Glint: Breeding Mobile Ambients with Actors

Final Report

Matthew Sackman

Supervisor: Susan Eisenbach
Second Marker: Sophia Drossopoulou

Copyright © Matthew Sackman, 2006. All rights reserved.
Abstract

Software Engineering is facing a crisis. Hardware designers are no longer able to make ever faster single-threaded CPUs and are instead building CPUs with thread-level parallelism. Simultaneously, clusters of computers are becoming far more popular for building high performance systems but making good use of these parallel and distributed systems is difficult to achieve with current tools. Mobile computation is also becoming increasingly important as computers shrink in size and become physically more portable. Engineering applications that can migrate between devices and share resources is a very difficult task given the current tools, platforms and languages available.

The methodologies and techniques for addressing the issues of concurrent, distributed and mobile computation are badly underdeveloped and unsuitable for the large software systems being built today. Typical locking mechanisms in modern programming languages are problematic and unsafe to use in large software systems because of the inability to analyse successfully their use and because they are not safe in composition. Distributed and mobile computation is similarly badly addressed by most programming languages which often present complex and misleading notions of mobility.

This report presents the GLINTVM, an interpreter for the Safe Boxed Ambient calculus of Merro and Sassone. Ambient calculi are designed to model concurrent, distributed and mobile computation but with some adaptation I use them as an assembly language for a virtual machine. I add mobility to the Actor paradigm of Hewitt and Agha and present the GLINT language based on the Actor model. The Actor model was similarly designed with concurrency and distribution in mind but is a more powerful abstraction, is more expressive than the Ambient calculi and is hence more suited for programming. I then present a compiler which translates from the GLINT language to the Safe Boxed Ambient calculus which can then be interpreted by the GLINTVM.

This complete tool chain of a language, compiler and interpreter contributes to the range of tools and techniques that are available for tackling the parallel architectures and the challenges of mobile and distributed computation. The resulting programming model is better suited for these purposes and more concisely and powerfully expresses movement than existing languages such as Java.
Acknowledgements

This project would not have been possible without the input from many, of whom here, I have memory to thank just a few. Firstly my supervisor, Susan Eisenbach for allowing me considerable free reign in my work on this project and for telling me when to walk away before hospitalisation became necessary.

Many thanks to Amy for tolerating the endless late nights, deterioration of my ability to hold a normal conversation and general coping with my devolution into something socially more primitive. Thanks too for Sophia Drossopoulou’s insistence that the implementation of a highly concurrent and complex system is not something to trivialise away.

Tim Wood for superb knowledge of Java as ever along with Tristan Allwood, Marc Hull, Steve Zymler, Francis Russell and others for inspiring and insightful conversation and political debate. Thanks to the SLURP group for their critical and good natured grilling and thanks to Will Knottenbelt for permitting me access to his high performance cluster. Thanks too to the French Alps for being suitably covered in snow and for providing a sufficiently soft landing after an eight-foot unsighted drop...
Contents

1 Introduction ............................... 1
  1.1 Background and Context ..................... 1
  1.2 Contributions ............................. 2
  1.3 Report Structure ......................... 2

2 Ambient Calculi ............................ 5
  2.1 Mobile Ambients ............................ 5
    2.1.1 Communication ......................... 5
    2.1.2 Objective and Subjective moves ......... 6
    2.1.3 Objective open ......................... 7
  2.2 Boxed Ambients ............................ 7
  2.3 Communication Interference ............... 9
  2.4 Typing Ambients ........................... 10
    2.4.1 Typing Mobile Ambients ................. 10
      2.4.1.1 Exchange Types ....................... 10
      2.4.1.2 Opening Control ....................... 11
      2.4.1.3 Crossing Control ....................... 11
    2.4.2 Typing Boxed Ambients ................. 12
      2.4.2.1 Typing process communication ....... 12
      2.4.2.2 Typing Ambient mobility ............. 12
  2.5 The Channel Ambient System ............... 13
  2.6 Conclusion ............................... 14

3 Actors .................................... 15
  3.1 Basic Concepts ............................ 15
  3.2 Benefits of Actors ......................... 16
    3.2.1 Shared Mutable State ................. 16
      3.2.1.1 Transactional Memory .......... 17
    3.2.2 Unshared Mutable State ............... 17
  3.3 Implementations ........................... 18
    3.3.1 SCHEME ............................... 18
    3.3.2 ERLANG ............................... 18
    3.3.3 SALSA ................................. 19
      3.3.3.1 Continuation Based Programming .... 19
      3.3.3.2 Token Passing Continuations ..... 20
      3.3.3.3 Join Continuations ............... 21
      3.3.3.4 First Class Continuations ....... 21
  3.4 Conclusion ............................... 22

4 The GLINTVM ............................... 25
  4.1 Which Ambient calculus? ................... 25
    4.1.1 From LISP to ASCII .................... 26
6.2 Encoding into Ambients .................................................. 64
  6.2.1 Basic Properties of the Encoding ................................. 64
  6.2.2 Marshalling Callers .................................................. 65
  6.2.3 Changing Behaviours ................................................. 66
  6.2.4 Receiving Method Calls .............................................. 67
  6.2.5 Method Callers ....................................................... 69
  6.2.6 Encoding Become .................................................... 70
  6.2.7 Local Method Calls .................................................. 72
  6.2.8 Anonymous Inner Actors ........................................... 73
  6.2.9 Methods with Predicates ............................................ 73
  6.2.10 True and False ...................................................... 74
  6.2.11 Encoding Numbers .................................................. 74
  6.2.12 Equality Testing .................................................... 75
6.3 Technologies Employed .................................................. 76
6.4 Conclusion .............................................................. 76

7 Evaluation ........................................................................... 79
  7.1 Two-tiered Design ......................................................... 79
  7.2 The GLINTVM ............................................................. 79
  7.3 The GLINT Language ..................................................... 80
  7.4 The Channel Ambient System .......................................... 84
  7.5 Performance .............................................................. 88
    7.5.1 Custom GLINTVM Code ............................................. 89
      7.5.1.1 Sequential-Sequential Communication ......................... 89
      7.5.1.2 Parallel-Sequential Communication ............................. 89
      7.5.1.3 Parallel-Parallel Communication ............................... 89
      7.5.1.4 Local Movement .................................................... 90
      7.5.1.5 Inter-host Movement .............................................. 90
    7.5.2 Generated GLINTVM Code ......................................... 90
      7.5.2.1 Method Calls ..................................................... 90
      7.5.2.2 Local Method Calls .............................................. 91
      7.5.2.3 Inter-host Method Calls ........................................ 91
    7.5.3 Comparison with JAVA .............................................. 92
    7.5.4 Conclusion .......................................................... 92

8 Conclusion ........................................................................... 93
  8.1 Ambients ................................................................. 93
  8.2 Actors ................................................................. 93
  8.3 Future Work ........................................................... 94
    8.3.1 Improving the GLINTVM .......................................... 94
      8.3.1.1 Garbage Collection ............................................. 94
      8.3.1.2 Debugging and Inspection ..................................... 95
      8.3.1.3 Security of Communication and Inter-System Interaction .. 95
      8.3.1.4 Typings in the GLINTVM ...................................... 95
    8.3.2 Extending the GLINT Language .................................... 95
      8.3.2.1 Asynchronous Messages ....................................... 95
      8.3.2.2 Interfaces .......................................................... 95
      8.3.2.3 Type Systems ..................................................... 96
      8.3.2.4 Named Parameters .............................................. 97
      8.3.2.5 Currying ........................................................... 98
      8.3.2.6 Traits .............................................................. 99
    8.3.3 Redesigning the Encoding .......................................... 100
  8.4 Final Thoughts ........................................................ 100
1.1 Background and Context

For several decades, Moore's law has been interpreted to mean that the speed of single-threaded execution will double every 18 months and until 2004, this was indeed the case. But since 2004, it is only the number of transistors per CPU that has been doubling every 18 months as Moore originally predicted. This inability to increase the speed of single-threaded execution has forced all leading processor designers to introduce multiple execution cores per CPU. Current CPUs for desktop computers from both Intel and AMD are dual-core and quad-core CPUs will be appearing very soon. By the end of the decade octal-core and higher CPUs will be available. For games consoles, the new Microsoft XBox 360 features the Xenon processor from IBM which has three cores with each core capable of supporting two execution threads simultaneously for a total of six threads. The CELL processor that will power the upcoming Sony Playstation 3 contains up to nine cores.

The effects of this shift from single to multi-core CPUs cannot be understated. Whilst CPUs have for years supported parallelism by being able to analyse instructions and data for operations that can be executed in parallel, they have always presented a single-threaded execution model to programmers. As a result, programming models and paradigms have been developed with this mind and consequently the models and techniques to deal with multi-threaded and parallel execution are not as well developed. Parallel computation presents difficult challenges for programmers: ensuring the correct behaviour of multi-threaded programs and preventing deadlocks and livelocks are problems that in the general case cannot be addressed by static analyses.

Additionally, the Internet continues to grow and is increasingly being recognised as a source of massive computational power that has yet to be fully utilised. The distributed and potentially disconnected properties of the Internet makes the modelling and implementation of applications which operate well throughout the Internet very challenging. Zones of administration and the ever changing topology of the Internet creates challenges for models to address and makes programming such a system extremely difficult.

Different architectures, levels of accessibility, CPU, storage and power resources all combine to obstruct programmers needing to write distributed applications for use throughout the Internet. Many of these difficulties are also faced when trying to program high performance clusters that combine multiple multi-core CPUs throughout several computers with a high speed network interconnect. Making best use of the available resources with typical error-prone fine grained locking techniques is very challenging. Debugging misbehaving parallel applications is a miserable task due to the non-determinism inherent in parallel computation and the inability to verify the success of a modification.

At the same time, the modern human tends to carry with them a growing array of devices: laptops, mobile phones, PDAs and music players amongst others. This growth in mobile devices has brought to the forefront models concerning the movement of processes between different devices. For example, one might need the ability to transfer phone functions to a PDA or music player if the phone is turned off, or to share resources such as memory, CPU power and battery-life amongst a range of devices. Cooperation between such devices is currently very poor and where it does exist it only amounts to transferring application data such as music files, phone book entries or calendar appointments. There is the potential however, to be able to transfer running applications or parts of particular applications: consider being able to transfer the audio components of a video game onto your mobile phone which is connected to a headset you are wearing whilst the rest of the game continues to run on your PDA. Imagine that this would be possible without the game programmers taking any special steps to accommodate this functionality. Alternatively, consider the ability to spread a long running scientific calculation across many computers as and when they become available and to be able to adapt dynamically to the available computational resources without having to rewrite the application. It is these challenges that this project seeks to address.

The Object Oriented paradigm and the Functional paradigm that are both so widespread today are...
not designed with concurrent or distributed or mobile applications in mind. **Java** comes with support for distributed computation in the form of RMI and serialisation allows for mobility of a certain kind but it is still badly flawed for concurrent computation because the threading model and the mechanisms it provides for regulation of concurrent access to data are so primitive. Traditional locking mechanisms such as mutexes, semaphores and monitors are not safe under composition, are very hard to statically analyse and are easily misused. Furthermore, the mechanisms **Java** provides for mobility and distributed computation are complex and restrictive. **Java** is not alone in this: very few languages support the mobility of threads or code.

**Alice**¹ and **Acute**² are functional languages which have been designed to support distribution and mobility. **Alice** extends ML with capabilities to allow better control over distributed and mobile computation but leaves the task of resolving references to non-local data to the programmer. **Acute** is another extension of ML which has support for explicit marshalling and unmarshalling of data structures and functions. Similarly however, no distributed references are supported within the language, meaning that references to local values within data structures or functions that are to be marshalled must be either copied to the destination runtime or are already known to exist at the destination.

The Object Oriented paradigm focusses on the Object as a passive data structure, abstracting away the execution thread and presuming that the threads can manage amongst themselves to arrange safe passage through the Object. The Actor paradigm instead encapsulates an execution thread with a data Object and presents a more restricted model of computation which makes more exacting demands of the behaviour of threads, preventing shared access to Objects and eliminating some forms of deadlock.

The Ambient calculi are relatively recent creations building on existing process algebras. They formally model concurrent, mobile and distributed computation and focus on threads (which they term *processes*), communication and movement instead of focussing on data structures. All the clarity that Object Orientation brings to data Objects is instead, with the Ambient calculi, brought to threads. However, the Ambient calculi have been proposed primarily for formal modelling and analysis purposes and there have been few implementations of the calculi. Of these implementations, fewer still have been designed for use on parallel architectures which is both ironic given the aims of the Ambient calculi and also a reflection of the difficulties of programming parallel systems with the languages and tools prominent today.

### 1.2 Contributions

In this project I investigate the variations of the Ambient calculi and examine the reasons why the Actor paradigm is better suited to parallel computation. I treat the Ambient calculi as an assembly language rather than just a modelling formalism and implement a reducing machine or interpreter for an Ambient calculus which I then use as a virtual machine. I design a language based upon the Actor paradigm and create a compiler that is able to translate between my language and the Ambient calculus. Finally I evaluate the software developed and through examples demonstrate how they present a safer model than Object Orientation for concurrent and distributed computation and a more feature-rich platform for mobility.

The platform I have created is capable of transparently adapting to the number of and distribution of the computers and devices on which it is running. It can be used to implement applications that consist of components that can be moved dynamically between devices whilst maintaining communication as necessary and supporting disconnected operation. It is capable of making good use of multi-core CPUs whilst providing a conceptually simpler model of concurrency to programmers. Additionally, in creating a link between Actors and Ambient calculi, I present a means for using the formal methods and logics developed for Ambients for reasoning about Actors and allow the higher levels of abstraction and greater power of expression of Actors to be translated into Ambients.

### 1.3 Report Structure

In chapter 2 I introduce the Mobile Ambient calculus [10] and explain how it is suitable for modelling parallel, mobile and distributed computation. I examine the variations of the calculus and explain the motivations behind them and what their advantages are. Communication and mobility can be statically analysed and given typings in many of the Ambient calculi and these type systems are investigated.

Chapter 3 introduces the Actor paradigm and considers why it is suitable for programming parallel systems. Several languages that have been influenced by or are based on the Actor paradigm are examined

¹[http://www.ps.uni-sb.de/alice/](http://www.ps.uni-sb.de/alice/)
²[http://www.cl.cam.ac.uk/users/pes20/acute/]
and examples show how the Actor paradigm provides solutions that are inherently concurrent and safely composable. Some of the difficulties of the Actor paradigm are also examined.

In chapter 4 I present the GLINTVM which is an interpreter for a variation of the Mobile Ambient calculus, the Safe Boxed Ambient calculus \[25\]. I explain the design and construction of the interpreter and show how its multi-threaded design makes good use of parallel hardware. I present a formal model of the central rendezvous mechanism of the interpreter which demonstrates the correctness of the design.

Chapter 5 presents the GLINT language, a language based on the Actor paradigm but with modifications to try to address some of the issues of the Actor model. Extending the Actor paradigm, I add mobility to the language allowing mobile processes and algorithms to be expressed. By example I show how both classical functions can be concisely expressed and how distributed systems can be constructed.

In order to combine the GLINTVM and the GLINT language, a compiler is needed which is presented in chapter 6. The compiler accepts as input programs written in the GLINT language and translates them into the Safe Boxed Ambient calculus ready for interpretation by the GLINTVM. The encoding of the each component of the language is explained and numerical and truth values are expressed.

Chapter 7 evaluates the design decisions of the GLINTVM and examines how performance of the GLINTVM can be improved. By comparison with JAVA I show how distributed applications are more concisely and easily expressed in the GLINT language and how the GLINT language is better suited for subsequent modification and development of such applications. I compare the GLINT language with the Channel Ambient System \[29\] and explain how the higher level abstractions of the GLINT language offers greater power of expression to the programmer whilst loosing some fine grained control over the mobility of Actors. Performance metrics are presented.

Finally, in chapter 8 I conclude, summarising the contributions of the project and suggesting future directions in which to develop this work.
Ambient Calculi

The Mobile Ambient calculus builds on the $\pi$-calculus [26], limiting the range of communication but adding primitives to model locations and movement that cannot be expressed in the $\pi$-calculus. This makes the Mobile Ambient calculus more suited to modelling parallel computation across a range of locations. The restriction of the range of communication eliminates the assumption made in the $\pi$-calculus that having arranged some shared secret between two processes, those processes can use the secret to communicate without regard for locality. This allows the Mobile Ambient calculus to model firewalls and other restrictions of communication and is thus more useful for modelling distributed computation.

There are several variants of the Mobile Ambient calculus, each successive calculus attempting to address perceived difficulties and weaknesses in the last, sometimes at the cost of expressiveness. What features and what level of expression would make programmers’ lives easier? In this chapter I investigate the various Ambient calculi that will influence the design of my language.

2.1 Mobile Ambients

The Mobile Ambient calculus is the first of the Ambient calculi and was originally presented in [10]. The calculus is simple and powerful: an Ambient is a container for any number of processes and sub-ambients and is named. Restriction rules and scoping issues are as per CCS or $\pi$-calculus and the definitions of free and bound names and variables are again similar to CCS and $\pi$-calculus. Communication is in the style of $\pi$-calculus, is asynchronous on output but does not use named channels. Instead a nameless implicit channel is defined within every Ambient and only processes directly within the Ambient can communicate with one another on this channel.

What Mobile Ambients add, beyond the definition of the Ambients themselves are capabilities. These allow a process to move its containing Ambient into a sibling Ambient, out of its parent Ambient or to open a child Ambient. $n[P] | m[in.n.Q] \rightarrow n[P|m[Q]]$ shows how the in capability works: the Ambient enclosing the in.n.Q process, named m, is transported inside the Ambient named n. This can only happen when such an Ambient as named in the capability exists and is a sibling: if the Ambient n didn’t exist or was at another location then the in.n.Q process would block until a time at which the capability could execute.

The out capability is equally straightforward: $n[P|m[\text{out}n.Q]] \rightarrow n[P|m[Q]]$ shows how the out capability moves the process’s enclosing Ambient (m) out of its own enclosing Ambient (n), provided the Ambient named in the capability does in fact enclose the process’s own Ambient. Again, were this not the case, the process would block.

Finally, the open Ambient dissolves the named process. $n[\text{open}m.P|m[Q]] \rightarrow n[P|Q]$ shows how the opening of the m Ambient has caused all of the processes that resided within m to become processes within n itself and the m Ambient no longer exists.

Example 2.1 shows all of these capabilities in action, along with communication between processes.

Because of the manner in which capabilities are defined, it is syntactically valid to construct Ambients named with a capability, for example, in $m[P]$. This is clearly nonsensical and it is left to a type system to prevent such Ambient names from being constructed.

2.1.1 Communication

In Mobile Ambients, communication takes place on anonymous channels, one per Ambient. The content of communication is defined as either names of Ambients or capabilities acting on an Ambient. It is assumed that if you have received a capability to act on an Ambient then from that you cannot recover the name of the Ambient itself: if this were not the case then upon receiving any capability you could recover the name of the Ambient on which it acts and form any new capability to act on that Ambient.
Example 2.1 Mobile Ambients Example

\[
\begin{align*}
n[\text{in } m . \text{open } l . (x) . x \mid Q] & \rightarrow m[\text{in } n . \langle \text{out } m \rangle] \mid R \\
\rightarrow m[\text{in } n . \langle \text{out } m \rangle] \mid R \\
\rightarrow m[\text{in } n . \langle \text{out } m \rangle] \mid R \\
\rightarrow m[\text{in } n . \langle \text{out } m \rangle] \mid R \\
\rightarrow m[\text{out } m \mid Q] \mid R \\
\rightarrow n[Q] \mid m[R]
\end{align*}
\]

Because communication can only occur between sibling processes within an Ambient, communication with other Ambients is more involved. There are two choices: either one of the Ambients is opened, or a new Ambient is issued from within one Ambient that moves inside the other, is opened by the other and then creates a new Ambient that returns. Whilst clearly more complex, this second form tends to be preferred because it doesn’t result in opening either of the Ambients: instead, only a small controlled Ambient, constructed specifically for such a purpose is opened. Nevertheless, this relies on the receiver of the Ambient trusting the contents of the received Ambient in some way such that it’s happy to open it. Example 2.2 shows both these strategies in use.

Example 2.2 Inter-Ambient communication strategies

(a) Opening the Ambient with whom to communicate

\[
\begin{align*}
n[\text{open } m . (x) . P] & \mid m[\text{in } n . \langle j \rangle \mid Q] \\
\rightarrow n[\text{open } m . (x) . P] & \mid m[\langle j \rangle \mid Q] \\
\rightarrow n[\langle x \rangle . P \mid \langle j \rangle \mid Q] \\
\rightarrow n[P\langle j/x \rangle \mid Q]
\end{align*}
\]

(b) Issuing a new Ambient to perform the communication

\[
\begin{align*}
n[l_1[\text{out } n . \text{in } m . (y) . l_2[\text{out } m . \text{in } n . \langle y \rangle]] & \mid \text{open } l_2 . (x) . P] \mid m[\text{open } l_1 . \langle j \rangle \mid Q] \\
\rightarrow l_1[\text{in } m . (y) . l_2[\text{out } m . \text{in } n . \langle y \rangle]] & \mid \text{open } l_2 . (x) . P] \mid m[\text{open } l_1 . \langle j \rangle \mid Q] \\
\rightarrow n[\text{open } l_2 . (x) . P] & \mid m[l_1[\langle y \rangle . l_2[\text{out } m . \text{in } n . \langle y \rangle]] \mid \text{open } l_1 . \langle j \rangle \mid Q] \\
\rightarrow n[l_2[\langle j \rangle . \text{open } l_2 . (x) . P] \mid m[Q] \\
\rightarrow n[\langle j \rangle . \text{open } l_2 . (x) . P] \mid m[Q] \\
\rightarrow n[P\langle j/x \rangle \mid m[Q]
\end{align*}
\]

Obviously, issuing a new Ambient leads to more complex code. However, note that in issuing the new Ambient, the process Q that originally started in Ambient m remains there, and does not migrate into n as is the case with the simpler strategy. Also note that in example 2.2b, when m receives the Ambient l1 it implicitly trusts it when it opens it: there is nothing to stop l1 from containing processes that monitor and interfere with the proper operation of m.

2.1.2 Objective and Subjective moves

The capabilities in m and out m are subjective rules in that they cause the Ambient enclosing the process to move rather than causing an external Ambient to move. Such movement would be called objective and there is some discussion as to how to best encode this. [10] discusses that it should be considered who
has the authority to move whom. If a malicious party were to compromise a system, what damage could they wreak? With subjective moves, they can move their own Ambient anywhere, but they cannot open or move any pre-existing Ambients. With objective moves however, they could. As such, objective moves are considered unsafe if they existed as primitives within the calculus.

However, a safer version of objective moves can be encoded via subjective moves. It is treated as safer as it depends on a naming convention, in fact, the very same convention-structure that was used in example 2.2b. There, it was required to move a process out of the Ambient \( n \) and into the Ambient \( m \). As discussed above, this was achieved through issuing a new Ambient which was moved into the target Ambient and then the target Ambient opening the newly arrived Ambient. Through restriction, this process can be made completely private as example 2.3 shows: the use of restriction makes private the name of the messenger Ambient \( l \) meaning that no hostile external Ambient can enter \( m \) and be opened by it as it won’t have the correct name.

**Example 2.3** A safe encoding of objective moves using restriction

\[
(\nu \nu v)(n[v[\text{out } n \text{ in } m . P] | Q] | m[\text{open } v]) \\
\rightarrow (\nu \nu v)(n[Q] | v[\text{in } m . P] | m[\text{open } v]) \\
\rightarrow (\nu \nu v)(n[Q] | m[\text{open } v] | v[P]) \\
\rightarrow (\nu \nu v)(n[Q] | m[P])
\]

### 2.1.3 Objective open

The `open` capability is essential for any non-trivial expression within the calculus: without it, the hierarchy of Ambients could move and grow, but could never shrink and worse, no communication could ever take place between processes that weren’t created as siblings within the same Ambient.

Given the arguments made for the use of subjective moves, it may appear surprising that `open` is objective. Again, the reason lies in security considerations: if `open` was subjective, for example: \( n[P|m[\text{open } Q]] \rightarrow n[P|Q] \), then it would be possible for any Ambient to move itself inside any other Ambient and then open itself, gaining access to the processes in the parent Ambient. Making `open` objective prevents this and gives the parent Ambient control over what it is and isn’t prepared to open, allowing the parent Ambient to perform, for example, typing checks prior to opening any Ambient.

The behaviour of the `open` capability leads to some difficulties when building type systems around Mobile Ambients. This is discussed in section 2.4.1 and is one of the chief motivations for dropping the `open` capability.

### 2.2 Boxed Ambients

In [7], Boxed Ambients are presented. The semantics of the `in` and `out` capabilities, the structure of Ambients, the rules for free and bound names and processes are all as for Mobile Ambients. However, the `open` capability has been dropped completely and in its place are four new synchronous communication primitives that can allow communication across Ambient boundaries. These define input from a named child Ambient: \( (x)^* \), input from the parent Ambient: \( (x)^! \), output to a named child Ambient: \( (M)^n \) and output to the parent Ambient: \( (M)^! \). These communication primitives provide functionality that is sufficient to replace the `open` primitive, but their behaviour is simpler making type systems easier to reason about and implement.

The communication primitives are not required to match in any kind of action/co-action manner: for communication across Ambient boundaries, only one of the sending and receiving processes needs to declare the cross-ambient nature of the communication. This makes it possible to send or receive to or from any process or child Ambient. The reduction rules are defined as in figure 2.4.

Also addressed is the issue of named communication channels: it is shown how to encode channels in the style of the \( \pi \)-calculus by using replicated Ambients as channels and then translating \( \pi \)-calculus input and output operations into corresponding operations on named Ambients as shown in example 2.5. Note however that this is not suitable for a general encoding as the channel Ambient \( c \) must be visible to both the receiver and sender, in particular, must be a child Ambient of the Ambients containing the sending and receiving processes. Thus some movement must be achieved in more complex situations.
open resources to the applet via APIs which the applet accesses from within its own Ambient. The lack of

Providing APIs to a downloaded applet in a safe manner using just Mobile Ambients

r

in such a way that no interaction with the applet can reveal to the applet the shared secret, the name

returning Ambients is private. This method is shown in example

with the host in the form of a restricted name which will serve to guarantee that any communication with

Example 2.5 Encoding of \(\pi\)-calculus communication primitives in Boxed Ambients

\[
\langle x | c(y).Q \rangle \Rightarrow \lambda e[!(z).\langle z \rangle | \langle x \rangle^c | \langle y \rangle^c . Q
\]

\[
\rightarrow \lambda e[!(z).\langle z \rangle | e[!(z).\langle z \rangle] | \langle x \rangle^c | \langle y \rangle^c . Q
\]

\[
\rightarrow \lambda e[!(z).\langle z \rangle | e[!(z).\langle z \rangle] | \langle x \rangle | \langle y \rangle^c . Q
\]

\[
\rightarrow \lambda e[!(z).\langle z \rangle] | e[!(z).\langle z \rangle] | Q(x/y)
\]

\[
\rightarrow \lambda e[!(z).\langle z \rangle] | Q(x/y)
\]

(By structural congruence)

The Boxed Ambient is in spirit quite similar to the Seal calculus \([35]\) in which the notation \(e^c(M)\) indicates a request to write \(M\) to channel \(c\) in Ambient (or Seal) \(n\). The Seal calculus can be simply encoded in Boxed Ambients.

Throughout \([7]\), the example of a process running on a host \(h\) downloading an applet \(a\) is used as motivation against opening. In particular: \(a[\text{in } h.Q] \parallel h[P]\). This reduces to \(h[a(Q) | P]\) which shows the result of downloading the applet \(a\). For communication between \(P\) and \(Q\) to occur, the authors argue that it is necessary for either the Ambient \(a\) to be dissolved, thus exposing \(h\) to any process that resided in \(a\), or for \(P\) (or a clone of \(P\)) to enter the applet Ambient \(a\), again exposing \(h\) to potentially harmful processes as the process \(P\) could become compromised.

What isn’t considered is a third possibility which most closely resembles the host providing APIs to the applet: the host is responsible for issuing Ambients that themselves move inside the applet and are opened by the applet. As shown in example 2.3, it is possible for the Ambients issued by the host to share a secret with the host in the form of a restricted name which will serve to guarantee that any communication with returning Ambients is private. This method is shown in example 2.6. It is crucial to design the api Ambient in such a way that no interaction with the applet can reveal to the applet the shared secret, the name \(r\).

Example 2.6 Providing APIs to a downloaded applet in a safe manner using just Mobile Ambients

\[
h[a[\text{open } \text{api}. \langle \text{myRequest} \rangle | Q] | (\nu r)(\text{open } r . x). R | \text{api}[\text{in } a . \langle \text{request} \rangle . r[\text{out } a . \langle \text{request} \rangle]]]\]

\[
\rightarrow h[(\nu r)(a[\text{open } \text{api}. \langle \text{myRequest} \rangle | Q | \text{api}[\langle \text{request} \rangle . r[\text{out } a . \langle \text{request} \rangle]] | \text{open } r . (x). R)]]
\]

\[
\rightarrow h[(\nu r)(a[Q | r[\text{out } a . \langle \text{myRequest} \rangle]] | \text{open } r . (x). R)]
\]

\[
\rightarrow h[a[Q] | (\nu r)[\text{open } r . (x). R | r[\langle \text{myRequest} \rangle]]]
\]

\[
\rightarrow h[a[Q] | (\nu r)(x). R | \langle \text{myRequest} \rangle]
\]

\[
\rightarrow h[a[Q] | (\nu r)[R[\text{myRequest}/x]]]
\]
and typing communication with the downloaded applet simpler.

### 2.3 Communication Interference

In [22], it is discussed that whilst non-determinism can be created in Mobile Ambients purposefully (mimicking CCS’s `+` operator), it can also arise unintentionally. Consider

\[ k[n[\mathit{in} \cdot m \cdot P | \mathit{out} \cdot k \cdot R] | m[Q]] \]

The nondeterminism is assumed to be unwanted and the authors treat it as a programming error. The authors classify a set of these non-deterministic situations they call grave interferences. They go on to develop a variant of Mobile Ambients called Safe Ambients in which the capabilities can only reduce if there exists a matching co-action. So \( n[\mathit{in} \cdot m \cdot P] | m[Q] \) can no longer progress, but \( n[\mathit{in} \cdot m \cdot P] | m[\overline{m} | Q] \) can. Note that the name in the co-action refers to the Ambient which is being crossed rather than the Ambient that is moving. A type system is then used to enforce a notion of *single-threadedness* in which at any time, Ambients can only engage in a maximum of one activity across boundaries.

Not only are these situations also possible in Boxed Ambients, but they are more complex because an input can potentially receive from an output in the current Ambient, in the parent Ambient or in a child Ambient, and dually for an output. Hence a similar grave interference can be constructed for Boxed Ambients:

\[ m[(x)^n \cdot P | n[\langle M \rangle | \langle x \rangle \cdot Q | k[(x)^1 \cdot R]]] \]

In [8] these issues are addressed for Boxed Ambients: input and output now must occur with appropriate co-actions as figure 2.7 shows. It is then necessary to be able to learn the name of any Ambient that enters an Ambient so that the parent Ambient is able to communicate with the new child. This is achieved through renaming the capability to `enter` and only allowing `enter` to reduce when a co-action is present. This is further modified so that the Ambient wishing to enter presents its name which is bound in the co-action by the runtime system. The same alteration is used for `exit`, creating an `exit` capability in which the Ambient leaving must unregister itself with its parent before departing. These rules are then further modified by the addition of a key parameter which must match in the action and co-action and provides the function of a password in a similar manner to the SAP calculus [24].

![Figure 2.7 Reduction rules for communication and mobility in the NBA calculus](image)

These rules, when combined with the presented type system, eliminate communication interference in Boxed Ambients without limiting expressiveness.¹ Note also that the refinement of Boxed Ambients into NBA leads to a simpler type system than is presented for Boxed Ambients in [7]. This is due to communication actions pairing unambiguously with co-actions which means only one side of any communication needs to be tracked and in the paper, this is the upward communication to the parent Ambient.

While communication interferences are effectively eliminated, capability based non-determinism is still present. It is possible to write in NBA:

\[ k[n[\mathit{enter}(m, a) \cdot P | \mathit{exit}(k, a) \cdot R] | m[\mathit{enter}(n, a) \cdot Q]] | \mathit{exit}(n, a) \]

In this form it is assumed however, that the presence of the co-capabilities implies that this was intentional and without either of the co-capabilities, the non-determinism is avoided.

¹Other than loss of expression relating directly to the elimination of grave interferences in communication. Indeed, the paper [8] demonstrates how augmenting NBA with a guarded form of non-determinism (similar to CCS’s `+` operator) specifically applied to communication results in a safe recovery of the loss of expression.
2.4 Typing Ambients

Communication interference can lead to communications occurring which exchange values not intended to be exchanged and which are subsequently misinterpreted. A process may be executing an input where it is expecting to receive the name of an Ambient into which it can move but instead it receives a different unanticipated name. Communication interferences are difficult to spot by hand but fortunately type systems for Ambient calculi can detect communication and movement interferences and can ensure that the values being exchanged are of the correct type.

There are several aspects of Ambients that can be typed and for which type systems have been proposed. The most common type systems are based around source code annotations giving types to messages. These types are called exchange types. There are other type systems for Ambients including the ability to type mobility. This allows the type system to verify that when Ambients move, they move to locations in which they are allowed and is more powerful than simply typing message exchanges.

All of these type systems are based on the same set of ideas. Firstly a type is simply a group or set which has elements. For example, the names of Ambients (note that it is the names of the Ambients rather than the Ambients themselves that are members of the groups). The analogy with Object Oriented languages is that the group is a type and the elements of the group are instances of that type. Furthermore, these groups tend to be parameterised with other types in a manner similar to parameterised polymorphism. The parameters represent for example, the types of messages that may be exchanged within Ambients named by the names in the group; a group exists, collecting together a number of names. Each name names Ambients which all exchange the same type of messages, thus the group is parameterised with that type.

There is always an exchange type called \( \text{Shh} \) which represents the absence of exchanges. For example, the names of Ambients in which no exchanges occur are members of a group \( G \), parameterised by the exchange type \( \text{Shh} \), or \( G[\text{Shh}] \).

The typing of message exchanges often allows for a very simple subtype relation in which the silent exchange type, \( \text{Shh} \), is a subtype of all other exchange types. This means that an Ambient that makes no exchanges is allowed anywhere on the grounds that the processes it contains will never interact with any other processes.

2.4.1 Typing Mobile Ambients

In \cite{9} a series of type systems are proposed, covering typing of messages and Ambient mobility. The type systems are presented in stages, initially giving types to messages, processes (including capabilities) and Ambients and controlling message exchanges. This is then extended to regulate the opening of Ambients and finally crossing control is presented in which the type system is further extended to allow Ambients to regulate which Ambients they may cross: this controls the in and out capabilities.

Some minor modifications made to the Mobile Ambient calculus in this paper. Firstly input and output are extended to polyadic communication; and secondly a simple objective move definition, \( \text{go} \), becomes a primitive operation. This however can be safely encoded in the original calculus as

\[
\langle \langle \text{go} \ N \ . \ M[ P \ ] \rangle \rangle \Rightarrow (\nu k)(k[ N \ . \ M[ \text{out} k \ . \ P \ ]])
\]

2.4.1.1 Exchange Types

There are some subtleties to exchange types. Whilst the typing of parallel processes, restriction, input and output follow in a straightforward manner, the null (0) process can be given any type. This also applies to any child Ambient, the justification for this is that the child Ambient cannot partake in exchanges in the current Ambient as it is a child, so it can be given any type. Similarly, the capabilities in and out can be given any type because they have no bearing on exchanges. This however, contrasts with the typing of Ambient names which must match the types of the processes enclosed by the Ambients named by the name in question.

The \text{open} capability unleashes the processes within a child Ambient and so must be given special attention. Firstly, the type of the \text{open} \( n \) capability must match the type of the Ambient named in the capability that it is opening and secondly, the type of any continuation of the \text{open} \( n \) process must be prepared to participate in any exchanges that the unleashed processes may perform, thus must also match the type of the Ambient opened. This explains why the Ambient names have more restrictive types than the Ambients named by those names: the names are used in capabilities and so have a bearing on the type of the current Ambient whereas a child Ambient itself has no such bearing.
In example 2.8, a possible typing derivation of the process \( n[ (x : W) . P | \text{open} m .0 ] | m[ Q | in n .0 ] \) is shown. Note that the two Ambient names \((n \text{ and } m)\) turn out to have the same type. This is predictable because the processes \( P \) and \( Q \) will eventually become part of the same Ambient and so will interact with each other. As such, they should have the same type.

**Example 2.8** A typing derivation of the process \( n[ (x : W) . P | \text{open} m .0 ] | m[ Q | in n .0 ] \)

\[
\begin{align*}
n[ (x : W) . P | \text{open} m .0 ] & | m[ Q | in n .0 ] \\
Q & : T \\
in n & : Cap[T] \\
in n .0 & : T \\
m & : G[T] \\
m[ Q | in n .0 ] & : S \\
\text{open} m & : Cap[T] \\
open m .0 & : T \\
P & : T \\
(x : W) . P & : T (T \equiv W) \\
n & : G[T] \\
n[ (x : W) . P | \text{open} m .0 ] & : S \\
n[ (x : W) . P | \text{open} m .0 ] & : S \\
m[ Q | in n .0 ] & : S
\end{align*}
\]

### 2.4.1.2 Opening Control

Opening control adds another parameter to the types of Ambients so Ambient types are now represented by \( G^c[G_1, \cdots, G_k, T] \). A member of this group \( G \) is a name which names Ambients within which processes may exchange messages of type \( T \) as before and now may open Ambients named within the set \( \bigcup_{i=1}^k G_i \). This modification to Ambient types similarly affects both capability types and process types which both gain the extra parameter. Thus exercising a capability of type \( Cap^c[G_1, \cdots, G_k, T] \) may unleash exchanges of type \( T \) as before and may now result in the subsequent opening of Ambients of a name within \( \bigcup_{i=1}^k G_i \). The modification to the types of processes can be interpreted similarly.

Due to complexities in subject reduction with opening control there is an additional requirement on the typing rule for the \text{open} capability: the Ambient name named in the capability must belong to a group which itself is a member of the set of groups that parameterise the Ambient type, representing the names of Ambients that may be opened by it. If this were not the case then subject reduction fails without additional subtyping rules, complicating the type system. The consequence of this additional requirement is that if the name of an Ambient is a member of the group \( G^c[H, T] \) and is also a member of the set \( H \) then it can mean two things: as already explained, it can mean that opening an Ambient of this name may subsequently open Ambients named with names within \( H \) or, it may simply mean that the Ambient name names Ambients that are open-able.

This side effect is unnecessary and is only included in order to avoid greater complexity elsewhere in the type system.

### 2.4.1.3 Crossing Control

CROSSING of an Ambient \( m \) occurs when another Ambient enters it via an in \( m \) capability or exits it via an out \( m \) capability. The type system for crossing control treats objective and subjective moves separately and this has a bearing on how types are further modified. The type of an Ambient name is represented by \( G^c[G]^c[H^c, T] \). An Ambient that has a name which is in group \( G \) has within it message exchanges of type \( T \) and may open Ambients named within the groups within \( H \) as before. It now can only cross objectively Ambients within groups \( G' \) and cross subjectively Ambients within groups \( G \). As before, these changes in notation propagate to the process and capability typings.

These changes result in changes to the typing rules for the in and out capabilities, requiring that the name of the Ambient which is named in the capability is a member of the groups \( G \) representing the names
of Ambients that can be subjectively crossed in the type of the capability. The open capability is unchanged other than for the obvious syntactic alterations.

One of the results of this typing system is that correct typing derivations allow the inspection of which Ambients are mobile and which are immobile. This is a conservative result: some Ambients may be indicated as mobile when in fact they are not. One of the main motivations for including objective moves as primitives in the calculus is the objective moves have separate typing rules. The separate typing rules allow the type system to provide a more accurate indication of mobile and immobile ambients for objective moves.

2.4.2 Typing Boxed Ambients

2.4.2.1 Typing process communication

One of the chief motivations for Boxed Ambients is that by dropping the open capability it becomes easier to type the calculus. In [7] a communication typing is presented. The basic structure is similar to the typing of Mobile Ambients, though there are important differences. Because of the different communication primitives from Mobile Ambients, communication needs to be tracked differently. A process is now typed by \( \text{Pro}[E, F] \) where \( E \) and \( F \) represent the types of local and upward exchanges respectively.

Ambients are represented in similar fashion, whilst capabilities are typed as \( \text{Cap}[E] \). The meaning of this is dependant on the capability: the type of in \( n \) is parameterised by the type of the local exchanges of the Ambient \( n \). Moving the surrounding Ambient of the capability, \( m \), into the Ambient \( n \) must not cause exchange typing failures, so the type of the upward exchanges of the \( m \) Ambient must match the type of the local exchanges of Ambient \( n \). The dual explains the typing of out \( n \): the type of this is parameterised by the upward exchanges of Ambient \( n \). This is because if Ambient \( m \) is moving outside Ambient \( n \) then the upward exchanges of \( m \) must be compatible with the upward exchanges of \( n \) as the two Ambients are to become siblings and thus may take part in exchanges with their shared parent.

Because of the ability for a process in an Ambient to target child Ambients for communication, it becomes possible for the typing to allow different types of communication between the parent and the child Ambients. This contrasts sharply with the dual in Mobile Ambients where, due to the need to open a child Ambient to be able to communicate with its enclosed processes, it becomes necessary for all child Ambients, if they are to be opened, to have the same type. This is motivated by the fact that any output must be received by a process in the same Ambient as the output and can be received by any such process. Thus all the processes that can do a receive must be of the same type as the output and by considering that the receiving process could have arrived as the result of opening a child Ambient, it is clear that the typing restriction also applies to child Ambients. This is very much more restrictive than the situation with Boxed Ambients.

2.4.2.2 Typing Ambient mobility

In [25], a novel type system is presented along with the Safe Boxed Ambient calculus. This type system not only types messages but also types mobility of Ambients. Co-capabilities are used in a very similar manner to the NBA calculus (albeit without the password parameter) as described in section 2.3. One difference with the co-capabilities from [8] is that it is allowed for a co-capability to use a wildcard rather than naming the Ambient it is to pair with. Such a co-capability expresses a general level of access, unrestricted by Ambient names.

The type system is based on the usual structure where names of Ambients belong to groups which are parameterised in various ways. Here, an Ambient group has a parameter which expresses the set of groups which contain names of Ambients in which the Ambient in question may reside. Processes and capabilities are further annotated to express the set of groups of Ambients that the process or capability may move the enclosing Ambient to.

Subtyping mobility is also covered in this paper. The idea is to allow a process that may move Ambients to Ambients within the group \( D \) wherever a process than moves Ambients to Ambients within the group \( D' \) is expected and \( D \subseteq D' \).

The addition of the co-capabilities are also accounted for within the mobility typing: the expression of the set of groups in which an Ambient may reside or a process or capability may execute is extended to express a second set of groups which are only populated by the (co-)capabilities and record the groups of Ambients that co-capabilities allow to enter or exit the system. This allows the type system to reject processes that try to enter Ambients for which there is no suitable co-capability.
2.5 The Channel Ambient System

In [29] a further Ambient calculus variant is presented along with an Ambient machine, runtime and simple
programming language.

The in and out capabilities exist without additional parameters but must be paired with co-capabilities
in and out to reduce. Whilst similar to the Boxed Safe Ambients presented in [25], the matching of names
for the in and out capabilities differs significantly, allowing the capability to target both the name of the
Ambient and the name in the co-capability with which to reduce. For example
\[
a[\text{in } b \cdot x \cdot P | Q] | b[\text{in } x \cdot R | S] \rightarrow b[P | Q | a[R | S]]
\]

This shows how the capability in \( b \cdot x \) targets both the Ambient \( b \) and the channel \( x \). This allows better
accuracy when targeting co-capabilities with which to pair at the cost of greater complexity of the mobility
primitives. In common with Boxed Ambients, there is no open capability.

Communication primitives are unique amongst the Ambient calculi, allowing specific targeting of sibling
Ambients for output. The communication primitives are shown in figure 2.9. Communication happens on
named channels as per the \( \pi \)-calculus and there are no primitives for local communication: all communica-
tion primitives are cross-boundary. Despite this, there is also no primitive for output to a child Ambient on
a named channel. For both of these cases, an encoding is provided, as shown in figure 2.10.

Given the presence of named channels, it seems unnecessary to further complicate the well known and
widespread communication model with directionality constraints on actions on the channel. In particular,
the addition of channels removes the programmer’s need to have Ambients move to be able to communicate,
thus reducing the necessary awareness of the location of Ambients. However, adding the directionality
constraints seems to complicates the communication model, requiring that the programmer maintains full
awareness of location, reducing the benefit of having named channels that can cross Ambient boundaries.

Furthermore, given the directionality constraints on the channels, it is impossible to use a channel if the
Ambient in question has moved away from the other Ambients that share the channel as the other Ambients
must not only know where the targets of communication are, but those targets must also be no further away
than a parent or sibling otherwise a messenger Ambient must be issued as with the other Ambient calculi.

On the other hand, one of the key motivations behind the Ambient calculus is to model the limitations
in communication enforced at administrative boundaries, such as firewalls. These limitations are difficult
to model in the \( \pi \)-calculus because of the lack of notion of place and the assumption that if the name of a chan-
nel can be agreed upon between two processes then those two processes can use that name to communicate.
With the Channel Ambient calculus, the limits upon effective use of named channels, defined in terms of the
closeness of two processes, correlates well with this basic motivation to model situations in which the range
of communication is limited. In practice, to avoid cross talk between different communicating processes
with other Ambient calculi, one is often reduced to using the arity of communication (assuming a polyadic
implementation) to uniquely identify and pair inputs and outputs. Adding named channels provides a more
robust and sophisticated solution to this problem.

**Figure 2.9 Communication primitives in the Channel Ambient calculus: definition and reduction rules**

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Sibling Output)</td>
<td>( a \cdot x(v) )</td>
</tr>
<tr>
<td>(Parent Output)</td>
<td>( x^\uparrow(v) )</td>
</tr>
<tr>
<td>(Internal Input)</td>
<td>( x(u) )</td>
</tr>
<tr>
<td>(External Input)</td>
<td>( x^\downarrow(u) )</td>
</tr>
</tbody>
</table>

\[
a[b \cdot x(v) \cdot P | P'] | b[x^\downarrow(u) \cdot Q | Q'] \rightarrow a[P | P'] | b[Q/v/u | Q']
\]

\[
a[x^\downarrow(v) \cdot P | P'] | x(u) \cdot Q \rightarrow a[P | P'] | Q/v/u
\]

The calculus is firmly built around computer networks and differentiates Ambients between sites (phys-
ical computers) and agents. It is impossible, without the aid of a large hammer, to nest physical computers
and that constraint is mirrored in the calculus. Similarly, an agent needs a computer on which to execute
so the hierarchy of Ambients must place sites at the roots of the trees. Other than these constraints, agent
Ambients can be nested as normal.
The author goes on to define the Channel Ambient Machine, an abstract machine for the direct execution of the calculus. The correctness of the abstract machine is verified with respect to the underlying calculus and a runtime is then defined as a direct mapping from the abstract machine to functional code. Finally, the Channel Ambient Language is defined which is a simple programming language for the calculus. The language is imperative but with some functional influences and has a skeleton library API.

The Channel Ambient calculus contributes a model of the nesting of Ambients that is informed by real world limitations of nesting and by adding named channels to communication provides greater power and ease of use to the programmer. Personally, I think that some of the decisions relating to the directionality constraints on communication are questionable and do not really benefit the programmer. Nevertheless, the model is faithful to the motivations behind the original Mobile Ambient calculus and contributes usefully an abstract machine and a useable language.

2.6 Conclusion

The various Ambient calculi discussed here differ primarily in the steps needed to achieve communication between processes in different Ambients. As a result of these variations, the ability to statically analyse properties of programs expressed in these calculi varies in complexity. I intend to use an Ambient calculus as an assembly language and would expect a type system to work in a similar way to the Java bytecode verifier in the JVM. This would aim to assert some safety properties of the code to be executed. However, the additional complexity of the type systems put their implementation outside the scope of this project.

The apparent simplicity of the Mobile Ambient calculus is offset by the complexity of the typing system whereas the more complex communication primitives of Safe Ambients, Boxed Ambients, NBA and Safe Boxed Ambients all provide easier typings at the expense of more complex and explicit programs.

In terms of suitability for programming, my opinion is that the basic Mobile Ambient calculus is unsuitable as the open primitive is so unfamiliar to programmers: there is no similar construct in any language I’m aware of. As such, I believe it would form an obstacle to the use of the language and should be avoided. The Boxed Ambient calculus and the safer variants thereof present more rigid structures that are more instantly familiar to most programmers and are conceptually simpler.
The first publications demonstrating Actors appeared in the late 1970s. It is a model that was ahead of its time and has since fallen out of fashion, presumably due to difficulties of efficient implementation and the delayed arrival of the massively parallel architectures, the programming of which Actors were designed to address. However, today, those architectures are on the horizon if not with us already and once more, much energy is being focussed on addressing the difficulties of parallel programming. The Actor paradigm is an established model which offers a usable solution to the problems surrounding the programming of parallel hardware.

The first available work¹ on Actors is [16], in which a formal semantics is developed with which to model computation with Actors. No language is presented. In [19], control structures are presented in terms of message passing, demonstrating how Actors can represent standard control flow mechanisms. The most complete and comprehensive work on Actors is [1] in which the author presents two simple languages for Actors, a formalism for modelling Actors and then discusses how Actors address the problems of programming for a distributed and concurrent architecture.

The attraction of Actors and hence my motivation to examine the model and its implementations, is that not only is the model designed to deal with some of the same problems I am trying to address, but that there are also strong similarities between Actors and Ambient calculi which suggest they will be a good fit for one another.

### 3.1 Basic Concepts

An Actor is a queue or buffer which is at any one instance associated with a behaviour. A behaviour has an input, this is in the form of receiving a message and it has a finite set of outputs: it must specify a replacement behaviour, it can output a finite number of messages and it can create other Actors.

A behaviour is associated with a queue of messages which is shortened by the behaviour receiving a message from that queue. When the behaviour specifies a replacement behaviour, that replacement is associated with the very same queue albeit one message shorter. The queue has a name and messages are added to the queue by behaviours (from any Actor) sending messages to the name of the queue. Thus a behaviour is a one-shot short lived computation which specifies a future behaviour to process messages from its queue and performs some actions of its own on the message it has received. The Actor is the combination of the tree of behaviours, the queue and the name of the queue. The name of the queue is often referred to as the name of the Actor, the address of the Actor or the mail-address of the Actor.

A behaviour need not receive the first message on the queue. A behaviour specifies some form of pattern which filters the available messages. It will block until a message is added to the queue that passes the behaviour's filter. It is only this message that is removed from the queue: all other messages remain on the Actor's queue because the future behaviour may provide a different filter which allows it to process a message that has already been added to the queue.

The use of a message queue or buffer implies that message sending is asynchronous, i.e. the target of the communication need not be in a ready-to-receive state for the sender to complete the message send action. The send action is complete as soon as the message is appended to the queue and as a result, the queue is modelled as being of unbounded size.

An Actor has a set of acquaintances. These are the names of Actors that it knows. There may be an initial set of acquaintances which are available to the Actor from its creation. Messages the current behaviour of the Actor receives may contain names of Actors which the Actor then adds to its acquaintances. Indeed, in a pure Actor system, everything is an Actor so a message can contain nothing other than names of Actors.

---

¹Actors were first proposed in [20], a paper that is not in any of the digital libraries and which I have not had the opportunity to read.
A behaviour can only send messages to Actors for which it knows the name. It is simple to see that the size of the set of acquaintances can not decrease.

An Actor is inherently concurrent. Upon receiving a message, the current behaviour need only to specify the replacement behaviour before that replacement can start work on the remainder of the queue. All the messages that the behaviour sends can be done in parallel as can the creation of new Actors. There are no requirements placed on the order in which a Actor processes messages and it is guaranteed that messages that are sent are delivered to the named Actor. There are no requirements or limits set on how long delivery will take, the ordering of arrival of messages nor the fairness of the delivery system other than the one guarantee of delivery. There are no requirements on the selection of messages from the queue that pass the behaviour’s filter: if multiple messages pass then any of them can be received by the behaviour.

Listing 3.1 shows the definition of a stack in a pseudo Actor language, adapted from [1]. The Actor defined is termed the receptionist for the stack because all messages sent to the stack will be sent to that Actor. This is easy to see. If the message is a pop message then the future behaviour specified in line 4 is the next Actor on the stack. That means that the value of link will be associated with the queue of the current Actor, thus messages sent to the name of the Actor will be available to the behaviour of link. If the message is a push then a new Actor is constructed that is identical to the current Actor and bound to P. The future behaviour is then defined as a new stack-node Actor with the content sent in the message and the link set to the clone, P, of the current Actor. Again, the consequence is that the new behaviour inherits the current Actor’s queue. Therefore after both possible operations, the new head of the stack is associated with the same queue and name as before the operation.

Listing 3.1 A stack in a pseudo Actor language

```plaintext
1 a stack-node with acquaintances (content, link)
2 receive message
3 if (message.operation is pop and content != NIL) {
4   become link
5   send content to message.customer
6 } else if (message.operation is push) {
7   P = new stack-node with current acquaintances
8   become new stack-node with acquaintances (message.newContent, P)
9 }
```

This behaviour can perhaps be better understood given a different description of a behaviour. Consider instead that a behaviour and an Actor are the same thing, each with their own queues and names. The act of becoming a different Actor, i.e. specifying a replacement behaviour is then defined as forwarding forevermore all messages in and sent to the original queue to the named future Actor. In [1], it is explained that with such a description, an implementation can safely short-cut whenever multiple forwardings are present in a chain. This is safe because there is no possibility of a forwarding either changing its behaviour or its target.

3.2 Benefits of Actors

Given that Actors are designed to address the difficulties of programming highly parallel architectures, it is important to investigate how they achieve this, for which it is necessary to understand precisely what these difficulties are.

3.2.1 Shared Mutable State

If state is immutable then there is no danger in allowing many threads access to the state. For example, in JAVA, strings that are declared using quote marks in the program source code may all be compiled to the very same String Object given the same text of the string. The String Object itself is immutable so there is no need for any mutual exclusion, locking or other concurrency limiting mechanisms: examination of the source code of the String class shows that the only use of synchronisation is in comparison with a StringBuffer which is a class designed for concurrent access and needs synchronisation for proper behaviour.

Immutable state is limiting for general purpose programming; mutable state is needed on the whole. The problem is then how to regulate sharing of that state between multiple threads, particularly at points of modification of the state, but also in languages like JAVA at the point of access given JAVA’s memory model.
3.2. Benefits of Actors

The standard situation of having interleaved access, decision making and updating of the same state from multiple different threads is so standard and well covered I shall not further explain it, needless to say that the case for regulation of access to shared state is undeniable.

The standard tools that have been developed for achieving this regulation are mutexes, semaphores and monitors. All suffer from the same problem: they are not safe under composition. This means that it is possible to construct components that are themselves entirely thread safe, perhaps even provably so, but that when incorporated in a larger system they are no longer thread safe. This is normally due to changes in locking order: in order to prevent deadlock, it is vital that all threads that need to obtain access to a set of shared resources obtain the necessary locks in the same order. This is easy to see: if this is not obeyed then it is possible for two threads to each obtain the very lock that the other thread needs thus preventing either thread from progressing. However, if they both obtain the locks in the same order then any contention between the threads must be for the same lock and significantly the outermost lock that both threads try to obtain, in which case there will be exactly one winner and one loser and no other possibility. The looser cannot then have any impact on the winner obtaining the remainder of the locks. A particular component or library may guarantee that all threads obtain locks in the same order, but there is nothing that the library can do to ensure that no thread enters component having already obtained one or more of the locks regulated by the component. Consequently the ordering of obtaining locks is altered and deadlock can occur. Thus our established mechanisms of regulation of access to shared state are not safe under composition.

3.2.1.1 Transactional Memory

Recently, software transactional memory, \([32, 18]\), has been proposed as a general purpose solution. With transactional memory, there is no attempt to regulate access to shared state at the point of access. Instead, in a manner similar to databases, a transaction is started during which all writes to shared variables are cached. At the end of the transaction, a commit action is executed. This checks to see if any other thread has written to any of the variables read by the current thread during the transaction. If so, the transaction is restarted.

This obviously has implications for side-effecting operations and ideally needs supporting from the operating system upwards (possibly even in the hardware) in order to achieve effective and efficient implementations. However, the performance of prototype implementations, for example in \textit{Haskell} shows very high performance, approaching that of the most cleverly designed traditionally regulated concurrent structures. This is because the model is inherently optimistic: with traditional locking, the locks must be obtained regardless of whether there really is any contention, which takes time. Effectively, with transactional memory, the “lock” is only obtained in the case of contention, although the action of restarting the transaction is potentially far more expensive than obtaining a lock. The cost of restarting the transaction is further amplified by the possibility that it may happen repeatedly: there is nothing to prevent any given transaction from spinning in this way should suitable conditions occur, whereas with traditional locking mechanisms threads block until the lock is available, consuming no unnecessary CPU cycles. Nevertheless, it would seem that the optimistic nature of transactional memory creates the opportunity for recovery of performance due to the dynamic behaviour of the component not causing conflicts requiring transactions to be restarted; behaviour which cannot be safely predicted for lock based code: it is precisely the non-deterministic behaviour of such components that allow transactional memory to make progress where locking system block.

The major advantage of transactional memory is that it is safe under composition: because there are no locks to obtain there are no ordering problems. However, very few databases have implemented nested transactions, there are complex implementation issues regarding nested transactions and the efficient implementation of transactional memory as a whole, as discussed in [15]. Nevertheless, this is a very promising development that may offer an effective solution to some of the problems of programming concurrent systems.

3.2.2 Unshared Mutable State

The chief argument behind the suitability of Actors for concurrent programming is that state is not shared. Consequently, no two threads can ever suffer from the read-decide-update race which motivates locking and thus locking itself is not needed. This would suggest that deadlocks cannot occur and in a precise syntactical sense, this is true. However, because of the ability of an Actor to filter the messages it is sent, it is still possible to create a deadlock in a semantic sense. Actor A sends a message to Actor B and then only accepts a reply from Actor B. Actor B then sends a message to Actor C and then only accepts a reply from Actor C. Actor C sends a message to Actor A and then only accepts a reply from Actor A. This last send will not be received by A and so the reply will never be sent and so a deadlock has occurred.
So if deadlocks can still occur in Actor systems then why are they considered to be a solution? The situation that results in a deadlock requires that messages are chained from one Actor to the next, eventually creating a cycle and that the Actors themselves are set as the continuation, specifying a replacement behaviour that can only accept the message representing the invocation of the continuation, in other words, a reply. If the Actors $A$, $B$ and $C$ instead issued new Actors to act as the continuation then this cannot occur because the new Actors are only known to the behaviour that created them and the Actor that receives the message marking the new Actor as the continuation. Firstly, the original Actors, $A$, $B$, and $C$ are freed up as soon as they have created the new Actors and sent their message, secondly with this technique a cycle will never occur: visualising the acquaintances results in a spiral rather than a cycle, and thirdly, the only way for a deadlock to occur is if one or more of the new Actors is sent more messages than just the one it was designed for, the invocation of the continuation, thus recreating a cycle.

Thus with careful design, it is possible to avoid deadlock in Actor systems. Achieving this is much easier that attempting the same with lock based systems and the resulting components are safe under composition.

3.3 Implementations

There are few languages implementing Actors. A good list of languages and extensions to languages can be found at [12] with further information available from [27]. Many of the Actor systems are very similar and it would not be particularly enlightening to examine many of these in detail. Instead I investigate here three languages, SCHEME, ERLANG and SALSA which have an increasingly close resemblance to pure Actor systems. This serves as an indication of how widespread and universal some of the key concepts of Actors really are.

3.3.1 SCHEME

SCHEME was inspired by Actors and demonstrates the strong links between Object Orientation and Actors. To quote Guy Steele in [33]:

“The SCHEME programming language was born from an attempt in 1975 to explicate Object Oriented programming in terms that Gerry Sussman and I could understand. In particular, we wanted to restate Carl Hewitt’s theory of Actors in words of one syllable, so to speak. One of the conclusions that we reached was that “Object” need not be a primitive notion in a programming language; one can build Objects and their behaviour from little more than assignable value cells and good old lambda expressions. Moreover, most of the Objects in Hewitt’s theory were stateless and unchanging once created; for those, lambda expressions alone were sufficient.”

Steele and Sussman, in their first SCHEME interpreters, implemented both functions and Actors, expressing control-flow in terms of Actors. They then noticed that the function interpretation and the Actor interpretation were essentially identical and therefore there was no need to include both. At this point, they dropped the interpretation of Actors and SCHEME developed in a different direction.

3.3.2 ERLANG

ERLANG² is not formally based around Actors, but the processes in ERLANG approximate Actors. Also, ERLANG is in popular use today predominantly in large-scale distributed soft real-time control applications such as telephony exchanges.

The development of ERLANG is documented in [2] and explains how ERLANG is based on PROLOG but with added support for concurrency. Erlang is a single-assignment language (i.e. within any given scope, only one binding of each variable can occur), with full predicate dispatch ([13, 11]) and a messaging system based on CSP ([5]).

In ERLANG the mechanism for filtering available messages is very similar to how Actors are described as filtering messages and the same basic model of a named queue to which messages are sent and from which messages are selected is used. However, no explicit modelling of behaviours is used and the basic behaviour of a process is sequential: between waiting for messages, there is a total ordering on the actions that a process will execute. This does not violate the Actor model and the actions that are available, creating new processes and sending messages, correlate with Actors. Eventually a process will either terminate or reach another receive statement causing the process to block until a correctly specified message is available. This can be argued is the equivalent of specifying a replacement behaviour, though of course, this only

²http://www.erlang.org/
happens when the current process is instructed to: there is no eager creation of the replacement behaviour as is possible with Actors. Again, this does not violate the Actor model.

Having written some simple programs in ERLANG (including a distributed multiplayer version of the classic Pong⁴), my chief complaint is the lack of structuring features of the language. In many ways similar to PERL, good, clean and maintainable code can be written but it is up to the programmer to enforce good programming practices; in effect, coding by convention. Lack of such structuring tends to result in a spaghetti of receive blocks the control flow between which can be difficult to unravel. This is not a unique problem to ERLANG: I suspect that having worked for so long predominantly in JAVA, I have become accustomed to the more rigid structuring that the type system and other features such as interfaces create. This is certainly not, however, a suggestion that it is harder to write bad code in JAVA, just that there is more help to write well structured code.

ERLANG maintains an explicit separation between function calls and communication. A process can call functions and the thread of that process will itself execute the function as in most traditional programming languages. Message sending is performed using a completely separate mechanism.

ERLANG is implemented by compiling to a byte-code and then using a stack machine to interpret the byte-code. The stack machine itself implements co-routines for the processes, [3, 21], thus not making use of operating system threads. The reason for this is that in the systems that are written in ERLANG it is often the case that many thousands of ERLANG processes exist and if standard POSIX threads were used then the amount of memory allocated for the threads would be crippling. Instead, by effectively implementing its own threading system, ERLANG is able to minimise memory usage. However, this is not without its drawbacks, the most critical of which is that as far as the operating system is concerned, ERLANG is single threaded as it only uses one operating system thread. Therefore, it cannot benefit from thread level parallelism in hardware.

The performance of ERLANG is stunning. This is a result of the domain for which it was developed: telephony exchanges with real-time requirements. Latency of communication is exceptionally low between processes within the same virtual machine instance and there is transparent support for communication between processes in different virtual machine instances. Whilst distributed processes are catered for, there is no mechanism for moving processes: they are trapped inside the virtual machine in which they are spawned.

3.3.3 SALSA

SALSA⁴ is a general purpose Actor based language. It is heavily influenced by JAVA and is implemented as a preprocessor, transforming code into JAVA which is then compiled and interpreted as normal for JAVA. SALSA, presented in [34], is not only based around Actors but also has support for the migration of Actors between different sites. The language builds upon JAVA and, whilst altering the syntax, the JAVA libraries are available to the user, under certain constraints.

The preprocessor produces code that makes use of JAVA threads which modern JVMs will map to operating system threads and hence will be able to make good use of parallel architectures. SALSA directly supports token passing continuations, join continuations and first class continuations. These mechanisms serve to overcome some of the problems of programming in the Actor model, discussed below.

Rather than be based on abstract messages which have to be explicitly received, SALSA’s messages are potential method calls. Thus when an Actor is ready to receive a message, messages on the queue are examined so that a message representing a valid method call to the current Object is selected. Then, by using reflection, the method is invoked.

SALSA does not support the notion of one Actor becoming another; of specifying a replacement behaviour. Instead, Actors are free to change their internal state and this is the only way in which an Actor can alter its behaviour. This makes comprehension of the tree of behaviours much simpler and more intuitive as all possible behaviours are contained in the same definition at the expense of conciseness as often a greater number of state variables will be needed to represent which of the possible states the Actor is in.

3.3.3.1 Continuation Based Programming

All Actor systems make use of message passing in order to affect the behaviour of other Actors. But these messages are one-way: there is no expected reply to a message, nor is that even necessarily possible: in [1], messages are defined such that the receiver of the message is not required to learn, upon receiving the message, who sent the message; it is up to the message sender to decide whether the receiver needs to know

⁴http://en.wikipedia.org/wiki/Pong
*http://www.cs.rpi.edu/research/groups/wwc/salsa/*
who sent the message. In order to achieve the processing of results of messages, continuation passing style must be used. In this, when sending a message to an Actor, the message must contain the name of an Actor to which to send any result. This Actor is the continuation.

When working with Actors in this way, there are broadly two options concerning the continuation:

- To use the existing current Actor as the continuation. This means that the current Actor is unable to process any other messages that are pending for it and is forced to wait until the Actor that it has called replies. At that point it may return to its previous behaviour.

- To create a new Actor specifically for the purposes of the continuation which captures the necessary state such that it is able to complete the processing and reply to the original caller once the reply from the Actor called arrives. This frees up the current Actor which is able to continue processing outstanding and incoming messages.

In neither case is dealing with the continuation as simple and intuitive as the traditional method-call and return-result system that we’re familiar with. Whilst continuation passing style is very elegant and useful in certain situations, it is not a solution that is intuitive and easy to work with in all situations. Thus this restriction made by the Actor systems can result in a loss of comprehension of code as a consequence of the data flow of what are essentially sequential operations passing through a complex and possibly nested sequence of continuations.

It is therefore interesting to see how a modern Actor system such as SALSA has come up with solutions to these problems.

3.3.3.2 Token Passing Continuations

Token passing continuations is the most typical form of continuations and corresponds to the chaining together of a sequence of functions, sending the output of the current function to the input of the next. Listing 3.2, adapted from [34], shows token passing. With token passing, the keyword token refers to the value corresponding to the result of the previous message sent. This is implicitly passed where necessary but can also be explicitly passed so as to correctly order parameters to method calls if necessary. As the token carries the most recent result, its type varies throughout the computation.

Listing 3.2 Token Passing Continuations in SALSA

```
1 behaviour HelloWorld {
2     void act(String[] args) {
3         standardOutput <- print("Hello") @
4         standardOutput <- println("World!");
5         5 <- add(4) @
6         "5 + 4 = " <- append(token) @
7         standardOutput <- println(token);
8     }
9 }
```

In listing 3.2, we see the use of the token passing continuation operator, @. The @ symbol, when placed between statements indicates that the continuation of the preceding statement is the succeeding statement. This ensures a total ordering between statements chained in this way. As can be seen, the token keyword is used explicitly to construct the correct string before sending the println message to the standardOutput Actor, an Actor that is defined by the SALSA system. Note that although the two chains of continuations are each totally ordered, there is no total ordering between the two chains. That means that the result of the listing could place the output of the sum from the second chain between the outputs of "Hello" and "World" from the first chain.

In this example, the current Actor will be able to return to its receive-message loop after performing the first send of each chain. There is no need for the current Actor to block, waiting for computations to complete, indeed, the only way for the current Actor to be further involved in the computation is if it is used as a continuation in the chain, as the next example will demonstrate. Of course, there is nothing to stop the Actor from changing its internal state so that the next message it can receive is the result of a stage of the calculation.
3.3.3.3 Join Continuations

Join continuations collect together the results of an array or list of message sends and only invokes the continuation when all the tokens from the message sends have arrived. The tokens are all sent to the continuation as an array. Listing 3.3, again adapted from [34] shows join continuations in use.

Listing 3.3 Join Continuations in SALSA

```java
behaviour Fibonacci {
  int n;

  Fibonacci(int number) {
    n = number;
  }

  int compute() {
    if (2 > n) {
      return n;
    } else {
      Fibonacci lhs = new Fibonacci(n - 1);
      Fibonacci rhs = new Fibonacci(n - 2);
      join (lhs <- compute(), rhs <- compute()) @
        add @
        return token;
    }
  }

  int add(int[] numbers) {
    return numbers[0] + numbers[1];
  }
}
```

Here, the two recursive calls to the two created Fibonacci Actors can occur in parallel or in sequence, depending on the hardware. All that is required is that the message add is not sent until both the calls to the Fibonacci Actors have completed and sent a result. Here we also see that sending a message to the current Actor does not require the naming of the Actor: the message add is sent to the current Fibonacci Actor.

The construction of the chain of continuations is now only partially ordered as there is no ordering between the two compute messages sent to the Fibonacci Actors. Again, the chain is started and then the current Actor can return to processing incoming messages. In this example, this is crucial: if the Actor blocked in some manner other than in its main receive-message loop then it would not be able to process the add message that it is sent in the course of the computation and thus the computation would block permanently.

3.3.3.4 First Class Continuations

First class continuations is a mechanism in which an Actor has explicit access to the continuation it has been sent as part of the message it is currently processing. This allows it to delegate, passing the continuation to a child computation. This is the default and most explicit form of continuation passing style.

Here too SALSA makes some changes from the traditional mechanism by implicitly binding the passed continuation to the variable currentContinuation. This ensures that the caller and callee do not have to agree in advance on some sort of protocol for passing the continuation around, nor do method parameters at either the callee site and caller site need to be modified to incorporate the continuation. Consequently, it is not at all obvious that a continuation is actually passed within the message as it is implicitly extracted from the call site. Listing 3.4 demonstrates first class continuations.

The program is invoked by a call to the act method. At this point, the compute method is called with the continuation set to print the result, as specified in the act method. In the subsequent recursive calls, the continuation of the compute method will be the join continuation; therefore all calls to compute, other than the first call will pass the result to the relevant join, whilst the first call will finish last, passing the complete result to the standardOutput Actor in a println message, in this case passed implicitly via the value of token.
Listing 3.4 First class continuations in SALSA

```java
1. behaviour Fibonacci {
2.     int n;
3. 
4.     Fibonacci(int number) {
5.         n = number;
6.     }
7. 
8.     int compute() {
9.         if (2 > n) {
10.            return n;
11.         } else {
12.             Fibonacci lhs = new Fibonacci(n - 1);
13.             Fibonacci rhs = new Fibonacci(n - 2);
14.             join (lhs <- compute(), rhs <- compute()) @
15.                 add @
16.                 currentContinuation;
17.         }
18.     }
19. 
20.     int add(int[] numbers) {
21.         return numbers[0] + numbers[1];
22.     }
23. 
24.     void act(String[] args) {
25.         n = Integer.parseInt(args[0]);
26.         compute() @ standardOutput <- println;
27.     }
28. }
```

3.4 Conclusion

Actors seem ripe for further development to take advantage of the available and soon to be available highly parallel architectures. The relative lull in development of Actor systems for the past 15 years provides an opportunity to reevaluate the decisions that were made in the design of previous Actor languages and to investigate what ideas and concepts that have been developed in the mean time, for example with regard to type systems, can be effectively incorporated into new languages based upon Actors. Many of the decisions that were made, for instance in the development of ERLANG, were based around the need for efficiency of implementation. This has resulted in a rich selection of primitive types and a fast, albeit currently slightly flawed, implementation. Given the changes in available memory and speed of processors, the developer of a new language would find themselves in a very different and less restricted situation.

SALSA is a very interesting and very useable system which has gone a long way to try to address some of the issues of the Actor model. As a modern system that draws on years of experience with Actors, it is remarkable how some of the rethinking of Actors, as it is implemented in SALSA, has resulted in a lower learning curve and a more intuitive system.

Nevertheless, there are still issues that I think need to be better addressed in order to make Actor based programming more intuitive. My main concerns are related to the mechanisms added for continuation passing. I personally find the implicit binding of variables such as token and currentContinuation problematic from a purity point of view and I also feel that the chaining of messages and continuations that is possible with token passing and the @ operator does not make it clear enough that the messages sent will not be sent from the current Actor, other than the initial message. On the other hand, the alternative, of lots of definitions of anonymous inner Actors and explicit passing of results and continuations is much less desirable. I would be keen to try to develop a solution between these two alternatives. As can be seen from the example of Fibonacci in listing 3.4, the semantics of the return keyword have also been altered. The semantics are to bind the value of the token variable and to invoke the continuation. This can be seen in the compute method.

ERLANG is a good example of a language that was written to fill a need and has turned out to be successful in a lot of environments. I personally dislike the disparate method calling and message sending mechanisms
and I think that additional language support for structuring the code would be beneficial. At the same time, the language is clearly a success from which much can be learnt.

Object Orientation has become a more familiar and universal paradigm of computer programming and I think it would be sensible to emphasise in any new language the similarities between Object Orientation and Actors, in many ways building on the ideas behind SALSA. This would potentially include the code reuse and structuring features of inheritance, interfaces and other, less widespread techniques.
Whilst the Actor paradigm provides a sound basis for programming highly parallel architectures, the axioms of the language are more complex than one would like from both a formal reasoning perspective and from an implementation perspective. Additionally, the concept of mobile Actors is not formally modelled and only adds to the richness of the axioms.

Therefore, I decided that instead of directly implementing an Actor based language, I would implement a reducing machine for an Ambient calculus and then create a compiler from a higher level Actor based language to the Ambient calculus chosen. The advantages of this are numerous:

- The Ambient calculi have been extensively formally modelled and are well understood. Providing a compiler to allow Actors to be expressed in terms of Ambients will provide the means to reason about Actors using the tools and techniques established and proven for Ambients.

- Ambient calculi are much simpler than Actors systems. Consequently, the implementation of a reduction machine will be more approachable and it will be more readily possible to thoroughly test and establish the correctness of the implementation.

- Whilst Ambient calculi are simpler than Actors, there are enough similarities to suggest that the amount of work the compiler will have to perform will not be prohibitive, i.e. these two models of computation are a close match to one another albeit at very different levels of abstraction and expression.

- Mobility is a core concept of Ambient calculi and so using Ambient calculi to add mobility to Actors is easier and semantically simpler than adding mobility directly to Actors: having defined Actors in terms of Ambients, reasoning about the behaviour of those Ambients once mobility has been added is a known quantity whereas reasoning directly about the mobility of Actors is less well established.

Additionally, there are very few implementations of reduction machines for Ambient calculi and those that do exist predominantly implement processes purely as data structures, using their own internal schedulers and co-processes to reduce Ambient processes in a way similar to Erlang, as investigated in section 3.3.2. Consequently, as with Erlang, such implementations cannot take advantage of parallel hardware.

Finally, no one has ever attempted to express Actors in terms of Ambients before and certainly no one has demonstrated whether it is possible and if so, how complex the encoding is. Thus this is very much new ground. The compiler of my language GLINT will therefore produce a byte-code which is expressed in terms of an Ambient calculus. This will then be interpreted by my implementation of an Ambient calculus reducing machine which is hence my virtual machine, the GLINTVM.

Inspired by the classic software engineering approach of building from the ground up, my first task was to implement a reducing machine for an Ambient calculus...

4.1 Which Ambient calculus?

I chose to implement the Safe Boxed Ambient calculus as defined in [25] and investigated in section 2.4.2.2. The reasons behind my choice of Ambient calculus are not particularly profound. Firstly there has not been an implementation of a reducing machine for the Safe Boxed Ambient calculus before; secondly the calculus has been designed so that it can be typed effectively, in particular the problematic open primitive has been dropped as explained in section 2.4.2.1; and thirdly I realised that the addition of co-actions means that there is great similarity between communication and movement: both require the pairing of instructions, i.e. the pairing of an input with an output or of an action with a co-action. This presents the possibility of reuse of software components which makes for a more attractive calculus to implement.
The Safe Boxed Ambient calculus is defined by the names, locations, values and processes shown in figure 4.1 and the reductions shown in figure 4.2 both of which are adapted from [25]. As is common with Ambient calculi, the asterisk is not used in practice to symbolise local communication. I follow this practice, thus $(x) \cdot P$ is more correctly written as $(x)^* \cdot P$ and similarly, $\langle V \rangle \cdot P$ is equivalent with $\langle V \rangle^* \cdot P$.

The only significant modification that I have made to the definitions given in [25] is that the definitions here support polyadic communication where $(\rightarrow x)$ indicates a comma separated list of zero or more names to be bound by an input and $\langle \rightarrow V \rangle$ indicates a comma separated list of zero or more values to be sent by the output. Correspondingly, reduction must match the arity of the input and output.

**Figure 4.1** Names, Locations, Values and Processes for the Safe Boxed Ambient calculus

| Names: $n, m, \ldots, x, y, \ldots \in \mathbb{N}$ | Values: $V, U ::= n$ name |
| Locations: $\eta ::= n$ names | $| \text{in } V$ enter into $V$ |
| $| \uparrow$ parent ambient | $| \text{out } V$ exit into $V$ |
| $| \ast$ local | $| \text{in } \alpha \quad \alpha \in \{n, *\}$ allow entry of $n$ or of all |
| | $| \text{out } \alpha \quad \alpha \in \{n, *\}$ allow exit of $n$ or of all |
| | $| V_1 \cdot V_2$ path |

| Processes: $P ::= 0$ nil process |
| $| (P_1 | P_2)$ parallel composition |
| $| (\nu n)(P)$ restriction |
| $| !P$ replication |
| $| V[P]$ ambient |
| $| V \cdot P$ prefixing |
| $| (\rightarrow x)^\eta \cdot P$ input |
| $| \langle \rightarrow V \rangle^\eta \cdot P$ output |

### 4.1.1 From \LaTeX to ASCII

Whilst in all the papers on Ambients and in all the examples in this report \LaTeX is used to nicely typeset Ambient calculus processes, a different method of representation is needed for writing programs that are to be executed by the reducing machine, the GLINTVM. Figure 4.3 shows how this encoding works, mapping from the formal modelling syntax to the more computer friendly ASCII representation.

### 4.2 Requirements

The aim of the GLINTVM is to implement the Safe Boxed Ambient calculus in such a way that real operating system threads are used thus allowing the utilisation of parallel hardware. This is in direct contrast to many of the implementations of the Actor model and variations upon it, for example Erlang. In these implementations, processes are purely data structures and are reduced as co-processes in which the language implementation itself chooses a process to reduce and selects between the available processes. As such, these implementations only use a single thread as far as the operating system is concerned and so they cannot take advantage of parallel architectures which is one of the main motivations of this work. Using multiple operating system threads in the GLINTVM makes the implementation much harder: as there are no established and widespread languages that are either based on Actors or other paradigms that are more

¹Or some inferior “equivalent”.
suitable for parallel programming, I must use the typical locking mechanisms for regulating access to state between the threads in the implementation as discussed in section 3.2.1.

As first mentioned in section 2.1, it is syntactically valid to construct Ambients with a capability as a name, for example in $m[P]$. This is clearly nonsensical and it will left to the implementation to detect such constructions. Consideration of the definition of values in figure 4.1 reveals that there are several other similar nonsensical constructions. The implementation must be able to detect these at runtime as they cannot always be statically detected.

The standard requirements of reliable, stable and useable software are very important: it is no good having a functioning and well written compiler if the virtual machine executing the generated byte-code keeps crashing, falling over or otherwise going wrong.

### 4.3 Design

The overall design is quite simple and is shown in figure 4.4. When execution starts, there will be a single thread. As soon as a forking instruction is reached, additional threads will be spawned as necessary. A forking instruction is an instruction that has within it more than one process, for example, in the process

$$(x).\text{in } x. (\nu a)(\langle a \rangle . 0 | a[\langle y \rangle . 0])$$

initially only one thread is needed: that thread is able to control the reduction of the first two instructions, namely the input and the movement. However, the restriction is a forking instruction because within it are two processes which are $\langle a \rangle . 0$ and $a[\langle y \rangle . 0]$. Each of these processes needs its own thread so one new thread will be needed and the original thread can be reused as it otherwise would have no further work to do. Consequently, you can imagine a tree of processes where branching occurs whenever a forking instruction is encountered. Sadly, this is not a finite tree as the effect of replication is of a node with potentially infinite child branches: whilst at any one point there will only be a finite number of branches there is no upper bound.

When examining the available expressions in the Safe Boxed Ambient calculus, it is easy to group the processes into those that require the pairing of processes from different threads (communication, mobility and value prefix) and those that don’t (nil, parallel, restriction, replication and Ambient). Of those that don’t, parallel, restriction and Ambient can be treated in a very similar way, whilst replication and nil require unique solutions. All of the processes that require pairings can be treated in a very similar way.

An Ambient itself, whilst it is a process, does not undertake any reduction. It will eventually terminate and die when all the processes within it have terminated but the thread reducing the last process alive within the Ambient can take care of any cleaning up that is needed when the Ambient itself should be terminated. Therefore an Ambient does not need a thread of its own. A similar argument can be made for a parallel process and restriction with the exception that restriction does have some work to do initially in creating a unique value before the processes within it are started.

![Figure 4.2 Reduction in the Safe Boxed Ambient calculus](image-url)

**Mobility:**

$$m[n[\text{in } m . P | Q] | m[\overline{m} \alpha . R | S] \rightarrow m[n[P | Q] | R | S] \quad \text{for } \alpha \in \{*, n\} \quad \text{RED IN}$$

$$m[n[\text{out } m . P | Q] | R] | \overline{\text{out }} \alpha . S \rightarrow n[P | Q] | m[R] | S \quad \text{for } \alpha \in \{*, n\} \quad \text{RED OUT}$$

**Communication:**

$$\langle \overline{x} \rangle . P | \langle \overline{V} \rangle . Q \rightarrow P[\overline{V} / \overline{x}] | Q \quad \text{where } | \overline{x} | = | \overline{V} | \quad \text{RED COMM LOCAL}$$

$$\langle \overline{x} \rangle^n . P | n[\langle \overline{V} \rangle . Q | R] \rightarrow P[\overline{V} / \overline{x}] | n[Q | R] \quad \text{where } | \overline{x} | = | \overline{V} | \quad \text{RED COMM INPUT}$$

$$\langle \overline{V} \rangle . P | n[\langle \overline{x} \rangle . Q | R] \rightarrow P | n[Q[\overline{V} / \overline{x}] | R] \quad \text{where } | \overline{x} | = | \overline{V} | \quad \text{RED COMM OUTPUT}$$

**Congruence:**

$$P \equiv Q \quad Q \rightarrow R \quad R \equiv S \quad \text{implies } P \rightarrow S \quad \text{RED STRUCT}$$
Figure 4.3 Encoding the Safe Boxed Ambient calculus into AsCI

\[
\begin{align*}
\langle \langle n \rangle \rangle & \mapsto n \text{ where } n \in \mathbb{N} \\
\langle \langle 1 \rangle \rangle & \mapsto ^{^*} \\
\langle \langle * \rangle \rangle & \mapsto \left\{ \begin{array}{ll}
\ast & \text{where used in mobility} \\
\text{where used in communication} & \\
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\langle \langle \text{in } V \rangle \rangle & \mapsto \text{in } \langle \langle V \rangle \rangle \\
\langle \langle \text{out } V \rangle \rangle & \mapsto \text{out } \langle \langle V \rangle \rangle \\
\langle \langle \text{in } V \rangle \rangle & \mapsto \text{IN } \langle \langle V \rangle \rangle \\
\langle \langle \text{out } V \rangle \rangle & \mapsto \text{OUT } \langle \langle V \rangle \rangle \\
\langle \langle V_1 \cdot V_2 \rangle \rangle & \mapsto \langle \langle V_1 \rangle \rangle \cdot \langle \langle V_2 \rangle \rangle \\
\langle \langle 0 \rangle \rangle & \mapsto 0 \\
\langle \langle (P_1 | P_2) \rangle \rangle & \mapsto \langle \langle P_1 \rangle \rangle \land \langle \langle P_2 \rangle \rangle \\
\langle \langle (\nu n)(P) \rangle \rangle & \mapsto \langle \text{new } \langle \langle n \rangle \rangle \rangle \land \langle \langle P \rangle \rangle \\
\langle \langle !P \rangle \rangle & \mapsto \langle \langle P \rangle \rangle \\
\langle \langle V[P] \rangle \rangle & \mapsto \langle \langle V \rangle \rangle \land \langle \langle P \rangle \rangle \\
\langle \langle x^n . P \rangle \rangle & \mapsto \langle \langle x \rangle \rangle \land \langle \langle \eta \rangle \rangle \land \langle \langle P \rangle \rangle \\
\langle \langle (V^n . P) \rangle \rangle & \mapsto \langle \langle V \rangle \rangle > \langle \langle \eta \rangle \rangle \land \langle \langle P \rangle \rangle
\end{align*}
\]

Mobility instructions will be implemented using exactly one thread for each of the action and co-action and the same is true for all of the communication instructions with the exception of inputs and outputs that target named Ambients. As is shown in section 4.7.2.4, these require more than one thread in order to successfully pair with a suitable output and input respectively.

4.3.1 From Static to Dynamic

Once the input, or *program*, has been parsed, it must be reduced, instruction by instruction. The static structure of the processes is different from the dynamic structure, in particular, the static structure does not convey information regarding the results of communication or movement, whereas at the point of reduction, at runtime, such information is crucial. This suggests that from the static structure, a dynamic, or runtime equivalent of each instruction must be generated which performs the reduction and acts on the consequences. In the rest of this section the example process shown in example 4.5 along with the reduction of the process will be used.

Whilst this example isn’t short, it is completely deterministic: there are no points at which a choice of reductions is available. The basic gist is that there are initially two Ambients, \(a\) and \(b\) and another process. This other process tells \(a\) to enter \(b\) and \(b\) to allow \(a\) in. \(a\) then moves inside \(b\) and then is told by \(b\) to exit \(b\). \(a\) instead, creates a new Ambient \(c\). \(c\) then tells \(a\) to tell \(b\) that \(c\) wants out of \(a\) and additionally out of \(b\). Finally \(c\) makes the moves it wants ending up outside all the Ambients. Note that there is no step in which the Ambient \(c\) is constructed or the restriction \((\nu c)\) is executed. Also notice that when \(b\) and \(a\) have no more processes in them then they terminate and vanish and finally observe that the replication is unwound as needed when \(b\) outputs to its parent. Again, no step explicitly corresponds to the unwinding of the replication. Unwinding is defined as the reduction from \(!P\) to \(P \mid !P\).

The graph in figure 4.6 shows the process diagrammatically. The nodes contain the process that the thread reaching the node is to reduce and the edges link the various stages of reduction. Forking instructions can very easily be identified here as nodes that have more than one edge departing the node. The diagram shows that the replication need only be unwound when the previously unwound replicated process (i.e. the process that is replicated: \(P\) in \(!P\)) has reduced in some meaningful way. Quite what constitutes meaningful will be defined later. Notice that in this form, there is explicit consideration of replication, restriction, parallel processes and Ambient construction.

Where the next instruction to be executed in a process requires a pairing, this is indicated by a node
with a double border. As can be seen, the vast majority of instructions involve a pairing. As mentioned earlier, a value prefix (i.e. an instruction that is a name) requires a pairing. By examination of the definition of values, it can be seen that the value prefix will be substituted, as a result of an input, into either an action, a co-action or a path. A path can only consist of actions or co-actions or paths and eventually, all the paths will be expressed in terms of actions and co-actions. Therefore ultimately, only actions and co-actions can be substitutions for a value prefix and thus a value prefix will require one or more pairings in order to reduce. It is a runtime error if the substitution of a value prefix is to a name as it is impossible to execute a name.

Diagrams such as this one represent the static reduction plan of the process: here no attempt is made
to identify what will pair with what, what substitutions or α-conversions will take place or the effect of replication or restriction. What is identified clearly is the relevant instruction that must be executed next for each process to reduce and the result of that reduction in terms of the remainder of the process still to be reduced.

As the diagram shows, in the case of replication, the same process may need to be reduced more than once as a result of the unwinding of the replicated process. This provides motivation for the progressive generation of a runtime equivalent of an instruction from its static representation so that no two threads executing instructions are required to share the same instruction Object. The runtime Objects that represent instructions contain state necessary to achieve the execution of that instruction. Therefore it is not safe for more than one thread or process to make use of the same runtime instruction Object (or in fact for any such runtime Object to ever be reused). In contrast to this, after parsing, the Objects representing the instructions statically are effectively immutable, thus it is safe for them to be shared by multiple threads and for them to generate unique runtime instruction Objects for the process’ threads.

Thus the general plan of reduction can now be stated:

1. Parse the input and verify it syntactically.

2. From the input, produce a static representation of the process. This is equivalent to the diagram in figure 4.6 where the edges represent pointers between Objects and the Object represents the instruction to be executed next.

3. Incrementally, on demand generate a runtime equivalent of the remaining process to be reduced. For example, given the process \((x) \cdot (x) \cdot x \cdot 0\), having created a static representation of this process, the runtime equivalent would be an Object representing the input \((x)\) with a pointer to the remainder of the static process, \((x) \cdot x \cdot 0\). Once the input instruction has been executed, the remaining process is
transformed into its runtime representation: a runtime Object that performs the output $\langle x \rangle$ with a
pointer to the remaining static process $x . 0$.

4. Whilst there are processes that have not been reduced to 0, continue the runtime-generation reduction cycle.

The result of this design is that it is safe for multiple threads to be reducing the same process as they
will not be sharing any of the Objects or state that actually represent the instructions at runtime, and that
no static instruction Object needs be aware of its location, for example which Ambient it is in, or what the
results of the substitutions performed so far are etc: in short, static instruction Objects have no knowledge
of the state of the reduction of the process. This correlates accurately with the diagram in figure 4.6 as the
remaining process is just successively shortened whilst being unaware of the effects of the instruction that
has been executed. This is very important because it means that, in the case of a replicated process, the
consequences of the reduction of one unwinding of the replication cannot affect the next unwinding which
is precisely as intended.

4.3.2 Substitutions, Names and $\alpha$-conversions

Process algebra use substitution to modify processes as the result of binding operators. A binding operator
is an instruction that causes a substitution in which the name indicated in the operator is replaced systemati-
cally throughout the remaining process with some value. As with CCS and the $\pi$-calculus, the Ambient cal-
culus has two binding operators, input and restriction. For example, given the process $b[(in\ a)\ .\ 0 ](x)\ .\ x\ .\ 0 ]$, the input $(x)$ is the binding operator and when it is paired with the output $(in\ a)$ and the two sub-processes are reduced, a substitution is performed in which all occurrences of $x$ in the remaining process are replaced with the value $(in\ a)$ leading to the resulting process $b[ 0 \mid in\ a\ .\ 0 ]$.

However, if the result of the substitution is that two names that were previously distinct are now no
longer distinct then an $\alpha$-conversion must take place to ensure the distinction between names is preserved.
For example, in the process $(x)\ .\ (y)\ .\ x\ .\ y\ .\ 0$, the two inputs bind different names, $x$ and $y$. However, if composed with another process to form $(y)\ .\ (g)\ .\ 0 \mid (x)\ .\ (y)\ .\ x\ .\ y\ .\ 0$ then the result of the first reduction would naïvely be $(g)\ .\ 0 \mid (y)\ .\ y\ .\ y\ .\ 0$ because all occurrences of the name $x$ that remain in the process after
the binding operator are replaced with the value, $y$. But consequently, what were previously two distinct
names, $x$ and $y$, are now a single indistinguishable name, $y$. The next reduction leads to the process $0\mid (g)\ .\ g\ .\ 0$ which is not what was intended.

Intuitively, the result of the two communications should be the process $0\mid y\ .\ g\ .\ 0$ and in order to achieve
this, the distinct names must be maintained so that the result of the second input cannot affect any of the
bindings made by the first input. This is the purpose of $\alpha$-conversion. With $\alpha$-conversion, at the point of
binding by the first input, you must analyse the remaining process for any name that is equal to the value
that is being used in the substitution. In this case, you would detect that the name $y$ is used already in the
process and that it is indeed equal to the value that is being used in the substitution. Therefore, an
$\alpha$-conversion is created in which all such names, in this case $y$, are substituted with some unused name that
is also not equal to the value of the substitution. This $\alpha$-conversion then takes place immediately prior to
the substitution created by the binding operator.

So having detected that $y$ is indeed used in the process and that it is equal to the value $y$ in the
substitution, a fresh name $z$ is chosen and a substitution of $z$ for $y$ is performed, creating the process
$(y)\ .\ (g)\ .\ 0 \mid (x)\ .\ (z)\ .\ x\ .\ z\ .\ 0$. Now the original substitution can safely take place in which $y$ is substituted in place of $x$ as a result of the input $(x)$ pairing with the output $(y)$, reducing to $(g)\ .\ 0 \mid (z)\ .\ y\ .\ z\ .\ 0$. Now the
number of distinct names is the same before and after the execution of the binding operator. The second
input can now reduce with the second output creating the process $0\mid y\ .\ g\ .\ 0$ which matches the intuition
about the intended result of the communication.

The advantage of this technique is that it makes the formal models simpler: it is not necessary to main-
tain any sort of mapping from names to values when performing reductions and the results of reductions,
in particular communication, are far more obvious as they are reflected immediately in the representation
of the process rather than in some lookup table. However, from an implementation perspective, this notion
of examining processes to detect clashes of values and names as a result of substitutions caused by binding
operators is complex and expensive. The purpose of $\alpha$-conversion is to maintain distinction between names
before and after substitutions. If no substitutions are ever made then $\alpha$-conversions are not needed because
nothing other than substitutions can affect the distinction between two names. So instead of using substitutions,
a simple mapping will be used. This will map the names in the processes to their values. Initially, a
name will be mapped to itself, but as a result of binding operators, the mapping will be modified.

Returning to the diagram in figure 4.6, initially the mapping is empty. As each name is encountered, if
there is no existing entry in the map for that name then a default entry will be added in which the name
is mapped to itself. Thus the first reduction on the left hand side, when creating the Ambient $a[\ldots]$ will
be to add a mapping from $a$ to $a$. The next reduction of the process in which the input $(x)$ is reduced will
create a mapping from $x$ to in $b$, following the reductions shown in example 4.5. At the forking points, the
mapping will be cloned and each branch will be started with its own mapping. The mapping will reflect
all the bindings up until the fork, or equivalently all the substitutions made by the binding operators until
the point of the fork, at which point the various processes created will reduce with independent mappings.
This corresponds to the behaviour of substitutions: a substitution only affects the remainder of the process
left after the current binding operator has been reduced and cannot affect any other process. By having
independent mappings, the effect of a binding operator in one process cannot alter the mapping in another process.

It is easy to see why this mapping is not used for formal modellings as it can lead to tremendous confusion and ambiguity. Given the process

\[ a[ (x) . (y) . (x) . (y) . 0 ] ] (\nu b)((\in b)^a . 0) ] (\nu b)((\in b)^b . 0) ] \]

the two restrictions produce communications that, along with a mapping, result in the process

\[ a[ in b . in b . 0 ] ] \]

The mapping would be required to distinguish between the two bs but visually it is impossible to assume that they refer to different values. Because the value of the bs are defined by restrictions in different processes and hence with different mappings, prior to the communication taking place, the in b actions that are to be transmitted capture the value to which b is mapped in their process’s map, hence capturing the value generated by the restriction previously reduced by their process and from that point on they never change the value which they have captured. This means that effectively they ignore the fact that they were defined with the name b and any mapping that may redefine the value of b in any process in which they are used. In this way they, whilst visually looking identical, carry within them the actual value that the name b was mapped to at the point at which the action was transmitted and hence the result of the substitution caused by the respective restriction. With the original model of substitutions, \( \alpha \)-conversions would ensure that the value created by the restriction and hence the value used by the actions would never be further substituted by a binding operator. In particular, with restriction, the restriction binding operator would float outwards and would then itself undertake \( \alpha \)-conversion in order to remain distinct. Once more, careful comparison with the behaviour of a substitution shows that the behaviour of the mapping correlates correctly with the behaviour of substitutions.

Thus there are some similarities with traditional programming languages and some dissimilarities. The similarity is that it is possible to implement a mapping from (effectively) variable names to values, with some provisions regarding the capturing of values for names that are going to be communicated between different process, or in more familiar terms, different scopes. These conditions are very similar to the behaviour of closures in a number of widespread programming languages. The main difference is that whilst there are scopes and whilst they can be thought of as being stacked on top of one another (and also splitting and forking for separate threads which is familiar to standard languages), the model presented always flattens the scopes so no stack is present because there is never the possibility of returning to the previous scope: there is no call and return structure, or, put another way, the number of mappings from names to values in any one map can never shrink.

4.3.3 Hosts

As briefly explained in section 2.5, it is quite natural to map particular Ambients onto hosts. This is crucial if the programmer wants to be able to express the movement of Ambients between hosts and given the nature and the targets of this project, this is certainly necessary.

None of the Ambient calculi apart from the Channel Ambient System treat host Ambients in any way differently from non-host Ambients: in fact, none make such a distinction at all. Consequently, the abstract models presented allow endless possibilities of nestings and hierarchies without any consideration of what changes and restrictions are necessary given some notion of host Ambients.

To be able to express movement between two hosts, an Ambient would have to target that host and treat it as an Ambient name. In particular it ideally should issue an out action to move it out of its current host Ambient and then an in action to move it into the target host. In such a model the host Ambients would be immovable and would form the roots of a many rooted tree expressing the hierarchy and nesting of the Ambients.

The problem with this model is that movement of an Ambient is subjective: the Ambient itself issues the instruction for it to be moved rather than some external process objectively moving the Ambient, see section 2.1.2 for further discussion of this point. Consequently, having moved out of a host Ambient, the Ambient is now not in any host and so cannot reduce: it does not have access to any mechanism or processing element that can execute its next instruction, namely the move into the target host. So having moved out of its originating host it is now stuck, unable to make any further progress. Indeed, if it were able to make further process then it would have to have access to some sort of processing element which would mean that it is indeed inside a host, some kind of all encompassing ethereal host. But such a concession only gains one additional level of computation: what if the Ambient wanted to move outside of this ethereal
host too? It would find itself back in the very same situation. So the only lasting solution to this model is an infinite nesting of ethereal host Ambients, which given their ethereal nature, would cause naming issues along with cost and physical construction issues: just how do you infinitely nest computers?

The problem is that the Ambient that wants to do the moving is required to move out of its current host before attempting to enter the target host. Dropping this requirement, whilst making a complete mess of the relationships between the host Ambients, allows the non-host Ambients to achieve their mobility aims. Consider the following:

\[
\text{host:rose} \alpha [\text{in host: holly}. \text{0}] | \text{host: holly} [\text{in host: rose} : \alpha . \text{0}]
\]

Here, there are two hosts called rose and holly and one non-host Ambient, \(\alpha\). With this new model, this reduces to

\[
\text{host:rose}[] | \text{host: holly} [\text{0}|\text{a}[\text{0}]]
\]

But to look at the semantics of reduction of \(\text{in}\) would show that for an \(\text{in}\) to pair with an \(\text{in}\) requires that the Ambient that containing the \(\text{in}\) instruction is a sibling of the Ambient containing the \(\text{in}\) instruction. Thus this implies that the \(\alpha\) Ambient is a sibling of the \(\text{host: holly}\) Ambient, which is clearly not the case.

Thus some alteration of the semantics of the mobility instructions for inter-host movement is required. Firstly, as already demonstrated, when the \(\text{in}\) instruction targets a host then that host must be a sibling of the \(\text{parent}\) of the current Ambient, or in other words, for reduction to occur, the Ambient must be directly within the top level host Ambient and not any deeper in the hierarchy of Ambients. At the receiving side, this is partially counteracted by the fact that the co-action must now name both a host and an Ambient in the form \(\text{in host: rose} : \alpha\). The host name is required as this seems sound from a security point of view but in a similar manner to how co-actions are already defined, the Ambient name \(\ast\) can be used indicating any Ambient from the named host is permitted entry.

It is also the case that the host Ambient itself can not execute any action at all as that would imply that the host Ambient is attempting to move. With this model of hosts, the hosts are always stationary and are always the top level Ambients. Only Ambients within hosts are capable of movement.

It turns out that these are the only changes needed: I have already discussed and rejected a model in which the \(\text{out}\) action is used for inter-host mobility and there are therefore no other possible changes to consider. It can be seen that the use of the notation specifying hosts is only valid when combined with the \(\text{in}\) action and co-action as shown and invalid when combined with the \(\text{out}\) action or co-action.

Finally, the ability to communicate host names must be considered. The solution I have chosen adds the two forms \(\text{host: rose}\) and \(\text{host: rose} : \alpha\) where \(\alpha \in \{n, \ast\}\) to the definition of values in figure 4.1. This allows communication to be as flexible as possible where concerning hosts. The only further complication is that substitution is required to be able to operate on both the host name part and the Ambient name part of a definition in the form \(\text{host: rose} : \alpha\). This allows the following reduction:

\[
\langle \text{host: rose} \rangle . (m). 0 | (h) . (n) . \text{in host: m} . 0 \\
\rightarrow (m). 0 | (n) . \text{in host: rose} : m . 0 \\
\rightarrow 0 | \text{in host: rose} : m . 0
\]

and the following is also permitted:

\[
\langle \text{host: rose} : \ast \rangle . 0 | (hn) . \text{in host: m} . 0 \\
\rightarrow 0 | \text{in host: rose} : \ast . 0
\]

On the other hand, the process \(\langle \text{rose} \rangle . 0 | (h) . \text{in host: m} . 0\) is not valid. This is not because of anything troublesome with this particular process, but consider specifying a fuller host-name, for example \(\langle \text{rose} . \text{bushes} . \text{garden} \rangle . 0 | (h) . \text{in host: m} . 0\). Here, the host-name \(\text{rose} . \text{bushes} . \text{garden}\) is indistinguishable from a path and so creates ambiguity. Thus in order to communicate a host-name, it must be explicitly marked at the output as a host-name.

### 4.4 Testing Non-determinism

The use of multiple threads in the reducing machine means that it becomes much harder to build a test harness around the machine as the actions of multiple threads must be tracked. The solution was to construct two observer-pattern implementations at different stages. The first is the \textit{generation} observer. This is triggered when a static instruction Object is told to generate a runtime instruction equivalent. The newly
created runtime instruction is sent to the observer before being returned. This allows the generation observer to be utilised by the second observer-pattern, the start-stop observer. Every single runtime instruction implements this start-stop observer-pattern and it allows for a subscriber to be informed when execution of the instruction starts and when it completes. For example, with an input, execution starts when the input is the next instruction to be executed and it ends when the input has been successfully paired, values exchanged and mappings updated. An Ambient starts as expected when it is created and stops when there are no more processes within it.

Thus the generation observer allows for the subscription to the generation of runtime instructions from static instructions and the start-stop observer allows for the subscription to when instructions start being executed and when they stop being executed. In a flash of software re-usage, these observers also turn out to be very important for replication, as explained in section 4.6.

Therefore, the testing system, by using these observers, is able to observe across all the different processes and hence threads just when individual instructions are started and stopped. Whilst putting these events into a list (suitably synchronised) would create a total ordering of events, the ordering would not be repeatable. For example, in the process $a[0] \mid b[0]$ it may be that $a$ starts before $b$ in one particular execution and that $b$ starts before $a$ in another. All that can be tested in this case is that $a$ starts before it stops and $b$ starts before it stops. A more complex example, $(\langle 1 \rangle . 0 \mid a[ (x) . 0 ]$ allows for the testing that $a$ starts before $a$ stops, the input starts after $a$ starts, the input stops before $a$ stops and most crucially, the output starts before the input stops and the input starts before the output stops. Note that there is no ordering between either the starting of the input and the output or the stopping of the input and output: either could start before the other and either could finish first. Indeed, if the thread scheduler or hardware chose then the whole of $a$ could stop before the output had stopped. What this would mean is that the input and output had paired correctly and exchanged values but then the output had received no more CPU time and so had been unable to fire the stop observer, or, in another situation, the rest of the input process had been able to run so fast that it completed and shut down a before the output process had stopped.

These subtleties make the testing of more complex processes very hard as it requires a detailed understanding of the partial orderings between instructions. There were many occasions when a newly written test would initially pass and then some days later would fail, without any modification of the relevant code. The culprit was the specification within the test of orderings that were not guaranteed. For some time, the behaviour of the hardware and operating system scheduler had ensured that these orderings were honoured but, for example upon using different hardware, different orderings occurred thus causing the test to fail.

Listing 4.7 shows one of the tests that I wrote. You can see the definition of the process to be executed plus three checks that ensure orderings. The checks list in chronological order events that should occur, either starting events or ending events. In many ways, the valid orderings correspond to the order in which instructions are executed as shown in the diagram in figure 4.6. In addition to these orderings which are obtained purely from the static structure of the process, other orderings can be ensured based on communication and movement. For example, the listing shows that it is checked that Ambient $x$ starts before restriction $g$. This is guaranteed because the restriction $g$ can only start after the input $z$ has stopped and that can only happen as a result of the communication with the output instruction in the process within the Ambient $x$. Therefore, $x$ must start before the restriction $g$ starts. Also notice that unlike in figure 4.6, information about the results of communications are available. Note that there is a check to ensure that name $y$ starts before the in $g$ action. The action is communicated and bound to $y$. Thus when the name $y$ is executed, it looks up the value to which it is bound and executes that. Therefore, to a large extent, the orderings can check pairings made as a result of communication.

### 4.5 Process Containers

A process container is a container for a number of processes, corresponding exactly to the definition of a forking process as described earlier. Parallel, restriction and Ambient are all process containers and replication is implemented as one although it is fundamentally distinct and so will be discussed separately.

The runtime process container Objects, when executed, generate runtime instructions for all the processes they contain and then start executing those processes all in separate threads. The processes are supplied with their own mappings from names to values which are duplicates of the mapping that the process container received upon execution. One process is kept aside and not started with its own thread, instead the thread that is currently executing the container will execute this particular process when it has finished starting up all the other processes in the container.

As each process is reduced, at the end of every instruction in the process, the instruction will contact
the container in which it exists and inform the container which instruction has been completed and what the next instruction is to be executed. There is no traversal of containers with this: the “last” instruction in a process is 0 or a container; an outer container will never be informed of the activities going on within an inner container.

In this way, a container keeps track of how many processes are executing within it. When every process within the container has informed the container that the last instruction of the process has been completed, the container itself is finished. Thus it contacts its container, i.e. the container in which it is “executing”, and informs the container that it has itself completed.

This is the basic behaviour of a process container and, given the simplicity of parallel, also the behaviour of the runtime Object representing the parallel instruction. However, Ambient and restriction instructions require slightly more complex behaviour.

### 4.5.1 Ambients

Ambients inherit their basic behaviour from the process container as defined above. The initial difference with the process container is that the Ambient must add a mapping for its name to the maps that are passed on to the contained processes. Upon closer inspection, the Ambient must provide a massive amount of extra functionality.

Communication is defined in terms of in which Ambient or between which Ambients is the communication taking place. Therefore the Ambient must provide mechanisms to facilitate the pairings of inputs with outputs. Similarly, movement uses the names of Ambients as targets and so once more, the Ambient is the logical place in which to put mechanisms for pairing actions with co-actions. These mechanisms are discussed in section 4.7.

### 4.5.2 Restriction

A restriction creates a new and unique value to which the name indicated in the restriction is substituted. In my model, this is simply a matter of either creating or updating the mapping from the name to its new value and then deferring to the general behaviour of the process container as described above.

Because substitutions are not used in my implementation, the unique value generated must be globally unique because there is no possibility of checking that the generated value doesn’t clash with another value in
4.6. Replication

Replication is an instruction that is crucial to the Ambient and \( \pi \)-calculus. Without replication, neither calculus would be Turing complete as it would not be possible to express any sort of loop in either calculus. CCS does not use replication, instead it allows for named processes to reference themselves, thus permitting recursion.

A replication can never terminate as it represents infinite copies of the replicated process in parallel, where the replicated process is \( P \) if the replication is \( !P \). The fact that they are all in parallel explains why the replication instruction is implemented as a process container. In practice, it is very much ill advised to try to create infinite copies of the replicated process out of consideration for available memory and CPU resources. The ideal solution is that the replicated process is replicated only when the previously replicated process performs a reduction. For example, given the process \( !P \), there is an initial replication, resulting in \( P!P \). Then, only when \( P \) is reduced, e.g. if \( P \) undergoes some reduction, \( P \rightarrow P' \) so resulting in \( P'!P \), is

any mapping to which it is added. Whilst such a check might be possible at the point at which the restriction is generated, it is not possible after communication of the restricted value as it would be necessary to track that the value was generated by a restriction and would then possibly need to modify itself or some other value if a clash were to be found, corresponding to an \( \alpha \)-conversion. Given the ability of an Ambient to move between hosts it would then require a large and complex protocol to make sure that where modifications of the restricted value are needed, they are applied globally. This would almost certainly require a global locking mechanism to ensure this happens safely which has massive ramifications for the complexity and success of the implementation.

Thus a much simpler scheme is employed. Firstly, syntactically, the value of a restriction is completely distinct from any value that can be expressed in the Ambient calculus. This is as a result of specifying tightly in the parser precisely what can and can't be a name or value and then using a much wider type in the actual implementation. To be precise, the parser only accepts as names text that starts with an underscore or a lowercase letter. But in the implementation, Java's general String class is used which can represent a much wider set of textual values. Therefore, by using a representation that cannot be approached by any other value in the Ambient calculus, the only possibility of a clash is between two values both generated by restrictions.

Within each host, there is a restriction generator that only permits access sequentially (i.e. no concurrent access). This then employs a tweak map which is effectively used as a set, caching generated values. By virtue of it being tweak, the reference to the generated value in the set does not preclude the value from being garbage collected. I.e. the garbage collection algorithm in Java effectively ignores entries in the set and so if there are no other pointers to the value that are reachable, the value will be garbage collected and removed from the set. Thus, when each value is generated, it is checked to see if that value already exists in the set. If it does then another value is generated and checked and so on until it is ascertained that the generated value is not in the set and is thus unique, at which point it is added to the set and returned to the restriction instruction.

But this is only local to each host, there is no guarantee that two hosts would not generate the same restricted value. This only becomes a problem if those two values meet and affect each other's behaviour in some unintended way. The solution to this problem is a probabilistic one. The generation of the value is performed by a random number generator. Two random 32-bit numbers are generated and concatenated textually thus creating a random 64-bit value. Random number generators are required to generate numbers with uniform distribution over an infinite sampling. The odds of any one particular value being generated are therefore 1 in \( 2^{64} \). The odds of that same number being generated twice are 1 in \( 2^{64} \) or 1 in \( 2^{128} \) which is otherwise expressed as a probability of about \( 3 \times 10^{-39} \). This is quite small and so it is assumed that for practical purposes the chances of two values generated being equal and of those two values causing undesired consequences are vanishingly small, so small in fact as to be ignored. Of course, if greater certainty is needed then a greater quantity of random numbers could be used for each value thus producing a bigger range of values and a smaller chance of a collision.

The reader may wonder why a simple counter is not used for the generation of restricted values. Whilst this would certainly result in an implementation free from collisions within the same host no such guarantee can be made for restrictions that were generated in different hosts. The existing solution already provides freedom of collision within the same host so nothing is gained. Furthermore, the most naïve solution would make the counters all start from the same number in each host which would massively increase the chances of a collision. Finally, a counter solution would be less secure as from one restricted value it would be possible to predict the next.
the next replicated process created and started, hence \( P' | P | !P \). In this way resources are not unnecessarily consumed.

In order for this to happen, the replication must be informed whenever any instruction in the current replicated process completes. But which instruction will complete first can be non-deterministic. For example, in the process \( !a[(x).0] | (n)^a \cdot 0 | (m)^a \cdot 0 \) either of the outputs or the input could be the first instruction to complete. It is furthermore vital that only one replicated process is created in total from any number of reductions that occur in the previously generated replicated process otherwise unrequired copies of the replicated process will be created.

By using the two observer patterns as described in section 4.4 the replication can learn of the first instruction to complete. But no matter how fast it unsubscribes from the rest of the instructions, one or more of those instructions may in the mean time have also completed and are waiting to tell the replication. The solution is a somewhat cunning algorithm.

Firstly, the replicated process is never started directly within the container that is the replication. Instead, artificially, another container, a simple parallel process container is used to contain the replicated process. This extra container is created for every copy of the replicated process made and is added to the replication’s container. Whenever an instruction is generated within the replicated process, the observer patterns ensure that the replication learns of the new process. Not only does the replication subscribe to the start-stop observer on the process itself, it also adds to a map it maintains a mapping from the newly generated instruction to the parent container in which the instruction is executing. For example, in the process \( !a[(x).0] | (n)^a \cdot 0 | (m)^a \cdot 0 \), firstly, the artificially added parallel container will be mapped to the replication. Then, the two outputs will be mapped to the parallel container as will the Ambient. The input, when it is generated as a result of the Ambient being started, will be mapped to the Ambient.

When any instruction completes and fires its stop observers, the map is destructively traversed and a new process is started only if a path through the map can be found which reaches the replication. For example, if the input completes first then the input to Ambient mapping will be removed then the Ambient to parallel mapping will be removed and finally the parallel to replication mapping will be removed. At this point the replication has been reached so a new parallel container with a new copy of the replicated process in it is created and added to the replication. Additionally, all subscriptions to all other instructions within the process just reduced are cancelled. If any other instruction had finished whilst this was going on then it would enter the same algorithm. For example, the algorithm that caused the input to complete might also have completed. It would find and remove the mapping from itself to the parallel container but then it would be stuck as there is no longer a mapping from the parallel container to the replication. Consequently, no additional replicated process is created.

With this algorithm, it is guaranteed that the first and only the first instruction to complete in the current copy of the replicated process causes a new copy of the process to be created. Note that it is only on completion of an instruction that the replication is unwound: if it were on starting an instruction then any Ambient that is created or any other process container that is started would result in the unwinding of the replication which is clearly not desired. What is now necessary is to see that this is the correct behaviour and that this on-demand unwinding of the replication does not preclude reductions that a more eager unwinding would allow.

Consider the following process:

\[ !((n).P | (x).Q) \]

With an eager unwinding, the communications could be paired between different copies of the replicated process, e.g., given two copies of the replicated process, the output in one may pair with the input in the other. This is not the case with the on-demand unwinding that I have implemented as the next copy will not be available until after the first copy has made some sort of reduction, which in this case means the only reduction that can be made which is the input pairing with the output. However, this does not alter the behaviour of the process at all. In either mode of unwinding, the value of the output is fixed before the replication is started. Therefore all the copies of the replicated process will output the same value. Thus all the inputs that pair with any of the outputs will input the same values. Therefore no change in the values communicated can possibly occur as a result of one mode of unwinding or the other. If other processes which also contain inputs and outputs are placed in parallel with this replication, any interference between communication that can occur is possible under either mode of operation.

Any process that does not reduce internally by itself cannot be affected by the mode of unwinding as it will depend on interaction with processes external to it. What’s more, it is impossible to express a process that when replicated, requires two or more copies in order to achieve the first reduction. Thus the only processes that need consideration are those that can reduce by themselves without external interaction and by examination of the available mechanisms of internal reduction (i.e. communication or mobility), it can
be seen that they ultimately result in equivalent, for the purposes of this argument, processes to the one examined above.

4.7 Pairings and Rendezvous

One of the most crucial components of the implementation is the mechanism by which an output and input are paired or an action and co-action are paired. The details specific to communication and mobility will be covered in their own sections, 4.7.1 and 4.7.2. Here I shall consider the general requirements and explain the solution implemented whilst in section 4.7.3 I shall present a model of the pairing mechanism and show its correctness.

Firstly, some requirements:

• It must be impossible to construct an invalid pairing. At the most superficial level, this means matching arities of communication and the Ambient names with targets in mobility. However, this also includes other constraints such as ensuring that the pairing occurs between processes that belong to Ambients that are in the correct relationship with one another, hierarchically for the communication or movement that is to be undertaken.

• Having achieved a valid pairing, the Ambients involved must be locked in some manner so that no movement in any other process can rearrange the hierarchy of the Ambients, thus invalidating the pairing. Having achieved the lock, all the checks must be repeated to make sure that the Ambients are still in the correct relationship before the pairing can be acted upon. It must be impossible for a pairing to be acted upon after movements caused by other processes have resulted in a hierarchy that invalidates the pairing. This point is in fact only necessary for movement. As will be shown, these conditions can be relaxed for communication.

• Either party of the pairing can register for pairing before the other: there cannot be any restriction requiring, say, an input to appear before an output.

• Until the movement lock is obtained, it is possible for some other process to cause movements of Ambients so that either half of a potential pairing is now trying to achieve a pairing in the wrong location. Thus upon movement, attempted pairings should be aborted as quickly as possible so that they can be restarted in the correct Ambient.

• Having achieved a pairing, it must be impossible for one half of the pair to believe the pairing is valid whilst the other half believes it is invalid. Similarly, it must be the case that both halves believe the other half is the other half of the pairing: it must be impossible for some sort of chain to occur where A believes it’s pairing with B which believes it’s pairing with C . . .

• It is required for a process that has achieved a pairing and the movement lock and then finds that movement had already occurred thus invalidating the pairing to release the movement lock.

The basic rendezvous between two threads is performed using the usual synchronise-wait synchronise-notify combination. The construction of the pairing is done through an ExchangePair Object. This enforces an ordering of events relating to the population of the two halves of the pairing and ensures that if the pairing is broken then it is broken consistently on both halves and can never be unbroken.

Let A and B be the names of two threads which wish to construct a pairing. C is the name of a thread which intends to cause havoc by moving Ambients around in such a way as to cause the pairings to become invalid. In order to do this C keeps pairing with other threads and they perform movements. It is important to remember that the process that C goes through with these other threads is the same process as is being examined in detail with A and B. All pairings are constructed in the appropriate Ambient, quite what constitutes the appropriate Ambient is defined separately for communication and movement and covered in the relevant sections.

A is responsible for creating the blank ExchangePair. It creates the pair and sets itself as one half of the pair. It then synchronises upon the pair and then synchronising on various data structures adds the pair to the data structures in the appropriate place so that B can find it. Having added the pair to the data structure, A then notifies all threads waiting on the data structures and exits the synchronising block on the data structures thus relinquishing access to those structures. It then settles down for a hopefully short wait, waiting on the ExchangePair for it to become fully populated by another half of the pair.

B synchronises on the data structures and searches for an appropriate ExchangePair. If B cannot find any such ExchangePair, and indeed in this case, B has managed to progress faster than A so there is no
ExchangePair waiting for $B$, then it waits on the data structures until it is awoken by the notification sent by $A$. $A$, however, is not the only thread that can cause these notifications to be sent. $C$, which is up to no good as normal, is causing movements to occur, making the Ambient in which $A$ and $B$ are trying to pair move. Upon completing the movement, $C$ synchronises on the same data structures and issues a notification. This causes $B$ to wake up at which point it checks that it is in the right Ambient and that all the necessary conditions for pairing are met. It finds they are not met so it aborts, returning to its instruction and it starts again, this time picking whatever Ambient is currently appropriate in which to pair. This is not however, the only work that $C$ is required to do. $C$ also traverses the data structures looking for ExchangePairs. For every such ExchangePair that it finds it synchronises on it and notifies it. Whilst $C$ has been busy, $A$ was able to add its ExchangePair to the data structures and is now waiting on its ExchangePair for the other half of the pair. When $C$ notifies $A$’s ExchangePair, $A$ wakes up and checks that all the preconditions for pairing are met. Like $B$, it finds that they are not. Instead of just aborting and restarting though, $A$ marks its ExchangePair as broken. This ensures that should any thread like $B$ come along and find it, or had already found it, it cannot complete the pairing because the ExchangePair is broken.

$A$ and $B$ now restart. $A$ once again creates an ExchangePair and once again adds it to the relevant data structures. It is important that the ordering of the checks and synchronisations are understood. It must not be possible for $C$ to cause notifications to occur but for $A$ to be able to accidentally ignore those notifications and continue blindly on. Therefore, after creating the ExchangePair, $A$ synchronises on the ExchangePair as already stated. At this point, $A$ is the only thread to have any pointer to the ExchangePair. Then, after $A$ synchronises on the data structures, it performs its checks. At this point it can abort before adding its ExchangePair to the data structures. If it does not abort then it adds the ExchangePair to the data structures. For $C$ to send notifications on the data structures and the ExchangePairs requires that $C$ synchronises on those structures and pairs. Thus either $C$ sent notifications before $A$ synchronised on the data structures, in which case $A$ will notice that the preconditions for pairing are not met before it adds its ExchangePair to the data structures or $C$ sent notifications after $A$ has added the ExchangePair to the data structures in which case the ExchangePair will be found by $C$ and notified by $C$, thus waking $A$ as described previously. The precise orderings of the synchronisations guarantee that these are the only two possibilities in which $A$ and $C$ interact: it is not possible for $C$ to send notifications in some ordering that $A$ will not notice. If $A$ is therefore guaranteed to notice these notifications then upon being notified, it is guaranteed to check its preconditions for pairing and will therefore notice that the situation has changed and it must restart.

$A$ progresses more quickly this time and has its ExchangePair created and added to the data structures before $B$ comes looking. $B$ finds the ExchangePair and removes it from the data structures before exiting its synchronisation on the data structures. $B$ then sets itself as the other half of the ExchangePair which in turn awakens $A$. Once more, upon awakening, $A$ checks its preconditions for movement and again finds that $C$ has been beavering away behind its back. This time, $C$ did its work after $B$ had removed the ExchangePair from the data structures. Therefore when $C$ went looking for ExchangePairs to notify, it did not find the ExchangePair that $A$ had created. But, by setting itself as the other half of the ExchangePair, $B$ awoke $A$ which finds the preconditions not met and so $A$ breaks the ExchangePair thus preventing $A$ or $B$ from making any further use of the ExchangePair. Once more, $A$ and $B$ restart.

This third attempt progresses as for the second except this time $C$ is even later at creating movement. The ExchangePair has been populated by $A$ and $B$ and a valid pairing was created. Both $A$ and $B$ marked the ExchangePair as sealed which ensures that both parties are awake and are aware that a pairing has been created. Having returned to their own runtime instruction Objects, $A$ and $B$ synchronise on their instruction Objects and mark the ExchangePair so that the other half of the pair can learn who they are: up until $A$ has released its half, there is no way for $B$ to learn with whom it is pairing and vice versa. The reason for this particular structure is that there is a potential deadlock if $A$ and $B$ synchronise their instruction Objects before requesting the pairing. Therefore they do not synchronise their instruction Objects whilst pairing but the exchange of data that will eventually happen at the climax of the pairing will involve direct contact between $A$ and $B$ in their instruction Objects. Therefore $A$ and $B$ cannot allow the other to learn of who they are before they have synchronised on their instruction Object otherwise there is the possibility for the other half to progress far more quickly and perform operations on the instruction Object that are not appropriate at that point.

Having consented to the exchange of identities, $A$ and $B$ confer between themselves and ensure that they are a valid pairing. At this point, the behaviour is slightly different for communication and movement. For communication, a valid pairing has been constructed and although it may no longer be the case, the Ambients in which the processes performing the communication exist certainly once were in a configuration in which communication was valid to progress. Therefore, the communication goes ahead. Whilst this may technically allow for communications to occur after a movement had made such a pairing invalid,
the resulting reduction does not violate any of the semantics of Ambient calculi: the Ambients were in a configuration in which the communication was valid to progress and any movement that occurs does not alter that fact. What cannot happen is for a communication that could only be started after a movement had finished to progress with a pairing that was never a valid pair. For example in the following,

\[ a[\text{in} b . (y) . 0 | (x) . 0 ] | \langle n \rangle^a . 0 | b[\text{in} a . 0 ] \]

it is perfectly valid for the \((x)\) to pair with the \(\langle n \rangle^a\) when the Ambient is in its initial position. It is valid for that pairing to result in a successful communication even if the movement actually occurs and completes before the communication does. What is not valid is for the \((y)\) to have anything to do with the \(\langle n \rangle^a\) because the \((y)\) cannot start until the movement has completed at which point it is invalid for the \((y)\) to pair with the \(\langle n \rangle^a\). It is also not valid for the pairing between the \((x)\) and the \(\langle n \rangle^a\) to be created after the movement has taken place. It is only if the pairing was created before the movement completed that it may be allowed to progress.

For movement, it is crucial that no two movements occur in the same Ambient at the same time. For example, the following two reductions are invalid:

\[ a[\text{in} b . 0 | \text{in} c . 0 ] | b[a[\text{in} b . 0 ] | c[\text{in} a . 0 ] \]

\[ \rightarrow b[a[\text{in} c . 0 ] | c[\text{in} a . 0 ] \]

\[ \rightarrow b[c[0 | a[0 ] ] ] \]

\[ a[\text{in} b . 0 | \text{in} c . 0 ] | b[\text{in} a . 0 ] | c[\text{in} a . 0 ] \]

\[ \rightarrow b[a[0 ] | c[0 | a[\text{in} b . 0 ] ] ] \]

\[ \rightarrow b[a[0 | c[0 ] ] ] \]

This is the equivalent of the previous situation which is permitted with communication. Here, with movement, the pairings are both constructed before any movement takes place. Both pairings are valid. They both reduce. Whilst this does not violate the semantics of Ambients when it concerns a pairing of movement and a pairing of communication, it certainly does violate the semantics of Ambients when both are movements as the above examples show.

Therefore, after achieving a valid pairing and releasing the identity of the two halves of the pair, the movement lock on both its own Ambient and the target Ambient and any Ambients directly between. If any of these locks cannot be obtained, for example because \(C\) is in the middle of causing other movement then \(B\) directly informs \(A\) that the movement is to be aborted. At this point \(A\) is not looking at the ExchangePair so it can not notice the abortion if \(B\) marks the ExchangePair as broken, hence the direct notification. \(A\) and \(B\) once more restart. If \(B\) is able to obtain the movement locks then it makes one final check that the relationship between \(B\)'s Ambient and \(A\)'s Ambient is correct for the type of movement being performed. Having obtained the movement locks, if these last checks pass then the movement can complete without further incident because for anyone else, e.g. \(C\), to cause other movement would require \(C\) to obtain the movement locks which it cannot. If these last checks fail then predictably, \(B\) directly informs \(A\) that the movement is aborted and both restart. Note that the movement locks are local to each Ambient. This allows multiple Ambients to engage in non-conflicting movement in parallel.

Not only is the ordering of synchronisations in establishing the ExchangePair critical, but the protocol that works around and after the ExchangePair is too. Up to seven locks are used during this process and ensuring that they all occur in the right order without any possible deadlock and that the rendezvous between up to three threads happens in such a way that information regarding the changing locations of Ambients cannot be lost was very tricky to achieve both conceptually, in terms of implementation and in terms of testing.

At the point at which \(B\) searches for a suitable ExchangePair within the data structures as part of the pairing mechanism, it is possible that it finds more than one suitable ExchangePair. If this is the case then it picks at random. This ensures that the pairings are non-deterministic and that there are no such first-come-first-served semantics in the construction of the pairings.

Also note that when \(C\) sends notifications that movement has occurred, it may not be necessary for \(A\) and \(B\) to restart. If their preconditions for pairing can still be met then they have no need to restart and can carry on. This reduces to a minimum the amount of work that gets repeated due to movement.
4.7.1 Mobility

An in action must pair with an \(m\) action. The two Ambients that contain the processes that are performing the in and \(m\) instructions must be siblings as can be seen in the definition of reduction in figure 4.2. Because they are siblings, they share the same parent and this is hence the most logical place for them to achieve the pairing: the parent Ambient of the Ambients in which they are executing. The main precondition for pairing of the in and \(m\) instructions is that the Ambient in which they are pairing is indeed the parent of their own Ambient. It’s possible for their own Ambient to be moved by another process. If it is moved then their own Ambient would be de-registered from its current parent Ambient which causes the notifications to be sent as described above, thus alerting the threads trying to achieve a pairing.

The main requirements for matching the instructions are that the names match. For example, in

\[
a[\text{in \, b, \, 0}] | b[\text{m, \, c, \, 0}] | b[\text{m, \, a, \, 0}] | d[\text{m, \, a, \, 0}]
\]

the Ambient named in the in action, i.e. \(b\) must match the name of the Ambient in which the \(m\) action is being performed. This precludes the \(m\) action in the \(d\) Ambient from being a candidate for pairing. Additionally, the Ambient name in the \(m\) action must match the name of the Ambient in which the in action is being performed. This precludes the first \(b\) Ambient on the grounds that \(c \neq a\). Thus there is just one valid pairing in the above.

An out action must pair with an \(\overline{\text{out}}\) action. The process that is performing the \(\overline{\text{out}}\) action is in an Ambient that is the grandparent of the Ambient in which the out action is being performed. Because of the ability of child and grandchild Ambients to be constantly moving around, the location for the pairing is in the Ambient in which the \(\overline{\text{out}}\) instruction is being executed. A moment’s consideration of the alternative should provide a convincing argument why. If the obvious alternative, which is parent of the Ambient in which the out instruction is being executed were chosen for the pairing then the \(\overline{\text{out}}\) instruction would have to locate every possible child Ambient of its own Ambient and look inside that Ambient for a matching out action. This is much more complex and difficult. In particular, without spinning, it cannot be done by just one thread.

The precondition for movement involving outward movement is that the Ambient in which the out instruction is being executed is a grandchild of the Ambient in which the \(\overline{\text{out}}\) instruction is being executed. Now movement can occur in two places. Firstly the child Ambient can move (this is the Ambient in the middle of the two Ambients in question, i.e. a child of the Ambient in which the \(\overline{\text{out}}\) instruction is being executed and the parent of the Ambient in which the out instruction is being executed). This would cause the notifications to be sent to the waiting \(\overline{\text{out}}\) thread in its parent in the same way as for \(m\). The other movement is if the grandchild moves. In this case, the notifications that would otherwise not reach the parent are chained through the child to the parent to ensure that the \(\overline{\text{out}}\) action is awoken. For example, in

\[
b[\text{a[\text{out, \, b, \, 0}] | \text{in, \, b, \, 0}] | \text{out, \, a, \, 0}] | b[\text{m, \, a, \, 0}]\]
\]

it’s possible for the out \(b\) action to start first. It finds that the parent of its Ambient is indeed named \(b\) so it then finds the parent of that Ambient which is the top level Ambient, or the host Ambient, and lies in wait for a suitable co-action. Of course, no such co-action is coming. In the mean time, the in \(b\) action is able to pair and progress and so now the \(a\) Ambient is inside the inner \(b\) Ambient. It is vital that the out \(b\) instruction is awoken and reevaluates the situation, so, upon de-registering from the outer \(b\) Ambient, the \(a\) Ambient also finds the parent of that outer \(b\) Ambient and sends notifications to the data structures and ExchangePairs as described in the previous section. This guarantees to wake the out \(b\) thread which then becomes aware of the movement that has occurred. It then restarts itself and now chooses the outer \(b\) Ambient in which to pair (as opposed to that \(b\) Ambient’s parent) because of the movement that has occurred and correctly finds that it can now pair with the \(\overline{\text{out}}\) a instruction. The pairing is made and the movement proceeds.

The requirements on the pairings for an out and \(\overline{\text{out}}\) action are a little more complex than for in and \(m\). For example, in the following,

\[
\overline{\text{out, \, a, \, 0}] | b[\text{c[\text{out, \, b, \, 0}] | a[\text{out, \, z, \, 0}] | a[\text{out, \, b, \, 0}]]}
\]

the Ambient name within the \(\overline{\text{out}}\) action, i.e. \(a\), must match the name of the Ambient in which the out action is being executed. This precludes pairing with the \(c\) Ambient. Additionally, the Ambient name within the out action must match the name of the parent Ambient of the Ambient in which the out action is being executed. This precludes the first \(a\) Ambient from the pairing as \(b \neq z\). Thus only the last out action is a valid partner for the \(\overline{\text{out}}\) action.
4.7.2 Communication

Communication turns out to be more complicated than movement. The first source of additional complexity is caused by the fact that an input can pair with three different forms of output and an output can pair with three different forms of input. Behold the following:

\[
\langle l \rangle^a . 0 \mid a[\langle x \rangle . P \mid \langle m \rangle . 0 \mid c[\langle n \rangle . 0 ]]
\]

\[
(x)^a . 0 \mid a[\langle n \rangle . P \mid \langle y \rangle . 0 \mid c[\langle z \rangle . 0 ]]
\]

In the first process, any of the outputs can correctly pair with the input and in the second process, any of the inputs can pair with the output. I'll tackle these combinations one by one. With all of the communications possible, the requirements for the validity of pairing, other than the correct relationships between the Ambients involved, are that the input arity and output arity match.

4.7.2.1 Local Input, Local Output

This situation occurs in both of the above processes. The input and output are both in processes that are within the same Ambient. Therefore there is little alternative to achieving the pairing within the Ambient. The input will request a pairing with some sort of output (deliberately not limiting itself to pairings with local outputs) and the output will make a similar request. The two will find each other, provided the arity of the input and output match and the communication will progress.

4.7.2.2 Local Input, Parent Output

This situation requires a pairing between the input and output in, for example, \( \langle x \rangle . P \mid a[\langle n \rangle . 0 ] \). Given that the local input is already waiting for a pairing in its own Ambient, the simplest thing is for the parent output to find the parent of its own Ambient and request a pairing with a local input within that parent Ambient. Note that it is required at this point that the parent output pairs with a local input, otherwise there is the possibility that \( a[\langle n \rangle . 0 \mid \langle x \rangle . P ] \) would reduce which is not permitted.

4.7.2.3 Parent Input, Local Output

This is the role reversal of the above, for example, \( \langle n \rangle . 0 \mid a[\langle x \rangle . P ] \). The same reasoning applies: the output will be waiting to pair in its own Ambient so the input must find the parent Ambient of its Ambient and request a pairing with a local output.

4.7.2.4 Local Input, Child Output

The communications involving child Ambients are the most complex. For example, consider

\[
\langle n \rangle^a . 0 \mid a[\langle x \rangle . P \mid \langle y \rangle . Q ] \mid a[\langle z \rangle . R ]
\]

Not only is there the possibility of more than one local input being suitable for pairing in a suitable child Ambient, but there is also the possibility of more than one such child Ambient. Given that the behaviour of the local input has been well established by now and is known to be waiting inside its own Ambient for a suitable output to pair with, it is the output side that must adapt. An observer pattern is added to the Ambients. This allows the output to be notified whenever an Ambient with a particular name is added or removed from the Ambient. Thus in this case, the output will contact its own Ambient (here, the top level host Ambient, not shown) and will register itself as an observer of all child Ambients named \( a \). If any Ambient named \( a \) enters or leaves the parent Ambient then the output will know about it. The output is also told at this point about all the child Ambients already present that have the correct name, thus the output learns of the two \( a \) Ambients. For each such Ambient, the output creates a new thread that contacts the child Ambient and waits for a local input to pair with. Once again, it is required that it pairs with a local input rather than just any type of input. Then, having created a pairing, the thread returns to the output and presents the input to the output. This is a race condition and suitable synchronisations ensure that no deadlocks occur, only one pairing is confirmed, that all unused pairings are marked broken, that all the extra threads created die safely after a successful communication has been achieved and that the output de-registers itself from the observer pattern after the communication.
4.7.2.5 Child Input, Local Output

Once again, this is the dual of the above and exactly the same complexities exist.

\[(x)°.0 | a[⟨n⟩.P | ⟨m⟩.Q] | a[⟨l⟩.R]\]

The exact same procedure is used but this time the input is the party searching for appropriately name child Ambients of its Ambients, spawning additional threads and pairing with local outputs in the child Ambients.

4.7.3 Modelling the Pairing Mechanism

Given how crucial the pairing system is to the correct operation of the GLINTVM I decided to model it using LTSA². LTSA is the Labelled Transition System Analyser and, given process definitions expressed as Finite State Processes, can display processes as labelled transition systems and perform exhaustive analysis to detect deadlock, amongst other things. A complete description and explanation of LTSA and Finite State Processes is beyond the scope of this report but a brief introduction is provided in appendix C which should prove sufficient for the purposes of this model.

The model is shown in figure 4.8. There are five processes in the model, one for the data structure which is modelled as a simple buffer and is hence named BUFFER, one for the ExchangePair, called PAIR, which enforces which methods can be called when, one for the preconditions for the pairing which is called CONDITION and corresponds to the actions of the thread C in the preceding discussion which caused havoc by moving the Ambients around and thus causing the conditions for pairing to fail and finally a process LEFT and a process RIGHT which represent the two halves of the pairing, previously referred to by the threads named A and B in the previous discussion.

The first process to understand is the CONDITION process. As discussed previously, the condition can fail at any point. However, after the ExchangePair has been sealed by both halves of the pairing, whilst the condition can fail, it doesn't directly affect the pairing, instead the condition is inspected by the instructions directly. One of the consequences of the design of the pairing system is that either of the pairing processes can be the first to seal their respective half of the pair. Therefore they have to accept that the pairing could fail after they have sealed. Rather than explicitly model the LEFT and RIGHT processes sharing both the sealing of the left and right halves, they only model the sealing of their own half. This then allows for the possibility that the condition could fail after both halves have been sealed which then would cause a deadlock as dealing with that situation isn’t modelled here. Instead, the CONDITION process is modified to prevent the condition failing after both halves have sealed.

The BUFFER process is a very simple and straightforward buffer. The PAIR process regulates access to the ExchangePair Object and ensures the correct ordering of method calls to the Object. The first thing that must happen after creation is that the left hand side of the pair is set. As discussed previously, at this point, the LEFT process will check the preconditions and may well abort at this point. Thus the PAIR allows itself to be marked broken. If it is not marked broken then it must be added to the data structure and is next used when the B thread, or RIGHT process, sets itself as the other half. At this point the A thread wakes up and will notice if the condition has failed or not, if so, it will mark the ExchangePair as broken. The PAIR can be sealed in any order and then will finally be released, allowing the two instructions to see with whom they are pairing.

The LEFT and RIGHT processes represent the two threads performing the pairing. As already discussed, the LEFT process creates the ExchangePair and adds it, if appropriate to the data structure. If there is no ExchangePair ready, the RIGHT process will simply wait. During this time the condition could fail so the RIGHT process is required to be able to catch this situation and restart without waiting for an ExchangePair to appear in the data structure. Note that in this particular case, there is an explicit pair.broken message. Clearly at this point the RIGHT process has no knowledge of the ExchangePair. However, firstly, if the condition does fail at this early point then it is guaranteed by the model that the ExchangePair has been created so it will need to be marked broken. Secondly, the RIGHT process uses the pair.broken message elsewhere and so it cannot opt out of sharing this action in this first case. Thus it is necessary for the process to explicitly share this action at this point. Again, the sealing and releasing can happen in a non-deterministic order.

The PAIRING_SYSTEM constructs the parallel composition of these processes and makes sure they’re all talking correctly about each other. The analysis LTSA performs shows that the model has no deadlocks. Given the small size of the BUFFER process (i.e. it can only model a buffer which has a capacity of one), it shows that the ExchangePairs that are created are removed in all possible situations from the buffer. This means that there can never be broken ExchangePair Objects left in the data structures.

Figure 4.8 The Pairing System modelled as a Finite State Process

\[
\text{CONDITION} = \begin{cases} 
\text{met} \rightarrow \text{CONDITION} \\
\text{unmet} \rightarrow \text{reset} \rightarrow \text{CONDITION} \\
\text{pair.sealLeft} \rightarrow \text{CONDITION\_BREAKABLE} \\
\text{pair.sealRight} \rightarrow \text{CONDITION\_BREAKABLE}, 
\end{cases}
\]

\[
\text{CONDITION\_BREAKABLE} = \begin{cases} 
\text{met} \rightarrow \text{CONDITION\_BREAKABLE} \\
\text{unmet} \rightarrow \text{reset} \rightarrow \text{CONDITION} \\
\text{pair.sealLeft} \rightarrow \text{CONDITION\_DONE} \\
\text{pair.sealRight} \rightarrow \text{CONDITION\_DONE}, 
\end{cases}
\]

\[
\text{CONDITION\_DONE} = \begin{cases} 
\text{met} \rightarrow \text{CONDITION\_DONE} \\
\text{pair.completed} \rightarrow \text{CONDITION}. 
\end{cases}
\]

\[
\text{BUFFER} = \begin{cases} 
\text{put} \rightarrow \text{remove} \rightarrow \text{BUFFER}. 
\end{cases}
\]

\[
\text{PAIR} = \begin{cases} 
\text{create} \rightarrow \text{setLeft} \rightarrow \text{SET\_RIGHT\_OR\_BREAK}, 
\end{cases}
\]

\[
\text{SET\_RIGHT\_OR\_BREAK} = \begin{cases} 
\text{setRight} \rightarrow \text{SEAL\_OR\_BREAK} \\
\text{broken} \rightarrow \text{PAIR}. 
\end{cases}
\]

\[
\text{SEAL\_OR\_BREAK} = \begin{cases} 
\text{sealLeft} \rightarrow \text{SEAL\_RIGHT} \\
\text{sealRight} \rightarrow \text{SEAL\_LEFT} \\
\text{broken} \rightarrow \text{PAIR}. 
\end{cases}
\]

\[
\text{SEAL\_RIGHT} = \begin{cases} 
\text{sealRight} \rightarrow \text{RELEASE} \\
\text{broken} \rightarrow \text{PAIR}. 
\end{cases}
\]

\[
\text{SEAL\_LEFT} = \begin{cases} 
\text{sealLeft} \rightarrow \text{RELEASE} \\
\text{broken} \rightarrow \text{PAIR}. 
\end{cases}
\]

\[
\text{RELEASE} = \begin{cases} 
\text{releaseLeft} \rightarrow \text{releaseRight} \rightarrow \text{completed} \rightarrow \text{PAIR} \\
\text{releaseRight} \rightarrow \text{releaseLeft} \rightarrow \text{completed} \rightarrow \text{PAIR}. 
\end{cases}
\]

\[
\text{LEFT} = \begin{cases} 
\text{pair.create} \rightarrow \text{pair.setLeft} \rightarrow \text{LEFT\_PUT\_OR\_ERROR}, 
\end{cases}
\]

\[
\text{LEFT\_PUT\_OR\_ERROR} = \begin{cases} 
\text{buff.put} \rightarrow \text{pair.setRight} \rightarrow \text{LEFT\_SEAL\_LEFT\_OR\_ERROR} \\
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{LEFT}. 
\end{cases}
\]

\[
\text{LEFT\_SEAL\_LEFT\_OR\_ERROR} = \begin{cases} 
\text{pair.sealLeft} \rightarrow \text{LEFT\_RELEASE\_LEFT\_OR\_ERROR} \\
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{LEFT}. 
\end{cases}
\]

\[
\text{LEFT\_RELEASE\_LEFT\_OR\_ERROR} = \begin{cases} 
\text{pair.releaseLeft} \rightarrow \text{pair.completed} \rightarrow \text{LEFT} \\
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{LEFT}. 
\end{cases}
\]

\[
\text{RIGHT} = \begin{cases} 
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{RIGHT} \\
\text{buff.remove} \rightarrow \text{pair.setRight} \rightarrow \text{RIGHT\_SEAL\_RIGHT\_OR\_ERROR}, 
\end{cases}
\]

\[
\text{RIGHT\_SEAL\_RIGHT\_OR\_ERROR} = \begin{cases} 
\text{pair.sealRight} \rightarrow \text{RIGHT\_RELEASE\_RIGHT\_OR\_ERROR} \\
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{RIGHT}. 
\end{cases}
\]

\[
\text{RIGHT\_RELEASE\_RIGHT\_OR\_ERROR} = \begin{cases} 
\text{pair.releaseRight} \rightarrow \text{pair.completed} \rightarrow \text{RIGHT} \\
\text{cond.unmet} \rightarrow \text{pair.broken} \rightarrow \text{cond.reset} \rightarrow \text{RIGHT}. 
\end{cases}
\]

\[
\text{PAIRING\_SYSTEM} = \begin{cases} 
\text{cond::CONDITION }|| \text{buff::BUFFER }|| \text{pair::PAIR} \\
\text{|| LEFT }|| \text{RIGHT }|| \text{(pair/cond.pair)}. 
\end{cases}
\]
Finally, there are liveness properties and a safety property that establish stronger results from the model. These are shown in figure 4.9. The SUCCESS and BREAKAGE expressions are liveness properties. They ensure that the actions they show happen infinitely often in the model. This guarantees that the model is modelling situations where the pairing completes and where the pairing is broken. The process CREATE_THEN_BREAK_OR_COMPLETE is a safety property. It ensures that before a pair.create action is performed again, the safety property process must loop. Therefore it guarantees that after the creation of the pair, either the condition fails, the pair is broken and the condition is reset or the two halves of the pair are released and the pair is completed. This safety property, in combination with the liveness properties ensure that the aims of the model are met: eventually a pairing is completed, the two halves of the pairing permit each other to learn with whom they are pairing, and no deadlock ever occurs.

Figure 4.9 Progress and Properties for the model in figure 4.8

property CREATE_THEN_BREAK_OR_COMPLETE = (pair.create → BREAK_OR_COMPLETE),

BREAK_OR_COMPLETE = (cond.unmet → pair.broken → cond.reset

→ CREATE_THEN_BREAK_OR_COMPLETE

| pair.releaseLeft → pair.releaseRight → pair.completed

→ CREATE_THEN_BREAK_OR_COMPLETE

| pair.releaseRight → pair.releaseLeft → pair.completed

→ CREATE_THEN_BREAK_OR_COMPLETE).

progress SUCCESS = pair.completed
progress BREAKAGE = pair.broken

||PAIRING_SYSTEM = (cond::CONDITION || buff::BUFFER || pair::PAIR

|| LEFT || RIGHT || CREATE_THEN_BREAK_OR_COMPLETE )

/(pair/cond.pair).

4.8 Paths

Paths are a very useful feature of the Ambient calculus and enable a sequence of actions to be constructed and communicated. Ultimately, a path consists of actions and co-actions only. The implementation of paths is reasonably straightforward. All that is needed when constructing a path is for the actions and co-actions specified to capture the current values of the names within them as discussed in section 4.3.2. If a name is used as a component of a path then by induction, that will already have captured its real value and so will be communicated correctly. Example 4.10 shows paths in use.

Example 4.10 Use of Paths in the Ambient calculus

(x). x. 0 | a[ (y). (out b . y) . 0 | b[ out a . 0 | (z)c. (out c . z) . 0 | c[ out b . 0 | (out d) . 0 | d[ out c . 0 ]]])
→ (x). x. 0 | a[ (y). (out b . y) . 0 | b[ out a . 0 | (out c . out d) . 0 | c[ out b . 0 | out d . 0 | d[ out c . 0 ]]]]
→ out b . out c . out d . 0 | a[0 | b[ out a . 0 | c[ out b . 0 | d[ out c . 0 ]]]]
→ out c . out d . 0 | a[0 ] | b[0 | c[ out b . 0 | d[ out c . 0 ]]]
→ 0 | c[0 ] | d[0 ]

In this example you can see how the co-actions are chained together by communication and ultimately
sent to the top level input process which then executes the co-actions. This then allows the Ambients to un
wrap themselves in a Russian-doll manner. The only point of note with the implementation of paths is that the same path can be executed in more than one place concurrently as a result of communication. Therefore, whilst in communication, no copying of the path is performed if no modifications are made. This ensures that in communication, where possible, no unnecessary copies of the path are made. At the point of execution however, the values (actions and co-actions) within the path are duplicated to ensure they are unshared. This prevents a race condition between multiple processes all trying to execute the same path at the same time.

4.9 Inter-host Mobility

Whilst movement of Ambients within the same host can be accomplished with relative ease, inter-host mo
bility is much harder. The primary problem is that threads themselves cannot be moved between machines. Therefore, the Ambient to be moved must first be stopped, all the threads within it must die and in doing so, leave the processes within the Ambient in known good states which accurately represent the reductions performed so far. Then the Ambient must be communicated to the target host and finally, having been received by the target host it must be restarted, continuing from exactly the same point as it was last at with the exception of the inter-host movement having taken place.

This freezing of the Ambient must only take place when it has been established that the target host is willing to accept the Ambient. No provision is made for the target host to renege on its commitment to accept. This can be explained by the consequences of three aspects of the reduction machine and Ambient calculi. Firstly, once issued, an instruction can not be cancelled, it must complete for the process to progress any further. Secondly, only movement can alter the context in which the instruction executes. And thirdly, as explained in section 4.3.3, accepting an inter-host movement can only be done by a process directly within the top level host Ambient, which cannot move. Therefore, once the in co-action has been issued, nothing can cause it to be cancelled or moved to a different Ambient.

Having established that the target host will accept the Ambient, a freeze message is propagated through the Ambient and all of its processes and child Ambients recursively so that every process within the Ambient, no matter at what depth is aware of the freeze request. The freeze message also causes all the Ambients to obtain their movement locks thus preventing any further movement within the Ambient. The freeze request informs all instructions currently being executed at any depth within the Ambient to either complete the current instruction or abort it. In either case, the thread executing the instruction must exit the instruction as soon as it can do so safely and mark the instruction as successfully frozen on its way out. This is achieved by making the thread run a continuation that it was sent as part of the freeze message. As a result it can be commandeered so as to perform additional work, for example if it is the last process within an Ambient to freeze then it is responsible for marking the Ambient as frozen. As is expected, the freeze message also sets off all the usual notify events to awaken threads that are sleeping, waiting for pairings. They then notice that the Ambient has frozen, they notice they too have received the freeze message and so rather than restarting, they batten down the hatches and freeze.

Once the entire Ambient is frozen it is serialised and sent to the target host. One of the properties of serialisation is that the Object graph is traversed and every Object that is reachable and not marked as transient will be serialised and sent to the target host. Therefore, great care was required when constructing this system to ensure that there could not be rogue pointers that reference Ambients or processes outside of the frozen Ambient for such pointers would result in much more than the frozen Ambient being serialised and sent to the target host. In particular, judicious use of the transitive field modifier and consistent overriding of the writeObject method which is used by the serialisation system ensures that no process or instruction Object which is not frozen can be serialised.

Having been received, it is a relatively simple matter to thaw the Ambient as much the same system as for when the Ambient was constructed can be used. The only difference is that now, the runtime instruction Objects have already been generated and so it's just a matter of marking the processes as thawed, creating new threads as necessary and setting them to work.

There is of course a major race condition if the Ambient is trying to move into two or more hosts. To solve this, upon receiving confirmation of a valid pairing with an in action on the remote host, the first thing the in action does is to obtain the movement lock on the Ambient to be moved. This prevents any other attempt to freeze the Ambient even being started and so there is no further conflict between any number of inter-host movement actions.

One of the more subtle issues is that it is considered bad practice for an Object to use itself for synchroni
sation. The reason for this is that some other class may make use of that Object for its own synchronisation
thus disrupting the access patterns designed for the Object. This is another example of traditional locking systems not being composable. The recommended solution is to use a private final `Object` within the class for synchronisation. This prevents the use of the Object by other classes for synchronisation from affecting the access regulation within the Object. When specific interactions and rendezvous are needed, this `Object` can be passed around in a controlled way so that other classes can have access to it as necessary. The `Object` class in Java is not serializable thus all such uses of `Object` had to be consistently replaced with a new class that contains no state, inherits without any modification or augmentation from `Object` but supports serialisation.

4.10 Interacting with the Real World

The Ambient calculus is a complete model of computation in which can be expressed any computable expression or algorithm. However, unless the entire world is rewritten from the ground up in terms of the Ambient calculus, some additional mechanism is required so that interaction with the real world can be achieved. This is similar to having native methods in Java: methods which are implemented in C and interact with the operating system, performing computations that could not otherwise be expressed in Java without the universal rewrite of software.

In the GlimtVM this is achieved through sending messages to a special Ambient, called `__gvm__`. This special Ambient is always present and can be messaged from any process, anywhere. It is always local to each host. Initially, there were just two functions accessible in this way, but this has grown to meet the needs of the Glimt language. To give a feel for how these functions can be used, I will here explain the original two functions. This mechanism is very non-intrusive, honouring the spirit of the Ambient calculus as much as possible. It is also very extensible and has been used to implement a number of crucial functions.

4.10.1 println

Where would we be without `println`? Other than using the Java debugger, this is the primary means of seeing any sort of output or response from the program the GlimtVM is executing. The following shows the `println` function in use:

\[(x). \langle \text{println}, x \rangle^{\text{gvm}}. P\]

After the input has been received, a message is sent to the special `__gvm__` Ambient asking it to print to the console the current value of \(x\). This it does and execution of \(P\) can then begin.

4.10.2 readln

The obvious dual of `println`, `readln` reads the next line of input from the console, blocking if necessary and returns it to the caller. This it does by creating an Ambient, the name of which is supplied to it and then adding a local output within that Ambient containing the value of the line just read. This new Ambient is then added to the same Ambient in which the process that called `readln` is executing, for example:

\[(\nu a)(\langle \text{readln}, a \rangle^{\text{gvm}}. (t)^a. P)\]

The use of restriction here guarantees that no other process can perform the input on \(a\) thus ensuring that the process captures in \(t\) the text that was read from the console.

A standard `echo` process can now be constructed as:

\[! (\nu a)(\langle \text{readln}, a \rangle^{\text{gvm}}. (t)^a. \langle \text{println}, t \rangle^{\text{gvm}}. 0)\]

4.11 Technologies Employed

Java is the main language of implementation. This is not a result of any particular love of the language, more a result of a level of confidence in being able to achieve a working implementation and a level of understanding of Java's inner workings and semantics. The other advantage of working in Java is the range and quality of tools and libraries available. Eclipse is my development environment of choice and I make use of both additional plug-ins such as CheckStyle\(^5\) and external tools such as FindBugs\(^4\), the latter in particular being extremely useful.

The lexer and parser generator is ANTLR\(^5\). ANTLR is a very powerful parser generator who's main

\(^5\)http://checkstyle.sourceforge.net/
\(^4\)http://findbugs.sourceforge.net/
\(^5\)http://www.antlr.org/
strength is in being able to use the very same grammar syntax to walk over the ASTs generated by
the parser. This removes any need to implement the visitor pattern, in itself a massive win for all concerned.
The real power comes from the flexibility of being able to add Java code at almost any stage of matching of
the AST and thus being able to transform the AST in very powerful and intuitive ways. In this particular
instance, the full power of the AST transformation system is not needed.

The unit testing system used is TestNG* which has several advantages over the more widespread JUnit.
JUnit forces a more rigid structure on the user, identifying methods to test by a naming convention of the
methods. TestNG uses annotations which have only recently been added to Java in version 5 to identify
tests and is more flexible in other areas too: the annotations can identify which tests belong to which test-
groups thus allowing the construction of groups of related tests with virtually no additional code cost. An
example of a TestNG test is shown in listing 4.7. Here the class is added to the “ambients” group as can
be seen in the annotation.

4.12 Conclusion

Whilst the GLINTVM may not have the same level of performance on hardware with little parallelism as
other reducing machines due to the overheads of operating system threads and rendezvous, it meets its
aims and performs well on parallel hardware. Details of using the GLINTVM can be found in appendix D.1
whilst some performance metrics are presented in section 7.5.

Whilst I have worked on bigger projects in terms of amount of code and whilst I have written software
which has more functions and features, I have not written anything that approaches the GLINTVM in terms
of complexity. The motivation for much of this project stems from the difficulties in writing concurrent
programs using traditional locking mechanisms for access regulation. The GLINTVM is a monumental
testament to these difficulties.

The most difficult problem to overcome was debugging race conditions that occur infrequently. The
only possible way to achieve this was to try to create tests that stress the races in particular areas of code
and then to run them over and over again automatically until the events occur in the correct order and
the deadlock happens. It then becomes a matter of forensics to be able to, from the resulting deadlock,
work backwards so as to uncover quite what the actual cause really was. It is possible to write multi-
threaded software that behaves correctly on single-threaded hardware but does not behave correctly on
parallel hardware. Therefore development and testing had to occur on machines with as much parallelism
as I could find, which amounted to dual-core processors.

The GLINTVM provides a useable platform for the execution of Safe Boxed Ambient calculus programs.
The design of the GLINTVM, principally the staggered generation of runtime instructions from the static
representation of the parsed program has proved to be a successful and reliable design which lends itself
well to decoupling the static and dynamic aspects of reduction of Ambient processes and is easily extended.
By not using substitution directly, the implementation was easier than it otherwise would have been and can
successfully migrate processes between hosts without succumbing to any notions of global substitution or
a global lock. The GLINTVM interprets programs written in the Safe Boxed Ambient calculus which, for
the purposes of this project, is treated as an assembly language but with vastly more powerful constructs for
concurrency and inter-host mobility of processes than typical assembly languages.

*http://testng.org/doc/
The GLINT language is a high level pure Actor-based language with additional support for mobility. The GLINT compiler compiles the language into the Safe Boxed Ambient calculus which is then executed by the GLINTVM, as discussed in chapter 4.

Given the evidence of the languages in use today, it would seem that language design is an almost impossibly hard task as there seems to be no language which is universally liked. In fact it would seem that every language is far closer to being universally disliked than liked given the ease with which one can find criticism of any language. That is not though to condemn every language ever conceived. Many languages genuinely do bring new concepts and ideas to the community and many represent existing ideas in novel forms. There is much to suggest that languages that are popular do not correlate with novel languages, instead correlate far more closely with established and widespread ideas assembled in particularly useable ways.

Languages take a long time to develop and if successful normally go through significant changes in their working lives. You only have to look at the differences between JAVA 1.1 and JAVA 5.0, a gap of about 10 years, to see how much can change and how much things can improve. Therefore, given the limitation on development time, it becomes very important to choose a set of features for the language that can be implemented in the remaining time and that serve to allow the assessment of the language and its runtime environment. In short, the language is better considered as a prototype rather than an attempt at a fully featured language: it is far more important to focus on the design and implementation of the new and non-obvious features of the language rather than the features that the language has in common with the majority of programming languages and for which good solutions are already known.

5.1 The Suitability of Actors

One of the primary aims of this project is to contribute to the diminutive set of tools, techniques and methods that exist to aid programmers as they program architectures with ever greater levels of parallelism. The Ambient calculi have presented a means for modelling formally both concurrent and mobile processes and the GLINTVM is a useable implementation of one such Ambient calculus. One of the main problems that occurs when programming using multiple threads in an Object Oriented language is that it is necessary to keep track of too many things. Firstly the classes and their inheritance hierarchies. Secondly the fields and the graph of pointers between all the Objects used. Finally an entirely separate graph must be mentally maintained in which orderings between method calls and threads are mapped. This is too much information and frequently causes brain failure in very concurrent programs.

Actors, presented in chapter 3, go a long way to solving this problem. By encapsulating the thread of execution within each Actor, two mental graphs are merged into one. Actors themselves are atomic units, similar to Objects with strong encapsulation (i.e. with all instance fields accessible only within the Object instance) but can also be modelled as Finite State Processes where Objects cannot because Objects only express a structuring of data whereas Actors express both a structuring of data and of processes.

Given the focus of Ambients and process algebra in general on communication between concurrent processes, it seems wrong to design a language based around Object Orientation in which a single thread is simulated, operating in the standard and widespread sequential manner. It would not serve to address any of the issues of concurrent programming and making better use of highly parallel hardware, and it would not make good use of the GLINTVM. Instead, it seems that Actors and their explicit focus on concurrency offers a much better fit with Ambients whilst at the same time offering opportunities for high level abstractions in line with those we are all used to from Object Oriented languages.

It is readily apparent that Actors and Ambients have much in common: Ambients encapsulate state, any changes to their own state is achieved purely through communication; communication is the primary means of reducing a program (giving movement a back-seat for the time being); and an Ambient can create
other Ambients. The only areas where they differ significantly from Actors is in the filtering and selection of messages and in the availability of control-structures. The only means of filtering messages that is available to Ambients is in the arity of the message along with some control over the directionality of the message, be it coming from a local process, a child Ambient or the parent Ambient. Further, with Ambients it is not possible for a single process to define a set of communications in which it is willing to partake, instead it can only specify one such communication, requiring a number of processes to emulate the behaviour of the Actor. Additionally, whilst an Actor can make use of traditional control-flow structures should the language permit, an Ambient process cannot: a simple if-then-else structure requires three processes in an Ambient calculus as example 5.1 shows. These differences suggest that there cannot be a one-to-one mapping of Actors to Ambients, instead, more than one Ambient process will be required to encode each Actor.

Example 5.1 Encoding an if-then-else structure in Ambients.

\[(x) \cdot \langle \rangle \cdot 0 | \text{true}[(.] \cdot P] | \text{false}[(.] \cdot Q)\]

Adding the mobility of Ambients to Actors is an attractive idea and conceptually simple. In an Object Oriented language, defining what happens when moving an Object from one host to another is complex and non-intuitive: it breaks the abstraction of having (if desired) a single thread doing all the work as a thread physically can’t move between hosts. Furthermore, if an Object is moved to a different host, then what is the meaning of pointers to that Object from the originating host: they clearly cannot be addresses (given a non-shared address space) so what exactly are they?

In contrast, given that all communication in Actors is done by message passing, sending a message from one Actor to another requires knowing the name of the target Actor. Whilst this amounts to the same thing, the name of an Actor carries less implied knowledge than the address of an Object: suddenly the user becomes less concerned with any internal routing that may go on to get the message to its target Actor and it becomes very much easier to think about and deal with Actors moving between hosts. Thus the underlying model of computation for Actors is more amenable to both concurrency and to mobility than is Object Orientation.

5.2 Modifying Actors

Implementing a pure Actor language that is totally faithful to the Actor paradigm would be possible but would not achieve much that is novel nor address any of the issues uncovered in section 3.4. Continuation passing style limits the level of comprehension of Actors and when performed almost behind the scenes, as it is in SALSA, it requires the programmer to be able to program via a level of indirection.

The primary reason for this need to program in continuation passing style is that there is no system for call-and-return: messages are one way only and it is difficult to adjust the Actor so that it can receive a reply from the callee without there being some pre-established call and return protocol with the callee which may very well involve a continuation in order to provide a wrapper around the callee. It is not enough to set the continuation of the message to be the caller because that doesn’t prevent the caller from processing any other message in the mean time by accident. What is needed is a shared secret in the message that the callee can send back to the caller thus allowing the caller to identify the message it is awaiting.

It is interesting to note that Erlang does not present easy ways to construct new Actors for the purpose of continuation passing but does manage to be a very successful and useable language. Sending is asynchronous as standard. By completely decoupling communication between processes and method calls, Erlang allows individual processes access to a larger domain of code than is the case with Actors. The code that an Actor can invoke is only the code that is within the current behaviour, though languages such as SALSA do rather blur the line between message sending and method calling when an Actor “calls” a method within itself.

One of the main reasons for asynchronous communication is to allow for a higher level of parallelism. But, if several results have to be assembled before any meaningful computation can take place then that amounts to the use of several continuations which may be expensive in terms of CPU resources and memory. On the other hand, by using several continuations, the computation is pipelined ensuring that under higher loading, more can be done in parallel.

Nevertheless, given a choice between greater parallelism and easier comprehension, comprehension should win every day. Compilers will get more powerful at performing transformations to achieve greater
parallelism and what is the point of greater parallelism if it cannot be understood? Therefore, I decided that the GLINT language would have a call-and-return mechanism incorporating inter-actor communication.

The consequences of this decision are quite extensive. Firstly, this means that message sending is no longer asynchronous. If communication is asynchronous then it means the sender can complete the sending of the message without the receiver receiving the message. But with a call-and-return mechanism, not only can the send not complete until after the receiver has received the message, it cannot complete until it has had some sort of result from the receiver. Secondly, with Actors, sending messages can be done in any order: by virtue of the asynchronous communication, there is no need to express any kind of ordering between the different messages. With synchronous communication an ordering is needed: it becomes much more important for an Actor to be able to express which result is sought first and so forth. Whilst this ordering can be expressed simply by orderings of statements, I have always enjoyed HASKELL’s where clauses which allow statements to be expressed in any order but for the compiler to analyse the dependencies between the statements and to reorder the statements so that the dependencies are satisfied.

Finally, there are situations in which anonymous Actors are called for specifically for the purposes of continuations and deadlock avoidance. I think a lightweight syntax is important here but not so light as with SALSA and certainly clear enough so that the programmer is aware additional Actors are being constructed.

5.3 Components of the GLINT Language

Any program in the GLINT language consists of a set of Actor definitions. Every Actor definition gives that Actor a name which can be thought of as the type of the Actor. The definition of an Actor is analogous to the definition of a class in JAVA. An Actor definition can simply define what actions the Actor performs when it receives a message or it can subdivide those actions into different actions which depend on what type of message the Actor receives. Method invocation is modelled as a particular type of message but there are further mechanisms for control flow allowing finer grained control of the mapping of message type, structure and content to actions the Actor performs.

The approach here is to firstly examine the individual components that allow for the expression of actions and to explain the meaning of these components. Following this, it will be shown how these components can be assembled to define Actors and the different types of Actor definitions.

5.3.1 Actor Types, Variables and Assignment

There are two different types of names. A variable starts with a lower case letter whilst the name of the type of an Actor starts with an upper case letter. Assignment happens with the standard ‘=’ symbol.

**Example 5.2 Assignments**

```
1. x = y
2. z = Foo
```

There is just one “special” variable which is called this which as you would expect, refers to the name of the current Actor.

5.3.2 Actor Instances and Singleton Actors

The language supports both Actors that have to be constructed explicitly via a new call and Actors that are singletons in the same way as SCALA has singleton Objects. Singletons are referenced by their type name whilst instances are constructed and have their instance name bound to a variable. As the above example shows, the name of a singleton, Foo, can be bound to a variable.

5.3.3 Method Calls

Inspired by the Ambient calculi, arguments are enclosed by angle brackets rather than parenthesis. There are four types of method call:

- Constructor call. This creates a new instance of the indicated Actor type and binds the name of the instance to the supplied variable.

- Actor call. This is a call directly to an Actor. This can be either a singleton Actor or an instance. The Actor is expected to be able to receive the message directly which means that the Actor is not expected to have any declared methods.
• Method call. This is a call to a singleton or instance Actor where a method name is provided by the
sender. It is expected that the Actor has declared a method matching the name of method called.

• Local method call. This is a call in the body of a method to another method within the same Actor.
This is interpreted as the Actor’s thread directly invoking the local method rather than sending a
message to itself to cause the method to be invoked later.

In all of these cases, if the Actor cannot receive the message sent then it is simply queued until such
a time as the Actor can receive the message. This means that it is valid to try and invoke methods on
a particular Actor, the type of which does not declare any such method because the Actor may, through
types it to its behaviour become of a type that does support such methods. For example, an Actor that is an
instance of Frog may, if very lucky, receive a call to its kiss method at which point it may become a Prince.
Princes can kneel, fight and ride horses whereas Frog’s can’t. But it is valid to try to tell the Frog to ride a
horse before it has become a Prince.

Example 5.3 Method Calls
1. frog = new Frog<Green,Slimey>
2. fiona = Princess
3. farquaad = frog.kiss<fiona>

Example 5.3 shows the use of a constructor to create the frog, the assignment of the name of the Princess
type to fiona and shows a method call.

5.3.4 Where Clauses

A where clause is a set of assignments. These assignments can be in any order as the compiler will reorder
them in order to meet dependencies. Each variable can only be assigned to once in a where clause. In
Example 5.4, the assignments from the previous example are reordered but are nevertheless acceptable.

Example 5.4 A Where Clause
1. where
2. farquaad = frog.kiss<fiona>
3. frog = new Frog<Green,Slimey>
4. fiona = Princess

5.3.5 Becoming Death

It is possible that an Actor should die. The original description of Actors stated that a behaviour should
specify its replacement, but it is commonly interpreted that if no replacement behaviour is specified then
the current behaviour is still valid for the next message. Therefore, explicit handling is required when the
Actor wants to die rather than remain as it is currently or become something else.

The keyword die causes the current Actor to die: it will never more accept any messages. The keyword
become specifies explicitly a replacement behaviour. The example 5.5 shows what might occur within the
kiss method of the frog, whereas the die keyword may feature in the slay method within the Prince Actor.

Example 5.5 Becoming
1. become prince
2. where
3. prince = new Prince<>

5.3.6 Simple Methods

A simple method is declared within an Actor and has a name and a list of parameters. It can have one where
clause which must come last, an optional become or die statement and it can have one return statement.
A return is simply represented by unprefixed angle brackets. Example 5.6 shows the kiss method in its
entirety. Note that there is no way of invoking a method and not capturing the result: this is why the final
method call is assigned to the variable any which plays no further role.
Example 5.6 Kissing

```plaintext
1   kiss (Princess princess)
2       kiss
3       become prince
4       where
5       prince = new Prince<princess>
6       title = Camelot.obtainNextTitle<>  // any = prince.setTitle<title>
```

Because this example has no return statement and because it is impossible to call a method and not capture the result, a result must be returned to the caller. Thus where no return statement is specified, a default return statement is used in which the name of the Actor is returned to the caller, i.e. returning this.

5.3.7 Complex Methods

The reader may have noticed a stunning lack of any ability to express control flow. Uniquely, control flow is only implemented at the top level of a method, creating a large switch-case statement. Each case has its own predicate and can have its own where clause, become or die statement and return statement. There must be a default case with no predicate which corresponds to the simple method described above.

Example 5.7 Complex Kissing

```plaintext
1   kiss (Princess princess)
2       kiss princess.isTall<>  // any = prince.addBeard<>  // kiss
3       become new ShortPrince<princess>
4       kiss princess.shoeSize<> .greaterThan<10>  // become prince
5       where
6       prince = new Prince<princess>
7       any = prince.setTitle<title>
8       become prince
9       where
10      prince = new Prince<princess>
11      any = prince.setTitle<title>
```

Example 5.7 shows a complex method with two predicated cases. Standard logical constructs and, or and not can be used in plain text. This example also demonstrates method chaining and the ability to specify in the become statement a method call or new instance. If the princess that kissed the frog is tall then the frog becomes a short prince, otherwise if the princess’s shoe size is more than 10 then the prince has a beard otherwise the frog becomes a prince as in the previous simple method example.

A complex method can have a where clause at the end of the method which defines variables which are accessible to all the predicates and their bodies. Furthermore, the predicate itself can access variables that are defined within the where clause of the predicate body itself: the compiler is required to extract from the predicate body’s where clause the minimum set of assignments necessary to be able to satisfy the predicate.

Predicates are executed in order. In [13], predicate dispatching is defining where the ordering between tests is constructed as a result of logical implication. Thus with such a system, the order that the predicates appear in is irrelevant. Instead, the predicates are analysed and an ordering of implication is established. Whilst this guarantees that there will not be cases that can be missed out due to incorrect orderings (i.e. the previous cases cover a wider range of values than the current case so if the current predicate is satisfiable then one or more of the previous predicates are satisfiable so the current case will never be executed), given complex predicates, it would seem to be potentially unobvious which order the predicates are tested in. Further, such a system was deemed too complex for consideration here so the more widespread mechanism of testing the predicates in the order in which they appear is used.
5.4 Putting it all together

Having introduced all the possible components, all that remains is for the definition of the Actors themselves. There are four possible types of Actors:

- Singleton Actor without methods.
- Singleton Actor with methods.
- Instance Actor without methods.
- Instance Actor with methods.

Because singletons are not instantiated, they can not require parameters to the Actor itself, whereas instances must have zero or more parameters. An Actor definition is introduced by the keyword define. Defining an Actor must give the Actor a type name unless the Actor is an anonymous inner Actor in which case it must be either part of a return or become expression or assigned to a variable in a where clause.

Example 5.8 shows a counter. By virtue of the fact that it has a value within it, i.e. the current value of the counter, it must be an instance Actor. This allows it to be started from any value.

Example 5.8 An Instantiable Counter Actor

```plaintext
define Counter(Number value)
  become new Counter<value'>
  where
  value' = value.plus<One>
```

This is the most confusing of all the examples. The parameters after the type name indicates the parameters that the constructor call must provide. In other words, creating a new instance simply provides values for these parameters. After that, the resulting instance is returned. Because the Actor defines no methods, the become and where clauses are executed only when the Actor instance next receives a message. Thus the definition here actually specifies the receiving of two messages: one to create the instance and one to cause the body of the Actor to execute. Example 5.9 shows the counter in use.

Example 5.9 The Counter Actor in use

```plaintext
define Something
  where
  c = new Counter<Zero>
  c' = c<>>
  c" = c<>>
```

In this example the Actor Something, upon receiving a message creates a new Counter, counting from zero and increments it twice. Something is a singleton Actor because its definition takes no parameters. Again, it has no methods. Thus for its body to be executed, it just needs to receive a single argument-less message. Note that every message sent to the counter instance will receive, as a result, the name of the counter instance. Thous the last call to the counter could have been a message sent to c'.

It might be considered that just sending a message to the Actor doesn't really express clearly enough what actions are expected of it. So it might be an idea to provide a wrapper around the counter which has an inc method which causes the counter to increment. This is shown in example 5.10. The method inc has a parameter which is the counter and it takes care of incrementing the counter. Note that it explicitly returns the result coming back from the counter.

Example 5.10 A Wrapper for the Counter

```plaintext
define CounterWrapper
  inc (Counter c)
  inc
  <c<>>
```

Finally, a more powerful counter is shown with two methods, inc and dec. As with the original counter, this must be instantiated but now has explicit methods rather than an implicit body. This is shown in example 5.11.
Example 5.11 A More Powerful Counter

```plaintext
define Counter (Number value)
  inc ()
  inc
  become new Counter<value.plus<One>>
  dec ()
  dec
  become new Counter<value.minus<One>>
```

5.4.1 Anonymous Inner Actors

Anonymous inner Actors follow the same four patterns as named Actors, the only difference being that they do not have a name and they are used directly where they are defined. But because they do not have a name, they cannot be used with a constructor call which reduces the four patterns to two. Having spent some time with the language, I would not describe this as a limitation: it is potentially quite confusing to define an anonymous inner Actor which then has to be instantiated before it can be used. Further, the main purpose of constructors is to provide state to the Actor instance. Because of the nesting of anonymous inner Actors within other Actors, at the point at which they are created, they inherit the surrounding scope and so can capture any state as they might need. Example 5.12 shows how such an Actor can be defined. Upon the LockableDoor being shut, the returned value of the shut method is an anonymous Actor. This Actor does not require instantiating and has a single method lock defined. Upon locking, the result is a LockedDoor. In this way the LockableDoor imposes an ordering on the method calls: it guarantees that the door cannot be locked until it has been shut.

Example 5.12 Example use of an Anonymous Inner Actor

```plaintext
define LockableDoor
  shut ()
  shut
  <lockable>
  where
  lockable = define
    lock ()
    lock
      <new LockedDoor>>
```

5.4.2 Movement

Movement is one of the simplest modifications I have made to the traditional Actor model. If it is desired to move an Actor to another host then it needs to be possible to express hosts and it needs to be safe to move the Actor; in particular, all the other Actors that know about the Actor that is to be moved must not have their knowledge invalidated. If it were possible to just grab hold of an Actor and move it to another host then all the other Actors that know the name of the moved Actor would find, when trying to message the moved Actor, that it doesn’t respond thus blocking the calls. Therefore, the only point at which it is safe to move an Actor is when it is certain that no more than one other Actor knows the name of the Actor. This corresponds to two situations: after an Actor has been created via a constructor call; and when an anonymous inner Actor is defined.

Further, movement must be specified as part of a become clause. This means that the Actor must be becoming an Actor that no one else knows of and that the becoming effectively establishes a link in between two hosts: any messages that get sent to the Actor’s name in the original host will be forwarded to the target host where they will be sent to the target Actor of the become clause and any response will be transported back to the original caller. Having moved to the external host, the names of any Actors that the Actor took with it will not be valid so the Actor will have to start afresh. What will be available to the Actor in its new host are all the singleton Actors that are loaded into the runtime environment of the host. This will typically include Actors for truth and numbers amongst others. Thus the primary reason for movement would be to gain access to singleton Actors that are not available on the current host. Imagine a situation where two hosts have their screens next to each other and there is a Ball Actor which draws itself on the screen as it...
moves. Here, Screen is a singleton Object as each host has only one screen. Having reached the edge of one screen, the Ball moves onto the other host and continues drawing. Eventually it will reach the far side, bounce and then when appropriate will move back to its original host. This is shown in example 5.13. The program is very simplistic and does not handle bouncing at all but it shows the principle of moving to the other host. Because numbers are singletons (i.e. there is never any need to have more than one Actor representing any particular number), the numbers that are in the Ball as it moves to the other host will refer to valid names on the other host.

Example 5.13 A Mobile Ball

```
define Ball (Number x, Number y)
draw ()
draw x>greaterThan<Zero> and x<lessThan<Ten>
become new Ball<x,y>
where
  any = Screen.clear<> . drawCircleAt<x,y>
draw
become new Ball<Zero,y> in host:otherHost
where
  x' = x.plus<One>
```

The limitation of not moving Actors that are known by an Actor that is moving is a sensible one. There is the possibility of under-the-bonnet magic but there is an argument for not moving Actors unless they want to be moved and there’s the issue of what happens if multiple Actors know the name of a particular Actor: should the Actor be moved or not if one, but not all, of the Actors that knows of it is moved? There is the possibility that there should be automatic management of known names so that any calls are forwarded back to the original host but such additions smack of the kind of built in black magic to which I really object: a language should not cloud a programmer’s mind with notions of automatic conversions and context sensitive semantics. It would also make it harder for the programmer to express their intent of what is to move and what isn’t.

Finally, this limitation is not insurmountable. If an Actor wants to move and there are a set of Actors that it wants to take with it then there is no reason why the Actor can’t ask its comrades to move with it too. The names of the Actors are not modified at all during transit so if the other Actors did decide to move too then the names would be preserved and communication between the Actors would still be possible on the far side. It is simply a matter of delegating to each Actor in question whether it is happy to move and precisely how it wants to go about achieving the movement. This technique is used in the extended example in appendix B.

5.5 Examples

The two classic examples that will be demonstrated here are Factorial and Fibonacci. There is a much more extensive example that demonstrates some of the more subtle aspects of moving multiple Actors simultaneously in appendix B. Example 5.14 shows Factorial expressed in the GLINT language. It looks and works just like a functional program. The reason for this apparent ease of expression is that every call other than the initial call is a call to the factorial function from within the very Actor in which it’s defined, i.e. the recursive calls are all local calls and are executed directly by the Actor’s thread rather than by message passing. Contrast this with example 5.15 which shows the same Factorial function but for a pure Actor language. Here, the recursive call must be expressed via a message sent to the Actor which cannot be processed until after the current message has been processed which means that a continuation must be used in order to get the eventual result back to the caller. The control flow is more complex, less intuitive and less comprehensible and is much further removed from standard languages. A novice programmer would almost certainly try to implement Factorial recursively and be quite stumped when it deadlocks and never completes.

Fibonacci is a more complex example because it requires two recursive calls. Again, the implementation in GLINT is quite straightforward and mirrors the classical functional definition closely. This is shown in example 5.16. Once again, the same function is rewritten for a more typical Actor language and once more, the implementation looses clarity and becomes much larger. This is shown in example 5.17. Note in particular that the continuation has to be constructed so that it can receive two messages representing the results of the two recursive calls. If Fibonacci did not use addition and used an operator that was not
Example 5.14 Factorial in the GLINT Language

```plaintext
define Maths
factorial (Number n)
  factorial Zero.eq <n>
  <One>
  factorial
  <n.times<r>>
where
  r = factorial<m>
  m = n.minus<One>
```

Example 5.15 Factorial in a standard Actor Language

```plaintext
actor Factorial {
  receive (number, consumer);
  if (0 == number) {
    send 1 to consumer;
  } else {
    let continuation = {
      receive (result);
      send result * number to consumer;
    } in {
      send (number - 1, continuation) to Factorial;
    }
  }
}
```

commutative then this would be much harder as the continuation would need to be able to identify which result came from which recursive call.

Example 5.16 Fibonacci in the GLINT Language

```plaintext
define Maths
fib (Number n)
  fib One.eq <n> or Zero.eq <n>
  <n>
  fib
  <l.plus<r>>
where
  l = fib<n.minus<One>>
  r = fib<n.minus<Two>>
```

5.6 Conclusion

The GLINT language is a small but useable Actor-based language with support for mobility, methods and predicate dispatch. The language departs significantly from Actors in terms of asynchronous versus synchronous communication and gains more concise and intuitive expressions. It also benefits from being able to directly invoke methods local to the Actor rather than having to send messages, meaning that recursion in particular can be expressed more easily. On the downside, the Actors are not as parallel as in a purer Actor language and it is necessary to ensure that method calls are made in the correct order. As far as is possible, the compiler is required to aid the programmer in this regard, allowing the programmer greater freedom of expression than would otherwise be the case.

It is possible in the GLINT language to define anonymous Actors which means that where necessary, continuation passing style can still be used. Additionally, it has added mobility of Actors in a tightly controlled and hopefully intuitive way.

The language is not without its faults though: the inability to pass parameters to a method-less Actor is an oversight though the solution of declaring a method explicitly generally leads to clearer code. Actors without methods can look quite confusing in terms of understanding which messages result in what actions.
Example 5.17 Fibonacci in a standard Actor Language

```plaintext
actor Fibonacci
receive (number, consumer);
if (2 > number) {
    send number to consumer;
} else {
    let continuation = {
        receive (resultOne);
        let innerContinuation = {
            receive (resultTwo);
            send resultOne + resultTwo to consumer;
        } in {
            become innerContinuation;
        }
    } in {
        send (number - 1, continuation) to Fibonacci;
        send (number - 2, continuation) to Fibonacci;
    }
}
```  

On the whole though, the language is expressive, does not hinder expression of intent in any particular way and presents some novel solutions to what are arguably detractions from standard Actor languages.
Compiling to Ambients from an Actor language is not something that has been attempted before and was much harder than I had anticipated. The GLINT Compiler is roughly in two halves. The first half contains the parser and nine AST walkers. These serve to both validate and modify the parsed AST, getting the AST closer, step by step to a form from which Ambients can be generated. However, the transformations applied are not significantly more advanced than rearrangements, reorderings and simplifications. As a result, there is still quite a jump from the most transformed state of the AST to Ambients and this is the second half of the GLINT Compiler.

Figure 6.1 Structure of GLINT Compiler
6.1 Transforming the AST

The parser generator I used, ANTLR, has the ability to walk over the generated AST with a transformer that is itself defined using the very same syntax as the grammar and hence is generated by ANTLR. The transformers are extremely flexible: it is possible to insert actual JAVA code at almost any point in the matching process and it is also possible to transform the AST not only directly from within JAVA, but also from a more natural syntax that ANTLR makes available. The result of this is that it becomes very easy to express complex transformations in a very readable and comprehensible manner: much improved from the alternative of having to use the visitor pattern.

Having been parsed, the AST created is fed into a series of transformers. Some simply make checks and generate errors if there are mistakes uncovered in the supplied program, whilst others transform the tree in a number of ways. In total, nine transformers are used.

6.1.1 Checking Method Definitions

The first transformer is a very simple checker. As shown in the previous chapter, all the possible cases within a method must start with the method name. For example, the method fib in example 5.16 uses two cases, one with a predicate and the default case, and both cases start with the method name, fib. This transformer simply checks that all methods are defined correctly in this particular way.

6.1.2 Checking for Variable Definitions

The next transformer does two jobs. Firstly it checks that all variables that are used are declared either within the current scope or within some parent scope and secondly it separates out Actor calls from local method calls. This is harder than it may sound as the algorithm used will assume that an Actor call is what is intended if the variable containing the name of the Actor being called is nearer (in terms of scope) than the method definition of the same name as the variable. The complexity is that because of the looseness of declarations of variables, the variable could very well be bound after (syntactically) the Actor call.

Example 6.2 shows that the binding of the variable post which appears after the use of the variable means that the use is interpreted as an Actor call, whereas if the where clause is removed then it is interpreted as a local method call. The keyword this can be deliberately used in these situations to resolve any ambiguity as, of course, can giving variables sensible non-clashing names.

If both the where clause and the local method post are removed then this transformer will complain that the variable post is not defined and no further processing will occur.

Example 6.2 Local Method Call versus Actor Call

```
1 define RoyalMail
2 createPostbox (Mail m)
3 createPostbox
4 <post<m>>
5 where
6 post = new Postbox<>
7 post (Mail m)
8 post
9 <Bin.deposit<m>>
```

6.1.3 Checking for Cycles in Variable Definitions

The third transformer is a simple checker. It ensures that there are no cycles in the definition of any variables, such as the ones shown in example 6.3.

6.1.4 Rewriting Predicates and Constructor Calls

The next transformer performs two jobs. Firstly it converts all constructor calls to method calls to a method called _new. This makes implementation easier as creating an Actor instance now becomes nothing more than calling a particular method. The transformation is safe because it is not permitted in the parser to declare methods that start with an underscore.
Example 6.3 Cycles in Variable Definitions

```java
Example 6.3 Cycles in Variable Definitions
1. define Cycles
2. 
3. where
4. a = new LeftPedal<b>
5. b = new RightPedal<a>
```

Secondly, the transformer rewrites all predicates into method calls. This, again, is more complex than I first thought. The idea is to get rid of the and, or and not logical connectives that are permitted in the predicates but different behaviour is required depending on what appears on the left hand side of the connective. If it is a simple variable then the predicate is transformed into a method call with that variable being the Actor on which the connective is called as a method, with the right hand side as the argument to the method. But if the left hand side is itself a method call then it cannot be wrapped in the same way as it breaks the format established for chains of calls and would thus complicate all the succeeding transformers. So the current method call AST for the left hand side must be traversed and then a new method call appended onto the end in order to correctly construct a chaining. The new method call uses the connective as the method name to be called.

6.1.5 Rewriting Inline Stuff

In the GLINT language, it is acceptable to have complex expressions as part of the return statement and become statement, to chain method calls and Actor calls together in almost any sensible fashion, to have complex expressions as arguments to calls and to have complex call chains for predicates. This fifth transformer removes all inline expressions and all method chains. It assigns to extra variables the intermediate values and ensures that there are no method call chains anywhere and that expressions within predicates, become and return statements are simple variables. The extra variable definitions are added to relevant where clause, creating where clauses if necessary.

At the end of this transformer, the resulting AST is much simpler and much more approachable in terms of Ambient generation as there are fewer types of expression to deal with and calls to methods or Actors can now only appear within where clauses.

6.1.6 Moving the Where Clauses

Up until this point, the where clauses have been at the end of bodies. This transformer moves them to the front. Ultimately, there is little need for this transformer, but from a conceptual perspective, it helps to have variables declared in a block before the body in which they are used. The one significant change this transformer makes is that the different cases of a method with predicates are modified so that the subsequent case is a child of the current case. This makes a substantial difference when it comes to the generation of the Ambients because it means that each case is responsible for either invoking itself or the next case. A standard recursive algorithm can then delegate to the next case the construction of the encoding for the remaining cases.

6.1.7 Further Modifying Predicates

In the language definition, predicates are allowed to access variables that are defined only within the where clause of the case the predicate guards. It is important though that only the variables that are needed for the predicate get bound before the predicate is tested otherwise the effect would be to execute much of the case before the predicate has even been evaluated. Therefore, this seventh transformer identifies which variables are and are not needed for the predicate and adds where clauses immediately in front of the predicates so that the predicates can be correctly executed. Similarly, where clauses are removed from cases if they have been emptied as a result of this shuffling of variable definitions.

6.1.8 Reordering Variable Definitions

This eighth transformer is the last that makes any modifications to the AST. It is also the most important. As with several of the other transformers, it analyses the variable definitions in the where clauses and models the dependences between the variables. However, unique to this transformer is that it then treats this model as a many rooted acyclic graph and traverses the graph breadth first, rewriting the where clauses. The
result is that the variable declarations are ordered by dependences so that a top to bottom reading of the
where clauses will never result in a variable being used before it is defined. The other consequence of the
breadth first traversal is that there is a greater degree of predictability than with a depth first traversal: with
depth first, the programmer would have no control over the eventual ordering of the variable definitions
other than that they are defined before they are used. With breadth first, the programmer knows that all
those variables that have one dependency on a previously defined variable will be defined before any that
have two dependencies on previously defined variables and so on.

6.1.9 Creating Ambient Generators

This last transformer was rewritten several times. Initially I had hoped to simply keep adding transformers,
getting ever closer to the target Ambient representation. However, it soon became apparent that the com-
plexity of the encoding would not permit this approach so having got this far, this last transformer interprets
the AST, creating class instances that can perform the resulting steps needed to generate the Ambients. This
last transformer also checks that where inter-host mobility is expressed, it is expressed correctly and applies
to Actors for which it is valid to move as discussed in section 5.4.2.

6.2 Encoding into Ambients

Ideally, the encoding would be able to take each component of the Gλλt language and transform it so as to
be expressed in Ambients, ready for the GLINTVM to execute. It would then be a simple matter of com-
posing the compilation of these separate components to achieve the compilation of the whole program. Sadly,
that is not possible with the encoding presented here: there are far too many cases where small differences,
for example in specifying a replacement behaviour, result in larger changes in Ambients, extending outside
of the scope of the encoding of the component itself.¹

One of the trickiest aspects of the encoding was ensuring that no communication interference occurs.
There are two reasons why this is hard to achieve. Firstly, a local input or output can pair with three
different types of output or input respectively; and secondly, it is impossible for this to be narrowed: if a
parent output was allowed to pair with a child input then it would be much safer as the following shows:

$$ (x)^a . P | a[ (n)^1 . Q ] $$

Here there is much less opportunity for interference, but such a pairing is not permitted: the parent output
must pair with a local input and the child input must pair with a local output. Consequently there are
many cases where I invented extra parameters so that communications carry unique arities thus avoiding
communication interference. But even this is not a full-proof solution: given a replication, it is possible for
two copies of the replicated process to have reached the same instruction which could be a communication.
Unless that communication targets a uniquely named child Ambient, it is impossible to avoid the potential
for communication interference.

Additionally, there is the potential for interference in actions too. Consider the following:

$$ \text{out} a . \text{out} b . 0 | \text{out} a . \text{out} c . 0 | d[ a[ \text{out} d . P ] ] $$

Depending on which out a action the out d action pairs with determines which of the subsequent co-actions
is executed. When constructing paths via communication and then executing the path, this can be hard to
spot, hard to debug and of course, is non-deterministic.

6.2.1 Basic Properties of the Encoding

The basic idea is that the top level host Ambient is treated as the heap within which Actors exist and Actor
instances are created. An Actor gets translated into an Ambient of the same name which is placed directly
within the top level host Ambient in parallel with all the other actor-ambients. Within each actor-ambient
exist several processes and Ambients that deal with marshalling of messages; an Ambient that represents
the current behaviour of the Actor; and within this behaviour-ambient, an Ambient for each method in the
Actor.

This is a simple encoding which does not make as extensive use of the nesting abilities of Ambients as
it could: with this encoding, all the actor-ambients exist at the same level and there is no notion of one

¹The consequence of this is that the resulting required data flow within the compiler is enormous. Whilst well engineered (as
demonstrated by the relative ease with which I could modify the encoding during development), it did somewhat become a “you know
you’re in trouble when…” moment after writing the nth method with m + 7 parameters!
Actor being owned by another by virtue of it being within another Actor, but such ideas certainly could be expressed: ownership type systems, amongst others, could be implemented by modifications to the encoding. The design and purpose of the encoding is to be comprehensible, reliable and successful, not at this point to explore experimental type systems or highly complex encodings.

In order to simplify the encoding as much as possible, Actors that do not have methods are encoded as Actors that do have methods, precisely one method, called _NONE_. This is reflected in both the encoding of the Actors and the encoding of the messages sent to method-less Actors. Therefore, with this modification in place, all Actors can be treated as having methods.

This now means that there are only two types of Actors to be considered: those that have parameters and need instantiating via a constructor call and those that are singleton. As already explained in the previous section, the construction of Actor instances via the new keyword is modified into a method call to a method called _new_. Therefore this _new_ method must create an instance of the Actor as defined and return the unique name of the generated instance. Thus the encoding of a singleton Actor and the instance of a parameterised Actor is identical: all that is needed for the parameterised Actor is that the construction of the instance is wrapped in a singleton Actor encoding so as to present the _new_ method. Whilst the encoding may be identical, the behaviour is not: a singleton Actor is lite and working from the start: there are no prefixes to the encoding to prevent the processes from being created and started. However, with the parameterised Actor, the purpose of the wrapper and _new_ method is precisely to prefix the encoding so that the processes are not started until after each call to the _new_ method.

To look at the solution in a different way, all Actors are singleton and respond to the same name as they are declared with in the program source. Actors that have parameters are singletons that have one method called _new_ and which, as a result of this method, return the name of a new and unique Actor instance which corresponds to the definition given in the program source.

Given that arguments and results are passed from and to Ambients that represent (or encapsulate) messages, those message-ambients need to be able to move in and out of the actor-ambients. The latter part is the tricky part because movement can only be achieved by a co-action pairing with the action so if a message-ambient wants to move out of an actor-ambient then it needs some help from the outside. Cue the first of three top-level replicated processes,

\[ !(\text{cap}) \cdot \text{cap} \cdot 0 \]

which exist to enable message-ambients to move around as they need. The idea is that upon letting the message-ambient go, the actor-ambient will send, as a message, an out co-action to its parent which will be picked up by the above process and duly executed, allowing the message-ambient to escape.

### 6.2.2 Marshalling Callers

Once a caller-ambient has made it out of the Ambient in which it was created it must enter the Ambient of its target Actor. Because the target actor-ambient is a sibling of the message-ambient at this point, the message-ambient cannot communicate directly with the actor-ambient. So a second top level replicated process is used, this time:

\[ !(\text{caller}, \text{method}, \text{actor}) \cdot (\text{caller}, \text{method})^{\text{actor}} \cdot 0 \]

This allows the message-ambient to send a message to its parent which has three arguments: the name of the message-ambient, the name of the method it hopes to call and the name of the Actor in which it wants to invoke the method. This is picked up by the above process and dutifully relayed on to the actor-ambient which receives the name of the message-ambient and the method. The actor-ambient must now check whether that method exists, if it does, obtain an exclusive lock on the actor-ambient to prevent any other message-ambient from being accepted whilst the method invocation is unfinished and permit the message-ambient to enter.

The first subsystem to explain is the locking mechanism, which is shown in figure 6.4 and exists directly within each actor-ambient. The basic idea is that to obtain the lock you must collect a magic token from the starter Ambient. This you carry with you and then, when you want to release the lock you create an Ambient named with the magic token which enters the starter Ambient and tells the starter what the name of the future behaviour is. In this way, whoever collects the magic token will also learn what the name of the current behaviour is. It should be clear that after the magic token and behaviour have been taken for the first time, the input at the start of the replication reduces with the output in the other process thus causing the co-action to be the next instruction to execute. In the meantime, the replicated process has unwound so when the output at the end of the replicated process occurs, it’ll pair with the waiting unwound copy which
will enter the same state, awaiting a pairing for the \texttt{out} co-action. Thus the unwound copy is first reduced by the output preceding the imminent death of the previously unwound process. The skeleton process at the bottom of the figure shows how this token is obtained and how it is returned.

\textbf{Figure 6.4} The Locking Mechanism employed for Actors

\begin{verbatim}
starter[ (\nu go)(
    \{starter\} . 0
    | \{go, behaviour\} . !(starter) . \in go . (behaviour)^go . \{go, behaviour\} . \{starter\} . 0
    )]

... (go, behaviour)^starter ... (future) . go[ in starter . (future) . 0 ]
\end{verbatim}

Given that the actor-ambient will be messaged, informing it of the caller and what the caller wants to invoke, the actor-ambient needs some way of checking that the method does indeed exist before it obtains the lock and lets the message-ambient in. This is the next subsystem and it is shown in figure 6.5. This also is a process directly within the actor-ambient.

\textbf{Figure 6.5} Checking for the Existence of Methods

\begin{verbatim}
checker[ !(methNameCap) . !methNameCap . 0 ]

| \in methOne)^checker . \in methTwo)^checker .

!(callerName, methodName).

\{ methodName[ in checker . out checker . 0 ]

| \out methodName . (go, behaviour)^starter . \in callerName ....
\}
\end{verbatim}

Here, the Ambient called \texttt{checker} is the Ambient doing the regulation. Upon receiving some sort of capability it will forever more execute that capability. The main marshalling process feeds into the \texttt{checker} the names of the methods in the Actor, here imaginatively called \texttt{methOne} and \texttt{methTwo}. Then, when the marshalling process is informed that a caller wants to call a particular method it creates an Ambient with the name of the method requested (bound to \texttt{methodName} by the input) which tries to enter and exit the \texttt{checker} Ambient. If it can’t get in then it will block, blocking the rest of the marshalling process. Fortunately, this is within a replication so other method calls can still be examined. If the Ambient can get inside the \texttt{checker} then it will be allowed back out as that is permitted by the marshalling process directly. Having got out, the marshalling process knows that the method call is valid so it grabs the magic token from the \texttt{starter} and allows the caller in.

Note that it doesn’t matter if two message-ambients try to invoke the same method at the same time: two Ambients named with the method name will be constructed which will both enter the \texttt{checker}. If they can both get in then they can both get out and it doesn’t matter if they pair with the other marshalling process at this point. Only one of the marshalling processes will be able to obtain the magic token and the other will block. All the message-ambients are required to have unique names so only one will be allowed in. When the token is returned, the other marshalling process will grab it and will then let its message-ambient in.

\subsection*{6.2.3 Changing Behaviours}

Within the actor-ambient sits the processes described above and a behaviour-ambient. The behaviour-ambient encapsulates the current behaviour of the Actor. When an Actor starts up, the behaviour-ambient will contain method-ambients, either the \_NONE\_ method in the case of a method-less Actor or the methods
as described by the Actor. As a result of a method call, the Actor can do three things with regard to its behaviour. Firstly it can keep the same behaviour in which case nothing in the actor-ambient needs to change, secondly it can die in which case the current behaviour of the Actor needs to change so that no further method calls can be accepted and thirdly it can become something else.

Having allowed a message-ambient in, the marshalling process passes the details of the message-ambient to the current behaviour which takes care of further processing of the message. Having been processed, the behaviour-ambient tells the marshalling process to allow the message-ambient back out and also informs the marshalling process of the name of the future behaviour-ambient. This is the purpose of the (future) input in example 6.4 and it ensures that the marshalling process that next receives a message-ambient for a method that exists gets the correct name of the current behaviour via the starter magic token system.

In the case of the Actor dying, any messages that are subsequently sent to the Actor result in the message blocking as the Actor refuses to accept any messages. Note that it doesn’t matter if the checker as described above accepts that the method exists or not: if the method does exist it simply means that the method existed at startup but in the meantime the behaviour has changed and the dead behaviour will not contain the method definitions so the message will block at the point at which the message-ambient tries to enter the dead behaviour. If the method never existed then the checker will inform the marshalling process of that and so the message-ambient will never even be allowed into the Actor. In either case, the Actor is behaving as if it is dead, not accepting any method calls whatsoever.

If an Actor becomes another Actor then it is interpreted as the behaviour of the Actor becoming such that all messages are forwarded to the target of the become statement. Therefore, the original actor-ambient must modify itself so that it will accept any message-ambient, wrapping the message-ambient and forwarding it on to the target. It is then up to the target actor-ambient to marshal and accept only valid method calls. Therefore, the checker needs modification: it is sent an in * action to allow it to accept as valid any method call and a new behaviour is constructed, the name of which is sent to the marshalling process which then ensures that the starter is informed of the new behaviour via the return of the magic token. After this, every message-ambient is accepted, it enters the actor-ambient and the new behaviour-ambient which wraps the message-ambient and forwards the message-ambient on to the actor-ambient that is the target of the become statement.

Note that having become another Actor, it is impossible for any further change to occur in the original Actor: the Actor that it has become can obviously undergo further change, but the original Actor forevermore forwards method calls to the target. Similarly with a dead Actor, it will never awaken from its slumber.

### 6.2.4 Receiving Method Calls

The message-ambient is one of the most crucial parts of the encoding and is quite complex. It must be capable of robustly moving out of the Ambient in which it is created, negotiating with the target actor-ambient, moving into the actor-ambient and the behaviour and ultimately the method-ambient, depositing any arguments, collecting any result and making the return trip. It must also be capable of being safely wrapped by a behaviour that wants to forward the message as a result of a become statement.

The idea behind the message-ambient is that one Ambient takes care of escaping from where it was made, negotiating with the target and entry into the target, behaviour and method where it then tells the method the name of the payload Ambient within it. This payload-ambient is then allowed to escape from the message-ambient, it tells the method the arguments, collects the result and makes the return trip on its own. In this way, a series of method calls are encased one inside the other with the payload-ambient from one being the message-ambient of the next. The reason for this design is that in the case of wrapping the message-ambient for forwarding, it is necessary to construct an Ambient which contains the processes that the payload-ambient contains for sending the arguments, collecting the result and continuing on. Without the open primitive, this cannot be done, so wrapping the message-ambient is the solution. Then, when the wrapper reaches the correct target method it tells the method how to access the payload. In a mechanism very similar to the Russian-dolls in example 4.10, the wrapper constructs a path via recursive descent into the wrapped message-ambient which then allows the payload-ambient to escape. In this way, multiple forwardings can work correctly.

Figure 6.6 shows the complete marshalling process within the actor-ambient. After verifying the method’s existence and letting the message-ambient in, it then asks the message-ambient for its leavingName, a.k.a. the name of the payload-ambient. It sends the message-ambient name, method name and payload-ambient name to the current behaviour and receives back a path which it sends on to the message-ambient. The message-ambient then executes this path which moves it either into the behaviour and then the method Ambient or into the behaviour and wrapping system if the behaviour is forwarding the message. To allow the subsequent method callers to do their work and to allow the payload-ambient to leave, the marshalling
process receives two communications from the behaviour. Firstly a triple of \texttt{capOut}, \texttt{capAll} and \texttt{_misc_}, as you might imagine, exists only to remove the possibility of communication interference. \texttt{capOut} is a path which contains only \texttt{out} actions. This is sent out of the actor-ambient and is picked up by the top level process that then executes the actions. These actions allow any method callers that are needed as part of the body of the method to exit the actor-ambient and to eventually allow the payload-ambient to exit. \texttt{capIn} is also a path but it not only contains all the \texttt{out} actions in \texttt{capOut} but also any necessary \texttt{in} actions. This path is executed by the marshalling process. It allows method callers and the payload-ambient out of the current behaviour, it allows method callers that are returning to reenter the actor-ambient and it allows any new behaviour-ambient to exit the current behaviour-ambient. Finally, the marshalling process receives the name of the future behaviour-ambient which it tells the \texttt{starter} when returning the magic token.

\textbf{Figure 6.6} Complete Marshalling Process

\!
(callerName, methodName).
\!
\!
(methodName \in checker . \texttt{out} checker . 0 ]
| \texttt{out} methodName . (go, behaviour)\texttt{starter} . \texttt{in} callerName .
| (leavingName)\texttt{callerName} . (methodName, callerName, leavingName)\texttt{behaviour} . (capIn)\texttt{behaviour} .
| \langle\texttt{capIn}\texttt{callerName} . (capOut, capAll, \texttt{misc}_\ldots) . \langle\texttt{capOut}\rangle . \texttt{capAll} .
| (future) . go[\texttt{in} starter . (future) . 0 ]

\!

\textbf{Figure 6.7} The Process in a Behaviour-Ambient containing Methods

\!
(methodName, callerName, leavingName) . \langle\texttt{callerName}, leavingName\rangle^{methodName} .
\!
\!
\langle\texttt{in} behaviour . \texttt{in} methodName\rangle . \texttt{in} callerName .
\!
\!
\langle\texttt{capOut}, capAll\rangle . \langle\texttt{capOut}, capAll, \texttt{misc}_\ldots\rangle . \texttt{capAll} . (future)^{methodName} . (future)^{future} . 0

Next to examine is the process within the method Ambients. This starts similarly for every method but then diverges depending on the actual definition of the method. Also, if it is detected at compile time that the behaviour \texttt{must} change after the method is called (i.e. there are no predicates and it either \texttt{dies or becomes} something else) then this method process is not replicated. If there is any doubt at all, it is replicated. The process is shown, unreplicated, in figure 6.8. First it receives the name of the message-ambient and the name of the payload-ambient, or \texttt{leavingName}. Here, this \texttt{leavingName} isn’t needed at all because the process, having let the message-ambient in, then communicates with the message Ambient and obtains
for itself the name of the payload-ambient. However in the behaviour resulting from a `become` statement, examined in section 6.2.6, this communication does not occur and so the `leavingName` is crucial in that case. The process asks the message-ambient for a path, `cap` which it executes. This consists entirely of `out` co-actions and allows the payload-ambient to escape from the message-ambient. It also receives the name of the payload-ambient, as already discussed and it receives a further capability. This is important for the `become` wrapping behaviour which will be discussed in due course. In the simplest case it is the `out` action on the name of the actor-ambient and is sent to the payload-ambient before it leaves the method. But in more complex cases this path can grow considerably and is used to ensure that a message-ambient that has travelled between hosts due to a wrapping behaviour returns to its original host.

Having allowed the payload-ambient out of the message-ambient the process then receives the parameters from the payload-ambient. These parameters match in name and quantity the parameters as defined (or not) in the Actor. At this point the process diverges but there remains one further action all the methods must do. This is to send a result back to the payload. As discussed, this may simply be the name of the current actor-ambient if there is no return statement. Along with the result, a path is sent which the payload executes. This path safely guides the payload out of the method, behaviour and Actor Ambients and leaves it as a sibling of the actor-ambient from where it continues by itself. This path will always have had appended to it the `capEscapeTail` path which the method process received from the message-ambient earlier. This allows for the message-ambient to influence the movements of the payload-ambient after it exits the actor-ambient which, as explained, allows the wrapping system to guide the payload-ambient back to its original host and actor-ambient from where it was wrapped. Thus the payload-ambient notices no difference in terms of location whether it is wrapped or not which is the motivation behind this design.

![Figure 6.8 Start of the Method Process](image)

```
(callerName, leavingName). in callerName. (\$callerName.
(cap, payload, capEscapeTail)\$callerName . cap . (parameters)payload ...
```

### 6.2.5 Method Callers

Having covered how the method caller or message-ambient is received and processed, the structure of the message-ambient itself can be explained. The structure is shown in figure 6.9 but the compiler will automatically replace the names `actorName` and `methodName` with the actual names of the current actor-ambient and method-ambient respectively and will replace the names `targetActor` and `targetMethod` with the real names of the actor-ambient and method that are being called. The message-ambient is created within the body of a method and so the first thing it must do is escape the current method, behaviour and actor-ambient. Having reached the great outdoors, it then sends its intentions to the parent process that relays them on to the indicated actor-ambient. If the marshalling process accepts the message-ambient then it moves in, receives and executes the path that gets it into the method and then forks. The process that has the `(escape)` instruction exists for the use of the `become` wrapping behaviour and will be discussed shortly.

The other process in the fork receives a no-argument communication causing to send the capability allowing the payload-ambient out, the name of the payload-ambient and the out action that will allow the payload-ambient to escape from the actor-ambient. This value is bound to the `capEscapeTail` name as discussed earlier. Sadly you cannot send a zero-length path which is why this action is sent even though the method-ambient with which the payload-ambient will communicate is capable of telling the payload-ambient how to escape the actor-ambient. In this example the payload-ambient is called `caller_inner`. Note that it is not started until after the communication has occurred: this is a general principle that I have employed throughout the encoding to ensure that Ambients are not created and not started until they really are needed. This reduces the loading placed on the GLINTVM as it means that a more steady and consistent use of resources is maintained rather than large spikes where a number of Ambients and processes simultaneously need to start.

Having started, the payload-ambient immediately tries to escape from the message-ambient and then it sends the arguments. As with the parameters in the method, these are replaced by the compiler, matching the names and arity from the actual method call in the source program. Having sent the arguments, the payload-ambient then waits for the result, again having the name `result` replaced by the target of the assignment in the relevant `where` clause in the source program. It also receives the exiting path which it
executes, allowing the payload-ambient to escape the actor-ambient. It then returns to the Actor, behaviour and method-ambient from whence it came. In this example it stops here. This corresponds to a single method call, no change to the current behaviour and no explicit return statement. If more method calls are required then the _caller_inner_ in this example would effectively take on the role of the _caller_outer_ and go and perform the next method call. If there was an explicit return statement then having returned to the method-ambient, it would output the value of the return expression which would be picked up by a process in the method-ambient and relayed on to the invoking payload-ambient. Finally, if a replacement behaviour was specified then that would appear within the _caller_inner_, the name of which would be sent to the method process which would allow the new behaviour-ambient out and communicate its name to the current behaviour process as already described.

The name of the innermost payload is always _caller_inner_. The names of the message/payload-ambients that surround it are generated automatically so that they never clash with one another. The whole method caller structure is wrapped in restrictions which guarantee the uniqueness of all the message/payload-ambient names and is invoked by the message process.

Figure 6.9 The basic structure of a Method Caller

```plaintext
caller_outer[
    out methodName . out behaviour . out actorName .
    ⟨caller_outer, targetMethod, targetActor⟩ . in targetActor .
    ⟨caller_inner⟩. (capIn). capIn .
    (escape). escape . 0
    |
    () . ⟨out caller_inner, caller_inner, out targetActor⟩ .
    caller_inner[
        out caller_outer. (arguments). (result, capOut). capOut .
        in actorName . in behaviour . in methodName . 0
    ]
]
```

6.2.6 Encoding Become

As demonstrated in the preceding discussion, many aspects of the encoding are due to the possibility of a _become_ statement. In the development of the encoding, I came up with several ways to achieve the necessary forwarding, many of which were simpler than the one presented here but they made use of replication. I have intentionally tried, even struggled, to avoid the use of replication where possible because a replication never terminates which prevents the Ambient which contains it and any parent Ambients from ever terminating thus using up resources unnecessarily.

The behaviour resulting from a _become_ statement is shown in figure 6.10. The _actorName_ and _targetActor_ names are replaced by the compiler with the actual names of the current actor-ambient and Actor name to which the current Actor has _become_. As it is a behaviour, it communicates with the marshalling process within the actor-ambient. After receiving details of the call it creates a unique name for the _wrapperInstance_ and allows the message-ambient into the behaviour. Next, the behaviour process sends the paths to the marshalling process that are bound to the _capOut_ and _capAll_ names which allows the _wrapperInstance_ to escape, then for the payload-ambient to reenter and finally for the payload-ambient to exit. Note that here the behaviour process does not communicate with the message-ambient in order to learn for itself the name of the payload-ambient. Thus this is the reason behind the payload-ambient name being communicated in advance as the _leavingName_.

Next the _wrapperInstance_ Ambient is constructed and once again the construction is delayed until absolutely necessary. After the message-ambient has entered, it sends to the message-ambient an _out wrapperInstance_ message which pairs with the _escape_ input within the message-ambient. This ensures that the message-ambient encased within the _wrapperInstance_ Ambient will try to escape just as soon as it can.
The *wrapperInstance* then leaves home and behaves as if it is itself a message-ambient, sending details of the method, target actor and its own name to the top level process to be sent on to the relevant actor. After entering the target actor and method ambient, its behaviour differs from a normal message-ambient. When prompted by the method process for the path, payload-ambient name and extra path, it relays the call into the wrapped message-ambient. When the result comes back, it modifies the result before sending it out to the method process. Consequently, the method process executes a path which allows the wrapped message-ambient out (which occurs because of the previous communication pairing with the *(escape)* input within the message-ambient) and then allows the payload out of the message-ambient. The name of the payload is unaltered, but the final path which is ultimately sent by the method process to the payload and executed by the payload is also modified: the wrapped message-ambient believes that it is inside the original actor-ambient and thus the extra path that it sends is to allow the payload-ambient to escape from the original actor-ambient. The modifications ensure that the payload really does return to the original actor-ambient before this extra path is executed. It should be clear from this how this system can be further modified in light of forwardings between hosts: the *wrapperInstance* takes the message-ambient and payload-ambient to the other host and then, by modifying this path ensures that the payload-ambient returns to the originating host before continuing on.

This mechanism can be viewed as the following: the initial message-ambient is the base case and any wrappers that enclose it become recursive cases. When the payload is required to escape from the wrappers, all the recursive cases that are the wrappers pass the communication down to the base case and then modify the results as they come out. In this way, the forwarding mechanism can be used as many times as necessary to forward a message-ambient as the result of an Actor executing a *become* statement without any changes being needed to either the message-ambient or the target Actor of the *become* statement.
Local Method Calls

Local method calls are calls to a method within an Actor by the very same Actor. The method is directly invoked by the Actor which makes for more straightforward ways of writing recursive methods by avoiding any sort of continuation passing. The method itself is unaltered: it is still a normal method. In fact, every single method can be invoked as a normal method, by an external Actor or as a local method if invoked by the Actor in which the method is defined. Therefore, the encoding of the method itself is not altered, only the message-ambient that invokes the method can be altered.

The requirement of the alterations is that the method can be invoked without having to obtain the magic token and that there is no communication interference with the partially evaluated processes which constitute the external invocation of the method containing the local method call. It is difficult to decide what to do if the local method call causes a behaviour change. The implementation here is that such behaviour changes are ignored, thus only externally invoked methods can cause the Actor to change methods. This seemed the most sensible solution but if carefully defined, other semantics would be possible.

As already seen, a message-ambient communicates initially with the target actor-ambient (in particular the marshalling process) via sending a message which is picked up by one of the three top level processes. In order to communicate with a different marshalling process there must be a different top level process. Hence the third and final top level process which is as follows:

\[
!(\text{caller}, \text{method}, \text{payload}, \text{behaviour}, \text{Actor}). (\text{caller}, \text{method}, \text{payload}, \text{behaviour})^{\text{actor}.0}
\]

The difference is an extra two parameters, namely the payload and behaviour Ambient names. These are sent on and picked up by a separate marshalling process within the actor-ambient as shown in figure 6.11. There are several things to take in here. Firstly, a local method can only occur when an Actor has methods. Thus it can't happen after a become or die statement. Secondly, the compiler ensures that the local method call is valid which means that there is no need to check with the checker that the method requested exists: it must exist for the compiler to have identified the method call as local. Thirdly, because there is no use of the magic token, there is no way for the extra marshalling process to find out what the name of the current behaviour-ambient is, but the local method caller knows it because the message-ambient was created within a method Ambient which is within the current behaviour-ambient. This is why the current behaviour name is sent from the message-ambient, relayed via the top level process and picked up by the extra marshalling process. Also, because of not using the magic token, there is no way to update the starter about any changes to the current behaviour. This is in truth, the main reason behind the decision to ignore behaviour changes that occur as part of local method calls. The (future) input still appears because the behaviour and invoked method don't know that they're dealing with a local method call, but having received the name of the future behaviour, nothing further is done and so it is guaranteed that the behaviour after a local method call is the same as before a local method call.

![Figure 6.11 Marshalling Process for Local Method Calls](image)

As usual, the paths of capabilities capOut and capAll are received and executed allowing for any movement that might be needed due to further method calls as a consequence of the invocation. As before, they ultimately allow the payload-ambient back out of the actor-ambient which it may then immediately reenter if there is no more work for it to do. Thus the message-ambient for a local method call by large behaves as for a normal method call. This is very advantageous as it means that the paths constructed by the method which requires the local method call do not have to take into account any special behaviour for local method calls.

Finally, it needs to be explained why communication interference can't take place. For the message-ambient of the local method call to escape from the method, behaviour and Actor Ambients in which it was created, requires that the processes within those Ambients are executing the relevant path, capAll, which contains the necessary _out co-actions. The capAll path finishes with allowing the caller_inner Ambient back in and then allowing the payload-ambient back out, so until that point, the processes will not be
receiving or sending communications. Therefore, during the execution of these paths, the communications in the Ambient that are caused by the local method call cannot pair with the partially evaluated processes dealing with the initial method invocation from an external Actor. Thus the communications must pair with new unw windings of the relevant replicated processes. This is why it is perfectly safe for, for example, the extra marshalling process to perform the \textit{(future)} input: it cannot pick up the actual future as a result of the initial method invocation because that will not have been communicated yet because those processes are still executing the \textit{capAll} path. Similarly, the \textit{(future)} input in the non-local marshalling process cannot pick up the future behaviour from a local method call because the input won’t occur until after the \textit{capAll} path has finished executing by which time any behaviour change caused by a local method call will have been picked up and ignored by the local marshalling process. By induction, this reasoning holds for any number of local method calls.

6.2.8 Anonymous Inner Actors

Anonymous inner Actors are implemented by hijacking the method calling system. In the normal way, the message-ambient has a payload-ambient within it and exits the method, behaviour and actor-ambient in which it is created. However, at this point, instead of trying to enter another actor-ambient and invoke a method, the payload-ambient exits the message-ambient. The message-ambient then starts up as a regular Actor, having become a sibling of all the other actor-ambients. Because of the use of restriction in the method-ambient, the name of the message-ambient and hence the anonymous actor-ambient is unique and is communicated to the payload-ambient immediately prior to the payload-ambient moving out of the message-ambient. The payload-ambient then continues with the next step, returning to the method-ambient from whence it came and then either calling another method, returning a value to the caller and possibly altering the behaviour of the Actor. This is a good example of reuse of components of the encoding.

6.2.9 Methods with Predicates

Methods with predicates are another example of reuse of components of encoding as they are converted into a sequence of nested anonymous Actors which determine which case is to be evaluated by method calls to Actors defined for the truth values, \textit{True} and \textit{False}. Figure 6.12 shows the basic premise behind the encoding of predicated methods into anonymous Actors. When a predicate evaluates to \textit{False} then it is simply a matter of invoking the anonymous Actor for the body guarded by the predicate. If the predicate evaluates to \textit{True} then the anonymous Actor invoked must recurse: testing the next predicate and invoking the appropriate anonymous Actor as a result. The base case concerns the default method body and the last predicate: if the predicate evaluates to \textit{True} then the guarded body is invoked as normal whilst if the predicate evaluates to \textit{False} then the default body is invoked.

\textbf{Figure 6.12 Converting Predicates into Anonymous Actors}

\begin{verbatim}
define Actor meth (params)
meth
where
  r = pred1.ifThenElse<t,f>
t = define body1
f = define where
  r' = pred2.ifThenElse<t,f'>
  t' = define body2
  f' = define where
    r'' = pred3.ifThenElse<t,f''>
    t'' = define body3
    f'' = define body4
\end{verbatim}
The definition of the ifThenElse method is covered in the next section, 6.2.10, but suffice to say that the predicates are required to evaluate to either True or False, the Actors of which both define the method ifThenElse.

Note that in this figure there are several parts missing. Firstly, there may be a sequence of method calls to be made in order to evaluate the predicates. These must be performed immediately prior to each call to ifThenElse. Also there may be the possibility of different bodies causing changes to the behaviour of the Actor but this cannot simply affect the generated anonymous Actors: the new behaviour-ambients must be capable of escaping from their anonymous Actors and entering the actor-ambient containing the method with predicates.

At a similar level of complexity is the different bodies returning different results to the caller. The most obvious idea is to chain the returning of the result via the calls to ifThenElse: the idea being that the executed body returns the correct result to the invocation of ifThenElse which returns it to its caller and so on, right the way up the chain. The problem is that the GLINT language has defined call-and-return semantics on top of the usual Actor semantics of each Actor being unable to deal with more than one message at a time. Thus assume that both pred1 and pred2 evaluate to False. The first call to ifThenElse will be sent to False which will invoke the relevant anonymous Actor and wait for the result. The anonymous Actor will find pred2 evaluates to False and so will try to send an ifThenElse message to False. But False is still waiting for the result of the first ifThenElse invocation and so cannot process the second. The second method call will therefore block.

Effectively, what is being performed is continuation passing: the ifThenElse method is invoked with two possible continuations and it is responsible for choosing which to invoke. As discussed in section 3.2.2, deadlocks can occur in Actor systems in precisely this situation: an Actor that is waiting for a response is invoked again as part of the calculation of that response. The solution is to prevent the result from being returned via the ifThenElse method. If from a method declaration it can be guaranteed that no computed result would under any circumstances be returned to the caller via the payload-ambient then the payload-ambient is sent the name of the Actor (as already discussed) and allowed to leave before any further method invocations take place. No further message will be processed until the method is completed because of the use of the magic token but it does solve this particular problem. Firstly, the invocation of ifThenElse returns before the next anonymous Actor (i.e. continuation) is invoked; and secondly, the invocation of the continuation allows the truth-value Actor (i.e. True or False) to return without waiting for a computed result: this means that if they are needed by the continuation itself they are available.

Finally, the result must get back into the original actor-ambient and be sent to the caller. The mechanism for this is the same for the result and the potential behaviour change: before the anonymous Actors are created, a restriction is created giving a unique value to the name _result_. When a guarded body is executed (and there must be exactly one), an Ambient is created called _result_ which is allowed back inside the originating Ambient, behaviour and method Ambients and from which the computed result is received by the method process which relays it onto the payload-ambient.

One further complication is the possibility of a local method being called from within a guarded body: the method is no longer local to the body because the body is now within an anonymous Actor. Fortunately, because the local method call system borrowed as much as possible from the normal method call system, it is a simple matter for the body to issue a local method call that invokes the necessary method in the original Actor. All that is needed is that the guarded body retains knowledge of the original Actor name and current behaviour name.

6.2.10 True and False

Figure 6.13 shows the implementation of the truth values in the GLINT language. Whilst these are assumed to exist by the compiler, in particular for the encoding of methods with predicates, they are nevertheless normal Actors and have no special behaviour as far as the GLINTVM is concerned.

The implementation is as efficient as possible: the ifThenElse method is largely discussed in the previous section but, upon receiving two Actor names, invokes either the first or the second via a simple message depending on the truth value. It is assumed that the evaluation of any predicate will result in either the True of False Actor. The other methods are easily understood and can be clearly seen to be as explicit and efficient as possible doing as little computation as possible to achieve the correct result.

6.2.11 Encoding Numbers

Numbers have always differed significantly from theory to implementation. In a purely functional manner, they can be defined recursively from zero in terms of successive pointers to the previous number and indeed,
6.2. Encoding into Ambients

Figure 6.13 True and False Singleton Actors in the GLINT language

(a) The True Singleton Actor

```plaintext
1 define True
2 ifThenElse(Any then, Any else)
3 where
4 x = then<> 
5 and(Bool b) and
6 <b>
7 or(Bool b) or
8 <True>
9 not() not
10 <False>
```

(b) The False Singleton Actor

```plaintext
1 define False
2 ifThenElse(Any then, Any else)
3 where
4 x = else<> 
5 and(Bool b) and
6 <False>
7 or(Bool b) or
8 <b>
9 not() not
10 <True>
```

in an analogous way are defined based on nesting depth in the Mobile Ambient calculus, [10]. However, maths based on such a system is extremely slow and inefficient. There is no reason why numbers could not be implemented in this pure way by the compiler, but for reasons of efficiency, they are not. Instead, changes are made to the GLINTVM to create additional built in functions as described in section 4.10. This allows for the mapping of names of Ambients to particular numerical values and then allows for mathematical operations between those names.

Initially, only two numerical Actors are defined, Zero and One. But from these two Actors any number can be constructed. Whenever a mathematical operation is invoked, the built in function returns two values to the caller. The first is a truth value which tells the caller whether there already exists an Actor that represents the result of the operation. If the result does exist as an Actor then the second value returned is the name of that Actor. If the result does not exist as an Actor then the second value is ignored. Instead, a new instance of the Number Actor is created which for constructor parameters is sent the name of the mathematical operation and the name of the two arguments. Having been created, the Number instance then uses these arguments to inform the built in mathematical functions that a new number has been created as a result of the given operation and arguments. The built in function evaluates the operation with the values of the arguments and then maps the name of the new Number instance to the result.

In this way it is (almost) guaranteed that there can only ever be one instance of any given number. The reason for the almost is that there is a race condition once the result of the operation reveals there is no existing Actor representing the result, the creation of the new instance and the registration of that instance with the result of the operation. There are various possible solutions to this problem which are not overly substantial to implement including mapping multiple instances to the same number, or at the point of registration of any number for which an Actor already exists, making the extra copies become the original Actor for that particular number.

6.2.12 Equality Testing

In the same way that the ability to test for pointer equality is crucial to Object Oriented language, name equality testing is crucial for Actor languages. Sadly, in the Ambient calculus, it is impossible to exclusively execute two different processes depending on whether two names are bound to the same value, the best that can be done is to invoke a supplemental process if the values are the same. Given two names, x and y, the first idea would be the following:

```
x[⟨True⟩.0] | y[⟨False⟩.0] | (r)y . P
```

the idea being that if x = y then the input would pair with the x Ambient and so receive True but given the non-determinism, there is nothing to stop the input choosing to pair with the output from the y Ambient. So if they are not equal then you will definitely get false but if they are equal you may still get false. Thus if
$r$ is bound to $False$ you know nothing whilst if $r$ is bound to $True$ you know that they are equal. In order to improve this situation, you might expect that if the converse can be constructed then enough information could be gained to conclusively determine between equality and inequality. The lack of precision comes from the fact that if the values of the names are equal then either Ambient could be paired with, thus for the converse it is required that if the values of the names are unequal then either Ambient could be paired with. One moment’s thought should convince that this is impossible to achieve on the grounds that non-deterministic choice of pairings can only occur as a result of equality, not inequality. This is the fundamental problem: whilst it is certainly easy to achieve reductions which are dependant on two names having the same value (such is the basis of movement), it is not possible to achieve reductions that only occur should two names have different values.

In short, to try and distinguish between two values one is forced to try to use the values to build structures and processes that when executed will reveal whether the values are equal or not. This is due to the fact that there is no primitive operation for determining whether two values are equal: the Ambient calculi take for granted some notion of name theory in which operations to determine equality, needed for example when executing substitutions, are already defined. Sadly, the processes and structures that can be built are not sufficient to achieve precise testing of equality so the underlying theory of names must be utilised.

This problem may lead the reader to believe that the Ambient calculus is not, on its own, Turing complete: a Turing Machine certainly can tell the difference between two names with total precision. In fact, the same situation exists in the $\pi$-calculus and CCS and CSP: none of these calculi are able to directly distinguish between names. The reason why it does not invalidate the Turing completeness of the calculi is because names are primitive to the calculi and so determining the difference between two names is rather similar to a Turing Machine determining whether two tape-movement directions are the same or not without examining the contents of the tape or the effect of the tape-directions. What is needed is to be able to represent the tape-movement directions by symbols on a tape so they can be directly compared which in turn prompts the use of a Turing Machine that receives as input the encoded definition of another Turing Machine, thus being able to inspect the definition supplied.

Much the same solution is available for the calculi: what is needed is to be able to express names in terms of processes. This is what led to the solution for the Turing Machine: by representing the program of a Turing Machine as input to another Turing Machine, the symbols on the input tape that represent the movement of the tape become directly comparable; they have been reflected into symbols upon which computation can be performed. In the calculi, expressing names in terms of processes again allows for computation to be performed on the structure of those names which allows for determination of equality. This is explored further and more rigorously in [23] and amounts to being able to reflect upon the structure of names.

6.3 Technologies Employed

The GLINT Compiler uses the same technologies as the GLINTVM though makes much more extensive use of the abilities of ANTLR, to walk over and transform the generated AST. Again, JAVA 5 was used as the only language and both FINDBUGS and CHECKSTYLE tools were used extensively in order to construct consistent, robust and reliable code.

6.4 Conclusion

The GLINT Compiler is a complex piece of work and reflects the complexity of the encoding. One of the first classes I wrote was a formatter for the generated Ambient code which makes it very easy to assemble chunks of Ambient code. It indents and pretty prints the code and places just one Ambient calculus instruction per line. This is one of the most useful classes I wrote as it makes the generated code as readable as humanly possible. This is important when you consider that a ten line program in the GLINT
language compiles to over 1000 instructions (and hence lines) for the GLintVM. Being able to navigate quickly through the generated code, recognising structures and patterns, often by their shape, became very important very quickly.

As examined in section 5.1, an Actor requires more than one process in the Ambient calculus to encode the Actor. The encoding used shows that many processes are needed in order to correctly model the Actor. No formal proof has been attempted to validate the correctness of the encoding, which given the complexity and size of the encoding would be a formidable undertaking. My chief regret is in defining a white-space sensitive language. This turns out to be harder to parse correctly and that on its own cost me significantly more time and suffering than had I defined a language with braces.⁵

The GLintCompiler works and generates valid Ambient code. To the best of my knowledge, the compilation respects the definition of the GLint language, but without a more formal semantics of the language and a formal proof of the compiler, it is difficult to have any certainty in the accuracy of the compilation. The programs that I have written and tested certainly behave as intended but it is obviously impossible to exhaustively test every possible program.

One of the hardest parts of designing the encoding was in trying to analyse the encoding for potential communication and movement interferences. There seemed little choice other than having large print-outs of compiled programs and working through all the different movements and arities of communication. Again, to the best of my knowledge, there are no communication or movement interferences but given the non-deterministic aspects of reduction of Ambients, the correct behaviour of compiled programs may not in fact correspond to an absence of interferences. It might be possible to formally prove the absence of interferences but once again, due to the complexity of the encoding, this has not been attempted. Type systems for the Ambient calculi may, if implemented, be sufficiently powerful to ensure there are no communication interferences in the encoding.

This is in all probability the largest amount of Ambient calculus code generated to date and in designing and creating the encoding I have come to understand the parts of the Ambient calculus that work well and are very useful in practical applications, for example paths, and those that do not work so well. The most predominant feature I feel is missing from the Ambient calculus is named channels. Even if limited in range as in the Channel Ambient System (section 2.5, [29]), named channels would make a substantial difference to ensuring that no communication interferences occur.

The GLintCompiler achieves its aims and the generated Ambient code does execute in the GLintVM. The encoding, whilst large is not excessively complex: hopefully this chapter has been able to explain how the individual components of the encoding work and how, to some extent, they slot together. Furthermore, I feel that were the encoding more efficient in terms of volume of generated code it would be less comprehensible: it is precisely the verbosity of the encoding that allows it to remain at all comprehensible. Details on using the GLintCompiler can be found in section D.2 and some performance measurements of the generated code can be found in section 7.5.

⁵Also, due to time constraints, the compiler is not as well structured as it could be: there are several opportunities where aggressive refactorings could be performed which would noticeably improve the quality of the code. That said, the compiler is not badly written.
Here I shall directly evaluate the overarching design and the GLINTVM before evaluating the GLINT language by example and comparison with implementations in other languages, in particular, JAVA and the Channel Ambient Language.

7.1 Two-tiered Design

It is not necessary to design a language, build a compiler and build an Ambient calculus reducing machine to investigate the mobility primitives of Ambient calculi for mobile and concurrent programming. Instead, I could have augmented the calculus so as to include some basic programming structures and patterns. However, having become accustomed to high level abstractions that modern programming languages provide, I felt that it would be too limiting to have a language based on an Ambient calculus and then supplement it with basic functional or procedural capabilities. Such a language would feel like an assembler language and that in itself would hinder the evaluation. Furthermore, the Ambient calculi are meant to be able to represent computation completely themselves and so adding additional features as primitives to the calculus is something I wanted to avoid from a purity point of view. The other possibility was to directly interpret the higher level language rather than compile it to a lower-level representation. As discussed at the start of chapter 4, there are many reasons why this would significantly complicate the implementation and leave me with a less reusable system which is harder to reason about.

The main advantage of the two-tiered design is in the ease of implementation. Given the tremendous difficulties of implementing the GLINTVM, a more complicated calculus which starts adding common programming structures would have added vastly to the complexity of the of the implementation. Instead, by focussing on a concise calculus, the implementation of the GLINTVM was kept as manageable as possible. The compiler then serves as a decoupling device, separating the higher level abstractions in the GLINT language from the concerns of the GLINTVM. By separating these concerns, two things are achieved: a simpler, more easily verifiable and more easily extendable compiler and virtual machine; and two products instead of one: the GLINTVM is a valuable contribution to the study of Ambient calculi on its own with or without the higher level GLINT language.

7.2 The GLINTVM

The GLINTVM is a faithful implementation of the Safe Boxed Ambient calculus. It is fully multi-threaded, allowing it to make good use of parallel hardware which results in a significant performance increase on suitable hardware. The performance increase cannot be a simple linear scaling based on the parallelism of the hardware however because of the fact that almost every operation requires the pairing of two processes which must rendezvous with one another. As such, during these moments of rendezvous, the potential for parallel execution is constrained due to the threads involved queueing for access to critical regions of the program. There are relatively few instructions in the calculus that do not involve rendezvous between two processes and it is only in these instructions that the parallelism of the hardware can be exploited.

That is not though to overly criticise the design, its approach to parallelism or the choice of calculus. As can be seen from the Ambient calculus code generated by the GLINTCompiler, the size of the encoding produces a large number of processes for a little code. Whilst much has been done in the design of the encoding to keep to a minimum the number of processes created, it is without any doubt the case that the number of processes created and the number of threads that are correspondingly spawned during execution massively exceeds the parallelism in any computer that is ever likely to be conceived. A simple counter written in the GLINT language that counts from 0 to 20 can easily result in over 1000 processes being created which corresponds to over 1000 threads when executed by the GLINTVM. Many of these threads
will have little work to do, starting and terminating quickly after having performed just a few instructions. It is only replicated processes that cause threads in the GLINTVM to be persistent. Given a smart operating system and a good JVM, the number of threads used will almost without exception be able to make full use of the available parallelism: whilst the requirements for rendezvous will scale linearly with the number of processes, so too will the number of instructions that do not require rendezvous and it is these instructions that will ultimately occur in such abundance as to be able to utilise parallel architectures whilst rendezvous occurs at a necessarily lower rate.

The implementation of the GLINTVM must be compared with other approaches of which two are significantly different. The simplest approach is to implement the processes in the Ambient calculus as passive Objects which are analysed and reduced by a single execution thread. This amounts to having the processes implemented as co-processes and then choosing between them as to which can reduce and what pairings to choose to reduce. This would have made the implementation easier and it would have allowed for a faster implementation on hardware with very low levels of parallelism. This approach is the same as that of the current implementation of Erlang and Erlang is well known for its low latency of communication and high performance. However, crucially, this approach cannot make proper use of parallel hardware limited as it is to just one real thread.

A hybrid of the two extremes is also possible. Instead of having one thread per Ambient process, one thread per CPU-core is used. Thus the number of threads is derived from the thread-level parallelism of the architecture, hence the design scales well with increasing parallelism. It is then up to the threads to concurrently reduce the Ambient processes between them; the Ambient processes are still passive co-processes, but there is the possibility of multiple reductions occurring simultaneously due to the multiple execution threads. This last hybrid approach offers both excellent performance, good utilisation of parallel hardware and a minimum of resource wastage. On the other hand, it is the hardest of the three approaches to implement as the scheduling and analysis of processes must be performed as per the single-threaded approach but in a thread safe way and all the difficulties of the implementation I have chosen are also present because theoretically, the hybrid design would be able to achieve the same level of concurrency as my approach given sufficiently parallel hardware.

Furthermore, this last approach amounts to re-implementing the role that the Operating System plays in my design, as in my design by using so many threads, I present to the Operating System all the processes and express in terms that the Operating System can understand (i.e. by blocking on monitors) which threads can execute and hence which processes can reduce. The Operating System is aware of the parallelism within the architecture and schedules threads appropriately. Of course, by utilising the Operating System in this way and by having to explain to the Operating System in terms it can understand which threads can and cannot execute, more resources are used and a less efficient implementation results. Erlang’s impressive performance comes from this re-implementation (albeit for a single threaded execution model) thus an efficient and tailored execution system can result in massive performance gains at the cost of a larger and more complex implementation.

Whilst the large number of threads used by the GLINTVM when interpreting compiled GLINT language programs is perhaps wasteful of memory, it is also crucial for ensuring the utilisation of parallel hardware. Although the hybrid approach is more desirable, it is also more complex and would not have been feasible in the time given. The number of processes used results from the complexity of encoding from the language to the Ambient calculus, and thus from the differences between the Actor model and the Ambient calculi. Without narrowing this gap, it is difficult to see how the encoding could be significantly improved in order to reduce the number of processes.

Testing of the GLINT language on dual-processor dual-core machines (i.e. four CPU-cores) shows that all the CPU cores are used. Using the Linux tool top reveals that CPU time is split evenly between user and system operations. System operations are operations that are performed by the kernel itself; these will be the Java monitors regulating access to critical regions which are implemented with kernel semaphores along with the creation and termination of threads which is implemented via the fork system call.

7.3 The GLINT Language

The GLINT language is unique, combining a set of features which have, to the best of my knowledge, not been assembled in such a way before. The purpose of the language is to combine features of the Actor model with a restricted form of mobility from the Ambient calculus in order to present programmers with a more powerful and intuitive language for writing concurrent and mobile systems. Therefore in this section I shall demonstrate by example the kinds of expressions that are benefited most by the design of the language, how they compare to equivalent programs in other languages and how these particular kinds of expressions
are beneficial for the purposes described.

At no point shall I claim that the GLINT language is a complete language or that it presents abstractions anywhere near the levels of abstraction and encapsulation provided by modern fully developed languages: the design of the language has focussed only on the goals of the project and not on mimicking typical features of modern programming languages no matter how useful and powerful I may think such features are. Thus the GLINT language will in the general case, struggle to present concepts, expressions and patterns as concisely and elegantly as the bulk of popular modern languages. The hypothesis is that in the areas of concurrency and mobility however, the GLINT language will allow for greater clarity and a much more powerful form of expression.

The example chosen is a game of hide-and-seek. The game starts by the hider choosing a computer to hide on. The computers are arranged in a grid, each computer knows its own location, knows whether the hider is in the computer and knows the host names of the computers to the North, South, East and West of it, as appropriate for each computer. The solutions presented here will not model creating the grid or the hider choosing a computer in which to hide. All that shall be shown is the seeker as it navigates the grid hunting for the hider. Also taken into account will be how easy it would be to have more than one seeker and how they could interact with one another in order to hunt down the hider more efficiently. The two languages considered will be the GLINT language and JAVA. The choice of JAVA is apt given its widespread use and the level of reader familiarity with the language and also because it was the first mainstream language to come with support for distributed computation.

The Computer and Seeker Actors for the GLINT implementation are shown in examples 7.1 and 7.2 respectively. It is assumed that the Seeker is initially started at the North-West corner. As can be seen from the main seek method, the Seeker works across and down the grid. There are several things to note from this implementation:

• The definition of the Actor Seeker needs to be present on all the machines. This is both similar and dissimilar to JAVA. The reason it needs to be present is because on each machine that gets visited, a new instance of the Seeker is created. For this, the definition is required and this mirrors the situation with JAVA: to create an instance of a class the class definition must be loaded by the JVM. On the other hand, if a new instance of the Seeker Actor was not needed and just one instance was able to travel across all the computers then the definition of Seeker would only need to be present on the first machine on which the instance is initially constructed. This contrasts with JAVA which must load the definition of a class before any instance of the class can be used at all in the JVM.

• The reason why the new instances are required is because it is impossible for an Actor to become another Actor and simultaneously invoke a method on itself. This is a limitation of the language. What is needed is to move the Seeker to the target host and to re-invoke the seek method but that is not possible: you cannot specify any expression that is to occur after the become statement. The work-around is very simple though: a clone of the current instance is made, that clone is moved to the new host and the seek method is invoked on the clone by the current instance. This is simple, the chain of events is straightforward and this technique is easily applied to a large number of situations where the computation is a continuous process spanning several hosts rather than a discrete computation limited to a single host which may result in some final movement.

• On each computer, the names of the computers that surround it are valid for the current computer. This is to say that invoking methods on the north, south, east and west computers correctly forward messages to the relevant host. That means that each Computer must visit its neighbours before settling in its host in order to establish valid forwarding mechanisms from each neighbouring host.

• Finally, note the use of the getLocalName method to allow the Seeker to get the name of the Computer valid for the host to which it is moving. This ensures that once it has arrived at the target host, it will still be able to communicate with the Computer on that host. If it did not do this then the name of the Computer that it would carry would be valid for the forwarding mechanism on the host from which it came but invalid for the current host.

Whilst the implementation is concise and quite simple, it is also very scalable. No changes are needed if more than one Seeker is to be used although currently they will all use the same algorithm and cover the same computers in the same order. The use of another boolean would allow for the choice of whether to go West to East or North to South. This would allow the second Seeker to cover the computers in a different order and if either the Computer Actors knew or the Seeker Actors knew by using counters when they are on a computer on the North-West to South-East diagonal then they would be able to ensure that they don’t both check the same Computer Actors.
### Example 7.1 The Computer Actor for Hide-and-Seek

```java
1. define Computer(Host myHost, Computer north, Computer south,
   Computer east, Computer west, Hider hider)
2. getHost ()
3. getHost
4. <myHost>
5. getLocalName ()
6. getLocalName
7. <this>
8. getNorth ()
9. getNorth
10. <north>
11. getSouth ()
12. getSouth
13. <south>
14. getEast ()
15. getEast
16. <east>
17. getWest ()
18. getWest
19. <west>
20. sheltersHider ()
21. sheltersHider
22. <Nil.notEq<hider>>
23. shelterHider(Hider h)
24. shelterHider
25. become me
26. where
27. me = new Computer<myHost, north, south, east, west, h>
```

It is also trivial for more than one Computer Actor to occur per host provided that the relationships in the grid are consistent. This facilitates development of distributed systems and algorithms across relatively few hosts prior to deployment on a much bigger network of hosts. However, whilst a system written for many hosts would on the whole work well on fewer hosts, the converse is not the case: care must be taken in the design of the system to account for the changing names of Actors which results from inter-host movement. If this is not taken into account then whilst the program may work on few hosts, it is unlikely to work on many hosts.

The Java equivalent is more complex. Instead of attempting a translation of the GLINT version I shall re-implement the Seeker in Java using the standard Java concepts and techniques. Rather than in advance propagating distributed pointers to the Computer instance to every neighbour, each computer knows only the URL to its neighbours. The seeker must choose the next host to check as before and it must then transport and reinvoke itself on the next host. It must be able to obtain the Computer and check the computer for its neighbours and whether or not it is sheltering the hider.

The implementation is shown in example 7.3, 7.4 and 7.5. RMI is used to achieve mobility here and this time, it really is the Seeker instance that moves between hosts. However, movement is a confusing concept with RMI for several reasons. Firstly, the movement of a serialisable class means that the receiver constructs a new instance of the class (requiring the receiver to know the class definition) and sets the instance fields of the class to match the instance being moved. Both the copied instance and the new instance are valid Objects simultaneously and both can be accessed by separate threads. Secondly, there is no movement of the thread executing the Seeker. The thread executing the Seeker in the originating host is blocked while the invokeSeeker method is being executed on the remote host; the thread that executes the invokeSeeker
method on the remote host is provided and controlled by the RMI system. Also note that in using RMI, it is not possible to simply move an Object to another host and invoke some method upon it; the Object can only be moved by being an argument to a method of an Object that is configured to be able to receive method calls via RMI. This is why the invocation of the seek method is chained through the Computer instance.

Whilst there are basic similarities between the GLINT and JAVA versions, it is clear that the JAVA version is more complex. Most of that overhead is in setting up and starting the RMI system and thus is a result of making the required functionality available from within an API rather than directly available from the language syntax. As discussed, the main difference is what the indicated movement amounts to but there are other differences relating to how well the JAVA version scales, how adaptable it is to different relationships between hosts and Computer instances and how well it copes with multiple seekers.

By ensuring that the hider has hidden itself before the seekers start looking for it, the Computer class does not change state whilst the seekers travel through it. This means that were there to be more than one Seeker there would be no need for synchronisation as there are no critical regions for which access must be regulated. However, if there was mutable state, for example if the Computer class kept a count of how many times it had been interrogated by the seekers, then synchronisation would be necessary. Furthermore, given
Example 7.3 The Computer interface for Hide-and-Seek

```java
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public interface IComputer {
    String SERVICE_NAME = "Computer";
    boolean sheltersHider();
    void shelterHider(Hider hider);
    String getNorth();
    String getSouth();
    String getEast();
    String getWest();
    void invokeSeeker(Seeker seeker)
        throws MalformedURLException,
                RemoteException, NotBoundException;
}
```

the way in which the Seeker is chained through the Computer, it would be very easy for the synchronisation to be added incorrectly causing the blocking of the second Seeker whenever it visits a Computer that has already been visited by the first Seeker. In the GLINT version however, this is not a concern due to both the Seeker not being invoked via the Computer and the fact that each Actor encapsulates a single thread and thus only deals with one method call or message at a time. Thus the GLINT language, by being influenced by the Actor model, has safer and more intuitive semantics for regulating access to shared Objects.

Not only is the JAVA version more prone to error if the Computer class changes state when visited, but it also cannot cope with changes to the relationship between hosts and computers as well as the GLINT version. As discussed, the GLINT version can easily cope with having multiple Computer Actors per host which makes development of distributed systems easier in an environment with fewer hosts than in the deployment environment. For the JAVA version however, there are two choices. Either another layer of abstraction must be used in order to allow multiple Computer instances to exist within the same Object configured to receive RMI messages, or each Computer instance must register directly with RMI using a unique non-clashing name. This would allow multiple distinct computers per host. Given the overhead of additional code to register with RMI, further modifications to ensure the use of unique names and the propagation of those names to the neighbours would add to the volume and complexity of the JAVA implementation.

However, the most significant difference between the two versions is in how knowledgeable the programmer must be in order to implement the two different versions. With the JAVA version, a good understanding of RMI is needed which is a complex subject and the documentation¹ for RMI is correspondingly large. The concepts are more complicated, more tools are needed, the ability to modify properties of the JVM is required and overall, the whole mechanism is much more complex. With the GLINT language, compiler and GLINTVM, the tool chain is no more complex than a non-distributed version and the concepts and ideas much simpler, more intuitive, easier to learn and perhaps most importantly, are better suited to further modification for a more complex system.

7.4 The Channel Ambient System

The Channel Ambient System [29] presents not only a variation of the Ambient calculus as discussed in section 2.5, but also a language based upon the calculus. The language takes a very different approach

¹http://java.sun.com/j2ee/1.5.0/docs/guide/rmi/index.html
Example 7.4 The Computer class for *Hide-and-Seek*

(a) top

```java
import java.net.MalformedURLException;
import java.rmi.AlreadyBoundException;
import java.rmi.ConnectException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public final class Computer extends UnicastRemoteObject implements IComputer {
    private static final long serialVersionUID = 4758636051176282319L;
    private final String northHost;
    private final String southHost;
    private final String eastHost;
    private final String westHost;
    private Hider hider = null;

    public Computer(final String north, final String south, final String east, final String west)
        throws RemoteException {
        super();
        northHost = north;
        southHost = south;
        eastHost = east;
        westHost = west;
    }

    public String getNorth() {
        return northHost;
    }
    public String getSouth() {
        return southHost;
    }
    public String getEast() {
        return eastHost;
    }
    public String getWest() {
        return westHost;
    }
    public boolean sheltersHider() {
        return null != hider;
    }
    public void shelterHider(final Hider h) {
        hider = h;
    }

    public void invokeSeeker(final Seeker seeker)
        throws MalformedURLException, RemoteException, NotBoundException {
        seeker.seek(this);
    }
}
```
public static void main(final String[] args) throws RemoteException,
        MalformedURLException {
    if (null == System.getSecurityManager()) {
        System.setSecurityManager(new RMISecurityManager());
    }
    Computer comp = new Computer(args[0], args[1], args[2], args[3]);
    rebindServer(IComputer.SERVICE_NAME, comp);
}

private static void rebindServer(final String compURL, final Computer comp)
        throws MalformedURLException, RemoteException {
    try {
        try {
            Naming.bind("//localhost/" + compURL, comp);
        } catch (AlreadyBoundException abe) {
            Naming.rebind("//localhost/" + compURL, comp);
        }
    } catch (ConnectException ce) {  // registry probably not running
        LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        try {
            Naming.bind("//localhost/" + compURL, comp);
        } catch (AlreadyBoundException abe) {
            Naming.rebind("//localhost/" + compURL, comp);
        }
    }
}
Example 7.5 The Seeker class for *Hide-and-Seek*

```java
import java.io.Serializable;
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public final class Seeker implements Serializable {
    private static final long serialVersionUID = -2359306394030951803L;
    private boolean goWest;

    public Seeker() {
        goWest = false;
    }

    public void seek(final IComputer comp) throws MalformedURLException,
        RemoteException, NotBoundException {
        if (comp.sheltersHider()) {
            System.out.println(comp);
            return;
        }

        final String nextURL;
        final boolean nextGoWest;
        if (goWest) {
            if (null == comp.getWest()) {
                nextURL = comp.getSouth();
                nextGoWest = false;
            } else {
                nextURL = comp.getWest();
                nextGoWest = true;
            }
        } else {
            if (null == comp.getEast()) {
                nextURL = comp.getSouth();
                nextGoWest = true;
            } else {
                nextURL = comp.getEast();
                nextGoWest = false;
            }
        }

        goWest = nextGoWest;
        final IComputer nextComp = (IComputer) Naming.lookup(nextURL);
        nextComp.invokeSeeker(this);
    }
}
```

The **migrate** type represents values which can be used for the migration of Ambients. The typing requires that the restricted **login** value is of type **migrate** so that it can be sent on the acknowledgement channel \( k \) and can be used for the co-action `-in login`.

An equivalent program is not possible in the GLINT language for two reasons: Actors are permitted to move between hosts without needing any sort of permission (other than the permissions granted when the GLINTVM is launched); and there is no nesting of Actors within the GLINT language: the top level host Ambient is used as a flat name-space and heap in which all Actors exist. The closest that can be achieved
is shown in example 7.7. Here, upon receiving a message, the server replies with a new anonymous inner Actor instance. This amounts to having a publicly known Actor, caller Server which corresponds to the register channel and the establishment of a unique shared secret with the client, in the GİLİN language case, the anonymous inner Actor instance which corresponds to the restricted login value in the Channel Ambient Language.

Example 7.7 Server handshake in the GİLİN language

```lisp
(define Server
  (<inner>
    where
    inner = define
      where
      any = System.println<this>
  )
)
```

The lack of control of inter-host movement is something that may well need reviewing in any future development of the GİLİN language. The mechanism currently employed is based on the premise of *the simplest thing that could possibly work* but it would be useful for both the user to dynamically alter with whom Actors can be exchanged and for the programs themselves to also carefully stipulate what mobility is permitted. This is however complex: if both an Actor explicitly grants permission in the form of a co-action and the user similarly grants explicit permission then when the Actor is migrated between hosts, if it pairs with the user granted permission then that may block the Actor process from progressing as its co-action has not been executed. The other issue is that with the Channel Ambient System, communication can occur between siblings which means that two processes on different hosts can communicate with one another without the need for any messenger Ambient. This is not possible in the Safe Boxed Ambient calculus and hence in the GİLİNVM and is only simulated loosely in the GİLİN language by the ability to forward method calls to Actors that have since moved to another host via a `become` statement.

There is no doubt in my mind that there are many features missing from the GİLİN language and that given more time, aspects of the language would be reworked to be more consistent whilst introducing abstraction features to make the language more complete and more powerful. However, as it stands, the GİLİN language presents the programmer with a better model and more powerful features than JAVA for tackling the construction of concurrent and distributed systems which are precisely the aims of the language and this project. The comparison with the Channel Ambient Language shows that the GİLİN language does not permit the same fine grained control of inter-host movement whilst presenting instead a clearer and more intuitive language.

### 7.5 Performance

The software developed throughout this project has been designed to make good use of parallel hardware rather than focussing on single-threaded performance. The design of the GİLİNVM maps all Ambient calculus processes to Operating System threads which is wasteful of memory and results in more system calls but eases the implementation. With Actor based languages and languages in which communication between different threads is a key concept, the latency, speed and overhead of communication is very important for achieving good overall performance. Here I present some simple measurements of the performance of the GİLİNVM interpreting hand crafted Safe Boxed Ambient calculus code and interpreting generated code from a GİLİN language program.
All the measurements here were performed on dual-processor dual-core AMD Opteron servers with 8GB of RAM running Linux kernel 2.6.11.9. Where relevant, the interconnection between the servers is InfiniBand. All tests were carried out using the 64-bit Sun JVM version 1.5.0_03. Timings were obtained by creating a built-in function in the GLINTVM which is callable from the input program which prints to the console the current time in milliseconds via the Java System.currentTimeMillis() method.

### 7.5.1 Custom GLINTVM Code

#### 7.5.1.1 Sequential-Sequential Communication

\[ a[ \langle time\_println \rangle^{gvm} . \langle a \rangle \ldots \langle a \rangle . 0 | \langle x \rangle \ldots \langle x \rangle . \langle time\_println \rangle^{gvm} . 0 ] \]

The above program was used to evaluate the speed of simple communication. In place of the two outputs or inputs in the above were 2000 sequential outputs or inputs. The calls to the time_println function are placed so that they occur before the communications start and after the communications end. The program was run five times and the results averaged.

<table>
<thead>
<tr>
<th></th>
<th>Average time for 2000 sequential communications</th>
<th>Average time for 1 sequential communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>253.6 ms</td>
<td>0.1268 ms</td>
</tr>
</tbody>
</table>

#### 7.5.1.2 Parallel-Sequential Communication

\[ a[ \langle time\_println \rangle^{gvm} . (\langle a \rangle . 0 | \ldots | (\langle a \rangle . 0) | \langle x \rangle \ldots \langle x \rangle . \langle time\_println \rangle^{gvm} . 0 ] \]

The above process was used and again, where two inputs or outputs are shown in the above, 2000 were used. This time, the outputs occur in parallel, thus every output is competing to pair with the available input. Again, the timings are placed to ensure they occur before and after the communications. The program was run five times and the results averaged.

<table>
<thead>
<tr>
<th></th>
<th>Average time for 2000 parallel-sequential communications</th>
<th>Average time for 1 parallel-sequential communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>421.6 ms</td>
<td>0.2108 ms</td>
</tr>
</tbody>
</table>

The results show that the increase in contention due to the 2000 competing threads cause the speed of communication to nearly halve.

#### 7.5.1.3 Parallel-Parallel Communication

This cannot be timed on its own because to do so would require that the second timing marking the end of the communications occurs after all the parallel inputs have been executed. This cannot be expressed in the Ambient calculi: you cannot have a process of the form \((P \mid Q \mid R) . S\). Instead, the following process is used.

\[ a[ \langle time\_println \rangle^{gvm} . (\langle a \rangle . 0 | \ldots | (\langle a \rangle . 0) | \langle x \rangle \ldots \langle x \rangle . \langle time\_println \rangle^{gvm} . 0 ] | (\langle y \rangle \ldots (\langle y \rangle . \langle time\_println \rangle^{gvm} . 0 \right) \]

The parallel-parallel communications occur first and then cause parallel-sequential communications to occur which cannot conflict with the ongoing parallel-parallel communications. As usual, 2000 instead of two inputs or outputs are used but here this results in a total of 8000 communication instructions instead of 4000 due to the extra process. The program was run five times and the results averaged.

<table>
<thead>
<tr>
<th></th>
<th>Average time for 2000 parallel-parallel and 2000 parallel-sequential communications</th>
<th>Average time for 1 parallel-parallel and 1 parallel-sequential communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>747.0 ms</td>
<td>0.3735 ms</td>
</tr>
</tbody>
</table>

|                                      | Average time for 1 parallel-parallel communication                                      | 0.1627 ms                                                                      |

The value for the parallel-parallel communication is found by removing the parallel-sequential communication time from the combined parallel-parallel and parallel-sequential time. This shows that parallel-parallel communication is faster than parallel-sequential which is what is expected: with parallel-sequential, only one communication at a time can take place but many outputs are contending for the one available input. With parallel-parallel communication, multiple pairings of inputs and outputs can occur at the same time and execute in parallel. This can occur thanks to the parallelism from the multiple processing cores of the computer. Note that the parallel-parallel communication time is worse than the sequential-sequential communication time.
communication. This highlights the overheads of using Operating System threads and contention for critical regions of the GLINTVM application.

This test was repeated on a Pentium-M computer with no thread-level parallelism and the results shown below.

| Average time for 2000 parallel-parallel and 2000 parallel-sequential communications | 18823.0 ms |
| Average time for 1 parallel-parallel and 1 parallel-sequential communication | 9.4115 ms |
| Speed up of four-core CPU over one-core CPU | 25.2 |

Whilst the single-threaded computer is not of the same class as the servers on which all the other testing occurred, it is close enough to indicate how well the GLINTVM takes advantage of parallel hardware.

7.5.1.4 Local Movement

\[
a[\langle time\_println\rangle^{gvm}\ldots\text{in } b\ldots\text{out } b\ldots\langle time\_println\rangle^{gvm}\ldots\text{.0 }]
b[\text{in } a\ldots\text{in } a\ldots\text{.0 }]
\text{out } a\ldots\text{out } a\ldots\text{.0 }
\]

The above process was used to time movement of Ambients within a single host. 1000 actions were used for every two actions in the above process resulting in the Ambient \( a \) making 2000 movements in total or 1000 trips in and out of the Ambient \( b \). The program was run five times and the results averaged.

| Average time for 2000 local movements | 410.6 ms |
| Average time for 1 local movement | 0.2053 ms |

7.5.1.5 Inter-host Movement

\( \text{hostB} \) ran the process:

\[
\text{in } \text{host:hostA} : a \ldots 0 \ldots \text{in } \text{host:hostA} : a \ldots 0
\]

While \( \text{hostA} \) ran the process:

\[
\text{in } \text{host:hostA} : a \ldots 0 \ldots \text{in } \text{host:hostA} : a \ldots 0
\]

\[
a[\langle time\_println\rangle^{gvm}\ldots\text{in } \text{host:hostB} \ldots \text{in } \text{host:hostA} \ldots \text{in } \text{host:hostB} \ldots \text{in } \text{host:hostA} \ldots \\
\langle time\_println\rangle^{gvm}\ldots\text{.0 }]
\]

\( \text{hostB} \) ran a total of 500 \( \text{in } \text{host:hostA} : a \) actions and \( \text{hostA} \) ran a total of 500 \( \text{in } \text{host:hostB} : a \) actions. The process in the \( a \) Ambient made a total of 1000 inter-host movements or 500 round trips. The program was run five times and the results averaged.

| Average time for 1000 inter-host movements | 13520.0 ms |
| Average time for 1 inter-host movement | 13.52 ms |

It can be seen that inter-host movement is expensive. This is due to having to establish a valid pairing between the two hosts and then to stop the process, serialise it, transfer it and then restart it.

7.5.2 Generated GLINTVM Code

7.5.2.1 Method Calls

The program shown in figure 7.8 was used to test inter-actor method call performance. 300 invocations of the call method were made in total and the generated code was modified to print the system time before and after the method calls. The program was run five times and the results averaged.

| Average time for 300 method calls | 3913.2 ms |
| Average time for 1 method call | 13.044 ms |

As the above table shows, the consequence of the complexity of the encoding is that method calls are not particularly fast. Dividing the method call time by the average of the communication and movement times suggests that there are around 74 pairing operations per method call.
7.5. Performance

![Figure 7.8 Testing Method Calls](image)

```plaintext
define Speed
where
aab = Callee.call<>
aac = aab.call<>
aad = aac.call<>
... 
aln = alm.call<> 
alo = aln.call<> 
```

1 define Callee
2 call ()
3 call
4 <Callee>

![Figure 7.9 Testing Local Method Calls](image)

```plaintext
define LocalSpeed
go ()
go
where
aab = call<> 
aac = call<> 
... 
aln = call<> 
alo = call<> call ()
call
<LocalSpeed>
```

7.5.2.2 Local Method Calls

The Glint program shown in figure 7.9 was used to measure local method call performance. A total of 300 invocations of the call method were made and the generated code was modified by hand to print the system time before and after all the calls had been made. The program was run five times and the results averaged.

Average time for 300 local method calls | 3530.4 ms
Average time for 1 local method call     | 11.768 ms

Dividing the method call time by the average of the communication and movement times suggests that there are around 65 pairing operations per local method call. Note that this is faster than the non-local method calls because the magic-token starter and checker mechanisms are not used for local method calls.

7.5.2.3 Inter-host Method Calls

![Figure 7.10 Testing Inter-host Method Calls](image)

```plaintext
define Speed
where
aaa = Callee<>
aab = aaa.call<> 
aac = aab.call<> 
... 
abx = abw.call<> 
aby = abx.call<> 
```

1 define Callee
2 become me in host:hostB
3 where
4 me = define
5 call ()
6 call
7 <Callee>

The programs shown in figure 7.10 were used to evaluate inter-host method call performance. A total of 50 invocations were made to the call method which amounts 100 movements or 50 round-trips from hostA to hostB and back. The generated code was modified by hand to print the system time before and after all the calls had been made. The program was run five times and the results averaged.

Average time for 50 inter-host method calls | 7776.8 ms
Average time for 1 inter-host method call   | 155.536 ms
The inter-host movement here is much more expensive than hand-written Safe Boxed Ambient calculus code. This is because the encoding nests successive method calls within one another and consequently the first inter-host movement is forwarding the static representation for all 50 method calls which is more expensive for serialisation and transportation. This decreases as more method calls are made.

7.5.3 Comparison with JAVA

The program of figure 7.11 was used to evaluate method call speed in Java. 20,000 method calls were made to the call method. A larger number would have been preferable, but I did not want to use a loop and counter because it would slow the calls down and Java places a limit on the maximum code size of a method. The program was run 10 times and the results averaged.

<table>
<thead>
<tr>
<th></th>
<th>Average time for 20,000 method calls</th>
<th>Average time for 1 method call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.0 ms</td>
<td>0.0001 ms</td>
</tr>
</tbody>
</table>

Thus Java and the JVM are somewhat faster than the GLINTVM with GLINT compiled code. This is expected given that the GLINTVM is implemented in Java and is very complex.

7.5.4 Conclusion

The performance of the GLINTVM is quite respectable and demonstrates that on parallel hardware, processes that can reduce in parallel do so. Whilst much slower than Java and the JVM, the GLINTVM’s performance is adequate and useable. The complexity and size of the encoding from the GLINT language to the Safe Boxed Ambient calculus causes the performance of the interpreted GLINT language program to be slower than it could be: with improvements to the encoding and optimisations of the generated code, the performance could be improved.
Conclusion

This report presents a reducing machine for the Safe Boxed Ambient calculus called the GLINTVM, a language which is based on the Actor paradigm called the GLINT language and a compiler which compiles the GLINT language into Safe Boxed Ambient calculus code. This provides a complete tool chain for both the study of Ambients and for the development of distributed, concurrent and mobile applications in the GLINT language.

8.1 Ambients

Chapter 2 provides an introduction to the Mobile Ambient calculus and reviews the major variations of the Mobile Ambient calculus, most of which are designed with potential type systems in mind. The type systems are examined and the typings of communications and mobility are explained.

Chapter 4 presents the GLINTVM and explains in detail its design and construction. The chosen Ambient calculus is extended with support for polyadic communication. The pairing rendezvous mechanism is examined in detail and a formal model verifying its correctness is shown. Finally an extensible mechanism for incorporating built in functions into the GLINTVM is examined.

The Ambient calculus does present a model of distributed and concurrent computation. However, the lack of consideration of immobile Ambients causes difficulties when implementing an Ambient calculus and avoiding communication and movement interference is problematic given the design of the primitive actions in the calculi. The Channel Ambient System uses richer communication primitives which are better suited to avoiding interference and further development in this direction would lead to an assembly language suitable for concurrent and distributed systems.

The Ambient calculus was designed in order to make analysis and formal reasoning of distributed systems possible. Consequently the available capabilities and actions in the language are minimalistic and not amenable to practical applications. This is shown by the complexity of the encoding from the GLINT language to the Safe Boxed Ambient calculus. Were the Ambient calculus to be extended with more powerful actions and capabilities then the encoding would become much simpler. It would also be possible to retain the ability to formally reason about programs even in light of more powerful primitive actions if the extensions of the calculus were designed sympathetically. Nevertheless, the GLINTVM has shown that a pure Ambient calculus can be successfully used to represent complex expressions and programs.

The GLINTVM is a useful contribution to the study of Ambients, its implementation emphasising the concurrency of Ambients. It is also easily extended to provide additional functions within the calculus and as such would is a good development environment for exploring extensions of Ambient calculi.

8.2 Actors

Chapter 3 introduces the Actor paradigm and examines why it is more suitable for concurrent programming and what problems it has faced. Several languages are examined which demonstrate the range of influence that the Actor paradigm has had on language design since its conception. Modern implementations such as SALSA go a long way to addressing some of the difficult areas of the Actor paradigm and present a more approachable Actor-based language.

Chapter 5 presents the GLINT language piece by piece and explains how it is both based on the Actor model but departs significantly for message passing in an attempt to make the language easier to use. Examples are used to demonstrate how these modifications make classical functions easier to express in the GLINT language than in other Actor-based languages. Mobility is added to the language to allow for mobile and distributed applications and the names that are and are not maintained after movement are examined.
Chapter 6 presents the compiler and the encoding from the GLINT language to the Safe Boxed Ambient calculus. The encoding is explained by considering the components of the language and showing how they can be expressed in Ambients. Parts of the encoding such as methods with predicates involves rewriting expressions in other forms within the language before encoding into Ambients. Finally truth values and numerical values are introduced.

Whilst Actors are not without their detractions, the GLINT language has shown how the addition of synchronous communication with explicit support for call and return semantics eases some of the difficulties of the Actor paradigm and makes for a more intuitive language. Additionally, the Actor paradigm and the GLINT language are both suitable for extension with modern abstraction techniques such as inheritance, interfaces and other code structuring and reuse features.

The GLINT language is a useful addition to the work on Actors and Actor-based languages. By presenting an encoding to the Safe Boxed Ambient calculus a link is established between these two separate models of computation providing a means to both study formally Actors by using tools and techniques established for studying Ambients and to generate Ambient calculus code from Actors allowing for more concise expression of intent.

8.3 Future Work

This section is perhaps longer than is the norm. This is not an indication of the lack of work encompassed by this project, more that this project touches on a large number of ideas and concepts that have not been brought together in this way before. Thus further investigation and development would lead to a better understanding of how GLINT combines and is positioned between these subject areas.

8.3.1 Improving the GLINTVM

The GLINTVM, whilst correctly implementing the Safe Boxed Ambient calculus, [25], is missing several features that limit its usefulness outside of purely experimental environments.

8.3.1.1 Garbage Collection

This is the most significant deficiency with the GLINTVM. The problem is the detection of stuck processes, i.e. processes that can never reduce further. Stuck processes have been defined for CCS and the π-calculus amongst other process algebra, but not to my knowledge for Ambient calculi. However, a straightforward reading of the ideas behind stuck processes in the π-calculus is sufficient to derive a suitable definition for the Ambient calculi and are shown in figure 8.1.

![Figure 8.1 Stuck Ambient processes](image)

Implementing a garbage collection system to detect and remove such processes is complex. The main problem is tracking the distribution of restricted values: this must occur both inside and outside the initial restriction as the side condition on the membership of the restricted value in the free names in figure 8.1 suggests. This becomes especially tricky when you factor in the possibility of movement of processes between hosts as now the information about the existence of restricted values must be tracked on a globally wide basis. Any naive implementation of such a tracking system would quite possibly start making assumptions around global synchronised time, an issue that has been repeatedly shown to destroy concurrency in a parallel system. To avoid this, the most natural solution is to write the monitoring of restricted names using properties of Ambients themselves to produce, ultimately, a concurrent garbage collection system. This does not appear to be a simple undertaking and would require significant additional work.
8.3.1.2 Debugging and Inspection

The GLINTVM does not currently provide mechanisms for the debugging or analysis of running processes. However, given the existence of the freeze-thaw mechanism, used for the distribution of Ambients between hosts, and the elaborate observer pattern used by the test framework, it would be quite straightforward to add such facilities. Additionally, given the internal structure of the GLINTVM it should be very easy to obtain, from a frozen process, a precise description of that process and to inspect and modify if desired, the current values associated with the free names of the process. Thus a relatively small amount of work would create a tool that would have greater use for studying, interactively, the reductions of Ambient calculi.

8.3.1.3 Security of Communication and Inter-System Interaction

Currently, the communication of Ambients between running instances of the GLINTVM are performed using JAVA Objects in their serialised form. Furthermore, the communication is completely unprotected and in the clear. The communication should be made secure in some suitable way, most easily by the use of SSL to encrypt the communication at the socket level. Additionally, it would be beneficial to make the communication expressed in an interchange format that has widespread couplings to languages other than just JAVA. This would allow for the movement of Ambients between Ambient virtual machines written in different languages with different implementations.

Whilst conceptually neither of these tasks is particularly difficult, both are likely to absorb more time than I would like in order to achieve a robust, reliable and secure language-agnostic interchange format.

8.3.1.4 Typings in the GLINTVM

In section 2.4, typing systems for various Ambient calculi are discussed. These typing systems give types to the communication between processes and the movement of Ambients and it would be possible to adapt the typings, given the design of the encoding of Actors into Ambients, such that the typing system plays a similar role to the verifier in the JVM. This would allow for a more formal study of the correctness of the encoding and the possible behaviours of the resulting Ambients. In particular, concerns about communication interference or movement interference in the encoding could be successfully reasoned about.

8.3.2 Extending the GLINT Language

The GLINT language, as explained in chapter 5 has purposefully implemented only the crucial, non-standard and novel aspects of what could be a much larger language. Here I shall discuss extensions to the language, some of which have much in common to other popular and widespread languages whilst others have seen little exposure so far.

8.3.2.1 Asynchronous Messages

A major departure from the Actor model, the GLINT language implements all method calls as not only synchronous but also having a result. There are situations where being able to asynchronously send messages and not expect a result would be beneficial. This would require either a modification to the where clause so that methods could be called but no result captured, thus indicating an asynchronous call, or a separate clause specifically for such calls.

The modifications needed for the encoding wouldn’t be that complex: the asynchronous caller could just be wrapped by the standard caller system allowing the asynchronous caller to escape and perform the call as soon as the standard caller had escaped the calling Ambient. The standard caller would then return and continue on. Such a modification would allow the GLINT language to be treated both as a more typical Actor language where necessary and as a significant departure from some of the key aspects of Actors. This would make it easier to compare and contrast the two approaches in terms of readability, maintainability, concurrency and performance.

8.3.2.2 Interfaces

Interfaces, in the JAVA sense, are an excellent means for decoupling the contract of an API from its implementation. I, amongst others, subscribe to a style of programming in which almost everything is first designed as an interface and then later implemented in classes. In JAVA this has other advantages to just decoupling: it is possible to express highly complex types in generics in the various layers of interfaces and then when it comes to implementing a target or leaf interface, to have massively simplified the typing, making very clear and concise the precise nature of the contract to be fulfilled.
As such, I believe interfaces are a powerful and important concept in programming and should be supported if at all possible.

8.3.2.3 Type Systems

Whilst the language syntactically supports types, there is no type system to make use of them. Adding a type system to enforce valid typings would be a valuable addition and would allow further formal study of the language and its behaviour.

Generics have solved a large class of typing problems with Java but are certainly not a complete solution. Whilst writing GLINT I have come across at least two situations which called for something more powerful than just generics. The first is the ability to specify a relationship between type parameters in a field or variable. This is not possible in generics and so the user must instead create an interface with the required relationship and then refer to the interface, if necessary using wild-cards. Listing 8.2 demonstrates the problem. The problem is a little surprising given that it is possible to express such a relationship both on a class and on a method: it seems odd that it was not implemented for fields. The main problem with the solution is that it is so inflexible. For each such relationship it requires a new interface to be created and for that interface to be suitably implemented. Further, at the point of use, the type parameters are hidden. There are situations where this encapsulation of type parameters is a good thing and other times where the loss of precision of the typing (due to the enforced use of wild-cards) is a bad thing.

Listing 8.2 Problems caused by inability to express relations between type parameters in fields in Java

(a) The desired typing.

```java
1 public class Something
2   <Type1 extends Foo<Type2>, Type2 extends Bar<Type1>> { ...
3 }
4
5 public class Anything {
6   private final
7       <A extends FooSubtype<B>, B extends BarSubtype<A>>
8         Something<A,B> aField = null;
9 }
10}
```

(b) The solution. Note that the relationship between types is hidden where the interface type is used.

```java
1 public interface SomeInterface
2   <Type1 extends FooSubtype<Type2>, Type2 extends BarSubtype<Type1>> { ...
3 }
4
5 public class Anything {
6   private final
7     SomeInterface<?,?> aField = null;
8 }
```

The other problem I came across is more complex and is shown in listing 8.3. The listing does not compile because it is not valid to pass this as a parameter to a super-constructor. It is also type incorrect to assign to the myself field the value of this in the super-constructor and for the same reason, the method cannot return the value of this in the base class DefaultFormatter. The most obvious solution, is to drop the final modifier on the field myself. But this is a truly terrible solution as it now means that any constructor of any subclass must explicitly remember to set the field itself: without a temporal type system, this cannot be enforced. The best that can be done is to modify the DefaultFormatter class so that the subclasses are required to provide the necessary value, but this then requires an additional method to be added to all subclasses. That said, the benefit of this solution is that the type system enforces the implementation of the abstract method; this solution is shown in listing 8.4. There are however, much better solutions: here I am effectively using generics to try and simulate the same properties as MyType, which offers a better and simpler solution to not only this situation but a wider class of problems too.

The purpose of these examples is to demonstrate that whilst a type system with generics is a large step up from one without, it is certainly not a solution to all problems: there are very real situations in which something that is more powerful than generics is called for. Given the unusual properties of Actor systems
8.3. Future Work

Listing 8.3 Problems resulting from an inability to pass the type of this

```java
public interface Formatter<ReturnType extends Formatter<ReturnType>> {
    ReturnType doSomething();
}

public abstract class DefaultFormatter<ReturnType extends Formatter<ReturnType>> implements Formatter<ReturnType> {
    private final ReturnType myself;
    public DefaultFormatter(final ReturnType me) {
        myself = me;
    }
    public ReturnType doSomething() {
        ...
        return myself;
    }
}

public interface AmbientFormatter extends Formatter<AmbientFormatter> {
}

public class DefaultAmbientFormatter extends DefaultFormatter<AmbientFormatter> implements AmbientFormatter {
    public DefaultAmbientFormatter() {
        super(this);
    }
}
```

and the ability for an Actor to change its behaviour and hence in some senses, its type, it would seem that GLINT is an ideal language to experiment with and develop more flexible and powerful type systems than have been seen in languages so far.

8.3.2.4 Named Parameters

The majority of programming languages use positional parameters in method invocation: the parameters are assigned to the parameter variables in the method body in the same order as they are listed by the method caller. In this, GLINT is no exception, nor are any of the Ambient calculi.

The main disadvantage of this is that at the call site, it is often difficult to determine which arguments play which roles for the method call. Given the emphasis within software engineering practices for the use of meaningful variable names, it seems unfortunate that the names of the parameter variables in the method cannot be seen by the caller.

Named parameters (also called keyword parameters) have been seen in several languages including SMALLTALK, ADA and more recently PYTHON and RUBY. When using named parameters, the caller must explicitly assign in the method invocation which values are assigned to which parameter names. In the method body, these parameter names are the parameter variables as normal.

Not only can the parameters be defined in any order by the caller, but the callee can also define default parameter values that are used if the caller does not define a value for that parameter.

In [17] positional parameters and named parameters are compared and a tentative endorsement of named parameters made. One of the chief criticisms is that if the name of the parameter is changed in the method body then that will force all calls to that method to need to be modified. Given the power and utility of the programming tools and environments we have today, this is no longer a valid criticism as, with sufficient support, the subsequent changes to the call sites could be entirely automated.

As a mechanism to increase readability of source code, named parameters are a good idea and it seems surprising that they have not seen greater support in language design. Good tool support is required for the
Listing 8.4 Solution to the changing type of \textit{this} problem

```java
public interface Formatter<ReturnType extends Formatter<ReturnType>> {
    ReturnType doSomething();
}

public abstract class DefaultFormatter<ReturnType extends Formatter<ReturnType>> implements Formatter<ReturnType> {
    private final ReturnType myself;

    public DefaultFormatter() {
        myself = getMyself();
    }

    protected abstract ReturnType getMyself();
}

public interface AmbientFormatter extends Formatter<AmbientFormatter> {
}

public class DefaultAmbientFormatter extends DefaultFormatter<AmbientFormatter> implements AmbientFormatter {
    public DefaultAmbientFormatter() {
        super();
    }

    protected AmbientFormatter getMyself() {
        return this;
    }
}
```

additional typing and refactoring overhead to become negligible. This is certainly a feature that I would be very keen to add to GLINT: the compiler could easily do all the necessary reordering so the encoding need not change and there is absolutely no need to modify the GLINTVM.

8.3.2.5 Currying

Currying is named after Haskell B. Curry, though he denied inventing the idea and it is commonly credited to Schönfinkel in [31]. With currying, a function is supplied with fewer parameters than it in needs to complete. Instead of returning an error, it returns a function that has captured the supplied parameters and awaits only the remaining parameters. When those parameters are supplied, the function will execute as if all parameters had been supplied simultaneously.

Currying has found its way into several functional languages, which is not surprising given that Curry himself used this idea in the $\lambda$-calculus. However, one of the chief complaints of currying is that it's too restrictive: because the parameters have to be supplied in the order dictated by the function, the function author is responsible for defining the order of parameters in an order which is of most use for currying. This is not always possible.

To solve this restriction, I propose that by using named parameters throughout, any set of parameters can be passed as they will be named and are therefore unambiguous. This then gives the caller complete freedom to curry any method with any subset of the parameters that they wish.

Currying is another feature that has made it into the SCALA language [28], though not without limitations. For a method in SCALA to be curry-able, it must be defined with multiple parameter sections explicitly. Sadly this means that none of the JAVA API libraries can be curried without explicit wrappers. Listing 8.5 shows currying in SCALA.

Again, currying would not be too difficult to implement. The encoding would need modification so that when a method is called with fewer parameters than expected, an anonymous Actor is returned which has
8.3. Future Work

Listing 8.5 Currying in Scala

```scala
// Listing 8.5 Currying in Scala

def sum(x: Int)(y: Int): Int = x + y;

val test: Unit = {
  var plus4Func = sum(4);
  var nine = plus4Func(5);
}
```

captured the supplied parameters and is ready to receive any number of remaining parameters either issuing more anonymous Actors or invoking the target method if enough parameters have been supplied. There are however semantic issues which have stopped me from attempting to implement currying so far. The main problem is that after currying a method, what happens if other methods are called on the Actor which alters the behaviour of the Actor: the curried method cannot now complete. Further, if the resulting behaviour has a similarly named method, supplying the curried method with the additional parameters would result in a completely different method being invoked with, quite possibly, unintended results. Such a feature would need careful evaluation to determine whether the potential dangers are outweighed by its utility.

8.3.2.6 Traits

Traits were first proposed in [30]. Traits succeed where mixins, [4], do not because traits enjoy the flattening property: the behaviour of a class which uses several traits is the same as if the contents of the traits had been copied directly into the class definition. It achieves this through several means: traits have no state of themselves and cannot directly access state from the classes in which they are used, instead all access must be through methods. These methods form requirements which must be satisfied by the class in some manner. If multiple traits are used and they provide the same method then that conflict must be handled explicitly by the class: there is no implied ordering favouring the method definition from one trait over another.

These design decisions serve to remove traits from the inheritance hierarchy in that using traits is no longer akin to providing a superclass to the trait as is the case with mixins. Much finer-grained control is provided with no ordering between traits. Three additional functions are defined: when using a trait, particular methods can be deliberately excluded; aliases can be defined, renaming methods in traits within the current class; and conflicts must be explicitly resolved by providing an overriding definition in the current class.

The authors of [30] implemented traits in the SMALLTALK machine Squeak and then refactored the collection hierarchy. The results were 10% fewer methods than in the original implementation and the methods that were refactored into traits were themselves 12% smaller. Although some classes made use of up to 22 traits, the flattening property means that this need not impede comprehension: tool support might allow for the display of the methods within those 22 traits within the class itself, potentially obeying exclusion, aliasing and conflict resolution annotations.

Because the initial implementation was performed on SMALLTALK which is dynamically typed, no static type system was given in [30]. In [14], such a static type is given in which the core of traits is expressed as a calculus and type soundness is shown. Traits are clearly a very successful mechanism for separating out different aspects of classes and providing a more flexible and comprehensible solution to code reuse than inheritance and subclassing provides. I am very keen to add traits to GLINT: again, this is a feature that can be entirely implemented in the compiler as traits can be thought of as controlled and configurable macro-expansion. I fully expect traits to become a widespread feature of future programming languages.

8.3.2.7 Libraries and Error Handling

Currently, there are no standard libraries, in certain cases generated Ambient code has to be tweaked by hand in order to make it work as it relies on special functions built in to the GLINTVM, and there are no mechanisms for error handling. All these deficiencies need to be addressed.

One of JAVA’s main strengths is the size and comprehensiveness of its libraries. Yet, there are many situations where additional functionality is required and there are no shortage of projects that provide that extra functionality, from more powerful collections libraries to better support for encryption. Having a comprehensive and successful set of base libraries is nevertheless a very strong attraction to working in JAVA. Development of such libraries takes a long time and a lot of effort. Unless GLINT becomes very popular and demand for a set of standard libraries is high, it’s difficult to justify the time needed to develop such libraries.
Error handling has always been an almost religious issue when it comes to the debate between the merits or otherwise of caught versus uncaught exceptions. Java was the first mainstream language to introduce caught exceptions, i.e. exceptions which the caller of the method must explicitly catch and deal with, and the propensity to create catch blocks which do nothing, effectively ignoring the exception, suggests that caught exceptions are not successful for achieving good error handling. On the other hand, the alternative of uncaught exceptions is even worse as without very good tool support, it is very hard to obtain which exceptions can be generated from any given code. Further, in Java, caught and uncaught exceptions have different semantics: caught exceptions are meant to symbolise exceptions that the caller should reasonably be able to recover from, whilst uncaught exceptions are situations in which the caller would not be able to recover from. It is no mistake that all the errors that the JVM generates (e.g. OutOfMemoryError) are uncaught exceptions. However, some of the choices that the Java library authors made as to where to use caught and uncaught exceptions seem odd at the very least. In many cases it would seem that the authors made certain exceptions uncaught to reduce the overhead on users making use of those particular methods.

Personally, I’m not sure if any of the current solutions are flexible enough or make the right balance between non-pollution of the calling code and safety. I think there is significant scope for further research and development in this area, and again, given some of the unique properties of Actor languages and GLINT, the GLINT language would be an excellent platform upon which to develop prototype solutions.

8.3.3 Redesigning the Encoding

As explained in chapter 5, an Actor can not be implemented by a single Ambient process. Whilst this cannot be avoided without significant alterations to the Ambient calculi, there are nevertheless areas in the encoding which are inefficient and wasteful of resources and which could be improved. The most significant issue is the use of replicated processes. These can never terminate and whilst they won’t eat up CPU cycles when they become stuck, they will use up memory. Many of these situations would be resolved by adding a garbage collector to the GLINTVM, as discussed in section 8.3.1.1 but several could also be solved by reworking the encoding so that it does not make use of replication. This however is very tricky and is not a possible solution in several situations.

Where possible, the encoding tries to make good reuse of simple components, for example, the encoding of predicates into anonymous Actors. However, in most cases, the reuse uses a component which is a more general solution than necessary and so resources are wasted. Again, with the example of the predicates, the anonymous Actors are full anonymous Actors and can deal with any number of calls. Yet they are only ever called once because they are generated afresh for every call to their parent Actor. Either they should be simplified so that they can only deal with one call and then stop or they should exist more permanently but are not generated afresh for each call. A careful examination of the encoding shows that there are many situations in which this kind of optimisation can be applied.

Further, currently the compiler applies no optimisations. The side-effecting nature of Actors means that many optimisations relating to code motion cannot be safely applied but there are nevertheless opportunities for classical optimisations which could improve the performance and efficiency of the encoding.

8.4 Final Thoughts

If the GLINT language was further developed and gained features that aided the structuring and reused of code, and if the encoding was improved to make it faster and more efficient than I would with some confidence use the GLINT language for future application development.

The simplicity of the model of concurrency and the improvements it makes over typical Actor languages makes development using the GLINT language easier and faster than writing equivalent applications in languages like Java. However, it would take some time to create a suite of tools that match Eclipse amongst others for the development of GLINT programs. Mobility is a growing area and I would be very keen to develop code for mobile applications in GLINT and to compare to equivalent solutions for real-world applications.


[31] M. Schönfinkel. Über die bausteine der mathematischen logik. Mathematische Annalen, 92:305–316, 1924. 8.3.2.5


[34] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. SIGPLAN Not., 36(12):20–34, 2001. 3.3.3, 3.3.3.2, 3.3.3.3

No metrics can possibly indicate with any reliability the complexity of a project but they can create an impression of the size of the software written. Here then are some statistics relating to both the GLINTVM and the Compiler. These were taken from the seventh release candidate on the 12th June 2006.

73,995 total lines of code
51,039 . . . of which is JAVA (.java)
  3,801 . . . of which is ANTLR grammar (.g)
  193 . . . of which is ANT build script (.xml)
    (the rest being auxiliary ANTLR output)

44,661 total lines of generated code
29,334 total lines of hand written code

  84 automated tests
  265 JAVA classes
  110 JAVA interfaces

  1 bug encountered in Sun’s JAVA JVM
  1 bug discovered in Sun’s JAVA compiler
  2 bugs discovered in ECLIPSE (relating to handling of generics)
Here I present a larger example of a system written in the GLINT language, featuring mobility of Actors. The example is an email server which delivers emails to users. When a user logs in to a workstation they contact the central server and authenticate which moves their mailbox out onto the workstation they're using. When emails arrive, they are delivered to the relevant mailbox which seamlessly incorporates any necessary movement from the central server to the workstations. When the user logs out, their mailbox is moved back into the central server. In this example, there are certain components that are not shown for purposes of conciseness, for example, the initial construction of the Actors for a new user.

### B.1 Email

Example B.1 shows the definition of the Email Actor which represents individual emails. Obviously in a more complete implementation there would be additional methods to access the individual components of the message rather than just the To field. Note how, in the display method, the method calls made are chained to guarantee a particular ordering. This takes advantage of the fact that methods that don't have an explicit return statement implicitly return the name of their Actor and so this allows for the chaining as seen in the method. If this weren't possible then there would be no guarantee between the ordering of the various println invocations.

Of most importance is the move method which allows the Actor to be moved between hosts. Note how as well as moving, the method returns the name of the new Actor instance. This is therefore returning the name that will refer to the Actor in the target host, thus it allows, when moving a structure of emails, for that structure to reassemble itself on the new host by having knowledge of the replacement names of all the Actors once they have moved to the new host.

#### Example B.1 The Email Actor definition

```plaintext
1 define Email (Text to, Text from, Text subject, Text body)
2 display ()
3 display
4 where
5   any = System.println<to>
6   any' = any.println<from>
7   any" = any'.println<subject>
8   any"" = any".println<body>
9
10 move (Host h)
11 move
12   <meMoved>
13   become meMoved in h
14 where
15   meMoved = new Email<to, from, subject, body>
16
17 getTo ()
18 getTo
19   <to>
```

---

105
B.2 Mailbox

Example B.2 shows the definition of the mailbox into which emails are delivered and which moves around between hosts. It works in much the same way as a stack or list: adding an email adds to the head of the stack whilst reading and deleting can act upon any element within the structure. Note that the mailbox acts as a receptionist making sure that if it is the head of the structure then it remains so after adding or removing emails. This is the reason for the clone within the add method.

Again, of real interest is the move method. The entire structure is moved and all the move methods in both the Mailbox and Email Actors share the same pattern of returning the name of the Actor that is going to move thus allowing the structure to reassemble itself on the far side. To make this clearer, consider what would happen if just the Mailbox moved and not the Email. Having arrived on the new host, it would find that it would not be able to communicate with the Email because the Email is still on the old host. If it told the Email to move but didn’t get the new name then whilst the Email would indeed move, the Mailbox would still not be able to communicate with the Email because it has its old name, i.e. the name of the Email that was (and still is) valid on the previous host.

B.3 Central Email Server

Example B.3 shows the definition of the central email server Actor. It relies heavily on a Database Actor which is not shown because it is just a straightforward map which really doesn’t demonstrate any of the interesting features of the GLINT language. Upon login, the user contacts the central server and tells it what host the user is working on. The server looks up the mailbox for the user and asks the mailbox to move to the host of the user. The database also records which host the user is working on.

When the user logs out, the server looks up which host the user was on and gets the mailbox to move back to the host the server is on. Note that the name in the database at this point is the name of the mailbox before it was moved, but forward references are permitted due to the become statements so the message to tell it to move is dutifully sent and forwarded to the mailbox on the host and result is returned. The result is, as discussed in the previous section, the name of the Actor that the mailbox will become, i.e. in this case, the name of the mailbox once it has arrived back on the central server host. It is this name that is stored back in the database. The reason for updating the database with this name is that if it were not then whilst it would still work, messages sent to the mailbox would go out to each workstation that the user has ever used and then back to the central server and if any of those workstations got turned off then the message would get blocked. So whenever the mailbox returns to the central server, the database is updated so that there will never be more than one inter-host transfer required when the central server communicates with the mailbox.

When an email is delivered, it is vital that the email is on the same host as the mailbox and that both the email and mailbox know each other’s names valid for the current host. Thus, after looking up the user to whom the email is addressed, the mailbox and host of the mailbox are found. The email is moved to that host and then added to the mailbox. Note that the name added is the result of the move which ensures that the name of the email that the mailbox receives is indeed the correct name of the email with the host in which the mailbox currently exists.

Note that there is never any need for synchronisation here, for example to combat interleaving of messages to the database should the user log in and out very quickly: only one message is ever dealt with at any one time in an Actor and so there can never be two threads in an Actor both trying to access critical regions. What may be important is to ensure that should a user log in and out very quickly, the messages are dealt with in the correct order, but some straightforward logic would be able to detect a logout occurring before a login and adjust appropriately. On the whole, the concerns and worries of the concurrent programmer don’t exist with the Actor model.

B.4 Comparing to JAVA

The equivalent Java implementation would be more complex. All the classes would have to make use of synchronisations to avoid multiple threads accessing the same data concurrently which requires a consistent ordering for obtaining the locks. For all inter-host communication and movement, RMI would be required which would create more code and would require good understanding of Java’s RMI mechanisms. The become statement in the GLINT language and its ability to make inter-host movement simplifies greatly the mobility of the emails and mailboxes in comparison to Java’s RMI system.
Example B.2  The Mailbox Actor definition

```
define MailBox (Email head, MailBox tail)
add (Email msg)
add
  <mb>
  become mb
  where
  clone = new MailBox<head, tail>
  mb = new MailBox<msg, clone>

read (Number n)
read Zero.eq<n>
  <head>
read
  <tail.read<n.minus<One>>>(
getTail ()
getTail
  <tail>

delete (Number n)
delete Zero.eq<n>
  become tail
delete One.eq<n>
  become mb
  where
tail' = tail.getTail <>
  mb = new MailBox<head, tail'>
delete
  where
  any = tail.delete<n.minus<One>>>(
move (Host h)
move Nil.eq<tail>
  <mb>
  become mb in h
  where
  mb = new MailBox<head',Nil>
  head' = head.move<h>
move
  <mb>
  become mb in h
  where
  mb = new MailBox<head',tail'>
  head' = head.move<h>
  tail' = tail.move<h>
```
Example B.3 The Server Actor definition

```java
1   define Server (Host serverHost)
2   login (User u, Host h)
3       login
4           where
5               mb = Database.getMailBoxForUser <u>
6               any = Database.setHostForUser <h, u>
7               any' = mb.move <h>
8
9   logout (User u)
10  logout
11     where
12        mb = Database.getMailBoxForUser <u>
13        mb' = mb.move <serverHost>
14        any = Database.storeMailBoxForUser <mb', u>
15        any' = Database.setHostForUser <serverHost, u>
16
17  deliverMail (Email msg)
18  deliverMail
19     where
20        user = Database.getUser <msg.getTo>
21        mb = Database.getMailBoxForUser <user>
22        host = Database.getHostForUser <user>
23        msg' = msg.move <host>
24        any = mb.add <msg'>
```
LTSA Tutorial

LTSA¹ is the Labelled Transition System Analyser and, given process definitions expressed as Finite State Processes, can display processes as labelled transition systems and perform exhaustive analysis to detect deadlock, amongst other things. Here I provide a brief introduction which should be sufficient to understand the models presented in this report.

Firstly, a process has a name and a sequence of actions which it can perform. The sequence can branch but should ultimately recurse so as to represent an infinite process. For example, the finite state process

\[ \text{TEAPOT} = ( \text{teabags} \rightarrow \text{hotWater} \rightarrow \text{stew} \rightarrow \text{pour} \rightarrow \text{empty} \rightarrow \text{clean} \rightarrow \text{TEAPOT} ) \]

describes a single process that goes through the actions shown in the prescribed order before looping and starting again. But what if it is wanted so that the teapot can pour more than once? For this, the process must be broken down into a number of sub-processes, thus

\[ \text{TEAPOT} = ( \text{teabags} \rightarrow \text{hotWater} \rightarrow \text{stew} \rightarrow \text{POUR\_OR\_FINISH} ) \],
\[ \text{POUR\_OR\_FINISH} = ( \text{pour} \rightarrow \text{POUR\_OR\_FINISH} | \text{empty} \rightarrow \text{clean} \rightarrow \text{TEAPOT} ) \].

Now, the sub-process, POUR\_OR\_FINISH is called by the main TEAPOT process and presents a choice: either the process can pour and then return to the same choice or it can empty, clean and then go back to the start again.

But this only models a teapot. How about a mug? The mug process would want to be filled, drunk and then rinsed, and it would need to be placed in parallel with the teapot. What's more, the pour action that the teapot does would correspond to the fill action on the mug.

\[ \text{TEAPOT} = ( \text{teabags} \rightarrow \text{hotWater} \rightarrow \text{stew} \rightarrow \text{POUR\_OR\_FINISH} ) \],
\[ \text{POUR\_OR\_FINISH} = ( \text{pour} \rightarrow \text{POUR\_OR\_FINISH} | \text{empty} \rightarrow \text{clean} \rightarrow \text{TEAPOT} ) \].
\[ \text{MUG} = ( \text{fill} \rightarrow \text{drink} \rightarrow \text{rinse} \rightarrow \text{MUG} ) \].
\[ \| \text{SYS} = ( \text{MUG} \| \text{TEAPOT} )/(\text{pour/ﬁll}) \].

Here, a concurrent composition of the MUG and TEAPOT processes is created called SYS and the fill action is renamed to pour. This now introduces a very important idea: where two processes perform an action with the same name, they share that action and are required to execute the action at the same time. This is the primary means of representing interaction between different processes. The final example demonstrates that if ever an action is shared between two processes then it must be explicitly used by all processes that ever use the action whenever the action is to be performed.

\[ A = ( \text{g} \rightarrow \text{h} \rightarrow \text{i} \rightarrow A ) \],
\[ B = ( \text{g} \rightarrow \text{i} \rightarrow B ) \],
\[ \| \text{SYS} = ( A \| B ) \].

In the above process, no deadlock occurs. This is because the action h is never used by B and so it is treated as being private to A. The two processes share the actions g and i and in between those actions h occurs

¹http://www.doc.ic.ac.uk/~jnm/book/index.html
because A requires it and B has no knowledge of h. However, modifying the process slightly to

\[
A = (g \rightarrow h \rightarrow i \rightarrow A).
\]

\[
B = (g \rightarrow i \rightarrow B
\]

\[| j \rightarrow h \rightarrow B).\]

||SYS = (A || B).

now causes \texttt{Ltsa} to report that a deadlock will occur after the \texttt{g} action. j is private to B so it cannot be j that is causing the problem. The problem is that j leads on to h which means that h is no longer private to A. As before, A wants to perform an h after the g but as a result of B now having some knowledge of h, the absence of h after g is interpreted as B explicitly not wanting any h to be executed: it wants to go straight from the g to i. This causes the deadlock. The reason for this example is that because of this property of \texttt{Ltsa}, the models I present will sometimes suggest that a particular process has knowledge of another process before the mechanism being modelled can have permitted that knowledge. The models have to be presented this way as a result of this particular property of \texttt{Ltsa}.
Using the GLINT VM and Compiler

D.1 Using the GLINT VM

Using the GLINT VM is quite straightforward. The GLINT VM is run from the command line and has a number of GNU style options. Most of these relate to causing additional debugging information to be printed. The only required option is either a filename for a file containing a program written in the Safe Boxed Ambient calculus or a ‘-‘ symbol (which can be omitted) indicating that the input is to parsed from the console. If used, a file must be plain text but have a .gvm extension. The standard -h option is available which will display help describing the various options and explaining the style of launching the GLINT VM.

If all processes terminate then the GLINT VM will exit correctly when it has no more work to do. A replication however, never terminates and so the GLINT VM must be forcibly terminated in the case of the program containing a replication or otherwise being unable to reduce all processes to nil.

Inter-host communication is achieved by socket connections that use TCP and operate on port 21212. You may need to modify your firewalls to allow for this communication. Every host within the network of hosts should ideally be able to initiate a connection to every other host, but with knowledge of the network topography and accessibility, processes can be designed to specifically work around limited connectivity as intended by the original Ambient calculus.

D.2 Using the GLINT Compiler

As with the GLINT VM, the GLINT Compiler is invoked from the command line and has several options, some of which are of interest only for debugging. A -o option specifies the output file name. If this does not end with the extension .gvm then such an extension will be added. If omitted then the compiled code is sent to the console. The required arguments must either be a ‘-‘ to indicate that the program to be compiled will be read from the console or it must be a valid path to a file in which the program to compile must exist. The file must have the extension .glint. Multiple files can be supplied and each file must contain exactly one named Actor definition. Due to issues with the parser, it is generally wise to append an empty line to each file from which an Actor definition is to be read.

Compiling will by default create code that is ready to be launched by the GLINTActorVMLauncher which will be explained shortly. The GLINT Compiler, if given the -b option will instead produce a library. This is very useful when compiling libraries of Actors such as truth values and numbers. The -k option takes an argument that points to a library and links with the library. Multiple -k options can be given. This is typically used to link the compiled truth values, numbers and other library functions either together to produce one large library if given the -b option, or with other Actor definitions to produce a launch-able compilation. Compilation to a launch-able target is closed-world, i.e. all needed Actors and libraries must be supplied, either as GLINT code or as a compiled library with which to link. Compilation is open-world when creating a library.

Having been compiled, if executed directly by the GLINT VM, nothing can possibly happen. This is because an Actor does nothing until it receives a message and the compiled code consists of nothing other than encoded Actors. Additionally, the hosts between which you want to move Actors need processes permitting such movement and the compiled code depends, for inter-host movement, on the name _originatingHost_ which contains the name or IP of the local machine. This is crucial for message-ambients that have been transported to other hosts as they need to know the name of the host from whence they came in order to return. These reasons explain the existence of the GLINTActorVMLauncher which has two required arguments, the first of which is the path to the compiled code and the second of which is the name of an Actor to which is sent a method-less and argument-less message. This is equivalent to specifying the name of the class to Java in which to find the main method and must name an Actor that does not have methods and does not need instantiating via a constructor call.
The GlintActorVMLauncher also has a couple of options. The first is -l and specifies the name or IP of the local machine: it is this that is bound to the _originatingHost_ name. If not supplied then it does its best to work out the IP but if the host has more than one IP then it’s reasonably likely that it’ll pick the _wrong_ one, for some definition of _wrong_. The other option is -e which can occur many times and takes an option, again a host name or IP. This is to allow the entry of Ambients from the named host. The relationship should be symmetric: if host A allows the entry of Ambients from host B then host B should allow the entry of Ambients from host A because otherwise the message-ambients will only be able to go in one direction and will get stuck in the other direction.

As usual, the -h option is available for both the GlintCompiler and the GlintActorVMLauncher and displays useful help on the options and invocation of the applications.