \quad V \longrightarrow V' \wedge \nu n V \in \text{CAM} \\
\text{A.2.1} \quad \xRightarrow{\text{IH}} V \in \text{CAM} \xRightarrow{\text{IH}} \llbracket V \rrbracket \dashrightarrow \llbracket V' \rrbracket
\end{array}
&
\begin{array}{l}
\text{[A.124]} \quad A \longrightarrow V' \wedge a \llbracket A \rrbracket :: C \in \text{CAM} \\
\text{A.2.1} \quad \xRightarrow{\text{IH}} A, C \in \text{CAM} \xRightarrow{\text{IH}} \llbracket A \rrbracket \dashrightarrow \llbracket V' \rrbracket \\
\text{(A.37;A.35)} \quad a \llbracket \llbracket A \rrbracket \rrbracket \mid \llbracket C \rrbracket \dashrightarrow a \llbracket \llbracket V' \rrbracket \rrbracket \mid \llbracket C \rrbracket \\
\text{A.2.7;A.2.17} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket A \rrbracket :: C \rrbracket \dashrightarrow \llbracket a \llbracket V' \rrbracket :: C \rrbracket
\end{array}
\end{array}$$

$$\begin{array}{ll}
C \succ b \llbracket x^\uparrow(m).Q :: B \rrbracket :: C' \wedge a \llbracket b \cdot x \langle n \rangle . P :: A \rrbracket :: C \in \text{CAM} & \text{A.2.1} \quad A, B, C' \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket A \rrbracket, \llbracket B \rrbracket, \llbracket C' \rrbracket \in \text{CA} \\
\text{(A.32;A.35)} \quad a \llbracket b \cdot x \langle n \rangle . P \mid \llbracket A \rrbracket \rrbracket \mid b \llbracket x^\uparrow(m).Q \mid \llbracket B \rrbracket \rrbracket \mid \llbracket C' \rrbracket \longrightarrow a \llbracket P \mid \llbracket A \rrbracket \rrbracket \mid b \llbracket Q_{\{n/m\}} \mid \llbracket B \rrbracket \rrbracket \mid \llbracket C' \rrbracket \\
\text{A.2.7;A.2.17} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket b \cdot x \langle n \rangle . P :: A \rrbracket :: b \llbracket x^\uparrow(m).Q :: B \rrbracket :: C' \rrbracket \longrightarrow \llbracket a \llbracket P :: A \rrbracket :: b \llbracket Q_{\{n/m\}} :: B \rrbracket :: C' \rrbracket \\
\text{A.2.15;A.38} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket b \cdot x \langle n \rangle . P :: A \rrbracket :: C \rrbracket \longrightarrow \llbracket a \llbracket P :: A \rrbracket :: b \llbracket Q_{\{n/m\}} :: B \rrbracket :: C' \rrbracket
\end{array}$$

$$\begin{array}{ll}
C \succ x(m).Q :: C' \wedge a \llbracket x^\uparrow \langle n \rangle . P :: A \rrbracket :: C \in \text{CAM} & \text{A.2.1} \quad A, C' \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket A \rrbracket, \llbracket C' \rrbracket \in \text{CA} \\
\text{(A.32;A.35)} \quad a \llbracket x^\uparrow \langle n \rangle . P \mid \llbracket A \rrbracket \rrbracket \mid x(m).Q \mid \llbracket C' \rrbracket \longrightarrow a \llbracket P \mid \llbracket A \rrbracket \rrbracket \mid Q_{\{n/m\}} \mid \llbracket C' \rrbracket \\
\text{A.2.7;A.2.17} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket x^\uparrow \langle n \rangle . P :: A \rrbracket :: x(m).Q :: C' \rrbracket \longrightarrow \llbracket a \llbracket P :: A \rrbracket :: Q_{\{n/m\}} :: C' \rrbracket \\
\text{A.2.15;A.38} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket x^\uparrow \langle n \rangle . P :: A \rrbracket :: C \rrbracket \longrightarrow \llbracket a \llbracket P :: A \rrbracket :: Q_{\{n/m\}} :: C' \rrbracket
\end{array}$$

$$\begin{array}{ll}
C \succ b \llbracket \text{in } x.Q :: B \rrbracket :: C' \wedge a \llbracket \text{in } b \cdot x.P :: A \rrbracket :: C \in \text{CAM} & \text{A.2.1} \quad A, B, C' \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket A \rrbracket, \llbracket B \rrbracket, \llbracket C' \rrbracket \in \text{CA} \\
\text{(A.33;A.35)} \quad a \llbracket \text{in } b \cdot x.P \mid \llbracket A \rrbracket \rrbracket \mid b \llbracket \text{in } x.Q \mid \llbracket B \rrbracket \rrbracket \mid \llbracket C' \rrbracket \longrightarrow b \llbracket Q \mid a \llbracket P \mid \llbracket A \rrbracket \rrbracket \mid \llbracket B \rrbracket \rrbracket \mid \llbracket C' \rrbracket \\
\text{A.2.7;A.2.17;A.2.16} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket \text{in } b \cdot x.P :: A \rrbracket :: b \llbracket \text{in } x.Q :: B \rrbracket :: C' \rrbracket \longrightarrow \llbracket b \llbracket Q :: a \llbracket P :: A \rrbracket \rrbracket :: B \rrbracket :: C' \rrbracket \\
\text{A.2.15;A.38} \quad \xRightarrow{\text{IH}} \llbracket a \llbracket \text{in } b \cdot x.P :: A \rrbracket :: C \rrbracket \longrightarrow \llbracket b \llbracket Q :: a \llbracket P :: A \rrbracket \rrbracket :: B \rrbracket :: C' \rrbracket
\end{array}$$

$$\begin{array}{ll}
B \succ \text{out } x.Q :: B' \wedge b \llbracket a \llbracket \text{out } x.P :: A \rrbracket :: B \rrbracket :: C \in \text{CAM} & \text{A.2.1} \quad A, B', C \in \text{CAM} \xRightarrow{\text{A.2.14}} \llbracket A \rrbracket, \llbracket B' \rrbracket, \llbracket C \rrbracket \in \text{CA} \\
\text{(A.34;A.35)} \quad b \llbracket a \llbracket \text{out } x.P \mid \llbracket A \rrbracket \rrbracket \mid \text{out } x.Q \mid \llbracket B' \rrbracket \rrbracket \mid \llbracket C \rrbracket \longrightarrow b \llbracket Q \mid \llbracket B' \rrbracket \rrbracket \mid a \llbracket P \mid \llbracket A \rrbracket \rrbracket \mid \llbracket C \rrbracket \\
\text{A.2.7;A.2.17;A.2.16} \quad \xRightarrow{\text{IH}} \llbracket b \llbracket a \llbracket \text{out } x.P :: A \rrbracket :: \text{out } x.Q :: B' \rrbracket :: C \rrbracket \longrightarrow \llbracket b \llbracket Q :: B' \rrbracket :: a \llbracket P :: A \rrbracket \rrbracket :: C \rrbracket \\
\text{A.2.15;A.38} \quad \xRightarrow{\text{IH}} \llbracket b \llbracket a \llbracket \text{out } x.P :: A \rrbracket :: B \rrbracket :: C \rrbracket \longrightarrow \llbracket b \llbracket Q :: B' \rrbracket :: a \llbracket P :: A \rrbracket \rrbracket :: C \rrbracket
\end{array}$$

□

A.2.3 Completeness

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 .

Lemma A.2.19 (Structural Correctness) ensures that structural congruence always produces a valid machine term.

Lemma A.2.20 (Structural Reduction) ensures that structurally congruent terms can perform corresponding reductions. This property needs to be proved explicitly for the machine, since structural congruence is not used in the definition of reduction. The theorem states that if the machine can reduce a term V to V' then it can reduce any term that is structurally congruent to V to a term that is structurally congruent to V' .

Once a structural congruence relation has been defined in this way, it is possible to prove the completeness of the machine. Calculus execution is defined in terms of structural congruence and reduction. Therefore, in order to prove the completeness of the machine it is necessary to prove that structural congruence and reduction are complete.

Lemma A.2.21 (Structural Completeness) ensures that if two calculus processes are structurally congruent then their encodings are structurally congruent. The lemma states that if a given process P is structurally congruent to process Q then the encoding of P is structurally congruent to the encoding of Q .

Theorem A.2.22 (Reduction Completeness) ensures that if a process can perform a reduction in the calculus then its encoding can perform a corresponding sequence of reductions in the machine, up to structural congruence. The theorem 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.

Lemma A.2.19. (*Structural Correctness*) $\forall V.V \in \text{CAM} \wedge V \equiv V' \Rightarrow V' \in \text{CAM}$

Proof. By induction on Definition A.2.8 of structural congruence in CAM:

$$\begin{array}{lll}
n \notin \text{fn}(V) \wedge \nu n V \in \text{CAM} & \xRightarrow{(A.83)} & V \in \text{CAM} \\
\nu n \nu m V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{(A.83)} & \nu m \nu n V \in \text{CAM} \\
A @ B @ C \in \text{CAM} & \xRightarrow{(A.84)} A, B, C \in \text{CAM} \xRightarrow{(A.84)} & B @ A @ C \in \text{CAM} \\
V \equiv V' \wedge \nu n V \in \text{CAM} & \xRightarrow{(A.83)} V \in \text{CAM} \xRightarrow{\text{IH}} V' \in \text{CAM} \xRightarrow{(A.83)} & \nu n V' \in \text{CAM} \\
A \equiv A' \wedge a[A] :: C \in \text{CAM} & \xRightarrow{(A.87)} A, C \in \text{CAM} \xRightarrow{\text{IH}} A' \in \text{CAM} \xRightarrow{(A.87)} & a[A'] :: C \in \text{CAM} \\
P \equiv P' \wedge \alpha.P :: C \in \text{CAM} & \xRightarrow{(A.5)} P \in \text{CA} \xRightarrow{A.1.13} P' \in \text{CA} \xRightarrow{(A.5)} & \alpha.P' :: C \in \text{CAM} \\
C \equiv C' \wedge A @ C \in \text{CAM} & \xRightarrow{\text{IH}} C' \in \text{CAM} \xRightarrow{(A.84)} & A @ C' \in \text{CAM}
\end{array}$$

□

Lemma A.2.20. (*Structural Reduction*) $\forall V.V \in \text{CAM} \wedge U \equiv V \wedge V \longrightarrow V' \Rightarrow U \longrightarrow \equiv V'$

Proof. By Definition A.2.4 of selection and by induction on Definition A.2.8 of structural congruence in CAM:

$$\begin{array}{ll}
[A.130] \quad n \notin \text{fn}(V) \wedge V \longrightarrow V' & [A.133] \quad U \equiv V \wedge \nu n V \longrightarrow \nu n V' \\
(A.122) \xRightarrow{\quad} \nu n V \longrightarrow \nu n V' \wedge n \notin \text{fn}(V') & (A.122) \xRightarrow{\quad} V \longrightarrow V' \\
(A.130) \xRightarrow{\quad} \nu n V \longrightarrow \equiv V' & \xRightarrow{\text{IH}} U \longrightarrow \equiv V' \\
& (A.122) \xRightarrow{\quad} \nu n U \longrightarrow \equiv \nu n V' \\
[A.131] \quad \nu m \nu n V \longrightarrow \nu m \nu n V' & \\
(A.122) \xRightarrow{\quad} V \longrightarrow V' & [A.134] \quad B \equiv A \wedge a[A] :: C \longrightarrow V' \\
(A.122) \xRightarrow{\quad} \nu n \nu m V \longrightarrow \nu n \nu m V' & \xRightarrow{A.2.4} a[B] :: C \longrightarrow \equiv V' \\
(A.131) \xRightarrow{\quad} \nu n \nu m V \longrightarrow \equiv \nu m \nu n V' & \\
[A.132] \quad A @ B @ C \longrightarrow V' & [A.135] \quad P \equiv P' \wedge \alpha.P :: C \longrightarrow V' \\
& \xRightarrow{A.2.4} \alpha.P' :: C \longrightarrow \equiv V' \\
& \xRightarrow{A.2.4} B @ A @ C \longrightarrow \equiv V' \\
[A.136] \quad C \equiv C' \wedge A @ C \longrightarrow V' & \\
& \xRightarrow{A.2.4} A @ C' \longrightarrow \equiv V'
\end{array}$$

□

Lemma A.2.21. (*Structural Completeness*) $\forall P \in \text{CA}. P \equiv Q \Rightarrow \llbracket P \rrbracket \equiv \llbracket Q \rrbracket$

Proof. By induction on Definition A.1.2 of structural congruence in CA, with induction hypothesis $P \equiv Q \Rightarrow \llbracket P \rrbracket \equiv \llbracket Q \rrbracket$:

$$\begin{array}{ll}
\begin{array}{l}
\text{[A.15]} \quad \mathbf{0} \mid P \equiv P \\
\Rightarrow P : \llbracket R \rrbracket \equiv P : \llbracket R \rrbracket \\
\stackrel{(\text{A.91})}{\Rightarrow} \mathbf{0} : P : \llbracket R \rrbracket \equiv P : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} (\mathbf{0} \mid P \mid R) \equiv (P \mid R)
\end{array}
&
\begin{array}{l}
\text{[A.21]} \quad (\nu n P) \mid Q \equiv \nu n (P \mid Q) \wedge n \notin \text{fn}(Q, R) \\
\Rightarrow \nu n (P : Q : \llbracket R \rrbracket) \equiv \nu n (P : Q : \llbracket R \rrbracket) \\
\stackrel{(\text{A.92})}{\Rightarrow} \nu n (P : Q : \llbracket R \rrbracket) \equiv \nu n ((P \mid Q) : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} (\nu n P) : Q : \llbracket R \rrbracket \equiv (\nu n (P \mid Q)) : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} ((\nu n P) \mid Q \mid R) \equiv ((\nu n (P \mid Q)) \mid R)
\end{array} \\
\\
\begin{array}{l}
\text{[A.16]} \quad P \mid Q \equiv Q \mid P \\
\stackrel{(\text{A.133}; \text{A.132})}{\Rightarrow} \nu \tilde{z} (A @ B @ C) \equiv \nu \tilde{z} (B @ A @ C) \\
\stackrel{\text{A.2.3}}{\Rightarrow} P : Q : \llbracket R \rrbracket \equiv Q : P : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket P \mid Q \mid R \rrbracket \equiv \llbracket Q \mid P \mid R \rrbracket
\end{array}
&
\begin{array}{l}
\text{[A.22]} \quad a \boxed{\nu n P} \equiv \nu n a \boxed{P} \wedge n \notin \text{fn}(a, R) \\
\Rightarrow \nu n (a \boxed{P : \square} : \llbracket R \rrbracket) \equiv \nu n (a \boxed{P : \square} : \llbracket R \rrbracket) \\
\stackrel{(\text{A.97})}{\Rightarrow} a \boxed{\nu n (P : \square)} : \llbracket R \rrbracket \equiv \nu n (a \boxed{P : \square} : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} a \boxed{(\nu n P) : \square} : \llbracket R \rrbracket \equiv \nu n (a \boxed{P : \square} : \llbracket R \rrbracket) \\
\stackrel{(\text{A.95})}{\Rightarrow} a \boxed{\nu n P} : \llbracket R \rrbracket \equiv \nu n (a \boxed{P} : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} a \boxed{\nu n P} : \llbracket R \rrbracket \equiv (\nu n a \boxed{P}) : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket a \boxed{\nu n P} \mid R \rrbracket \equiv \llbracket (\nu n a \boxed{P}) \mid R \rrbracket
\end{array} \\
\\
\begin{array}{l}
\text{[A.17]} \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
\Rightarrow P : Q : R : \llbracket S \rrbracket \equiv P : Q : R : \llbracket S \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} P : (Q \mid R) : \llbracket S \rrbracket \equiv (P \mid Q) : R : \llbracket S \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket P \mid (Q \mid R) \mid S \rrbracket \equiv \llbracket (P \mid Q) \mid R \mid S \rrbracket
\end{array}
&
\begin{array}{l}
\text{[A.23]} \quad P \equiv P' \wedge P \mid Q \equiv P' \mid Q \\
\stackrel{\text{IH}}{\Rightarrow} \llbracket P \mid Q \rrbracket \equiv \llbracket P' \mid Q \rrbracket
\end{array} \\
\\
\begin{array}{l}
\text{[A.18]} \quad !\alpha.P \equiv \alpha.(P \mid !\alpha.P) \\
\Rightarrow \alpha.(P \mid !\alpha.P) : \llbracket R \rrbracket \equiv \alpha.(P \mid !\alpha.P) : \llbracket R \rrbracket \\
\stackrel{(\text{A.94})}{\Rightarrow} !\alpha.P : \llbracket R \rrbracket \equiv \alpha.(P \mid !\alpha.P) : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket !\alpha.P \mid R \rrbracket \equiv \llbracket \alpha.(P \mid !\alpha.P) \mid R \rrbracket
\end{array}
&
\begin{array}{l}
\text{[A.24]} \quad P \equiv P' \wedge \nu n P \equiv \nu n P' \wedge n \notin \text{fn}(R) \\
\stackrel{\text{IH}}{\Rightarrow} \llbracket P \mid R \rrbracket \equiv \llbracket P' \mid R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} P : \llbracket R \rrbracket \equiv P' : \llbracket R \rrbracket \\
\stackrel{(\text{A.133})}{\Rightarrow} \nu n (P : \llbracket R \rrbracket) \equiv \nu n (P' : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} (\nu n P) : \llbracket R \rrbracket \equiv (\nu n P') : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket \nu n P \mid R \rrbracket \equiv \llbracket \nu n P' \mid R \rrbracket
\end{array} \\
\\
\begin{array}{l}
\text{[A.19]} \quad \nu n \mathbf{0} \equiv \mathbf{0} \wedge m \notin \text{fn}(R) \\
\stackrel{(\text{A.130})}{\Rightarrow} \nu n \llbracket R \rrbracket \equiv \llbracket R \rrbracket \\
\stackrel{(\text{A.91})}{\Rightarrow} \nu m (\mathbf{0} : \llbracket R \rrbracket) \equiv \mathbf{0} : \llbracket R \rrbracket \\
\stackrel{(\text{A.93})}{\Rightarrow} (\nu n \mathbf{0}) : \llbracket R \rrbracket \equiv \mathbf{0} : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket (\nu n \mathbf{0}) \mid R \rrbracket \equiv \llbracket \mathbf{0} \mid R \rrbracket
\end{array}
&
\begin{array}{l}
\text{[A.25]} \quad P \equiv P' \wedge a \boxed{P} \equiv a \boxed{P'} \\
\stackrel{\text{IH}}{\Rightarrow} \llbracket P \mid \mathbf{0} \rrbracket \equiv \llbracket P' \mid \mathbf{0} \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} P : \mathbf{0} : \square \equiv P' : \mathbf{0} : \square \\
\stackrel{(\text{A.91})}{\Rightarrow} P : \square \equiv P' : \square \\
\stackrel{(\text{A.134})}{\Rightarrow} a \boxed{P : \square} : \llbracket R \rrbracket \equiv a \boxed{P' : \square} : \llbracket R \rrbracket \\
\stackrel{(\text{A.95})}{\Rightarrow} a \boxed{P} : \llbracket R \rrbracket \equiv a \boxed{P'} : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket a \boxed{P} \mid R \rrbracket \equiv \llbracket a \boxed{P'} \mid R \rrbracket
\end{array} \\
\\
\begin{array}{l}
\text{[A.20]} \quad \nu n \nu m P \equiv \nu m \nu n P \wedge m, n \notin \text{fn}(R) \\
\stackrel{(\text{A.131})}{\Rightarrow} \nu n \nu m (P : \llbracket R \rrbracket) \equiv \nu m \nu n (P : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} (\nu n \nu m P) : \llbracket R \rrbracket \equiv (\nu m \nu n P) : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \llbracket (\nu n \nu m P) \mid R \rrbracket \equiv \llbracket (\nu m \nu n P) \mid R \rrbracket
\end{array}
&
\end{array}$$

□

Theorem A.2.22. (*Reduction Completeness*) $\forall P.P \in \text{CA} \wedge P \longrightarrow P' \Rightarrow \llbracket P \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \rrbracket$

Proof. By Lemma A.2.20, Lemma A.2.21 and by induction on Definition A.1.3 of reduction in CA, with induction hypothesis $P \longrightarrow P' \Rightarrow \llbracket P \mid R \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \mid R \rrbracket$:

$$\begin{array}{ll}
\begin{array}{l}
\text{[A.38]} \quad Q \equiv P \wedge P \longrightarrow P' \wedge P' \equiv Q' \wedge Q \longrightarrow Q' \\
\stackrel{\text{IH}}{\Rightarrow} \quad \llbracket P \mid R \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \mid R \rrbracket \\
\stackrel{\text{A.2.21}}{\Rightarrow} \quad \llbracket P \mid R \rrbracket \equiv \llbracket Q \mid R \rrbracket \wedge \llbracket P' \mid R \rrbracket \equiv \llbracket Q' \mid R \rrbracket \\
\stackrel{\text{A.2.20}}{\Rightarrow} \quad \llbracket Q \mid R \rrbracket \longrightarrow \longrightarrow \equiv \llbracket Q' \mid R \rrbracket
\end{array} &
\begin{array}{l}
\text{[A.35]} \quad P \longrightarrow P' \wedge (P \mid Q) \longrightarrow (P' \mid Q) \\
\stackrel{\text{IH}}{\Rightarrow} \quad \llbracket P \mid Q \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P' \mid Q \rrbracket \\
\text{[A.36]} \quad P \longrightarrow P' \wedge (\nu n P) \longrightarrow (\nu n P') \wedge m \notin \text{fn}(P, R) \\
\Rightarrow \quad P_{\{m/n\}} \longrightarrow P'_{\{m/n\}} \\
\stackrel{\text{IH}}{\Rightarrow} \quad \llbracket P_{\{m/n\}} \mid R \rrbracket \longrightarrow \longrightarrow \equiv \llbracket P'_{\{m/n\}} \mid R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \quad P_{\{m/n\}} : \llbracket R \rrbracket \longrightarrow \longrightarrow \equiv P'_{\{m/n\}} : \llbracket R \rrbracket \\
\stackrel{(\text{A.122})}{\Rightarrow} \quad \nu m (P_{\{m/n\}} : \llbracket R \rrbracket) \longrightarrow \longrightarrow \equiv \nu m (P'_{\{m/n\}} : \llbracket R \rrbracket) \\
\stackrel{(\text{A.93})}{\Rightarrow} \quad (\nu n P) : \llbracket R \rrbracket \longrightarrow \longrightarrow \equiv (\nu n P') : \llbracket R \rrbracket \\
\stackrel{(\text{A.92})}{\Rightarrow} \quad \llbracket (\nu n P) \mid R \rrbracket \longrightarrow \longrightarrow \equiv \llbracket (\nu n P') \mid R \rrbracket
\end{array}
\end{array}$$

$$\begin{array}{l}
\text{[A.31]} \quad a \boxed{b \cdot x \langle n \rangle . P \mid P'} \mid b \boxed{x^\dagger(m) . Q \mid Q'} \longrightarrow a \boxed{P \mid P'} \mid b \boxed{Q_{\{n/m\}} \mid Q'} \\
\stackrel{(\text{A.122}; \text{A.109}; \text{A.106})}{\Rightarrow} \quad \nu \tilde{z} a \boxed{b \cdot x \langle n \rangle . P :: A} :: b \boxed{x^\dagger(m) . Q :: B} :: C \longrightarrow \nu \tilde{z} a \boxed{P : A} : b \boxed{Q_{\{n/m\}} : B} : C \\
\stackrel{\text{A.2.3}}{\Rightarrow} \quad a \boxed{b \cdot x \langle n \rangle . P : \llbracket P' \rrbracket} : b \boxed{x^\dagger(m) . Q : \llbracket Q' \rrbracket} : \llbracket R \rrbracket \longrightarrow a \boxed{P : \llbracket P' \rrbracket} : b \boxed{Q_{\{n/m\}} : \llbracket Q' \rrbracket} : \llbracket R \rrbracket \\
\stackrel{\text{A.2.2}}{\Rightarrow} \quad (a \boxed{b \cdot x \langle n \rangle . P \mid P'} \mid b \boxed{x^\dagger(m) . Q \mid Q'} \mid R) \longrightarrow (a \boxed{P \mid P'} \mid b \boxed{Q_{\{n/m\}} \mid Q'} \mid R)
\end{array}$$

$$\begin{array}{l}
\text{[A.32]} \quad a \boxed{x^\dagger \langle n \rangle . P \mid P'} \mid x(m) . Q \longrightarrow a \boxed{P \mid P'} \mid Q_{\{n/m\}} \\
\stackrel{(\text{A.122}; \text{A.113}; \text{A.110})}{\Rightarrow} \quad \nu \tilde{z} a \boxed{x^\dagger \langle n \rangle . P :: A} :: x(m) . Q :: C \longrightarrow \nu \tilde{z} a \boxed{P : A} : Q_{\{n/m\}} : C' \\
\stackrel{\text{A.2.3}}{\Rightarrow} \quad a \boxed{x^\dagger \langle n \rangle . P : \llbracket P' \rrbracket} : x(m) . Q : \llbracket R \rrbracket \longrightarrow a \boxed{P : \llbracket P' \rrbracket} : Q_{\{n/m\}} : \llbracket R \rrbracket \\
\stackrel{\text{A.2.2}}{\Rightarrow} \quad (a \boxed{x^\dagger \langle n \rangle . P \mid P'} \mid x(m) . Q \mid R) \longrightarrow (a \boxed{P \mid P'} \mid Q_{\{n/m\}} \mid R)
\end{array}$$

$$\begin{array}{l}
\text{[A.33]} \quad a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\text{in } x . Q \mid Q'} \longrightarrow b \boxed{Q \mid a \boxed{P \mid P'}} \mid Q' \\
\stackrel{(\text{A.122}; \text{A.117}; \text{A.119})}{\Rightarrow} \quad \nu \tilde{z} a \boxed{\text{in } b \cdot x . P :: A} :: b \boxed{\text{in } x . Q :: B} :: C \longrightarrow \nu \tilde{z} b \boxed{Q : a \boxed{P : A}} : B : C' \\
\stackrel{\text{A.2.3}}{\Rightarrow} \quad a \boxed{\text{in } b \cdot x . P : \llbracket P' \rrbracket} : b \boxed{\text{in } x . Q : \llbracket Q' \rrbracket} : \llbracket R \rrbracket \longrightarrow b \boxed{Q : a \boxed{P : \llbracket P' \rrbracket}} : \llbracket Q' \rrbracket : \llbracket R \rrbracket \\
\stackrel{\text{A.2.2}}{\Rightarrow} \quad (a \boxed{\text{in } b \cdot x . P \mid P'} \mid b \boxed{\text{in } x . Q \mid Q'} \mid R) \longrightarrow (b \boxed{Q \mid a \boxed{P \mid P'}} \mid Q' \mid R)
\end{array}$$

$$\begin{array}{l}
\text{[A.34]} \quad b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \text{out } x . Q \mid Q'} \longrightarrow b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \\
\stackrel{(\text{A.122}; \text{A.121}; \text{A.118})}{\Rightarrow} \quad \nu \tilde{z} b \boxed{a \boxed{\text{out } x . P :: A} :: \text{out } x . Q :: B} :: C \longrightarrow \nu \tilde{z} b \boxed{Q : B'} : a \boxed{P : A} : C \\
\stackrel{\text{A.2.3}}{\Rightarrow} \quad b \boxed{a \boxed{\text{out } x . P : \llbracket P' \rrbracket} :: \text{out } x . Q : \llbracket Q' \rrbracket} : \llbracket R \rrbracket \longrightarrow b \boxed{Q : \llbracket Q' \rrbracket} : a \boxed{P : \llbracket P' \rrbracket} : \llbracket R \rrbracket \\
\stackrel{\text{A.2.2}}{\Rightarrow} \quad (b \boxed{a \boxed{\text{out } x . P \mid P'} \mid \text{out } x . Q \mid Q'} \mid R) \longrightarrow (b \boxed{Q \mid Q'} \mid a \boxed{P \mid P'} \mid R)
\end{array}$$

□

A.2.4 Liveness

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. 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 CAM^\vee and is defined by placing constraints on the set of machine terms CAM .

Lemma A.2.23 (Deadlock-Free Subset) ensures that deadlock-free terms are a subset of machine terms. The lemma states that if a given term V is deadlock-free then it is a valid machine term.

Once the set of deadlock-free terms has been defined in this way, it is possible to prove that the machine is deadlock-free. Machine execution is defined in terms of construction, selection, unblocking and reduction. Therefore, in order to prove that the machine is deadlock-free it is necessary to prove that construction, selection, unblocking and reduction are deadlock-free.

Lemma A.2.24 (Deadlock-Free Construction) ensures that construction cannot cause a term to deadlock. The lemma states that if a process P is added to a term V then the result is deadlock-free. Similarly, if an ambient a with unblocked contents $[A]$ is added to a term V then the result is deadlock-free. Note that a direct consequence of this lemma is that $\llbracket P \rrbracket \in \text{CAM}^\vee$, since $\llbracket P \rrbracket = P : []$.

Lemma A.2.25 (Deadlock-Free Selection) ensures that selection cannot cause a term to deadlock. The lemma states that if the machine selects a term A' from a deadlock-free term A then A' is deadlock-free.

Lemma A.2.26 (Deadlock-Free Unblocking) ensures that unblocking cannot cause a term to deadlock. The lemma states that if a deadlock-free term V is unblocked then the result is deadlock-free.

Finally, Lemma A.2.27 (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.

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 A.2.28 (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 A.2.29 (Decoding Encoding) ensures that a process is structurally congruent to the decoding of its encoding.

Lemma A.2.30 (Encoding Decoding) ensures that a term and the encoding of its decoding both have the same decoding. Note that by Definition A.2.7, if two terms have the same decoding then they are equal up to blocking of actions and ambients.

Once these properties have been proved for reduction, decoding and encoding, it is possible to prove the liveness of the machine. Theorem A.2.31 (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.

Lemma A.2.23. (*Deadlock Subset*) $\forall V.V \in \text{CAM}^\vee \Rightarrow V \in \text{CAM}$

Proof. By induction on Definition A.2.9 of deadlock-free terms CAM^\vee . \square

Lemma A.2.24. (*Deadlock-Free Construction*)

$$\forall P.\forall V.P \in \text{CA} \wedge V \in \text{CAM}^\vee \Rightarrow P:V \in \text{CAM}^\vee$$

$$\forall A.\forall V.A \in \text{CAM}^\vee \wedge V \in \text{CAM}^\vee \Rightarrow a[\boxed{A}]:V \in \text{CAM}^\vee$$

Proof. By induction on Definition A.2.3 of construction in CAM . The proof is similar to that of Lemma A.2.10. \square

Lemma A.2.25. (*Deadlock-Free Selection*) $\forall A.A \in \text{CAM}^\vee \wedge A \succ A' \Rightarrow A' \in \text{CAM}^\vee$

Proof. By Definition A.2.9 of deadlock-free terms CAM^\vee . The proof is similar to that of Lemma A.2.11. \square

Lemma A.2.26. (*Deadlock-Free Unblocking*) $\forall V.V \in \text{CAM}^\vee \Rightarrow [V] \in \text{CAM}^\vee$

Proof. By induction on Definition A.2.5 of unblocking in CAM :

$$\begin{array}{llll}
\boxed{} \in \text{CAM}^\vee & \xRightarrow{\text{(A.103)}} & & \boxed{} \in \text{CAM}^\vee \\
\alpha.P::C \in \text{CAM}^\vee & \xRightarrow{\text{(A.143)}} C \in \text{CAM}^\vee \xRightarrow{\text{IH}} [C] \in \text{CAM}^\vee \xRightarrow{\text{(A.143)}} & & \alpha.P::[C] \in \text{CAM}^\vee \\
& \xRightarrow{\text{(A.104)}} & & [\alpha.P::C] \in \text{CAM}^\vee \\
a[\boxed{A}]:C \in \text{CAM}^\vee & \xRightarrow{\text{(A.144)}} A,C \in \text{CAM}^\vee \xRightarrow{\text{IH}} [C] \in \text{CAM}^\vee \xRightarrow{\text{(A.144)}} & & a[\boxed{A}]:[C] \in \text{CAM}^\vee \\
& \xRightarrow{\text{(A.105)}} & & [a[\boxed{A}]:C] \in \text{CAM}^\vee
\end{array}$$

\square

Lemma A.2.27. (*Deadlock-Free Reduction*) $\forall V.V \in \text{CAM}^\vee \wedge V \longrightarrow V' \Rightarrow V' \in \text{CAM}^\vee$

Proof. By Lemma A.2.24 (Deadlock-Free Construction), Lemma A.2.25 (Deadlock-Free Selection), Lemma A.2.26 (Deadlock-Free Unblocking) and by induction on Definition A.2.6 of reduction in CAM:

Most of the cases are straightforward, apart from the case where $A \longrightarrow V'$ and $a[A]::C \in \text{CAM}^\vee$. In this case we want to show that $a[V']::C \in \text{CAM}^\vee$. We know that $A, C \in \text{CAM}^\vee$ by rule A.144, and that $V' \in \text{CAM}^\vee$ by induction. Assume $V' = \nu\tilde{n} A'$. By simple inspection of the reduction rules in Definition A.2.6, the term A' cannot contain any blocked actions at the top-level that were not already in A (apart from $x(m)$), neither can it contain any top-level ambients with a blocked leave that were not already in A . Thus $a[A']::C \in \text{CAM}^\vee$ by rule A.144. Therefore, assuming $\tilde{n} \notin \text{fn}(C)$ we can show that $\nu\tilde{n} a[A']::C \in \text{CAM}^\vee$ and consequently $a[V']::C \in \text{CAM}^\vee$.

$$\begin{array}{llll} V \longrightarrow V' \wedge \nu n V \in \text{CAM}^\vee & \xRightarrow{\text{A.140}} & V \in \text{CAM}^\vee \xRightarrow{\text{IH}} V' \in \text{CAM}^\vee & \xRightarrow{\text{A.140}} \nu n V' \in \text{CAM}^\vee \\ B \succ A \wedge A \longrightarrow V' \wedge B \in \text{CAM}^\vee & \xRightarrow{\text{A.2,25}} & A \in \text{CAM}^\vee \xRightarrow{\text{IH}} & V' \in \text{CAM}^\vee \\ A \longrightarrow V' \wedge a[A]::C \in \text{CAM}^\vee & \xRightarrow{\text{A.144}} & A, C \in \text{CAM}^\vee \xRightarrow{\text{IH}} V' \in \text{CAM}^\vee & \xRightarrow{\text{A.144}} a[V']::C \in \text{CAM}^\vee \end{array}$$

$$\begin{array}{ll} C \succ b[\overline{x^\dagger(m)}.Q::B]::C' \wedge a[b \cdot x\langle n \rangle.P::A]::C \in \text{CAM}^\vee & \xRightarrow{\text{A.2,9}} A, C \in \text{CAM}^\vee \xRightarrow{\text{A.2,25}} B, C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.144}} a[A]::b[B]::C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,24}} a[P:A]::b[Q_{\{n/m\}}:B]::C' \in \text{CAM}^\vee \\ C \succ x(m).Q::C' \wedge a[x^\dagger\langle n \rangle.P::A]::C \in \text{CAM}^\vee & \xRightarrow{\text{A.2,9}} A, C \in \text{CAM}^\vee \xRightarrow{\text{A.2,25}} C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.144}} a[A]::C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,24}} a[P:A]::Q_{\{n/m\}}:C' \in \text{CAM}^\vee \\ C \succ b[\underline{\text{in}} x.Q::B]::C' \wedge a[\underline{\text{in}} b \cdot x.P::A]::C \in \text{CAM}^\vee & \xRightarrow{\text{A.2,9}} A, C \in \text{CAM}^\vee \xRightarrow{\text{A.2,25}} B, C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,26}} [A] \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.144}} b[a[A]::B]::C' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,24}} b[Q:a[P:[A]]:B]::C' \in \text{CAM}^\vee \\ B \succ \overline{\text{out}} x.Q::B' \wedge b[a[\overline{\text{out}} x.P::A]::B]::C \in \text{CAM}^\vee & \xRightarrow{\text{A.2,9}} A, B, C \in \text{CAM}^\vee \xRightarrow{\text{A.2,25}} B' \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,26}} [A] \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.144}} b[B']::a[A]::C \in \text{CAM}^\vee \\ & \xRightarrow{\text{A.2,24}} b[Q:B']::a[P:[A]]:C \in \text{CAM}^\vee \end{array}$$

□

Lemma A.2.28. (*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 A.2.6 of reduction in CAM. By Definition A.2.7, two terms with the same decoding are equal up to blocking of actions and ambients. By Definition A.2.9, well-formed terms cannot have a blocked action and a corresponding blocked co-action simultaneously:

$$\begin{array}{ll}
\text{[A.122]} & U, V \in \text{CAM}^\vee \wedge V \longrightarrow V' \\
& \wedge \quad \nu n V \longrightarrow \nu n V' \wedge \llbracket W \rrbracket = \llbracket \nu n V \rrbracket \\
\text{(A.126)} & \Rightarrow W = \nu n U \wedge \llbracket U \rrbracket = \llbracket V \rrbracket \\
& \xRightarrow{\text{IH}} \exists U'. U \longrightarrow U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket \\
\text{(A.122)} & \Rightarrow \nu n U \longrightarrow \nu n U' \wedge \llbracket \nu n U' \rrbracket = \llbracket \nu n V' \rrbracket \\
\text{[A.123]} & B, B' \in \text{CAM}^\vee \wedge B \succ A \\
& \wedge \quad A \longrightarrow V' \wedge B \longrightarrow V' \wedge \llbracket B' \rrbracket = \llbracket B \rrbracket \\
& \Rightarrow \quad B' \succ A' \wedge \llbracket A' \rrbracket = \llbracket A \rrbracket \\
& \xRightarrow{\text{IH}} \exists U'. A' \longrightarrow^* U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket \\
& \xRightarrow{\text{(A.124)}} U \longrightarrow^* a \llbracket U' \rrbracket : C' \wedge \llbracket a \llbracket U' \rrbracket : C' \rrbracket = \llbracket a \llbracket V' \rrbracket : C \rrbracket
\end{array}$$

$$\begin{array}{ll}
\text{[A.106]} & U, V \in \text{CAM}^\vee \wedge V \longrightarrow a \llbracket P : A \rrbracket : b \llbracket Q_{\{n/m\}} : B \rrbracket : D = V' \\
& \wedge \quad V = a \llbracket b \cdot x \langle n \rangle . P :: A \rrbracket :: C \wedge C \succ b \llbracket x^\dagger(m) . Q :: B \rrbracket :: D \wedge \llbracket V \rrbracket = \llbracket U \rrbracket \\
\text{(A.129, A.128)} & \Rightarrow U = a \llbracket b \cdot x \langle n \rangle . P :: A' \rrbracket :: C' \wedge C' \succ b \llbracket x^\dagger(m) . Q :: B' \rrbracket :: D' \wedge \llbracket A \rrbracket, \llbracket B \rrbracket, \llbracket D \rrbracket = \llbracket A' \rrbracket, \llbracket B' \rrbracket, \llbracket D' \rrbracket \\
\text{(A.106, A.108)} & \Rightarrow U \longrightarrow^* a \llbracket P : A' \rrbracket : b \llbracket Q_{\{n/m\}} : B' \rrbracket : D' = U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket
\end{array}$$

$$\begin{array}{ll}
\text{[A.110]} & V \in \text{CAM}^\vee \wedge V \longrightarrow a \llbracket P : A \rrbracket : Q_{\{n/m\}} : D = V' \\
& \wedge \quad V = a \llbracket x^\dagger \langle n \rangle . P :: A \rrbracket :: C \wedge C \succ x(m) . Q :: D \wedge \llbracket V \rrbracket = \llbracket U \rrbracket \\
\text{(A.129, A.128)} & \Rightarrow U = a \llbracket x^\dagger \langle n \rangle . P :: A' \rrbracket :: C' \wedge C' \succ x(m) . Q :: D' \wedge \llbracket A \rrbracket, \llbracket D \rrbracket \wedge \llbracket A' \rrbracket, \llbracket D' \rrbracket \\
\text{(A.110, A.112)} & \Rightarrow U \longrightarrow^* a \llbracket P : A' \rrbracket : Q_{\{n/m\}} : D' = U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket
\end{array}$$

$$\begin{array}{ll}
\text{[A.114]} & V \in \text{CAM}^\vee \wedge V \longrightarrow b \llbracket Q : a \llbracket P : [A] \rrbracket : B \rrbracket : D = V' \\
& \wedge \quad V = a \llbracket \text{in } b \cdot x . P :: A \rrbracket :: C \wedge C \succ b \llbracket \text{in } x . Q :: B \rrbracket :: D \wedge \llbracket V \rrbracket = \llbracket U \rrbracket \\
\text{(A.129, A.128)} & \Rightarrow U = a \llbracket \text{in } b \cdot x . P :: A' \rrbracket :: C' \wedge C' \succ b \llbracket \text{in } x . Q :: B' \rrbracket :: D' \wedge \llbracket A \rrbracket, \llbracket B \rrbracket, \llbracket D \rrbracket = \llbracket A' \rrbracket, \llbracket B' \rrbracket, \llbracket D' \rrbracket \\
\text{(A.114, A.116)} & \Rightarrow U \longrightarrow^* b \llbracket Q : a \llbracket P : [A'] \rrbracket : B' \rrbracket : D' = U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket
\end{array}$$

$$\begin{array}{ll}
\text{[A.118]} & V \in \text{CAM}^\vee \wedge V \longrightarrow b \llbracket Q : D \rrbracket : a \llbracket P : [A] \rrbracket : C = V' \\
& \wedge \quad V = b \llbracket a \llbracket \text{out } x . P :: A \rrbracket : B \rrbracket :: C \wedge B \succ \text{out } x . Q :: D \wedge \llbracket V \rrbracket = \llbracket U \rrbracket \\
\text{(A.129, A.128)} & \Rightarrow U = b \llbracket a \llbracket \text{out } x . P :: A' \rrbracket : B' \rrbracket :: C' \wedge B' \succ \text{out } x . Q :: D' \wedge \llbracket A \rrbracket, \llbracket C \rrbracket, \llbracket D \rrbracket = \llbracket A' \rrbracket, \llbracket C' \rrbracket, \llbracket D' \rrbracket \\
\text{(A.118, A.120)} & \Rightarrow U \longrightarrow^* b \llbracket Q : D' \rrbracket : a \llbracket P : [A'] \rrbracket : C' = U' \wedge \llbracket U' \rrbracket = \llbracket V' \rrbracket
\end{array}$$

□

Lemma A.2.29. (*Decoding Encoding*) $\forall P. P \in \text{CA} \Rightarrow \llbracket \langle P \rangle \rrbracket \equiv P$

Proof. Follows from Lemma A.2.17:

$$P \in \text{CA} \xRightarrow{\text{A.2.17}} \llbracket P : [] \rrbracket \equiv P \mid \mathbf{0} \xRightarrow{\text{A.2.7}} \llbracket \langle P \rangle \rrbracket \equiv P \mid \mathbf{0} \xRightarrow{(\text{A.15})} \llbracket \langle P \rangle \rrbracket \equiv P$$

□

Lemma A.2.30. (*Encoding Decoding*) $\forall V. V \in \text{CAM} \Rightarrow \llbracket \llbracket \langle V \rangle \rrbracket \rrbracket = \llbracket V \rrbracket$

Proof. By induction on Definition A.2.7 of decoding in CAM. As usual, machine terms are assumed to be equal up to alpha-conversion of bound names:

$ \begin{array}{ll} \text{[A.126]} & \nu n V \in \text{CAM} \xRightarrow{(\text{A.83})} V \in \text{CAM} \\ \xRightarrow{\text{IH}} & \llbracket \llbracket \langle V \rangle \rrbracket \rrbracket = \llbracket V \rrbracket \\ \xRightarrow{(\text{A.3})} & \nu n \llbracket \llbracket \langle V \rangle \rrbracket \rrbracket = \nu n \llbracket V \rrbracket \\ \xRightarrow{(\text{A.126})} & \llbracket \nu n \llbracket \langle V \rangle \rrbracket \rrbracket = \nu n \llbracket V \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket \nu n \llbracket \langle V \rangle : [] \rrbracket \rrbracket = \nu n \llbracket V \rrbracket \\ \xRightarrow{(\text{A.93})} & \llbracket (\nu n \llbracket V \rrbracket) : [] \rrbracket = \nu n \llbracket V \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket \llbracket \nu n \llbracket V \rrbracket \rrbracket \rrbracket = \nu n \llbracket V \rrbracket \\ \xRightarrow{(\text{A.126})} & \llbracket \llbracket \llbracket \nu n V \rrbracket \rrbracket \rrbracket = \llbracket \nu n V \rrbracket \end{array} $	$ \begin{array}{ll} \text{[A.127]} & \mathbf{0} \in \text{CAM} \\ \xRightarrow{(\text{A.127})} & \llbracket [] \rrbracket = \mathbf{0} \\ \xRightarrow{(\text{A.91})} & \llbracket \mathbf{0} : [] \rrbracket = \mathbf{0} \\ \xRightarrow{\text{A.2.2}} & \llbracket \langle \mathbf{0} \rangle \rrbracket = \mathbf{0} \\ \xRightarrow{(\text{A.127})} & \llbracket \llbracket [] \rrbracket \rrbracket = [] \end{array} $
$ \begin{array}{ll} \text{[A.128]} & \alpha.P :: C \in \text{CAM} \Rightarrow C \in \text{CAM} \\ \xRightarrow{\text{IH}} & \llbracket \llbracket \langle C \rangle \rrbracket \rrbracket = \llbracket C \rrbracket \\ \xRightarrow{(\text{A.2,A.5})} & \alpha.P \mid \llbracket \llbracket \langle C \rangle \rrbracket \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.128})} & \llbracket \alpha.P :: \llbracket \langle C \rangle \rrbracket \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.96})} & \llbracket \alpha.P : \llbracket \langle C \rangle \rrbracket \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket \alpha.P : \llbracket C \rrbracket : [] \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.92})} & \llbracket (\alpha.P \mid \llbracket C \rrbracket) : [] \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket \llbracket \alpha.P \mid \llbracket C \rrbracket \rrbracket \rrbracket = \alpha.P \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.128})} & \llbracket \llbracket \llbracket \alpha.P :: C \rrbracket \rrbracket \rrbracket = \llbracket \alpha.P :: C \rrbracket \end{array} $	$ \begin{array}{ll} \text{[A.129]} & a \llbracket A \rrbracket :: C \in \text{CAM} \Rightarrow A, C \in \text{CAM} \\ \xRightarrow{\text{IH}} & \llbracket \llbracket \langle A \rangle \rrbracket \rrbracket, \llbracket \llbracket \langle C \rangle \rrbracket \rrbracket = \llbracket A \rrbracket, \llbracket C \rrbracket \\ \xRightarrow{(\text{A.4,A.2})} & a \llbracket \llbracket \langle A \rangle \rrbracket \rrbracket \mid \llbracket \llbracket \langle C \rangle \rrbracket \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.129})} & \llbracket a \llbracket \langle A \rangle \rrbracket :: \llbracket \langle C \rangle \rrbracket \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.99})} & \llbracket a \llbracket \langle A \rangle \rrbracket : \llbracket \langle C \rangle \rrbracket \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket a \llbracket A \rrbracket : [] : \llbracket C \rrbracket : [] \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.92,A.95})} & \llbracket (a \llbracket A \rrbracket \mid \llbracket C \rrbracket) : [] \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{\text{A.2.2}} & \llbracket \llbracket a \llbracket A \rrbracket \mid \llbracket C \rrbracket \rrbracket \rrbracket = a \llbracket A \rrbracket \mid \llbracket C \rrbracket \\ \xRightarrow{(\text{A.129})} & \llbracket \llbracket a \llbracket A \rrbracket :: C \rrbracket \rrbracket = \llbracket a \llbracket A \rrbracket :: C \rrbracket \end{array} $

□

Theorem A.2.31. (*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 A.2.29 (Decoding Encoding), Lemma A.2.30 (Encoding Decoding), Lemma A.2.28 (Decoding Reduction) and by Theorem A.2.22 (Reduction Completeness):

$$\begin{array}{ll}
 V \in \text{CAM}^\vee \wedge \llbracket V \rrbracket \longrightarrow P' & \xRightarrow{\text{A.2.22}} \llbracket \llbracket V \rrbracket \rrbracket \longrightarrow \longrightarrow \equiv (P') \\
 & \xRightarrow{\text{A.2.30}} \llbracket V \rrbracket = \llbracket \llbracket \langle V \rangle \rrbracket \rrbracket \\
 & \xRightarrow{\text{A.2.28}} \exists V'. V \longrightarrow^* V' \wedge \llbracket V' \rrbracket \equiv \llbracket \langle P' \rangle \rrbracket \\
 & \xRightarrow{\text{A.2.29}} \exists V'. V \longrightarrow^* V' \wedge \llbracket V' \rrbracket \equiv P'
 \end{array}$$

□

Appendix B

Language Syntax

The Channel Ambient Language is summarised below, where optional elements are enclosed in braces as *{Optional}*:

<i>Program</i>	<i>::=</i>	<i>Name</i> { <i>Address</i> } [<i>Process</i>] <i>Address</i> { <i>Address</i> } [<i>Process</i>] <i>Address</i> [<i>Address</i> { <i>Address</i> } [<i>Process</i>]]	Local Site Network Site Nested Site
<i>Process</i>	<i>::=</i>	<i>()</i> <i>Process</i> <i>Process</i> new <i>Name</i> : <i>Type</i> <i>Process</i> <i>Agent</i> [<i>Process</i>] <i>Action</i> { ; <i>Process</i> } ! <i>Action</i> { ; <i>Process</i> } if <i>Value</i> { -> <i>Value</i> } then <i>Process</i> { else <i>Process</i> } type <i>Name</i> = <i>Type</i> <i>Process</i> let <i>Pattern</i> = <i>Value</i> in <i>Process</i> let <i>Macro</i> ({ <i>Pattern</i> }) = <i>Process</i> in <i>Process</i> <i>Macro</i> < { <i>Value</i> } > <i>(Process)</i>	Null Parallel Composition Restriction Agent Action Replicated Action Pattern Matching Type Definition Value Definition Macro Definition Macro Instantiation Nested Process
<i>Action</i>	<i>::=</i>	<i>Ambient</i> . <i>Channel</i> < { <i>Value</i> } > <i>Ambient</i> / <i>Channel</i> < { <i>Value</i> } > <i>Channel</i> ^ ({ <i>Pattern</i> }) <i>Channel</i> ^ < { <i>Value</i> } > <i>Channel</i> < { <i>Value</i> } > <i>Channel</i> ({ <i>Pattern</i> }) in <i>Ambient</i> . <i>Channel</i> -in <i>Channel</i> out <i>Channel</i> -out <i>Channel</i>	Sibling Output Child Output External Input Parent Output Local Output Internal Input Enter Accept Leave Release
<i>Pattern</i>	<i>::=</i>	- <i>()</i> <i>Name</i> { : <i>Type</i> } <i>Pattern</i> , <i>Pattern</i> <i>(Pattern)</i>	Wildcard Pattern Void Pattern Name Pattern Tuple Pattern Nested Pattern

<i>Type</i>	::=	void	Void Type
		string	String Type
		int	Integer Type
		float	Float Type
		char	Character Type
		bool	Boolean Type
		site	Site Type
		agent	Agent Type
		migrate	Migration Type
		<{Type}>	Channel Type
		<i>Name</i>	Type Variable
		'Name	Polymorphic Type
		sub Type	Sub Type
		<i>Type</i> , <i>Type</i>	Tuple Type
		<i>Upper</i>	Type Constant
		<i>Upper Type</i>	Type Constructor
		<i>Type</i> <i>Type</i>	Data Type
		<i>Type</i> list	List Type
		rec <i>Name</i> . <i>Type</i>	Recursive Type
		(Type)	Nested Type
<i>Value</i>	::=	()	Void Value
		<i>String</i>	String Value
		<i>Integer</i>	Integer Value
		<i>Float</i>	Float Value
		<i>Character</i>	Character Value
		true	Boolean True
		false	Boolean False
		<i>Address</i>	Site Value
		<i>Name</i> : agent	Agent Value
		<i>Name</i> : <{Type}>	Channel Value
		<i>Value</i> , <i>Value</i>	Tuple Value
		<i>Upper</i>	Data Constant
		<i>Upper Value</i>	Data Constructor
		[]	Empty List
		<i>Value</i> :: <i>Value</i>	Value List
		<i>Name</i>	Variable
		<i>Value</i> + <i>Value</i>	Addition
		<i>Value</i> - <i>Value</i>	Subtraction
		<i>Value</i> * <i>Value</i>	Multiplication
		<i>Value</i> / <i>Value</i>	Division
		<i>Value</i> = <i>Value</i>	Equal
		<i>Value</i> <> <i>Value</i>	Different
		<i>Value</i> < <i>Value</i>	Less Than
		<i>Value</i> > <i>Value</i>	Greater Than
		<i>Value</i> <= <i>Value</i>	Less Than or Equal
		<i>Value</i> >= <i>Value</i>	Greater Than or Equal
		-Value	Negation
		show <i>Value</i>	String Representation
		(Value)	Nested Value

Regular Expressions Regular Expressions (*Regexp*) are used to describe the syntax of Constants and Variables in the Channel Ambient Language:

$Regex$	$::=$	c	Character
	$ $	$c \cdots c$	Character Range
	$ $	$\neg c$	Character Complement
	$ $	$Regex\ Regex$	Concatenation of Expressions
	$ $	$Regex Regex$	Alternative Expressions
	$ $	$Regex^?$	Optional Expression
	$ $	$Regex^*$	Repetition of Expression
	$ $	$Regex^+$	Strict Repetition of Expression
	$ $	$(Regex)$	Nested Expression

Constants An *Integer* constant consists of an optional negative sign followed by one or more digits:

$$Integer ::= (-)^?(0 \cdots 9)^+$$

A *String* constant consists of a sequence of zero or more characters enclosed in double quotes. The sequence can only contain a double quote if it is preceded by a backslash:

$$String ::= "((\neg) | (\backslash))^{*}"$$

A *Float* constant consists of an *Integer*, followed by a decimal point and one or more digits, followed by an optional exponent. The exponent consists of e or E , followed by $+$ or $-$, followed by one or more digits:

$$Float ::= Integer.(0 \cdots 9)^+((e | E)(+ | -)(0 \cdots 9)^+)^?$$

An *IP* address constant consists of a sequence of four numbers between 0 and 255, separated by a decimal point. The IP address 0.0.0.0 is short for the default IP address of the host machine:

$$IP ::= (0 \cdots 9)^+.(0 \cdots 9)^+.(0 \cdots 9)^+.(0 \cdots 9)^+$$

A *Character* constant consists of any character enclosed in single quotes, apart from the single quote character. It can also consist of a backslash, followed by a special *escaped* character or a three-digit decimal number, enclosed in single quotes:

$Character ::=$	$'(-)'$	Regular Character
	$ \backslash$	Single Quote
	$ \backslash\backslash$	Backslash
	$ \backslash n$	Linefeed
	$ \backslash r$	Carriage Return
	$ \backslash t$	Horizontal Tabulation
	$ \backslash b$	Backspace
	$ \backslash(0\dots9)(0\dots9)(0\dots9)'$	ASCII Character Code

Variables A *Name* variable consists of a letter followed by zero or more letters, digits, underscores or single quotes:

$$Name ::= (a\dots z)(A\dots Z | a\dots z | 0\dots9 | _ | ')*$$

An *Upper* case variable consists of an upper case letter followed by zero or more letters, digits, underscores or single quotes:

$$Upper ::= (A\dots Z)(A\dots Z | a\dots z | 0\dots9 | _ | ')*$$

Channel, *Agent* and *Site* variables are *Names* representing *Channel* values, *Agent* values and *Site* values, respectively. An *Ambient* variable is a *Name* representing either an *Agent* value or a *Site* value. A *Macro* variable is an *Upper* case variable representing a macro:

$$\begin{aligned} Channel &::= Name \\ Agent &::= Name \\ Site &::= Name \\ Ambient &::= Name \\ Macro &::= Upper \end{aligned}$$

The following variables are reserved keywords of the language:

agent	bool	char	else	float	false	if	in
int	let	list	migrate	new	out	rec	show
site	string	sub	then	true	type	void	

Comments A comment starts with the sequence of characters `(*` and ends with the sequence of characters `*)`. Comments can be nested, but they cannot occur inside single or double quotes.

Bibliography

- [1] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.
- [2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Journal of Information and Computation*, 148(1):1–70, 1999. An extended abstract appeared in the *Proceedings of the Fourth ACM Conference on Computer and Communications Security (Zürich, April 1997)*. An extended version of this paper appears as Research Report 149, Digital Equipment Corporation Systems Research Center, January 1998, and, in preliminary form, as Technical Report 414, University of Cambridge Computer Laboratory, January 1997.
- [3] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.
- [4] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.
- [5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
- [6] Ken Arnold, James Gosling, and David Holmes. *The Java Language Specification 3rd Edition*. Addison-Wesley, June 2000.
- [7] Martin Berger. Basic theory of reduction congruence for two timed asynchronous π -calculi. In *CONCUR'04*, volume 3170 of *LNCS*, pages 115–130. Springer, January 2004.
- [8] 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.

- [9] 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.
- [10] Gérard Boudol. Asynchrony and the π -calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.
- [11] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01*, number 2215 in LNCS, pages 38–63. Springer, 2001.
- [12] 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.
- [13] 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.
- [14] Michele Bugliesi and Giuseppe Castagna. Secure safe ambients. In *Proceedings of POPL '01*, pages 222–235. ACM, January 2001.
- [15] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part ii). In Luboš Brim, Petr Jančar, Mojmir Křetinský, and Antonín Kučera, editors, *Proceedings of CONCUR 2002*, volume 2421 of *LNCS*, pages 209–225. Springer, August 2002.
- [16] Cristiano Calcagno, Luca Cardelli, and Andrew D. Gordon. Deciding validity in a spatial logic for trees. In *TLDI'03*, volume 38, pages 62–73. ACM Press, January 2003.
- [17] Luca Cardelli. Mobile ambient synchronization. Technical Report SRC-TN-1997-013, Digital Equipment Corporation, July 25 1997.
- [18] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In Jiří Wiederman, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of ICALP '99*, volume 1644 of *LNCS*, pages 230–239. Springer, July 1999.
- [19] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In van Leeuwen et al. [78], pages 333–347.
- [20] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of POPL '00*, pages 365–377. ACM, January 2000.
- [21] 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.

- [22] Giuseppe Castagna and Francesco Zappa Nardelli. The seal calculus revisited: Contextual equivalence and bisimilarity. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 85–96. Springer-Verlag, 2002.
- [23] Giuseppe Castagna, Jan Vitek, and Francesco Zappa. The seal calculus. 2003. Available from <ftp://ftp.di.ens.fr/pub/users/castagna/seal.ps.gz>.
- [24] 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.
- [25] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [26] 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.
- [27] S. Crafa, M. Bugliesi, and G. Castagna. Information flow security for boxed ambients. In *F-WAN'02*, number 66(3) in *ENTCS*, 2002.
- [28] 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.
- [29] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
- [30] Cédric Fournet, Jean-Jacques Lévy, and Alain Schmitt. An asynchronous distributed implementation of mobile ambients. In van Leeuwen et al. [78], pages 348–364.
- [31] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. Internet Engineering Task Force: RFC 1519, September 1993.
- [32] S. Funfrocken. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In K. Rothermel and F. Hohl, editors, *Mobile Agents: Proceedings of the Second International Workshop*. Springer Verlag, 1998.
- [33] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software-Practice and Experience*, 1-5, 1999.
- [34] Philippa Gardner and Lucian Wischik. Explicit fusions. In Mogens Nielsen and Branislav Rován, editors, *Proceedings of MFCS 2000*, volume 1893 of *LNCS*, pages 373–382. Springer, 2000.

- [35] Jason Hallstrom Gerald Baumgartner and Xiaojin Wang. Reliability through strong mobility. Technical report, Dept of Computer and Information Science, Ohio State University, June 2001.
- [36] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.
- [37] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of MTTCPE*, number 794 in LNCS, pages 353–368, Vienna, May 1994. Springer-Verlag.
- [38] Andrew D. Gordon and Luca Cardelli. Equational properties of mobile ambients. In Wolfgang Thomas, editor, *Proceedings of FoSSaCS '99*, volume 1578 of LNCS, pages 212–226. Springer, 1999.
- [39] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [40] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In M[ario] Tokoro, O[scar] Nierstrasz, and P[eter] Wegner, editors, *Object-Based Concurrent Computing 1991*, volume 612 of LNCS, pages 21–51. Springer, 1992.
- [41] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995. An extract appeared in *Proceedings of FSTTCS '93*, LNCS 761.
- [42] ITU-TS. *Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 1996.
- [43] 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.
- [44] 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.
- [45] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*. ACM Press, 2000.
- [46] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *CONCUR'02*, volume 2421 of LNCS, pages 304–320. Springer, 2002.
- [47] Massimo Merro and Matthew Hennessy. Bisimulation congruences in safe ambients. In *POPL'02*, pages 71–80. ACM Press, 2002.

- [48] Robin Milner. *Communicating and Mobile Systems: the π -Calculus*. Cambridge University Press, May 1999.
- [49] 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.
- [50] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [51] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, Berlin, Germany, 1997.
- [52] Andrew Phillips. *The Channel Ambient System*, 2005. Runtime and documentation available from <http://www.doc.ic.ac.uk/~anp/ca.html>.
- [53] Andrew Phillips. *The Stochastic Pi Machine*, 2005. Simulator and documentation available from <http://www.doc.ic.ac.uk/~anp/spim/>.
- [54] Andrew Phillips and Luca Cardelli. A correct abstract machine for the stochastic pi-calculus. In *Bioconcur'04*. ENTCS, August 2004.
- [55] Andrew Phillips, Susan Eisenbach, and Daniel Lister. From process algebra to java code. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.
- [56] Andrew Phillips, Nobuko Yoshida, and Susan Eisenbach. A distributed abstract machine for boxed ambient calculi. In *ESOP'04*, LNCS. Springer, April 2004.
- [57] I.C.C. Phillips. CCS with priority guards. In *Proceedings of 12th International Conference on Concurrency Theory, CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2001.
- [58] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, May 2000.
- [59] C. Priami, A. Regev, E. Shapiro, and W. Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, 2001.
- [60] 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.

- [61] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*. ACM, 1998.
- [62] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999. Full version as CogSci Report 4/98, University of Sussex, Brighton.
- [63] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2004. Draft, <http://www.ps.uni-sb.de/Papers/>.
- [64] 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.
- [65] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.
- [66] D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. In *ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.
- [67] Davide Sangiorgi and Andrea Valente. A distributed abstract machine for safe ambients. In *Proceedings of ICALP 2001*, volume 2076 of *LNCS*. Springer, July 2001.
- [68] Davide Sangiorgi and David Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- [69] Peter Sewell. Applied π – a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, August 2000.
- [70] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *ICFP'05*, September 2005.
- [71] Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location independence for mobile agents. In H.E. Bal, B. Belkhouche, and L[uca] Cardelli, editors, *Proceedings of ICCL '98, Workshop on Internet Programming Languages (Chicago, IL, USA, May 13, 1998)*, volume 1686 of *LNCS*. Springer, September 1999. Full version with title *Location-Independent Communication for Mobile Agents: a Two-Level Architecture* appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

- [72] Niranjani Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An overview of the nomads mobile agent system. In *ECOOP'00*, 2000.
- [73] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, pages 29–43, 2000.
- [74] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).
- [75] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *Transactions on Software Engineering and Methodology*, 13(1):37–85, January 2004.
- [76] Asis Unyapoth. Nomadic π -calculi: Expressing and verifying communication infrastructure for mobile computation. Technical Report UCAM-CL-TR-514, University of Cambridge, Computer Laboratory, June 2001.
- [77] Asis Unyapoth and Peter Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *POPL'01*, pages 116–127, 2001.
- [78] J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses, and T. Ito, editors. *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000 (Sendai, Japan)*, volume 1872 of *LNCS*. IFIP, Springer, August 2000.
- [79] Björn Victor and Faron Moller. The Mobility Workbench — a tool for the π -calculus. In David Dill, editor, *Proceedings of CAV '94*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.
- [80] M.G. Vigliotti and I.C.C. Phillips. Barbs and congruences for safe mobile ambients. In *F-WAN: Foundations of Wide Area Network Computing*, volume 66 of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.
- [81] Hagen Volzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. In *CONCUR'05*, volume 3653 of *LNCS*, pages 458–472. Springer, August 2005.
- [82] Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: the*

- 11th European Symposium on Programming (Grenoble)*, LNCS 2305, pages 278–294, April 2002.
- [83] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, June 2000. Also appeared as Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.
- [84] Paweł T. Wojciechowski. *The Nomadic Pict System*, 2000. Available electronically as part of the Nomadic Pict distribution.