DynamiTE:
A 21st-Century Framework for Concurrent Component-Based Design

Andrew John Hughes

11th of December 2009

This thesis is submitted for the degree of *Doctor of Philosophy*

Department of Computer Science
University of Sheffield

Abstract

The free ride for software developers is over. In the past, computer programs have increased in performance simply by running on new hardware with ever increasing clock speeds. Now, however, this line of development has reached its end and chip designers are producing new processors, not with faster clocks, but with more *cores*. To take advantage of the speed increases offered by these new products, applications need to be redesigned with parallel processing firmly in mind.

The problem is that mainstream designs are still *inherently sequential*. Concurrency tends to be an afterthought that may be useful to gain a performance boost, not an essential part of the design process. The current vogue for object-oriented designs tends to also have the side-effect of making them heavily *data-oriented* which doesn't scale well; each shared element of data has to be protected from simultaneous access, resulting in operations becoming sequential again. In addition, the usual methods for protecting data tend to be very low-level and error-prone.

In this thesis, we introduce a new design method whereby applications are constructed from small sequential *tasks* connected by intercommunication primitives. Our approach is based on a two-stage process; first, the individual tasks are created as independent entities and tested with appropriate inputs, then secondly, the communication infrastructure between them is developed. We provide support for the latter via the DynamiTE framework, which allows the interactions to be defined using the terms of a process calculus. Depending on the developer's background, they can treat this as just another API, as a design pattern or as an algebraic expression which can be property checked for issues such as deadlocks. Either way, the communication layer can be developed, tested and evaluated separately from the tasks once it is known how the tasks will interface with one another.

To supplement DynamiTE, we define our own process calculus, Nomadic Time, using a carefully chosen novel selection of constructs. Among the features of the calculus are the ability to perform communication both locally (*one-to-one*) and globally (*one-to-many*), and the flexibility to change the location of tasks during execution. Security is paramount to the design of Nomadic Time and migratory operations can be limited in two ways; by simple enumeration of possibilities or by the optional typing of constructs to allow restriction on a task-by-task basis.

While it can't eradicate all the problems inherent in designing concurrent applications, DynamiTE can make things easier by reducing the dependency on shared resources and enhancing the reusability of concurrent components.

Acknowledgements

This work was supported by a grant from the Engineering and Physical Sciences Research Council (EPSRC) for the first three years.
Thanks to everyone else for keeping it bonkers!

Contents

1 Introduction **1**
- 1.1 Rationale . . . 1
- 1.2 Status Quo . . . 2
  - 1.2.1 Multiprogramming . . . 2
  - 1.2.2 Resource Contention . . . 3
  - 1.2.3 Semaphores and Monitors . . . 5
  - 1.2.4 Interprocess Communication . . . 11
- 1.3 Our Proposed Solution . . . 14
  - 1.3.1 A Prototypical Application . . . 16
- 1.4 Contributions to Knowledge . . . 18
- 1.5 Structure of the Thesis . . . 20

2 Algebraic Process Calculi **24**
- 2.1 Introduction . . . 24
- 2.2 The Calculus of Communicating Systems . . . 26
  - 2.2.1 The Dining Philosophers . . . 29
- 2.3 Advantages and Limitations of CCS . . . 31
- 2.4 Conclusion . . . 33

3 Global Synchronisation **35**
- 3.1 Introduction . . . 35
- 3.2 Temporal Process Language (TPL) . . . 35
- 3.3 Extending TPL . . . 37
- 3.4 The Calculus of Synchronous Encapsulation . . . 37
  - 3.4.1 Timeouts . . . 38
  - 3.4.2 Clock Stopping and Insistency . . . 39
  - 3.4.3 Encapsulation . . . 40
- 3.5 Conclusion . . . 41

4 Mobility **44**
4.1 Introduction . . . 44
4.2 Scope Mobility . . . 45
4.2.1 The π Calculus . . . 45
4.2.2 Variants of the π Calculus . . . 46
4.2.3 Advantages and Limitations of the π Calculus . . . 52
4.3 Distribution and Migration . . . 53
4.3.1 The Distributed Join Calculus . . . 58
4.3.2 The Ambient Calculus . . . 59
4.3.3 Variants of the Ambient Calculus . . . 62
4.3.4 Advantages and Limitations of the Ambient Calculus . . . 65
4.3.5 P Systems . . . 65
4.4 Comparing Modelling Approaches . . . 68
4.5 Bigraphs . . . 71
4.6 Conclusion . . . 72

5 Nomadic Time **74**
5.1 Introduction . . . 74
5.2 Localising the Calculus . . . 76
5.3 Adding Mobility . . . 78
5.3.1 Location Mobility . . . 80
5.3.2 Process Mobility . . . 83
5.4 Bouncers . . . 84
5.5 The Semantics . . . 85
5.6 A Simple Example . . . 91
5.7 A Prototypical Application in NT . . . 95
5.8 Conclusion . . . 98

6 The DynamiTE Framework **100**
6.1 Introduction . . . 100
6.2 Why Java? . . . 101
6.2.1 Concurrency Provision . . . 103
6.2.2 Popularity . . . 108
6.3 Mapping Theory to Practicality . . . 109
6.4 The Context of the Calculus . . . 114
6.4.1 The Plugin Abstraction . . . 116
6.5 The Evolver Framework . . . 118
6.6 A Prototypical Application in DynamiTE . . . 123
6.7 Related Work . . . 131
6.7.1 Obliq . . . 131
6.7.2 Nomadic Pict . . . 134

6.7.3 The Safe Ambients Abstract Machine . . . 135
6.7.4 JavaSeal . . . 136
6.8 Conclusion . . . 138

7 Typed Nomadic Time **140**
7.1 Introduction . . . 140
7.2 Existing Typed Calculi . . . 140
7.2.1 Type Systems for the π Calculus . . . 141
7.2.2 Type Systems for the Ambient Calculus . . . 143
7.3 Mobility Types for Nomadic Time . . . 145
7.4 DynamiTE and the Type System . . . 149
7.5 Typed Musical Chairs . . . 153
7.6 Conclusion . . . 154

8 Contributions and Future Work **156**
8.1 Our Contributions . . . 156
8.2 Future Work . . . 160
8.2.1 Nomadic Time . . . 160
8.2.2 DynamiTE . . . 162
8.2.3 The Type System . . . 163
8.2.4 Other Applications . . . 164

Bibliography **165**

A Progress **174**

B Preservation **178**

List of Figures

1.1 The Traditional Software Development Process 16
1.2 DynamiTE Variant of the Software Development Process 16
1.3 Structure of the Prototypical Application 18

2.1 Graph of $a.0 \mid \overline{a}.0$ 28
2.2 Graph of $a.0 + \overline{a}.0$ 28

4.1 Spatial diagram of $m[in\ n.out\ n.P] \mid n[]$ 60
4.2 Spatial diagram of $n[m[out\ n.P]]$ 60
4.3 Spatial diagram of $m[P] \mid n[]$ 60
4.4 Example P System 67

5.1 Derivation of Nomadic Time 75
5.2 Clock Hiding in $n[E \mid m[F \mid k[G]_{\{\sigma\}}]_{\{\rho\}}]_{\{\gamma\}}$ 78
5.3 The Musical Chairs Environment 92

7.1 Derivation of Typed Nomadic Time 145

List of Tables

2.1 CCS Semantics 29

3.1 CaSE Semantics 42
3.2 Derivation of CaSE from CCS 43

4.1 LCCS Dynamic SOS Rules 54

5.1 Feature Summary from the Literature Survey 75
5.2 Core Semantics 87
5.3 Mobility Semantics 88
5.4 Structural Congruence Laws 88
5.5 Derivation of Nomadic Time from CaSE 89
5.6 Summary of Processes and Derived Syntax for Musical Chairs 93

6.1 Translation Schema from NT to DynamiTE `Process` Subclasses . . . 110
6.2 Translation Schema from NT to DynamiTE Classes 110

7.1 Typing Rules from (Sangiorgi, 2002) 143
7.2 Types 147
7.3 Timeout Types 147
7.4 The Bouncer Type 147
7.5 Mobility Types 148
7.6 Structural Congruence Laws with Types 149

Chapter 1

Introduction

1.1 Rationale

Recent changes in the direction of computer hardware development have created an impasse in the domain of software engineering. Over the past few years, new microprocessors have not seen the same increase in clock speed that has prevailed over previous decades. Instead, the use of multiple 'cores' has become common, due largely to physical limitations which prevent the individual elements of a single processor core becoming any smaller. As a result, the performance benefits of these new processors arise not from being able to execute a single task faster than before, but from the parallel execution of many such tasks.

However, this leads to a problem. The existing dominant methods for designing software systems are inherently sequential. Current imperative and object-oriented programming languages are still founded on the principles of early computational models, such as the Turing machine (Turing, 1936). These take an idealised view of events where they always occur sequentially and in isolation. Programs are thus still effectively written as a sequence of reads and writes to a form of memory. The problem with this approach is that it runs into major issues when the execution of other programs may cause changes to memory outside the remit of the program. Imagine Turing's model but with multiple heads, each running separate programs yet still sharing the same tape – what happens if more than one head writes to the same area of the tape?

In this thesis, we advocate a move towards systems where the focus is on interaction between minimal sequential subsystems. Rather than building huge monolithic structures, the same result can be achieved using a number of smaller components, running in parallel. Such a strategy has been suggested in varying forms over the years, but due to the perceived future evolution of the microprocessor, this is now an essential requirement, rather than a design ideal or optimisation. We also provide

a formal grounding for such designs, based on academic research which has been largely overlooked in the industrial sector. Security also forms an inherent part of both the design and formal model by allowing restrictions to be imposed on the communication between individual components.

In the remainder of this chapter, we provide a brief overview of the evolution of concurrent processing, highlighting current issues arising from the flawed approach of maintaining a sequential design which is becoming more and more distant from reality. We also look at how restricted mutability and an emphasis on intercommunication between smaller, more specific processes can provide a better solution, and how this approach has been adopted in the past with varying success. We close with a summary of the novelty of this work, and an overview of how this will be covered in the later chapters of this thesis.

1.2 Status Quo

1.2.1 Multiprogramming

Concurrency is nothing new. The concept of executing multiple programs at the same time has been in use since *multiprogramming* was first introduced back in the 1960s. But the same underlying model has remained. Parallelism is still seen as an optimisation, beholden to the maintenance of the sequential standard. Utilising concurrency within a program remains relegated to study as an advanced feature, seldom taught and even less often practised. If parallelism is to become the dominant means of exploiting the power of future hardware, this needs to change.

Multiprogramming was introduced as an efficiency measure. At the time, machines were available only on a per-institution rather than per-user level, so a batch of *jobs* were submitted to the machine, each consisting of the program to run and any associated data it needed to do so. The machine ran a relatively simple *operating system*, which would take each job in turn and execute a specified series of commands written in a batch job language. Such jobs would usually consist of reading in the program, compiling it if necessary, and then executing it. During execution, data was read in and the results of computation output for the user to digest later.

It soon become clear that having an expensive processor sit idle while input and output (I/O) operations took place was wasteful. To solve this problem, a new generation of machines were introduced which provided a *scheduler* as part of the operating system. Instead of running each job to completion before attempting the next, the system read in multiple jobs to begin with, each forming a *process* in memory. These processes consisted not only of the code to be executed, but also included contextual information, such as the next instruction to execute (the

program counter) and environmental data (e.g. open file handles).

If a process being run by the system reached a point where it had to wait for an I/O operation, the scheduler would move the process into a *blocked* state and perform a *context switch* to begin the execution of another. Once the I/O operation was complete, the blocked process would be reassigned to a *ready* state, making it again eligible for execution.

All this remained completely invisible to the running processes, each of which appeared to be running in complete isolation. The hardware provided memory protection, which prevented a process from accessing data outside its own memory space and they remained largely oblivious to the fact that their execution was effectively being paused and then resumed later. This was achieved by the scheduler storing the register contents and program counter at the continuation point, and then restoring these values before handing control back to the resumed process. The effect of such operation was only noticeable if the running time of the process was recorded, as such results were now dependent on factors such as system load and the arbitrary choices of the scheduler. This was a big change in design, which paved the way for the the introduction of objects and threads, and the same concepts are still used today.

Over time, schedulers have been extended so as to switch jobs when a quantum of time allocated to a process has been depleted. This ensures a greater degree of fairness; a processor-intensive task which rarely blocks can no longer become overly dominant. For batch systems, this wasn't of great importance (provided the process eventually terminated) as users submitted a job and then collected the results later on. In this context, just utilising the time when a process was blocked had a significant impact on perceived performance. However, with a move towards first time sharing and then personal computer systems, it became necessary to ensure that each process was given time to execute on a regular basis, so the system remained responsive. This concept is referred to as *preemption*.

Finally, further performance enhancements were made possible by allowing processes to have multiple threads of control and extending the scheduler to enable switching between the individual threads within a process. The advantage of such threading is that the threads share the same memory space and thus may interoperate more easily and more efficiently. The disadvantage is that it makes the possibility of *contention* much more likely.

1.2.2 Resource Contention

Concurrency issues arise when multiple processes or threads contend for access to the same resource. With threads, this is a frequent occurrence as they run the same code and access the same variables. It also occurs with processes; although they have their own memory space in which to operate, the resources provided by the

underlying operating system are shared by them all. An obvious example is the filesystem. What happens if more than one process tries to access a file at the same time? Unless only reads occur, the possibility of data corruption arises.

Such bugs, known as *race conditions*, are difficult to reproduce as they are heavily dependent on timing. This is especially true of single processor systems, where concurrency is merely simulated by the scheduler switching between processes. Whether or not file corruption occurs depends on the choices made by the scheduler, which in turn depend on a number of factors, such as system load. If many processes are competing for the processor, then there is more chance of one which accesses the same file being picked.

A print spooler is a program which allows a printer (another shared resource) to be used by many processes while maintaining separation between individual jobs. Without such a mechanism, one program may write a few lines to the printer, and then be suspended by the scheduler. The program which is allowed to run next may then also write to the printer, causing the user to end up with output from different jobs mixed together.

Instead, the spooler tackles this concurrency problem by acting as an mediator between the programs and the printer. However, such an application must be carefully designed to ensure it doesn't fall foul of the same issue. Imagine the spooler operates by reading a list of files to print from a shared file. When a process wants to add a new job to the queue, it writes the filename as a new entry at the end of the file:

```
int fd = open("/var/spool/print_jobs");
seek(fd, END_OF_FILE);
write(fd, "my_print_job");
close(fd);
```

Problems arise because such an operation is *non-atomic*; it is possible that the process may be stopped by the scheduler while adding a job to the list (e.g. after the `seek` function above), just as it may be stopped while writing to the printer. If this happens, there is a possibility that whichever other process is scheduled in its place could also choose to alter the queue. The result of such a collision depends on the timing:

1. If the first program only opened the file, or was just about to close it, then there will be no consequence. In the first case, the first program will move to the end of the (now longer) file when it resumes and write its entry. In the latter case, closing the file is just a matter of freeing resources and has no effect on the file itself.

2. If the first program seeked to the current end of the file, then on resumption, it will overwrite any data added in the meantime. If the new data is longer than the older data, then the old data will simply be lost. If it is shorter, the file will be corrupted.

The solution to these sort of problems is to limit access to a resource, so that a process is forced to wait its turn.

1.2.3 Semaphores and Monitors

Such access limitations can be imposed by a *semaphore*, a solution first proposed by Dijkstra (Dijkstra, 1968). A semaphore maintains an integer count which is manipulated by two operations: `up` and `down`. The count can be used to limit the number of threads of control active in a particular region. In effect, this is akin to the scenario where a gate requires a token in order to allow someone (a thread) to pass through, but the number of such tokens is limited. When a thread wants to pass through the gate, it attempts to acquire a token by executing the `down` operation. If the count maintained by the semaphore is greater than zero, then it will be decremented and the thread can proceed through the gate. However, if it is zero, there are no tokens left so the thread is forced to wait until one of the existing tokens is returned. Tokens are returned by executing the `up` operation.

The `up` and `down` operations must be atomic; it should not be possible for such an operation to be interrupted. If they can be, then the whole purpose of the semaphore is defeated; a further solution would be needed to resolve the possible concurrency issues that may occur inside the semaphore itself. Most operating systems provide such atomicity by using support available at the processor level; a Compare And Swap (CAS) operation updates a memory value only if the current value matches the one given as an argument (i.e. it hasn't been changed by another process or thread).

Binary semaphores, where the count is either zero or one, are very common. Such semaphores can be implemented in a simplified form known as a *mutex*, which maintains a binary state (locked/unlocked) rather than a count. Locking a mutex is equivalent to decrementing the count to zero via a `down` operation, and unlocking it is the same as performing an `up` to return its value to one. The usage pattern is the same for both: a thread first locks the mutex, does its work and then unlocks the mutex to allow others access.

Mutexes can also be implemented at the file level as file locks, providing a solution to the problem we encountered in the previous section:

```
int fd = open("/var/spool/print_jobs");
flock(fd, LOCK_EX);
```

```
seek(fd, END_OF_FILE);
write(fd, "my_print_job");
flock(fd, LOCK_UN);
close(fd);
```

The first call to `flock` acquires an exclusive lock (`LOCK_EX`) on the file referenced by `fd` (the file descriptor returned by the operation which opens the file). Let's assume that this process is stopped by the scheduler after the `seek` function executes and another process is allowed to run. This second process executes the same program. While it can successfully acquire a file descriptor for the file through the `open` function, the `flock` function will block trying to obtain an exclusive lock. This is because the lock is still held by the original process which has been descheduled but has not yet relinquished the lock. When the original process is chosen again by the scheduler, it can continue to write to the file, safe in the knowledge that no other process has altered its contents in the interim. The final call to `flock` releases the lock so the second process may now proceed.

Semaphores also have signalling capabilities; threads waiting to perform a `down` operation are woken when an `up` occurs on the same semaphore. They can then retry the `down` operation again and return, having decremented the value of the semaphore, should the operation succeed on this attempt. Given that there may be multiple waiting threads, there is no guarantee that a thread will become active; for each `up` operation, only one `down` operation will be successful and any other threads will again be forced to wait. Again, these race issues are why it is essential that the `up` and `down` operations themselves are atomic.

Suppose we want to implement a bounded buffer which is accessed by multiple threads. We need to use semaphores both to prevent possible race conditions when modifications are made to the buffer, and to stall threads when the buffer is full (in the case of adding a new item) or empty (when retrieving an item).

As in our previous example, a binary semaphore or mutex can be used to make modifications to the buffer appear atomic; a thread wanting to operate on the buffer needs to first acquire the token and will be unable to do so if another thread has already taken it. Semaphores can also be used to monitor the state of the buffer, and provide notifications to the producer and consumer threads when the buffer empties or fills up, respectively.

```
produce()
{
  item = produce_item();
  down(empty);
  down(mutex);
  add_item_to_buffer(item);
```

```
  up(mutex);
  up(full);
}

consume()
{
  down(full);
  down(mutex);
  item = remove_item_from_buffer();
  up(mutex);
  up(empty);
  consume_item(item);
}
```

The above example provides an example implementation of such a buffer, using three semaphores: `mutex`, `empty` and `full`. The `mutex` semaphore is a binary semaphore, which ensures a thread has exclusive access to the buffer by making modifications to the buffer appear atomic; although the thread can still be interrupted, any other threads trying to execute `down(mutex)` will be blocked until the original thread relinquishes control.

The other semaphores are used to maintain a count of how many empty or non-empty slots are available in the buffer. As the buffer is filled, the number of empty slots goes down and the number of non-empty slots goes up. The inverse is true when the buffer is emptied by the `consume` function. The `empty` mutex is initialised with a value equal to the size of the buffer, while the `full` mutex begins with a value of zero.

In the `produce` function, the thread first checks if there are any empty slots by performing a `down` operation on the `empty` mutex. If the `empty` semaphore has a non-zero value, as at the beginning, then there are available slots in the buffer and the operation will return after decrementing the value by one. In this case, the thread can then proceed to lock the buffer using the `mutex` and add an item to it. It then releases the `mutex` and performs an `up` operation on the `full` semaphore, increasing the number of slots in use and potentially allowing those threads waiting in the `consume` function to proceed. The `consume` function is effectively the inverse of the `produce` function; it checks the number of full slots to begin with, using the `full` semaphore, and increases the number of empty slots when done.

The examples above are fairly simple, but already demonstrate some of the problems inherent with the use of semaphores. A successful strategy for using them requires placing acquisition and release calls in all affected locations and is extremely prone to error. Suppose one of the processes above never relinquishes the lock on the file. Or a thread never performs an `up` on the mutex. Other threads or processes

wishing to acquire the lock or mutex will be blocked forever. Similarly, it takes only one miscreant to access the shared resource without attempting to acquire a lock to make the whole process of locking redundant.

Semaphores don't scale well either. For even a small program like the buffer example above, three semaphores are required. In such a situation, the order of acquisition also becomes important. If the order is wrong or differs between code segments, deadlock can occur. Deadlocks happen when each process or thread is waiting on a resource held by another waiting process. In the buffer example, simply altering the order of the `down` calls in the `consume` function is enough to create a potential deadlock situation. If a thread manages to acquire the mutex but is then forced to wait for an `up` on the `full` semaphore, no other thread will be able to acquire the mutex in the meantime. Only in the unlikely situation that a thread has been stopped between the `up(mutex)` and the `up(full)` calls in the `produce` function would this deadlock be resolved. In most cases, the other threads will attempt to acquire the mutex before reaching the required `up(full)` call and so are left waiting forever.

By far the biggest issue with these kind of problems is reproducibility. Just as with the race conditions they are trying to avoid, bugs relating to semaphores may not always manifest themselves. The example above is very likely to result in deadlock, as it just requires the `consume` function to be called when the buffer is empty and no other thread is accessing it. Other issues can be much harder to diagnose.

Take two processes, A and B, both of which are trying to acquire a lock on the two files, `/etc/passwd` and `/etc/shadow` in order to add a new user to the system. If both processes acquire the locks in the same order, then all is well. If they don't, a deadlock may occur.

Let's assume process A runs first. It acquires a lock on `/etc/passwd`. At this point, we assume A has used its allocated quantum of processor time and so is descheduled. A context switch occurs and process B begins to run. If B begins by trying to acquire a lock on `/etc/passwd`, then it will simply block as A already holds this lock. If, however, it tries to acquire a lock on `/etc/shadow` first, this will succeed. We then get stuck in a deadlock; B blocks trying to acquire the lock on `/etc/passwd` held by A, which will never be relinquished because A will be blocked trying to acquire the lock on `/etc/shadow` held by B. Such problems occur simply through an ordering mismatch, but can be extremely difficult to catch through simulation or execution; in many situations, the process will acquire both locks without being descheduled inbetween.

The solution to these problems is to abstract away from such intimate details and allow the programmer to work at a more amenable level. One attempt at doing so can be seen in the use of *monitors* (Hansen, 1973; Hoare, 1974). Rather

than worrying about the placement and sequencing of individual acquisition and release calls, the programmer simply denotes which sections of code must be run in mutual exclusion from one another. The compiler or virtual machine (depending on whether the code is pre-compiled or not) then handles the process of adding the required statements to ensure this. The concept of monitors is strongly linked to the idea of *objects*, with the same common idea of data encapsulation; all variables are private to the object and inaccessible from the outside. To read or modify the data held by a monitor, one of its methods must be called. Once a thread is running code in a particular method, no other thread may enter a method belonging to that monitor. This ensures the thread safety of the data without the issues of acquiring locks and lowers the potential for deadlocks.

While this provides a better alternative to the use of binary semaphores or mutexes, for a scenario such as the buffer example a notification mechanism is required so that threads can wait for a particular event to occur and be notified by other threads when it does. Monitors provide for this via the use of *condition variables* and the `wait` and `signal` primitives. Just as with semaphores, one thread calls the `wait` operation on a particular condition variable and then another thread calls `signal` on the same variable when the situation has changed. The problem with this approach is that it is just as prone to error as the use of semaphores; if the `wait` and `signal` primitives are not used appropriately, then threads may be stalled. It is still a very low-level solution.

Another issue with monitors, as implied above, is that they are heavily reliant on support from the programming language being used. While semaphores just require some means of performing an atomic change to an area of memory, monitors need the compiler or virtual machine to be intelligent enough to parse the monitor structures and convert them into appropriate uses of more low-level locking constructs. One language in which support is provided is Java, as can be seen in the example below:

```
public class Buffer
{

  public static final int BUFFER_SIZE = 5;

  private int used = 0;
  private Object buffer[BUFFER_SIZE];

  public void produce()
  {
    Object item = produceItem();
    synchronized
    {
```

```
      while (used == BUFFER_SIZE)
        wait();
      buffer[used] = item;
      ++used;
      notifyAll();
    }
  }

  public void consume()
  {
    Object item;
    synchronized
    {
      while (used == 0)
        wait();
      --used;
      item = buffer[used];
      notifyAll();
    }
    consumeItem(item);
  }
}
```

This is an implementation of the buffer example using monitors rather than semaphores. There are two main differences between the Java implementation of monitors and that proposed in the academic literature: the mutual exclusion is limited to blocks of code marked with the `synchronized` keyword, rather than encompassing the whole class, and the `wait` and `signal` operations are realised as the `wait` and `notifyAll` methods of the `Object` class rather than being functions applied to condition variables. One downside of these changes is that the addition of selective mutual exclusion makes it prone to error; although it is more efficient not to lock the entire class whenever any method is called, this also means that one may forget to use the `synchronized` keyword just as one may forget to perform the appropriate operation on a semaphore.

The similarities and differences between monitors and semaphores can be clearly seen by comparing the two buffer examples. In the Java version, the use of the `empty` and `full` semaphores is replaced by a while loop and the use of `wait()` and `notifyAll()`[1]. The value these depend on is also made explicit in this version (see the variable `used`), whereas it is an implicit part of the operations on the semaphores

[1]The use of the `while` loop, although slightly inefficient, is essential in ensuring that the condition is still false once the thread has awoken.

in the earlier example. When `produce` is called, it tests to see if the buffer is full (the `used` count is equal to the size of the buffer). If it is, then `wait` is called. The test takes place in a `while` loop rather than a single `if` statement so that the condition is tested again when the thread is awoken by the `notifyAll()` call. As before, if many threads are waiting, it may be the case that the buffer is already full again by the time a particular thread is allowed to execute.

The `synchronized` blocks behave in a way equivalent to those protected by the `mutex` semaphore; the opening bracket is the `down` operation, while the closing bracket is the `up`. Once a thread is executing code inside one of these blocks, no other thread may enter such a block, whether this be the same one or another in the same class. Modifications to the `buffer` and `item` variables only take place within these blocks, thus ensuring that only one thread can change things at a time. Both variables are marked `private`, making them invisible to code outside this class.

What is clear from our comparison is that there are few advantages to using monitors; they are prone to similar low level errors to those we saw with semaphores, and they also require support from the language being used, which may not always be available. Ideally, we instead need to take a step back and limit the need for such locks altogether by reducing the number of shared resources and the amount of mutability inherent in our designs. Not only are existing designs prone to error, but they also reduce the advantages of concurrent processing (having to acquire a lock effectively makes operations single-threaded once again) and are reliant on the existence of some form of shared memory. In distributed systems, shared resources do not exist naturally but must instead be created artificially and may make processing more inefficient. In the future, we want to be able to utilise the advantages of massively parallel systems and this can only be achieved by reducing the need for resource contention.

1.2.4 Interprocess Communication

To reduce resource contention, we need to focus on more short-lived processes which interact directly with one another, rather than via the means of shared resources. This is nothing new. However, it has never achieved universal acceptance as a design paradigm because having to deal with the kind of concurrency issues outlined above has traditionally been avoidable. This is no longer the case.

Although mainstream development has migrated from procedural programs to the *object-oriented* paradigm, programs, once compiled, still tend to be monolithic entities, with generally only a single thread of control. The notion of objects we see being used is not that of Simula (Dahl, Myhrhaug & Nygaard, 1968), where they are *task-centric* units with their own behaviour. Instead, it is one which is much more *data-centric*. These objects allow data to be separated out into neat little bundles and stimulate reuse by allowing hierarchies of derived behaviour to be created. But

there is no relationship between objects and threads; when a method of an object is called, control switches from one object to another. If multiple threads are in use, then the objects are shared between them and we see the kind of problems described above.

Solving this takes more than simply establishing a one-to-one relationship between threads and objects, because each unit is designed with a focus on the data being stored and not on the task being performed. Thus, for most designs, having an object per thread would be terribly inefficient and, in some cases, preposterous. For example, an implementation of a library system would have a `Borrower` object. A typical system may have thousands of such borrowers, many of which are inactive for weeks or months at a time. Having a thread for each would be ridiculously wasteful.

Instead, the solution is again to use objects which are task-centric. In the library example, the objects would focus on jobs such as issuing and returning books, and dependent tasks such as obtaining data on a borrower or book from the database. In either scenario, there will be contention for database access, but in the task-centric variant, an object can be given the job of a database guardian, centralising all data storage issues in one place.

There are many existing examples of this kind of *component* or *service*-based design, but they have so far failed to become the mainstream approach. One of the earliest is the notion of pipelines between processes, which originated from UNIX systems[2]. Early UNIX programs were developed with the aim of doing a single task and doing it well, unlike the feature bloat apparent in many of today's applications. For example, the command `du`, which calculates disk usage, doesn't include an option to sort the results. This is because there also exists a command, `sort` which can order an arbitrary block of text in a number of ways. As such, there is no point adding duplicate functionality to `du` when its output can just be fed in as input to `sort` for those who desire this feature.

A pipeline is created in the shell by separating the two programs with a `|` symbol. For our example, `du -h | sort -n` would do the job of outputting disk usage in human-readable form (`-h`) and then sorting it numerically (`-n`). A similar solution can be applied programatically using system calls such as `pipe`, `fork` and `execve`. The pipe allows the output of one program (`du`) to become the input of another (`sort`). Neither of the individual programs needs to be aware that this is happening. As far as `du` is concerned, it is still sending output on its standard output channel. The difference is that this channel has been changed externally so as to feed instead into a pipe, the other end of which forms `sort`'s standard input channel.

[2]Other systems have since adopted this technique, including those such as MS-DOS which are single-tasking and thus can not actually pass data between two processes. Instead, they make use of *pseudo-pipelines*, where the first program outputs data to a temporary file and the second then reads its input from that file.

This is a very simple solution, yet it elegantly solves the problem of sharing the data between the two processes. If a pipe was not used, `du` would have to store its results somewhere for `sort` to access. This could then result in contention between the two processes for access to the resource. Instead, here the two are working together rather than against each other by synchronising the passage of data between them. Each is independent of the other and specific to its purpose.

Microkernels such as Mach (Rashid, Julin, Orr, Sanzi, Baron, Forin, Golub & Jones, 1989), the GNU HURD (Bushnell, 1994) and MINIX 3 (Tanenbaum & Woodhull, 2006) also utilise this idea of synchronous communication rather than a monolithic design based around shared resources. In this context, it provides an essential stability and security advantage; many services, such as device drivers, file systems and network protocols, can operate at a similar level to user processes.

Some elements of the kernel require specialised operations which are only available when the processor is in a *privileged* mode of operation. However, these restrictions need not apply to the entire kernel. Device drivers are particularly notorious for causing system instability by having this level of control. This is especially true when such drivers are provided by third parties who are not as familiar with the operating system code as the core developers.

To combat this, in MINIX 3, device drivers operate as separate privileged processes. Unlike normal user-level processes, they have the ability to request direct access to hardware but such access is achieved by passing messages to a minimal kernel. The majority of the driver's operation takes place in userspace and any low-level access can be monitored and potentially prohibited. Other components can operate with even fewer privileges; file systems and network protocols need only the means to transfer a sequence of bytes to disc or down the wire.

The Mach kernel, developed at Carnegie Mellon University, takes a similar approach with the central mantra being one of multiple servers, which provide different operating system services. The GNU HURD kernel is currently based on Mach, though a number of more recent microkernels are now being considered, due to issues with Mach's design (Brinkmann & Walfield, 2007). Apple also adopted this design for XNU, the Mac OS X kernel, but, while basing it on Mach, they greatly reduced the design to a single server running a monolithic BSD-based kernel. MINIX 3, XNU and the HURD all try to implement a component-based design while retaining compatibility with existing monolithic UNIX systems, and so compromises have to be made. While Mac OS X is easily the most widely used of these examples, it has had to sacrifice the most to achieve this.

The traditional objection against such designs has been performance. Designs based on intercommunication have always tended to be more elegant, but their usage has tended to be restricted to distributed systems such as web services. In these circumstances, any design approach necessitates utilising a potentially slow

connection to another system, and having a central resource upon which all others rely becomes disadvantageous, due to the potential for failure. That said, the most popular web services in use today do not follow the component-based design that would allow the dream of composite web services (Norton, Foster & Hughes, 2005) to become a reality; the likes of *Facebook*, *Twitter* and *Last.fm* (Various, 2009a; Various, 2009b; Last.fm, 2009) all provide web service interfaces which simply wrap an earlier monolithic object-oriented design. Others, such as *Amazon* (Amazon, 2009), now focus on providing a utility service, offering processing power and storage for a price, while *Google* (Google, 2009) prefer to target users with complete applications.

We believe it is time to reëvaluate the benefits of systems focused on intercommunication between specialised components. With modern systems, the potential performance disadvantage is becoming outweighed by the benefits of a cleaner and more sustainable design. With the increasing prevalence of truly concurrent systems, monolithic designs will face a clear disadvantage, as the potential for parallelism is severely reduced by contention for shared resources.

1.3 Our Proposed Solution

There are already many examples of computational models which represent concurrent behaviour and its issues in the academic literature. We will cover some of these in depth in chapters 2, 3 and 4. However, these have been largely ignored by the software industry, as has one of their main uses; formal verification. This is primarily due to inertia; developers have little time to invest in learning new techniques and so stick to those they know and which have proved successful in the past.

Change does occur when there is little other sensible choice and it makes good business sense to do so. We have already seen this with object-oriented programming (OOP). It took about twenty years for OOP to become widely adopted from its initial inception in academia, and even then, as we discussed in 1.2.4, it was in a different form much closer to existing sequential models. The change happened as programs became larger and their design made them more and more unmaintainable, to the point where the cost of continuing to use existing models was more than adopting a different technique, in this case OOP.

We have reached an equivalent juncture now with relation to concurrency. Programs have continued to become larger and more bloated with features, but the increasing speed of microprocessors has allowed a state of equilibrium to be maintained. This is no longer so. Now, when users go out to purchase a new computer, they are likely to get one with twice the number of processors than the one they had before, rather than twice the speed. Because their programs will be largely monolithic, they won't see much of a performance increase in their new purchase; the same application will still be running on a single processor of about the same

speed.

We are not the only ones to observe this need to make concurrency more central to the design process. With the recent release of Mac OS 10.6 (*Snow Leopard*), Apple have introduced a new application programming interface (API) called *Grand Central* (Apple, 2009), which shifts the responsibility for managing threads to the operating system. Developers instead design their code as a series of tasks, which are submitted to the operating system through the API. They are then scheduled and later executed using a pool of threads; this allows threads to be reused and thus increases performance by reducing the amount of thread creation that takes place. A similar approach is available to Java developers, which we discuss in detail as part of 6.2.1.

Thus, software designers need to seriously start thinking about how they can best utilise this new hardware and this undoubtedly requires a shift in the underlying design. What we propose here is a compromise; we introduce a new framework, DynamiTE (see chapter 6) with a task-oriented design methodology, which retains as many familiar ideas as possible. Unlike efforts such as (Cardelli, 1995), (Turner, 1996), (Wojciechowski, 2000) and (Giannini, Sangiorgi & Valente, 2006), we avoid introducing a completely new programming language. Instead, we build on top of an existing one (Java) which is already familiar to many software developers and which uses constructs with which they are already familiar. In doing so, we remove a huge barrier to adoption; the implementation of the framework is no longer some mysterious mass of code written in an obscure functional language, but a Java library like any other which developers may even be able to contribute to with time. In this form, it still provides the advantage of abstracting away from many of the low-level details we saw in 1.2.3, while also being much more approachable.

We still follow these earlier examples in basing the framework on a theoretical model. This allows us to leverage years of academic work in this area, and allows for the possibility of reasoning over such programs in the future. However, we approach this from the point of view of a software developer wanting an implementation with the benefits of a theoretical basis, rather than as a process algebraist looking to write code in their favourite calculus.

To this end, we base our framework on our own calculus, which comprises what we believe to be some of the best of the existing ideas present in the literature. We believe our particular combination to be novel, as are the way in which some features are presented, in particular the notion of 'bouncers'; its formation and use is discussed at length in chapters 5 and 7. However, our primary aim is not to provide a vastly superior calculus, but one which best suits its position at the core of our framework.

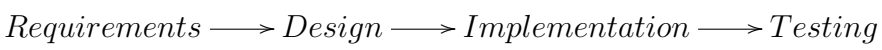


Figure 1.1: The Traditional Software Development Process

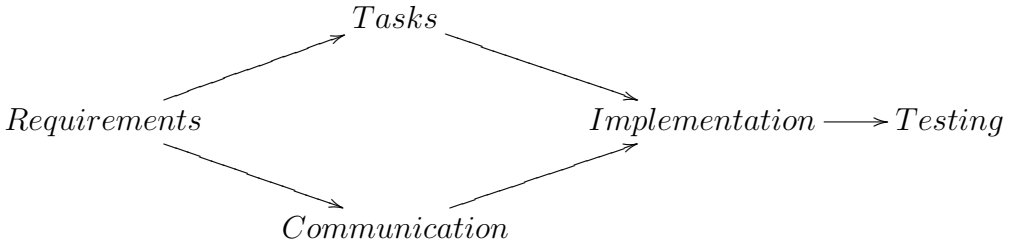


Figure 1.2: DynamiTE Variant of the Software Development Process

1.3.1 A Prototypical Application

The best way to demonstrate the use of a framework is through example. Hence, through the course of this thesis, we present a music player application and show how different elements of it may be developed using DynamiTE. We expect DynamiTE to be usable with most existing software development processes, such as the traditional one shown in Figure 1.1. The change we envisage is shown in Figure 1.2; the singular design process is split into two separate stages, *Tasks* and *Communication*. We expect each task to go through the design process independently; it should have a defined set of inputs and outputs and be verifiable without the others or the communication infrastructure being in place. Similarly, we expect the communication infrastructure to be developed and tested separately before being integrated with the tasks. As the design of the communication infrastructure is expressed in a process calculus, it should be possible to use simulation techniques to verify its behaviour and, in some cases, to prove properties such as the absence of deadlock, prior to it being used with the tasks.

This independence within the development process should mean that development itself can proceed in parallel. It is only at the final integration stage that the whole system needs to be tested. The independent development of each task also means that it should be more suitable for reuse in other systems as a result. For our music system, we elide the design of the individual tasks, and focus on just the design and implementation of the communication infrastructure as the part that's relevant here; these are covered in sections 5.7 and 6.6. At this juncture, we specify the requirements for it as follows:

- The application should provide some form of interface with which the user can interact.
- It should be able to take a wave file and return a sequence of sound data for playback.
- It should be able to output the sound data through the speakers.
- It should be able to generate a spectral analysis of the sound data as a form of visual feedback.

This is a minimal set, but is more than enough to demonstrate the process of building up an application. Further features could be added, such as playlists, more visualisations, support for further file formats, the use of tags and web services to provide song metadata, etc.

Central to designing an application with DynamiTE is keeping two things in mind; firstly, the application should be composed of components, each capable of performing their own task, and secondly, the application itself should be capable of being used as a component by others. The latter comes with the implicit assumption that the application's features are accessible by others, and that it remains relatively lightweight so as not to introduce unnecessary and burdensome requirements.

In an object-oriented application design, the focus would be on the data i.e. the songs being played. With a focus on function, we instead split the application up by task as follows:

- The `Inputter` receives a file name as input, and produces a stream of wave data from it as output.
- The `Outputter` receives a stream of wave data as input and produces output via the speakers.
- The `Visualiser` receives a stream of wave data as input and produces a graphical display as output.
- The `Interface` receives input from the user and uses this to provide input to and control the other components.

Figure 1.3 provides a diagrammatic illustration of how data flows between the various tasks. In later chapters, we will demonstrate how these components can be formally modelled using our process calculus and how they may be implemented using DynamiTE.

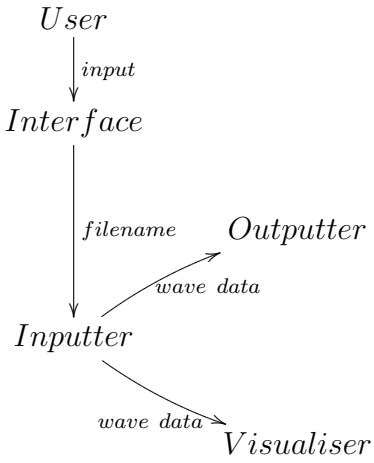


Figure 1.3: Structure of the Prototypical Application

1.4 Contributions to Knowledge

Through this thesis, we present the following contributions to knowledge which we believe to be novel:

C1. The development of Nomadic Time, an algebraic process calculus with *compositional global synchronisation*, *mobility* and security provision via the notion of '*bouncers*' (see chapter 5). This includes:

- **C1.1** The merging of clock hiding from the CaSE process calculus (Norton, Lüttgen & Mendler, 2003) with the notion of distribution, so that the boundaries of a locality (an *environ* in Nomadic Time) encapsulate the behaviour within them. This makes a locality effectively an opaque reusable component which can be integrated into other systems.
- **C1.2** The combining of this localised form of CaSE with structural mobility primitives from the ambient calculus (Cardelli & Gordon, 1998) to give a new mobile calculus with the local and global synchronisation properties of CaSE.
- **C1.3** The addition of a pair of process mobility primitives which allow direct process movement by synchronising on a name.
- **C1.4** The introduction of 'bouncers', which add a security mechanism to the calculus by taking the general idea of co-mobility primitives from the safe ambient calculus (Levi & Sangiorgi, 2003) and using them in a new specialised process, attached to individual environs.

C1.5 The creation of a set of structural congruence laws for Nomadic Time (Table 5.4), allowing process terms to be simplified and the number of semantic rules to be reduced.

C1.6 The provision of structured operational semantics for Nomadic Time (Tables 5.2 and 5.3). These extend those from CaSE as demonstrated in Table 5.5, extending the notion of prioritisation and adding new rules to handle the introduction of environs and mobility primitives.

C1.7 The demonstration of the properties of *prioritisation* and *time determinacy* inherent in the new calculus.

C2. The realisation of the aforementioned calculus as a *design framework*, DynamiTE, through the implementation of its constructs as programmatic elements in the Java programming language (see chapter 6). This allows the specification of system interactions to be shifted directly from the theoretical domain into an implementation backed by a formal methodology, with the intention of improving industrial adoption of concurrent techniques. This includes:

C2.1 The creation of a translation schema (Table 6.1), mapping process terms in Nomadic Time to Java objects.

C2.2 The implementation of the operational semantics as methods in the appropriate Java objects defined in the schema.

C2.3 The design and implementation of a plugin framework, allowing the use of different process calculi and different *side effects* as the result of performing a transition.

C2.4 The design and implementation of the *evolver* framework, allowing the execution semantics to be both clearly denoted and interchangeable.

C3. The optional addition of a *type system* to Nomadic Time in order to allow movement restriction to be based on the group membership of processes (see chapter 7); we refer to this extended version as Typed Nomadic Time (TNT). This includes:

C3.1 The design of a group type which can be applied to a Nomadic Time process to restrict movement.

C3.2 The provision of typing rules (Tables 7.2, 7.3, 7.4 and 7.5) for the new typed form of Nomadic Time.

C3.3 Proofs of type safety for the type system.

C3.4 The extension of DynamiTE to handle type systems in general, and specifically TNT.

This work has already produced two peer-reviewed papers (Hughes, 2006; Hughes, 2007) and several presentations, both internal and external (at the British Colloquium of Theoretical Computer Science (BCTCS) 2006, the Relational Methods in Computer Science (RelMiCS) PhD workshop 2006, the University of York and Principles and Practice of Programming in Java (PPPJ) 2007).

1.5 Structure of the Thesis

The first half of this thesis focuses on existing research in order to provide the necessary background material for the novel work presented in later chapters. Through this evaluation, we make clear the motivation for our work and also allow this thesis to remain relatively self-contained.

We begin the next chapter with a focus on the need for abstract models which can represent concurrent systems and the problems inherent therein, as covered earlier in this chapter. We then introduce existing research in this area in the form of algebraic process calculi and proceed with an exploration of the Calculus of Concurrent Systems (CCS) (Milner, 1989b) in 2.2. This includes, in 2.2.1, an example of how CCS can be used to model scenarios involving parallel behaviour, using Dijkstra's 'Dining Philosophers' problem (Dijkstra, 1971). We close the chapter with a critique of CCS (2.3), demonstrating how, while it provides a good foundation for concurrent modelling, it does have its limitations.

One specific limitation of CCS covered at the end of the second chapter is its inability to represent a *compositional* broadcast agent; by this, we mean one that can synchronise with an undefined number of recipients and then proceed to do something else. CCS can produce such agents that work with a known number of recipients, but the semantics of the agent have to be changed if the number of recipients changes. In chapter 3, we move onto the topic of *global synchronisation* (as opposed to local synchronisation between two parties), which allows this problem to be solved.

In 3.2, we introduce the Temporal Process Language (TPL) (Hennessy & Regan, 1995), which adds an abstract clock and a timeout operator to CCS. Together, these allow *prioritised choice*; the progress of time has a lower priority than the progress of work being done, so we can construct agents of the form 'do X until there is no more work to be done, then do Y'. Thus, we can design a compositional broadcast agent in TPL as 'broadcast until there are no more recipients, then do something else'.

The following section (3.3) briefly covers a number of calculi that extend TPL and provide the necessary stepping stones to the Calculus of Synchronous Encapsulation (Norton et al., 2003) covered in 3.4. CaSE retains the notion of a timeout operator (3.4.1), but incorporates multiple clocks and the notions of clock stopping (3.4.2)

and encapsulation (3.4.3). We provide examples and the operational semantics of the calculus in 3.4, and return to CaSE further in 5 as the basis for our calculus.

Chapter 4 covers a number of *mobile* calculi, where the context of a process can change during execution. There are two ways this can happen; via a change in scope (4.2) or a change in physical structure (4.3). The foremost example of scope mobility is the π calculus (Milner, 1999) which we explore in 4.2.1. This extends CCS so that the name of a channel may become known to a process during the course of execution, extending its scope. This is akin to passing a pointer or memory reference to a function in a programming language; the recipient then has knowledge of and can access something which it could not, prior to the call.

The following section (4.2.2) covers a number of variations of the π calculus, including its asynchronous and polyadic forms, and some theoretical results with regard to the expressivity and equivalence of its various forms. Another such extension is the Join calculus (Fournet & Gonthier, 1996), which is briefly covered in 4.2.2. Section 4.2.3 closes the discussion of scope mobility with a critique of the π calculus and its derivatives.

In 4.3, we cover the notion of *localisation*; the process of giving a physical position to a process. The initial discussion covers the use of localities within the area of equivalence theory (Boudol, Castellani, Hennessy & Kiehn, 1993), before we move onto the distributed form of the Join calculus (Fournet, Gonthier, Lévy, Maranget & Rémy, 1996) (4.3.1) and the foremost proponent of located mobility, the ambient calculus (Cardelli & Gordon, 1998) (4.3.2). We give examples of the various operators of the ambient calculus, before discussing the various extensions and derivatives (4.3.3), as we did with the π calculus. Of particular note are safe ambients (Levi & Sangiorgi, 2003), from which the form of mobility in our calculus, Nomadic Time, is most closely derived. Again, we close with a critique of the various ambient-based calculi in 4.3.4.

The final part of chapter 4 covers some more leftfield examples. In 4.3.5, we explore P Systems (Păun, 2002), a dominant modelling language in the field of biology which has some parallels with the ambient calculus previously discussed. We look more closely at the comparison between the two in 4.4 where we compare their different approaches to modelling biological scenarios. The final section (4.5) deals with bigraphs (Jensen & Milner, 2004), a framework proposed by Milner which aims to unify the various process calculi using a common representation.

The second part of the thesis introduces our own research and novel contributions. Chapter 5 introduces the Nomadic Time process calculus, a novel convergence of ideas from CCS, CaSE and the safe ambient calculus. We begin by localising the calculus (5.2), combining CaSE's notion of encapsulation (3.4.3) with the concept of localisation (4.3) to introduce *environs* and give our first contribution, **C1.1**. We then add a form of structural mobility to this localised form of CaSE using the

primitives of the ambient calculus (5.3.1); this is contribution **C1.2**. Contribution **C1.3** occurs in 5.3.2, where we add two additional mobility primitives that allow processes to be moved directly, without altering the hierarchical structure of the localities. These reuse the names introduced by CCS, with the new mobility primitives synchronising in the same manner as a co-name, but causing the process to move location. The final addition to the calculus comes in 5.4, where we take the idea of co-mobility primitives from the safe ambient calculus, but use them in a new special type of process we refer to as a 'bouncer' or guardian of an environ; this is contribution **C1.4**.

The following section 5.5 provides Nomadic Time with a set of structural congruence laws and a structured operational semantics; these form contributions **C1.5** and **C1.6**. We close this section by showing how the properties of prioritisation and time determinacy hold; this is contribution **C1.7**. We close the chapter with two sections which give examples of the calculus in use; 5.6 demonstrates the way compositional broadcast can be used with mobility through the example of a game of musical chairs, while 5.7 provides the next stage in the development of our prototypical application from 1.3.1 by providing the design of the communication infrastructure in Nomadic Time.

Chapter 6 covers the development of the DynamiTE (Dynamic Theory Execution) framework, beginning with an explanation (6.2) of why we chose Java for our initial implementation work. In 6.3, we then show how the process terms from Nomadic Time are mapped onto Java objects and how these objects are then implemented so as to return the possible transitions for each construct; these give contributions **C2.1** and **C2.2**. The next section, 6.4, describes the plugin framework which allows DynamiTE to be used with different process calculi, and allows the implementation of side effects, which may optionally be called when a transition is followed; this represents contribution **C2.3**. In 6.5, contribution **C2.4**, the evolver framework, is introduced, which allows the user to choose the execution semantics to apply to the system they have developed. Section 6.6 returns to the prototypical application with the implementation stage, showing how the design from 5.7 can be directly implemented in DynamiTE and how this is simpler and less error-prone than creating a bespoke communication structure for the application. We close the chapter in 6.7 with coverage of related work in the area of concurrent frameworks, including Obliq (Cardelli, 1995) (6.7.1), Nomadic Pict (Wojciechowski, 2000) (6.7.2), the safe ambients abstract machine (Giannini et al., 2006) (6.7.3) and JavaSeal (Vitek & Bryce, 2001) (6.7.4).

The following chapter (7) demonstrates how Nomadic Time may optionally be extended with a type system to create TNT (Typed Nomadic Time). We begin by looking at existing type systems. The first section, 7.2.1, covers type systems in the context of the π calculus, including 'sorts' which enforce the requirement that the

number of items being sent matches the number being received. We then turn to typing systems for the ambient calculus in 7.2.2 and cover the notion of groups. In 7.3, we take this idea of a group type and create our own for the Nomadic Time process calculus, which is then used to give typing rules for the new Typed Nomadic Time calculus; this forms contributions **C3.1** and **C3.2**. Contribution **C3.3** is found in appendices A and B in the form of proofs of type safety for TNT. Section 7.4 returns to DynamiTE and shows how it can be extended to perform type checking, using the example of supporting TNT; this is contribution **C3.4**. The chapter closes with an example use of the type system (7.5) in the context of the musical chairs example from 5.6.

Finally, chapter 8 summarises our contributions (8.1) and provides some suggestions for future work. A number of ideas are proposed for the further development of Nomadic Time (8.2.1), DynamiTE (8.2.2) and the type system (8.2.3), along with other possible applications for the calculus (8.2.4).

Chapter 2

Algebraic Process Calculi

2.1 Introduction

In this chapter, we focus on concurrency from a theoretical perspective and introduce one of the main algebraic models for modelling concurrent systems. The topics and ideas discussed here lay the foundations for the calculi we will discuss in chapters 3 and 4, and, as a result, are of great importance in understanding the novel work presented in chapters 5, 6 and 7.

Early computational models took a simple idealised view of the world, where events occur sequentially and in isolation. Such a model is the universal Turing machine (Turing, 1936) which has proven to be computationally complete; it is capable of simulating all recursive functions. However, it does not directly model concurrent execution.

If a model can have this level of computational power without attempting to represent concurrent behaviour, why is it necessary to model concurrency at all? Even though a method of modelling phenomena exists, and has a certain level of expressivity, it doesn't imply that it is the most appropriate for a particular context. The existence of both Turing machines and the λ calculus already demonstrates this point. While both have proven equivalent in power, they take different approaches to achieving this. However, neither model can represent the possibility of two or more events occurring at the same time, and thus can not be used to capture and evaluate the potential problems which may occur, such as the race conditions illustrated in the previous chapter.

To see the effect of concurrency on computation, consider a simple prototypical example, as demonstrated by Milner (Milner, 1993a). Observe the following programs,

```
x= 2;                                    (P1)
```

```
x= 1;
x= x + 1;                                (P2)
```

where we assume that each line is an atomic action[1].

In a sequential system, such as may be modelled by a Turing machine or the λ calculus, both these programs set `x` to 2. In such a system, there is only a single flow of control, so nothing else can modify the value of `x`.

However, in a concurrent system, multiple control flows or processes exist, each running in parallel with the others. With P1, the value of `x` will always be equal to two immediately after execution, as the assignment takes place within a single atomic action. However, in P2, another process is free to modify `x` (assuming `x` is globally accessible) between the assignment of the value 1 and the later summation which makes `x` equal to 2.

Thus, if P2 is run in parallel with a third program,

```
x = 3;                                   (P3)
```

then `x` may end up being either 2, 3 or 4, depending on whether P3 executes before the first line, after the completion of P2, or after the first line respectively. With P1 and P3, only 2 or 3 can result (which one depends on the order the two programs are run). Again, we have a *race condition*; the final value of `x` depends on the timing of the various modifications to its value performed by the two programs. As we saw in 1.2.3, the solution to this problem is to require each program to obtain exclusive access to `x` (a lock) for the extent of its use.

This example demonstrates that modelling concurrency is not so much about multiple programs executing at the same time, but instead concerns how they may interact. If each program exists in its own isolated environment and doesn't access any common resources, then no interactions will take place and a sequential model for each would be suitable. Indeed, this is the way most operating systems handle running multiple programs. Thus, it follows that sequential models are not distinct from concurrent models, but form a subset where this additional restriction of isolation applies.

The problem is that using sequential models to design and evaluate programs is becoming more and more detached from the reality of modern implementations. For example, many programs now include a graphical user interface, which must have

[1] This is a simplification; for example, `x = x + 1` actually involves three atomic actions – reading the value of x, computing the value of x plus one and writing the result to x

at least two concurrent threads of control to be operable; one is needed to await and handle any user interaction, while the other actually executes the operations of the program, even if that simply involves updating the display while idle. As multi-core processors make more machines capable of *true concurrency*[2] and distributed computing paradigms, such as services, become more prevalent, the need to accurately model the possible interactions increases. Concurrency raises issues outside the reach of traditional sequential models of computation, so to adequately work with concurrent systems, we need appropriate formal models to highlight potential flaws and to allow us to account for any issues that may arise in the design of the program. Many such models have been developed, and over the next three chapters, we will consider a subset of these.

2.2 The Calculus of Communicating Systems

Algebraic process calculi model the interaction of concurrent processes using a (usually small) set of algebraic operators, as opposed to the true concurrency of Mazurkiewicz trace theory (Mazurkiewicz, 1977) or the graphical style associated with Petri nets (Petri, 1962) and Hewitt's Actor model (Hewitt, Bishop & Steiger, 1973). Interaction between processes is via message-passing, rather than via a common shared memory[3] or a tuple space (Carriero & Gelernter, 1989).

The foundational calculi in this field are Hoare's Communicating Sequential Processes (CSP) (Hoare, 1978), Milner's Calculus of Communicating Systems (CCS) (Milner, 1989b) and Bergstra and Klop's Algebra of Communicating Processes (ACP) (Bergstra & Klop, 1984), all of which were first developed in the late 1970s to early 1980s. CSP was originally developed as a programming language with a relatively large syntax, while Milner aimed for a minimal calculus. Both calculi have influenced each other, with CSP later being refined and given a theoretical basis, following Milner's work. ACP shares many of the ideas of CCS, and can be regarded as an 'alternative formulation' (Bergstra & Klop, 1984), using a similar set of operators to achieve a different goal.

In our work, the focus is on CCS, as it forms the basis for most of the other calculi considered, including the π calculus (Milner, 1999) (see 4.2.1) and CaSE (Norton et al., 2003) (see 3.4). Of the three foundational calculi, CCS has the smallest syntax with additional features such as failure (represented in both CSP and ACP) needing to be derived from or appended to this core set. From a theoretical perspective,

[2] In truly concurrent systems, operations are actually performed simultaneously, rather than this being emulated by the system scheduler.

[3] Shared memory and message-passing are not orthogonal; a shared memory space may be represented as a communicating resource in a message-passing system, while message queues can be implemented using shared memory.

this is advantageous, as it makes reasoning over the calculus simpler, and, as will be seen in later chapters, further syntax can be added to represent further features if necessary.

In CCS, processes can be modelled as terms ranged over by E, F. These process terms have the following syntax:

$$E, F \ ::= \ 0 \mid \alpha.E \mid E \backslash\, a \mid E \ + \ F \mid (E \mid F) \mid X \mid \mu X.E \mid E[f] \qquad (2.1)$$

where α, a and f are explained below.

External behaviour is described using members of $\mathcal{N}$, an infinite set of names, and $\overline{\mathcal{N}}$, the corresponding set of co-names $\{\overline{a} \mid a \in \mathcal{N}\}$. These names are usually used to represent *channels* upon which the processes communicate. The internal behaviour of the processes is abstracted, represented simply by the silent action τ. Under such an interpretation, $a.E$ (where $a \in \mathcal{N}$) represents a process whose first action is an input on the channel a, whereas $\overline{a}.E$ (where $\overline{a} \in \overline{\mathcal{N}}$) represents a process which initially outputs on a. The behaviour of a process is thus described in terms of atomic actions. This can be seen in the first two cases above, where 0 represents the empty process (which exhibits no behaviour) and $\alpha.E$ represents action prefix (used for the limited sequential composition of actions), where $\alpha \in \mathcal{A} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$.

Thus, a basic process is defined as a sequence of inputs, outputs and silent actions. Each step in the sequence is a state from which the process may *perform* an action in order to transition to a new state. For example, $a.E$ may perform the action a and become E. We represent this using the notation $a.E \xrightarrow{a} E$.

For two processes to communicate with one another, they must *synchronise*; they must perform a corresponding pair of actions (e.g. a and $\overline{a}$) at the same time. For this to occur, the two processes must be running in parallel. Parallel composition in CCS is represented by the $\mid$ operator. When two processes are composed in this way, they may perform their corresponding input and output actions simultaneously, resulting in a single τ transition which changes the state of both processes.

For instance, if E is considered to be $a.E'$ and F to be $\overline{a}.F'$, then the process formed by the composition of these two processes, $E|F$ may initially perform one of three actions, a, $\overline{a}$ or τ, to give three possible derivations:

1. $E \mid F \xrightarrow{a} E'|F$
2. $E \mid F \xrightarrow{\overline{a}} E|F'$
3. $E \mid F \xrightarrow{\tau} E'|F'$

This is illustrated in Fig. 2.1. To make the derivation of $E|F$ deterministic, the scope of a can be restricted. In CCS, an action or co-action can be paired with any complementary action which is within its scope. To force the input of E to be paired

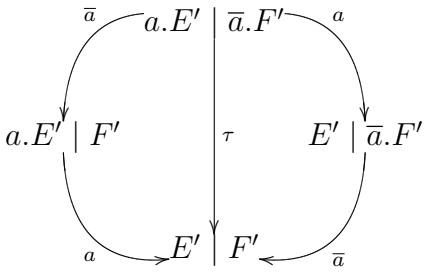


Figure 2.1: Graph of $a.0 \mid \overline{a}.0$

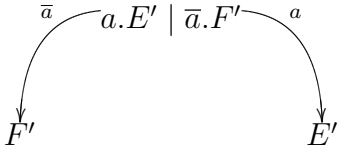


Figure 2.2: Graph of $a.0 + \overline{a}.0$

with the output of F above, the scope of a must be restricted so as to include only E and F. This is handled by another operator in the core syntax, $\backslash$. Its operand a is the name of a channel whose scope is restricted to the process given as its left operand. So, in this case, $(E|F)\backslash a$ appropriately limits the possible derivations to just $\xrightarrow{\tau}$.

The remaining binary operator is $+$, which provides non-deterministic choice between two processes. While the parallel composition operator represents two processes running in parallel, $+$ corresponds to the familiar idea of branching found in sequential models. E and F thus represent two possible behaviours which may or may not occur. Using the same two exemplar processes again, $E + F$ may derive as follows:

1. $E + F \xrightarrow{a} E'$
2. $E + F \xrightarrow{\overline{a}} F'$

Again, this is illustrated in Fig. 2.2. There are clearly similarities between the possible derivations from $E|F$ and $E + F$, but with choice, there is no possibility of synchronisation and only one of the two transitions, a and $\overline{a}$ is ever performed. The other is lost after the process makes its decision, whereas with composition, it is possible to perform both actions, one after the other.

Table 2.1: CCS Semantics

$$\text{Act}\ \frac{-}{\alpha.E \xrightarrow{\alpha} E} \qquad \text{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'}$$

$$\text{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'} \qquad \text{Par1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

$$\text{Par2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \text{Par3}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{Rec}\ \frac{E \xrightarrow{\alpha} E'}{\mu X.E \xrightarrow{\alpha} E'\{\mu X.E/X\}} \qquad \text{Res}\ \frac{E \xrightarrow{\alpha} E'}{E \setminus b \xrightarrow{\alpha} E' \setminus b}\ \alpha \neq b$$

$$\text{Ren1}\ \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]} \qquad \text{Ren2}\ \frac{E \xrightarrow{\overline{a}} E'}{E[f] \xrightarrow{\overline{f(a)}} E'[f]}$$

The remaining operators in CCS handle recursion and relabelling. The process $\mu X.E$ binds X to E, so that later occurrences of X are replaced with E. For example, $\mu X.a.X$ can perform an a transition to become $\mu X.a.X$ again. The function, f, in $E[f]$ has the type $\mathcal{N} \to \mathcal{N}$ and is used to rename actions and their complements. For example, $a.\overline{a}.\tau.0[a \to b]$ is $b.\overline{b}.\tau.0$.

An operational semantics for CCS can be given in terms of a labelled transition system, $(\mathcal{P}, \mathcal{A}, \to)$, where $\mathcal{P}$ is the set of CCS expressions formed from the above syntax, $\mathcal{A}$ is as defined above and $\to \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ is the transition relation defined in Table 2.1. We use E and F to range over process terms ($\mathcal{P}$), α over the set of actions ($\mathcal{A}$), σ over the set of clocks ($\mathcal{T}$), a and b over the set of names ($\mathcal{N}$) and and γ over $\mathcal{N} \cup \mathcal{T}$.

2.2.1 The Dining Philosophers

To fully appreciate CCS, it is necessary to see how it may be used to model an example scenario.

Dijkstra's classic 'Dining Philosophers' problem (Dijkstra, 1971) illustrates further issues which may arise in a situation where multiple processes must interact to achieve their goal. In this scenario, five philosophers are seated around a table, each with a plate of spaghetti and a fork. The philosophers divide their time between thinking and eating. In order to eat, a philosopher must obtain two forks, necessitating some form of interaction. This is a common situation in concurrency,

where multiple parallel processes (the philosophers) need to gain access to shared resources (the forks).

In cases where things go awry, deadlock or starvation may result. For example, if all the philosophers simultaneously pick up the forks on their left, then none of them will be able to eat; they will all end up waiting for a fork held by another philosopher. The system is said to be *deadlocked*, as none of the processes can obtain a lock on the resource it needs, as a lock is already held by one of the other processes[4]. Alternatively, *starvation* may result (literally in this case) if one of the philosophers never stops eating and consequently never releases the forks; the resources are unfairly distributed to the deficit of one of the processes.

Modelling this in CCS involves first ascertaining which processes form the basis of the system. Clearly, each philosopher plays a part, so they should be represented by processes. Returning to the original definition of the problem, each philosopher may choose to eat or think. In CCS, this can be represented as:

$$Philosopher = Eating + Thinking \tag{2.2}$$

where the philosopher is recursively defined as making the choice between $Eating$ or $Thinking$. Defining the latter is simple; thinking is simply some internal process of the philosopher:

$$Thinking = \tau.Philosopher \tag{2.3}$$

The focus of the model is on the eating process, which requires access to the system's shared resources: the forks. Modelling this necessitates defining a protocol whereby the philosopher may interact with the resource in order to obtain access to it.

$$Eating = \overline{take}.\overline{take}.\tau.\overline{replace}.\overline{replace}.Philosopher \tag{2.4}$$

which needs to synchronise with two available forks if the philosopher is to be able to eat (represented by τ) and then replace the forks again. It follows that the forks must also be represented using the process

$$Fork = \mu X.take.Fork' \tag{2.5}$$

with two communication channels, *take* and *replace*. The fork begins its life on the table from which it may be *taken*, represented here by the receipt of an input on the *take* channel. Once this has occurred, the process becomes

[4]The solution to breaking this deadlock is to break the symmetry; if the fifth philosopher tries to take the fork on the right first, he or she will be unable to proceed, but the first philosopher will, using the fifth philosopher's left fork.

$$Fork' = replace.X \tag{2.6}$$

which represents the state where the fork is in use by a philosopher. The fork can't be used again until it has received an input on *replace*, which causes X to be expanded and the fork to wait for input on *take* again.

The system as a whole is modelled by running a number of philosophers and forks in parallel, and restricting the scope of the fork channels in order to enforce synchronisation.

Note that this CCS representation of the problem only models the narrative version of the problem above. There is no attempt to resolve any of the competition problems, and a strong element of non-determinism, as to which philosopher gets which fork, still exists. It does, however, give a formal representation of the problem and allows the effects of varying the relative numbers of philosophers and forks to be observed via simulation.

Modifying this slightly gives a model that corresponds exactly to a specified number of philosophers and forks, n. From the definitions above, multiple variants may be generated, such that each philosopher and fork process has a unique subscript. For example, *Philosopher* becomes $Philosopher_i$, where $i = 1 \ldots n$. The same subscripting also applies to the *take* and *replace* channels, so that they now correspond to a specific fork. The original solution can thus be represented, as the case where each $Philosopher_i$ initially performs the action $take_i$ (to take the left fork) and then $take_{i-1}$ (with the exception that when $i - 1 = 0$, we use n)[5].

This model restricts which fork is taken by which philosopher (limiting the possible actions, and thus removing some non-determinism), but is still prone to questions of non-deterministic choice (some philosophers may arbitrarily choose to think instead) and fairness, with regards to action performance (if the actions are performed in a depth-first manner[6], only one philosopher may end up eating). These may be regarded as implementational aspects of the model.

2.3 Advantages and Limitations of CCS

From its syntax, it is clear that CCS can model sequential behaviour using sequential composition ($\alpha.E$), non-deterministic choice ($+$) and $\mathbf{0}$. This further confirms the intuition noted earlier that sequential programs are a subset of the larger set of concurrent programs. This is illustrated by the $+$ operator, which returns a smaller set of possible derivations, from the same initial pair of processes, when compared

[5]Again, it is necessary to reverse the actions of $Philosopher_n$ in order to obtain a solution that does not deadlock.

[6]i.e. if an implementation always chooses to execute a particular philosopher's choices first.

with parallel composition (|). These sequential operators can also be used to convert a set of parallel-composed processes into their equivalent interleavings.

CCS can model both sequential and concurrent programs, while still maintaining a minimal syntax. A finite axiomatisation can even be defined, if the simultaneous presence of parallel composition and recursion is avoided (Milner, 1989a). However, one fairly obvious limitation is that there is no data in the model. The processes discussed so far don't explicitly communicate anything when they send or receive signals. Instead, behaviour arises purely from synchronisation. It is possible to extend CCS to represent this by adding the concept of value passing between processes. A host of other process calculi have been based on such a variant of CCS, and we will consider this in more detail as part of chapter 4.

CCS models are also relatively static; while processes may evolve (e.g. $a.P$ may become P), the communication structure doesn't. Notably, if a process, E, knows about the channels x and y initially, while F only knows about x (due to restriction on y), this status can not change during the course of the various transitions inherent in the system. The effect of restriction is more generally known as *scoping* and occurs frequently with reference to variables in programming languages. CCS doesn't allow dynamic changes to the scoping of channels. Instead, scoping is fixed to the static arrangement provided by the initial system, prior to any transitions. The addition of dynamic scoping, often referred to as mobility, is the major contribution of the π calculus, a language based on CCS covered in 4.2.

To conclude, there is another limitation of CCS which is less to do with a particular concept being absent from the language, instead being more related to its central aspect: **synchronisation**. The problem here lies in the *compositionality* of processes. While the structure of a CCS system remains compositional, because the result of parallel composition is determined by the behaviour of the composed processes together with the rules of the | operator, this is not true of the synchronisation of arbitrarily many processes.

Consider broadcasting a signal to an arbitrary number of processes. Ideally, a general *broadcast agent* should be defined which provides this behaviour. In CCS, there are at least two ways of defining semantics for the agent, but not one that provides a suitably compositional solution. Perhaps the most obvious is simply to extend the familiar synchronisation of two processes. An input and output pair can synchronise, so why not just create multiple pairs, one for each receiving process? For example, transmitting a signal to two processes can be written simply as

$$\overline{\mathbf{o}}.\overline{\mathbf{o}}.\mathbf{0} \mid o.P \mid o.Q \tag{2.7}$$

where the process on the left (in bold) forms the semantics for the broadcast agent and the processes, P and Q, are the continuations of the input processes

This will work, but what happens when the broadcast agent needs to transmit

the signal to three processes?

$$\overline{o}.\overline{o}.\overline{o}.\mathbf{0} \mid o.P \mid o.Q \mid o.R \tag{2.8}$$

The semantics of the broadcast agent have to change. Simply composing the third input will lead to one of the three being ignored by the original definition of the broadcaster given above. So, simply enumerating multiple synchronisation pairs is not sufficient to provide a compositional broadcast agent.

A second solution lies in recursion. If the problem with the previous solution lies in the broadcasting agent doing too little (i.e. not transmitting to all the possible receivers), then, by making it recurse, it will keep sending the output to whoever will synchronise with it. Thus, the example for three inputs above becomes

$$\mu\mathbf{X}.\overline{o}.\mathbf{X} \mid o.P \mid o.Q \mid o.R \tag{2.9}$$

which works, and will continue to do so if a further input process is parallel composed.

But there is still a problem for much the same reasons as the first solution. This works fine on this small scale, but what happens when this agent is placed in the context of a larger system? Once the agent starts its cycle of outputs, it won't stop as there exists no base case for this recursion[7]. An output on o will always be available (within the scope of any restriction placed on that particular channel) and the broadcasting process can never do anything else. The result is a constantly cycling process, which, in an implementation of this model, would continue to consume resources.

The true solution to this problem is to enable some form of *global synchronisation*. This requires a separate entity, distinct from the processes involved in the communication, which can be used to co-ordinate the synchronisation. In the next chapter, a branch of process calculi is considered which provides just such a facility.

2.4 Conclusion

In conclusion, this chapter has taken a brief look at the field of concurrency modelling, largely from the perspective of process calculi. Initially, it was shown that, while universal Turing machines and the λ calculus can simulate any recursive function, their inherent sequential behaviour makes them unsuitable for modelling concurrent systems. CCS, in contrast, can model this kind of behaviour and in a succinct manner. However, its minimal syntax also leads to some limitations. In the next two chapters, we will look at some more process calculi, many of which use CCS as

[7] A base case may be introduced using non-deterministic choice, but there is no guarantee when this will be invoked, if ever.

their basis, and observe the benefits of features such as global synchronisation (see chapter 3) and mobility (see chapter 4).

Chapter 3

Global Synchronisation

3.1 Introduction

In this chapter, we cover a number of process calculi that incorporate abstract time. The notion of 'time' is generally associated with concrete real values, in units such as minutes and seconds. Real-time process calculi, such as those described in (Moller & Tofts, 1989; Satoh & Tokoro, 1993; Aceto & Murphy, 1996; Satoh, 1996; Beaten & Middelburg, 2001; Lee & Zic, 2002; Lee, Philippou & Sogolsky, 2005), attempt to model this. Instead, this section focuses on calculi that use *clocks* to provide global synchronisation, as introduced in 2.3. While *local synchronisation* occurs between two processes, the global form synchronises any number of processes using a clock signal. For this, we don't need to measure time itself; we just need an external reference point that can be used to co-ordinate events.

Our central focus here is on CaSE (3.4) which we extend to create our own calculus in chapter 5. However, we first turn our attention to its predecessor, TPL.

3.2 Temporal Process Language (TPL)

Hennessy's Temporal Process Language (TPL) (Hennessy & Regan, 1995) extends the CCS language discussed in 2.2 with a single clock, akin to a hardware clock which emits a signal at an arbitrary point in time. These signal emissions are controlled by a concept known as *maximal progress*, which allows each process to make as much progress as possible before the clock ticks. Formally, this means that all silent actions (τs) are performed before a σ action (which represents the clock signal) occurs.

This is of little use unless the actions of the processes can actually depend on the behaviour of the clock. The two are related via the addition of a *timeout* operator. This takes the form

$$\lfloor E \rfloor (F) \tag{3.1}$$

where E and F are processes. A process of this form can either follow a transition from the process E, leaving only the continuation of E, or perform a σ transition to become F. In short, F acts if E *times out* on the clock, σ. This is similar to non-deterministic choice, in that we lose either E or F as the result of performing a transition.

Here, however, the choice is determined by the clock, and thus effectively by the other processes, as it is their behaviour which controls when the clock will tick. If E can perform a silent action (τ), then the clock σ will be prevented from ticking by maximal progress, forcing one of E's transitions to be taken. For example, $\lfloor \tau.E' \rfloor (F)$ has only one transition $\xrightarrow{\tau}$ which leads to the state E'. If, however, we replace $\tau.E'$ with $\mathbf{0}$ to give $\lfloor \mathbf{0} \rfloor (F)$, we have two possibilities:

1. $\lfloor \mathbf{0} \rfloor (F) \xrightarrow{\sigma} \mathbf{0}$

2. $\lfloor \mathbf{0} \rfloor (F) \xrightarrow{\sigma} F$

as both $\mathbf{0}$ and the timeout produce σ transitions. Thus, what we have is *prioritised choice*; two alternatives are offered as with $+$ but the higher priority of τ transitions over σ transitions is instrumental in the choice of which transition to follow to the next state.

With these additions, the problem of defining a suitable compositional broadcast agent, as mentioned in 2.3, can be solved. Recall the second solution, which used recursion. Now, with the addition of an external entity (the clock) and a way of relating it to the processes involved (timeouts), a base case may be provided via recognition of the point when no more synchronisations may occur. This can be added to the earlier recursive solution

$$\mu \mathbf{X}.\lfloor \overline{\mathbf{o}}.\mathbf{X} \rfloor \sigma(\mathbf{0}) \mid o.P \mid o.Q \mid o.R \tag{3.2}$$

by simply adding a timeout which stops the recursion. This works because the synchronisations of the input processes with the output of the broadcast agent generate silent actions and thus invoke maximal progress. While there is a choice between a silent action (due to the broadcasting agent synchronising with an input) and a clock tick, the silent action always takes precedence and thus every possible synchronisation occurs. Once no more synchronisations are possible, the clock is allowed to tick and the recursion stops.

3.3 Extending TPL

The extensions to TPL considered here focus on expanding the scalability of the language. As demonstrated above, TPL adequately provides for situations where an arbitrary number of processes must synchronise. But what happens when a solution, like the one above, is integrated into a larger system? With only one clock, further problems occur. The use of the clock in one subsystem may conflict with its use in another, and there is no clock available to co-ordinate the subsystems themselves.

The Calculus for Synchrony and Asynchrony (CSA) (Cleaveland, Lüttgen & Mendler, 1997) extends TPL with the idea of multiple clocks, drawn from PMC[1] (Andersen & Mendler, 1994). However, while having multiple clocks allows the use of differing patterns of synchronisation, it increases the number of clock ticks present within the system. With five clocks, even the nil process has five possible transitions (as clocks idle over nil).

CSA solves this to a limited extent by localising maximal progress to a predefined scope for each clock. A more elegant solution is provided in the Calculus for Synchrony and Encapsulation (CaSE) (Norton, Lüttgen & Mendler, 2003), which introduces a clock hiding operator into the syntax. The effect of this is the introduction of *synchronous encapsulation*, as hidden clocks emit τ actions (as opposed to ticks) outside the operator's scope. This can be used, in conjunction with restriction, to produce a hierarchy of components. The actions of these subsystems can be represented purely as silent actions, and, when combined with the global form of maximal progress introduced by TPL and retained in CaSE, integrated into the 'synchronous cycle' (Norton et al., 2003) of clocks at the level above.

3.4 The Calculus of Synchronous Encapsulation

The syntax for CaSE, given in (Norton, 2005), is as follows:

$$\begin{array}{c} \mathcal{E}, \mathcal{F} \ ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid \mathcal{E} \,|\, \mathcal{F} \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid \mathcal{E}/\sigma \end{array} \tag{3.3}$$

where $\mathcal{E}$ and $\mathcal{F}$ define possible process terms. We assume a countable set of actions, $\mathcal{A} = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, ranged over by α, where the elements of $\mathcal{N}$ are drawn from an infinite set of *names*, and $\overline{\mathcal{N}}$ is the corresponding set of *co-names*, $\{\overline{a} \mid a \in \mathcal{N}\}$. $\mathcal{T}$ is a countably infinite set of *clocks* over which σ ranges. X ranges over a countably infinite set of variables, which are used to bind process behaviour in recursive process

[1] PMC also differs from TPL in its use of *insistent* actions; all must be performed before a clock tick.

definitions. $\mathbf{0}$, $\alpha.\mathcal{E}$, $\mathcal{E}+\mathcal{F}$, $\mathcal{E}\,|\,\mathcal{F}$, $\mu X.\mathcal{E}$, X and $\mathcal{E}\setminus a$ retain their behaviour defined in CCS, but now exhibit additional actions due to the presence of clocks.

There are now transitions for the $\mathbf{0}$ process, as, while the process has no explicit behaviour, it can idle over the ticks of the clocks. This also applies to actions in general:

$$a.0 \xrightarrow{\sigma} a.0 \tag{3.4}$$

assuming a clock context containing just the one clock, σ. Similarly, parallel composition and non-deterministic choice exist through time, so both sides can evolve due to a clock tick, while the operator remains in place. This gives the following possible derivations for $a.0 \mid b.0$ (where $b \neq \overline{a}$):

1. $a.0 \mid b.0 \xrightarrow{a} 0 \mid b.0$
2. $a.0 \mid b.0 \xrightarrow{b} a.0 \mid 0$
3. $a.0 \mid b.0 \xrightarrow{\sigma} a.0 \mid b.0$

with the same clock context as above. The third derivation is duplicated for each available clock that can tick over both sides of the composition. In cases where both sides may synchronise, causing a τ transition, this takes precedence over the clock transitions, due to *maximal progress* (see 3.1) and the original set of derivations for parallel composition (see 2.2) are available instead.

The changes to non-deterministic choice are simpler, as the operator itself does not generate silent actions. So, if both sides allow the clock to tick, then the following derivations will occur:

1. $a.0 \;+\; b.0 \xrightarrow{a} 0$
2. $a.0 \;+\; b.0 \xrightarrow{b} 0$
3. $a.0 \;+\; b.0 \xrightarrow{\sigma} a.0 \;+\; b.0$

again with the single clock, σ, as the context.

3.4.1 Timeouts

Moving on to the new operators, CaSE, as presented in (Norton, 2005), includes two variants of the timeout operator, first seen in TPL. Recall from 3.1 that the operator essentially allows a decision to be made, based on the presence of a clock tick. In the general scenario,

$$\lfloor E \rfloor \sigma(F) \tag{3.5}$$

F will act if E fails to, prior to a clock tick. If E can perform a τ action, then this will prevent the clock tick and E will evolve. Both operators in CaSE maintain this core behaviour, which is central to the concept of global synchronisation explained earlier.

The difference between the two operators in CaSE lies in their behaviour with regard to other clocks. With the fragile timeout, $\lfloor E \rfloor \sigma(F)$, any possible transition on E will cause the removal of the timeout. So, with $\lfloor a.0 \rfloor \sigma(b.0)$ and a clock context of σ and ρ, the following derivations can occur:

1. $\lfloor a.0 \rfloor \sigma(b.0) \xrightarrow{a} 0$
2. $\lfloor a.0 \rfloor \sigma(b.0) \xrightarrow{\sigma} b.0$
3. $\lfloor a.0 \rfloor \sigma(b.0) \xrightarrow{\rho} a.0$

where both the a and the ρ transition leave only the left-hand side of the timeout.

The stable timeout differs by continuing to exist through time until some action occurs. While it exhibits the same behaviour in response to actions or the tick of the specified clock, the ticks of other clocks only cause the left-hand side to evolve; the timeout itself is retained. Thus, $\lceil a.0 \rceil \sigma(b.0)$ gives a different set of derivations:

1. $\lceil a.0 \rceil \sigma(b.0) \xrightarrow{a} 0$
2. $\lceil a.0 \rceil \sigma(b.0) \xrightarrow{\sigma} b.0$
3. $\lceil a.0 \rceil \sigma(b.0) \xrightarrow{\rho} \lceil a.0 \rceil \sigma(b.0)$

where the ρ transition no longer causes the dissolution of the timeout.

3.4.2 Clock Stopping and Insistency

The remaining operators further control the behaviour of the clocks. Δ prevents all clocks from ticking, while Δ_σ prevents only the ticks of the specified clock, σ. Δ is similar to the CCS version of $\mathbf{0}$, as it has no possible transitions. Δ_σ exhibits transitions for all other clocks within the current context. So, for a context containing both σ and ρ, Δ_σ has a single transition,

$$\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma \tag{3.6}$$

which is replicated for any other clocks in the context, which are not equal to σ.

The stopping of clocks is used to provide *insistency*. Normally, a process $a.P$ has two possible derivations:

1. $a.P \xrightarrow{a} P$
2. $a.P \xrightarrow{\sigma} a.P$

with a clock context containing only σ. To ensure that the first of these two derivations occurs, or, in other words, to *insist* that a is performed before the next tick of the clock, σ, Δ is used. The semantics for an insistent prefix, $\underline{\alpha}.P$, may be given as:

$$[\![\underline{\alpha}.P]\!] \stackrel{\text{def}}{=} \alpha.P + \Delta \tag{3.7}$$

where the presence of Δ prevents a σ transition from occurring on the right-hand side of the choice, and thus for the choice as a whole (as both sides must move through time simultaneously). This leaves only one available action, $\xrightarrow{a}$, as required. Clearly, insistency relative only to one particular clock may also be defined in a similar manner, using Δ_σ instead.

$$[\![\underline{\alpha}_\sigma.P]\!] \stackrel{\text{def}}{=} \alpha.P + \Delta_\sigma \tag{3.8}$$

While on the subject of derived syntax, it is also possible to define a clock prefix, akin to the existing action prefix:

$$[\![\sigma.P]\!] \stackrel{\text{def}}{=} \lceil \mathbf{0} \rceil \sigma(P) \tag{3.9}$$

where the stable timeout ensures that the $\sigma.P$ will be retained until σ ticks, despite the ticks of other clocks. As the only transitions for $\mathbf{0}$ are clock ticks, only a tick from σ will cause the process to evolve and become P.

The two notions of a clock prefix and insistency can then be combined to give an insistent clock prefix:

$$[\![\underline{\sigma}.P]\!] \stackrel{\text{def}}{=} \lceil \Delta \rceil \sigma(P) \tag{3.10}$$

which differs from a standard clock prefix by only ever allowing the one transition, $\underline{\sigma}.P \xrightarrow{\sigma} P$, whereas $\sigma.P$ allows an arbitrary number of transitions from other clocks before this occurs.

3.4.3 Encapsulation

Clock hiding is used to provide scoping for the ticks of a clock. Take the following situation,

$$(P/\sigma) \mid Q \tag{3.11}$$

where $/\sigma$ hides the clock, σ, so that its ticks may only be seen by P. Q instead sees a silent action each time σ ticks. Such clock hiding is central to the encapsulation

of components present in CaSE. When coupled with restriction, components can be made to emit only silent actions from the perspective of external processes.

An operational semantics for CaSE can be given in terms of a labelled transition system, $(\mathcal{P}, \mathcal{A} \cup \mathcal{T}, \rightarrow)$, where $\mathcal{P}$ is the set of CaSE expressions formed from the above syntax, $\mathcal{A}$ and $\mathcal{T}$ are as defined above and $\rightarrow \subseteq \mathcal{P} \times (\mathcal{A} \cup \mathcal{T}) \times \mathcal{P}$ is the transition relation defined in Table 3.1. We use E and F to range over process terms ($\mathcal{P}$), α over the set of actions ($\mathcal{A}$), σ and ρ over the set of clocks ($\mathcal{T}$), a and b over the set of names ($\mathcal{N}$) and γ over $\mathcal{N} \cup \mathcal{T}$. In Table 3.2, we show how the semantics for CaSE expand on those for CCS[2], presented in Table 2.1.

3.5 Conclusion

The main advantage of the timed calculi we have discussed here is that they allow, via the introduction of *global synchronisation*, the construction of systems on a larger scale than those that could be created purely with CCS. With CaSE, components can be created which consist of multiple processes and clocks. These can then be successfully integrated together to form new components.

Global synchronisation allows the problem of defining a compositional broadcast agent, cited earlier in 2.3, to be solved, but these timed calculi still retain the other problems with CCS we mentioned there. None of TPL, PMC, CSA or CaSE explicitly includes data within the model. This is not necessarily a disadvantage, as it is possible to model data implicitly, via the use of silent actions. Including data explicitly in the model complicates formal reasoning and equivalence theories, so we will also adopt the implicit approach.

More importantly, these calculi all still retain a static structure. The scope of restriction or clock hiding doesn't change as the processes evolve. This prevents these calculi from being used to model mobile systems where these elements do change, although they are perfectly suited to modelling static dataflow-oriented systems such as those in (Norton & Fairtlough, 2004) and (Norton et al., 2005).

In contrast, the following chapter contains a discussion of calculi which, while lacking the scalability of the timed languages just illustrated, can model *mobile systems*.

[2]We go directly from CCS to CaSE in the table, but as noted above, some concepts were introduced by intermediate calculi such as TPL.

Table 3.1: CaSE Semantics

$$\text{Idle}\ \frac{-}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}} \qquad \text{Act}\ \frac{-}{\alpha.E \xrightarrow{\alpha} E}$$

$$\text{Patient}\ \frac{-}{a.E \xrightarrow{\sigma} a.E} \qquad \text{Stall}\ \frac{-}{\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma}\ \rho \neq \sigma$$

$$\text{Sum1}\ \frac{E \xrightarrow{\alpha} E'}{E + F \xrightarrow{\alpha} E'} \qquad \text{Sum2}\ \frac{F \xrightarrow{\alpha} F'}{E + F \xrightarrow{\alpha} F'}$$

$$\text{Sum3}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'} \qquad \text{Par1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

$$\text{Par2}\ \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \text{Par3}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{Par4}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F', E \mid F \overset{\tau}{\not\to}}{E \mid F \xrightarrow{\sigma} E' \mid F'} \qquad \text{FTO1}\frac{E \overset{\tau}{\not\to}}{\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F}$$

$$\text{FTO2}\frac{E \xrightarrow{\gamma} E'}{\lfloor E \rfloor \sigma(F) \xrightarrow{\gamma} E'}\ \gamma \neq \sigma \qquad \text{STO1}\frac{E \overset{\tau}{\not\to}}{\lceil E \rceil \sigma(F) \xrightarrow{\sigma} F}$$

$$\text{STO2}\frac{E \xrightarrow{\alpha} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\alpha} E'} \qquad \text{STO3}\frac{E \xrightarrow{\rho} E', E \overset{\tau}{\not\to}}{\lceil E \rceil \sigma(F) \xrightarrow{\rho} \lceil E' \rceil \sigma(F)}\ \rho \neq \sigma$$

$$\text{Rec}\ \frac{E \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'\{\mu X.E/X\}} \qquad \text{Res}\ \frac{E \xrightarrow{\gamma} E'}{E \setminus a \xrightarrow{\gamma} E' \setminus a}\ \gamma \neq a$$

$$\text{Hid1}\ \frac{E \xrightarrow{\sigma} E'}{E/\sigma \xrightarrow{\tau} E'/\sigma} \qquad \text{Hid2}\ \frac{E \xrightarrow{\gamma} E'}{E/\sigma \xrightarrow{\gamma} E'/\sigma}\ \gamma \neq \sigma$$

Table 3.2: Derivation of CaSE from CCS

Rule in CCS	Use in CaSE
New	$Idle$; Allows clocks to tick over $\mathbf{0}$
Act	As in CCS
New	$Patient$; Allows clocks to tick over the prefix $\alpha.E$
New	Prevents the clock σ from ticking
$Sum1$	As in CCS
$Sum2$	As in CCS
New	$Sum3$; Allows clocks to tick over a summation
$Par1$	As in CCS
$Par2$	As in CCS
$Par3$	As in CCS
New	$Par4$; Allows clocks to tick over $\mid$ when not prevented by τs
New	$FTO1$; Allows σ to tick, leaving F, when there are no τs
New	$FTO2$; Allows anything but σ to precede, leaving E'
New	$STO1$; Allows σ to tick, leaving F, when there are no τs
New	$STO2$; Allows α transitions to precede, leaving E'
New	$STO3$; Allows other clocks to tick over E, keeping the timeout
Rec	As in CCS, but generalised to γ
Res	As in CCS, but generalised to γ
New	$Hid1$; Converts the hidden clock σ's transitions to τs
New	$Hid2$; Allows anything but σ to precede, retaining clock hiding

Chapter 4

Mobility

4.1 Introduction

Within the field of algebraic process calculi, there are two clear ways in which the dynamic nature of a system is modelled. The most well-known is the form of mobility present within Milner's π calculus which allows the scope of a name to change as the system evolves. This concept can be thought of in a similar way to the reference passing that occurs in most programming languages; part of the program begins with no knowledge of an entity, and later gains knowledge by obtaining a reference to it.

Models in the π calculus are not really mobile in the sense of something moving from one place to another. This isn't possible, as there is no real notion of 'place' to begin with. However, the addition of this mechanism does allow the modelling of dynamic systems, such as a mobile phone network (Milner, 1993a), and is sufficiently expressive as to allow it to encode Church's λ calculus (Milner, 1992).

A more naturalistic form of mobility is found in calculi which allow entities to *migrate*. One of the primary exponents of this is Cardelli and Gordon's ambient calculus (Cardelli & Gordon, 1998), which groups composed processes inside *ambients*. These ambients can be moved up and down a nested hierarchy of such objects, or be destroyed. The calculus differs from those previously considered, in that it lacks communication primitives. Surprisingly, the base syntax is sufficient to allow communication to be encoded within them, and indeed the entire asynchronous form of the π calculus can be represented.

The following two sections consider examples of both types of mobile calculi in more detail.

4.2 Scope Mobility

4.2.1 The π Calculus

The π calculus (Milner, 1999) follows on from Milner's earlier work on CCS discussed in 2.2. Essentially, it is a value-passing form of CCS in which values and channels are replaced simply by *pure names*. Thus, channels can be passed between processes, as well as values, which means that their scope may change during execution.

To make this clearer, consider the syntax of the form of π calculus given in (Milner, 1992)

$$E, F \ ::= \ 0 \mid \overline{x}y.E \mid x(y).E \mid (a)E \mid (E \mid F) \mid !E \tag{4.1}$$

which is a minimal version containing replication as opposed to recursion, with a a channel name and x and y being defined below. Compare this with the syntax given for CCS in Eqn. 2.1. The nil process, 0, is still present, as is parallel composition and restriction (although in a new form, $(a)E$). Non-deterministic choice is present in the original version of the π calculus presented in (Milner, Parrow & Walker, 1989), but is removed from the version given in (Milner, 1992) due to the formulation of semantics used there. $!E$ is the syntax for replication, which replaces recursion in this particular variant of the calculus to give a simpler theoretical treatment, while still doing much the same job.

The main distinction between the two lies in the remaining element of the syntax: prefixing. In CCS, a more general syntax, $\alpha.E$, where $\alpha \in \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, is used and includes input, output and silent actions. In the syntax given above for the π calculus, the input ($x(y)$) and output ($\overline{x}y$) syntax are given separately, and the input prefix is *binding*[1] like restriction. x and y are both names, where 'x [is] the *subject* and y the *object*' (Milner, 1992). Silent actions no longer appear in prefix form, but do occur as $\tau.E$ in some variants of the π calculus.

The distinction between the π calculus and value-passing forms of CCS, which also use this form of prefixing, lies in x and y being drawn from the same set in the π calculus. In contrast, value-passing forms of CCS keep the two sets distinct, so that the channel and value names do not intersect. This change is what gives π calculus its power, as channels can now be used as the object of an input or output. Thus,

$$x(y).\overline{y}x.0 \tag{4.2}$$

becomes perfectly valid.

[1] When an input is received on x, y is bound to the value of that input, which is then substituted for y in the continuation of that process.

This also has an effect on restriction. Recall that, in CCS, $(a.0|\overline{a}.0)\backslash a$ restricts the scope of a to just the two processes, $a.0$ and $\overline{a}.0$, making a synchronisation the only possible action which may be performed. Now consider the following processes defined using the π calculus:

$$(a)(a(x).\overline{x}a.0 \mid \overline{a}y.0) \mid y(z).P \tag{4.3}$$

where the scope of a is again restricted, this time to the two processes $a(x).\overline{x}a.0$ and $\overline{a}y.0$. If these two processes synchronise, the system evolves to:

$$(a)(\overline{y}a.0 \mid 0) \mid y(z).P \tag{4.4}$$

with x becoming bound to the channel name, y. This shows how the π calculus allows channel names to be passed between processes, but it is the next transition that is really interesting. $\overline{y}a.0$ will pass the channel name, a, to $y(z).P$, which is outside the scope of the restriction imposed on a. As a result, the scope of a is *extruded*:

$$(a)(0 \mid 0 \mid P\{a/z\}) \tag{4.5}$$

so as to include the process, P, in which a is now substituted for z. Further, one of the structural congruence rules of the π calculus (Milner, 1992):

$$(x)(P \mid Q) \equiv P \mid (x)Q \text{ if x not free in P} \tag{4.6}$$

may be used to perform *scope intrusion*, giving:

$$0 \mid 0 \mid (a)(P\{a/z\}) \tag{4.7}$$

as the channel a no longer occurs in the other two processes. These changes in scope are central to the concept of mobility within the π calculus. They reflect the dynamic environment of the processes represented, and give the calculus a greater expressivity.

4.2.2 Variants of the π Calculus

Multiple variants of the π calculus exist, including various evolutions of the syntax and semantics. As noted above, replication is only introduced in the version of the calculus given in (Milner, 1992), which also defines a reduction-based semantics. The earlier tutorial papers (Milner et al., 1989) instead use recursion and a structured operational semantics, based on a labelled transition system.

The polyadic π calculus (Milner, 1993b) is a more distinct variant. Essentially, this involves a syntactic change to input and output, so that a tuple is used, as

opposed to the single names used in the monadic π calculus[2]. Having this as a core part of the syntax provides advantages in representing abstractions and giving a natural sort discipline[3]. However, it is also possible to simply provide an encoding of this in the monadic variant.

Doing so is not simply a matter of transmitting each value in sequence; the operation needs to respect the atomicity implicit in the use of multiple names. Observe the following example from (Milner, 1993b):

$$x(yz) \mid \overline{x}y_1z_1 \mid \overline{x}y_2z_2 \tag{4.8}$$

where the process on the left should receive either y_1 and z_1 or y_2 and z_2. With the following semantics,

$$[\![x(yz)]\!] \stackrel{\text{def}}{=} x(y).x(z) \tag{4.9}$$

$$[\![\overline{x}yz]\!] \stackrel{\text{def}}{=} \overline{x}y.\overline{x}z \tag{4.10}$$

the two sending processes can interfere with one another. y will become bound to either y_1 or y_2 on the first synchronisation, which is fine, but z may then receive whichever of these two remains instead of the second element in the tuple. This happens because there is no link between the two synchronisations. Thus, each subsequent transmission results in a new competition between the two processes as to who actually synchronises with the receiver.

The solution to this problem is to make use of a *private channel*. Before transmitting any of the names that form part of tuple, the sending process passes a reference to a new channel to the receiver. The receiver then uses this channel to receive the contents of the tuple, rather than relying on an existing channel, which may be prone to interference. Thus, the semantics become:

$$[\![x(yz)]\!] \stackrel{\text{def}}{=} x(w).w(y).w(z) \tag{4.11}$$

$$[\![\overline{x}yz]\!] \stackrel{\text{def}}{=} (w)(\overline{x}w.\overline{w}y.\overline{w}z) \tag{4.12}$$

where w is the new private channel created to facilitate the process of transmitting the tuple. This ability to encode the polyadic variant in the original monadic calculus implies that the new syntax fails to yield any greater expressivity, but this is not really the motivation behind this extension. Instead, what this provides is a more natural way of transmitting information, which makes modelling relatively complex systems easier.

[2]This is a term used to refer to the original π calculus in retrospect.

[3]Sorts are a way of applying typing to the π calculus, which will be covered further in section 7.2 on typed calculi.

The asynchronous π calculus (Honda & Tokoro, 1991; Boudol, 1992; Sangiorgi, 2001) deliberately reduces the level of expressivity in order to simplify reasoning and provide a better framework for distributed implementations. The output prefix, $\overline{x}y.E$ is replaced with $\overline{x}y.0$, so that there is no continuation after an output. In the original synchronous π calculus, the behaviour of the continuation, E, is blocked until a synchronisation with a recipient can occur. This doesn't occur in the asynchronous variant, as there is no longer any behaviour dependent on this output occurring.

Synchrony can be emulated in the asynchronous polyadic π calculus, just as synchronous messaging frameworks, such as TCP, can be implemented on top of an asynchronous network. The receiver simply has to acknowledge receipt of the message by replying to the sender. The following semantics are given for the monadic prefixes in (Bugliesli, Castagna & Crafa, 2001):

$$[\![\overline{c}x.P]\!] \stackrel{\text{def}}{=} (r)(\overline{c}xr \mid r.P) \tag{4.13}$$

$$[\![cy.P]\!] \stackrel{\text{def}}{=} c(yr).(\overline{r} \mid P) \tag{4.14}$$

where r is not free in P and $r.P$ is a syntactic abbreviation for $r().P$ i.e. the input is an empty tuple. The output is encoded as the transmission of a tuple containing two names: x, the original name being sent, and r, a new channel created to receive the acknowledgement from the recipient. This runs in parallel with another process that awaits an input on r before continuing with P. For example,

$$\begin{array}{ll} & \overline{c}x.P \mid cy.Q \\ \equiv & (r)(\overline{c}xr \mid r.P) \mid c(ys).(\overline{s} \mid Q) \\ \rightarrow & (r)(r.P \mid \overline{r} \mid Q\{x/y\}) \\ \rightarrow & P \mid Q\{x/y\} \end{array} \tag{4.15}$$

Thus, the original synchronous behaviour is emulated, as P will not evolve until the receiver has obtained the private channel, r, and replied.

Other changes to the calculus are also commonly adopted to reduce its expressivity, thus making more proofs feasible. These include:

- *input localisation* (Merro, 2001), whereby a link received from another process can not be used for input. For example, a process $a(x).P$ may not use x as a channel upon which to receive input in P.

- *uniform receptiveness* (Sangiorgi, 1999), where the input end of a link occurs only once syntactically and is replicated so as to be always available.

- *input-guarded replication*, which is not just restricted to uniform receptiveness variants, but is generally used as a more restricted form of replication (so the replication operator becomes $!a(x).P$ rather than $!P$).

The final variant of the π calculus considered here is the extension to higher-order operations. The most obvious change to make in this direction is to allow processes to be exchanged. Such a second-order form of the calculus is given by the *Calculus of Higher Order Communicating Systems* (CHOCS) (Thomsen, 1989), which actually predates the π calculus itself. This extended CCS with mobility by allowing processes, rather than channel names, to be transmitted.

The more general area of higher-order π calculus, and the theory behind it, is covered in Sangiorgi's thesis (Sangiorgi, 1993). It defines an extension to the π calculus, HOπ, which not only allows the transmission of names (first-order) and processes (second-order), but also parametrised processes of arbitrarily high order (ω-order). This is best illustrated by some examples, drawn from (Sangiorgi, 1993). In the simplest case, an 'executor' process can be defined, $x(X).X$, which will receive and then execute an arbitrary process. Placing this in an appropriate context,

$$\overline{x}P.Q \mid x(X).X \tag{4.16}$$

the process on the left, $\overline{x}P.Q$, will transmit the process, P, to the executor before continuing as Q. Thus, following the synchronisation of the two processes, this system evolves to become:

$$Q \mid P \tag{4.17}$$

where the process P having being substituted for X.

A more complex example is given by considering Milner's encoding of the natural numbers (Milner, 1993b). A natural number, n, is encoded as a series of outputs on y, the number of which is equal to n (represented as $\overline{y}^n$), followed by a transmission on z to indicate zero and thus, the end of the number:

$$[\![n]\!] \stackrel{\text{def}}{=} (y, z)\overline{y}^n.\overline{z} \tag{4.18}$$

Using HOπ, the addition of these numbers can be encoded in a very simple way. In the π calculus, summation is achieved via an indirect reference to the two numbers, using channel names. In HOπ, the parametrised processes or *agents* that represent the numbers can be used directly in the representation of addition. Thus, actually adding the two numbers together becomes a simple matter of running the two concurrently, and linking them via a common channel.

A *Plus* agent, which performs the addition of two numbers, can be defined as follows:

$$Plus \stackrel{\text{def}}{=} (X, Y)(y, z)((x)(X\langle y, x\rangle \mid x.Y\langle y, z\rangle)) \tag{4.19}$$

where both X and Y are agents with two parameters, corresponding to y and z respectively in the definition of $[\![n]\!]$ above. The operation of this agent is best demonstrated by example. Assume X is two and Y is three, represented in HOπ as:

$$X(y, z) \stackrel{\text{def}}{=} \overline{y}.\overline{y}.\overline{z} \tag{4.20}$$

$$Y(y, z) \stackrel{\text{def}}{=} \overline{y}.\overline{y}.\overline{y}.\overline{z} \tag{4.21}$$

and retaining the same representation used for $[\![n]\!]$ above. When X and Y are passed to the $Plus$ agent, X is instantiated with a new private channel, x, in place of z in the above. Y is then prefixed with an input on this same channel, so that the y outputs occurring in Y only execute after those in X. This leads to the following sequence of transitions:

$$\xrightarrow{y}\xrightarrow{y}\xrightarrow{\tau}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{z} \tag{4.22}$$

which is close to the sequence that occurs for the representation of five in HOπ:

$$\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{z} \tag{4.23}$$

Formally, the two are *weakly bisimilar*. A *bisimulation* is a symmetric binary relation between two processes, which exists if each process can simulate the behaviour of the other. R is such a relation iff, for all pairs of processes (p, q) in R and all actions, α[4]:

1. $P \stackrel{\alpha}{\rightarrow} P' \implies \exists Q' \textit{ such that } Q \stackrel{\alpha}{\rightarrow} Q' \textit{ and } (P', Q') \in R$
2. $Q \stackrel{\alpha}{\rightarrow} Q' \implies \exists P' \textit{ such that } P \stackrel{\alpha}{\rightarrow} P' \textit{ and } (P', Q') \in R$

For a weak bisimulation, τ transitions are effectively ignored. A series of such transitions, $\stackrel{\tau}{\rightarrow}\stackrel{\tau}{\rightarrow}\stackrel{\tau}{\rightarrow}\ldots$ is abbreviated to $\stackrel{\tau}{\Rightarrow}$ and $\stackrel{\tau}{\Rightarrow}\stackrel{a}{\rightarrow}\stackrel{\tau}{\Rightarrow}$ is deemed equivalent to $\stackrel{a}{\rightarrow}$. As the additional τ transition in the $Plus$-based derivation is the only difference between the two, the two can be deemed equivalent under the rules of weak bisimulation.

Returning to HOπ, the most interesting point about this calculus is not that it provides the means to formulate abstractions of the type just demonstrated, but that, in doing so, it adds no further expressivity. Indeed, Sangiorgi, in his thesis

[4]The bisimulation definition given here is more applicable to the static systems of CCS. Although it holds for this simple example, a more detailed method of bisimulation is required to handle the dynamic binding that occurs in the π calculus and its derivatives.

(Sangiorgi, 1993) demonstrates how a HOπ calculus can be represented in the π calculus. Thus, just as with the polyadic variant, the benefit of using HOπ comes not from increased expressivity, but from the additional ease it provides in modelling certain scenarios.

The Join Calculus

The Join calculus (Fournet & Gonthier, 1996) takes the asynchronous π calculus as its basis, and focuses on providing a formalism better suited as the basis for a distributed implementation.

Take the following example of a π calculus process given in (Lévy, 1997):

$$x(y).P \mid x(z).Q \mid \overline{x}a \tag{4.24}$$

where two processes are waiting to receive input on x. The problem with implementing this in a distributed setting is that there is no concept of location with the π calculus. Each of the two receiving processes or *receptors*[5] may be located at an arbitrary distance both from each other and from the transmitter, $\overline{x}a$. As a result, a *distributed consensus problem* arises as to which of the two receptors will receive the transmission.

The join calculus provides a solution to this problem by altering the syntax of the π calculus. The asynchronous variant of the syntax given in Eqn. 4.1 becomes:

$$P, Q \ ::= \ 0 \mid \texttt{def}\ D\ \texttt{in}\ P \mid (P \mid Q) \mid x\langle\tilde{v}\rangle \tag{4.25}$$
$$D, E \ ::= \ J \rhd P \mid D \wedge E \mid \mathbf{T} \tag{4.26}$$
$$J, J' \ ::= \ x\langle\tilde{v}\rangle \mid (J \mid J') \tag{4.27}$$

with $\mathbf{T}$ being the empty definition and a clear focus on linking the receptors in D to the emissions occurring in P (both represented by the same syntax, $x\langle\tilde{v}\rangle$). The use of this is most clearly demonstrated by example:

$$\texttt{def}\ (x\langle y\rangle \rhd P) \wedge (x\langle z\rangle \rhd Q)\ \texttt{in}\ x\langle a\rangle \tag{4.28}$$

which has essentially the same behaviour as the π calculus example presented earlier. $x\langle y\rangle \rhd P$ receives an input, y, on x and then continues as P. $x\langle y\rangle$ is said to guard P, and multiple such guards may be applied to a single such process. Multiple such receptors may be defined via use of the $\wedge$ operator.

It is impossible to provide an exact equivalent to the earlier series of π calculus processes, as the changes in the join calculus now prevent such scenarios from being created. Instead, the equivalent of this join calculus example in the π calculus is:

[5]The join calculus uses an analogy with chemistry to describe its behaviour, based on the *CHemical Abstract Machine* (CHAM) (Berry & Boudol, 1992).

$$(x)(!(x(y).P \mid x(z).Q) \mid \overline{x}a) \tag{4.29}$$

where the scope of x is restricted to the `def` expression and the inputs are replicated, so as to be always available. Thus, a channel x is always *localised* to a particular set of emitters and receptors.

Clearly, the join calculus, as a reformulation of the asynchronous π calculus with a new syntax, can not be used to express anything which can't be expressed in the π calculus. However, it has a lot of advantages in endowing the calculus with distributive properties at the syntactic level.[6]

4.2.3 Advantages and Limitations of the π Calculus

The π calculus is a powerful formalism drawn from a minimal abstract syntax. As noted at the start of this section, it is capable of encoding the λ calculus and so it follows that it is also capable of simulating any recursive function.

The problem is that this makes it a little too powerful in some cases. From (Sangiorgi, 2002), we can see how much more difficult the additional power given by the π calculus makes proving termination. In contrast, a sufficiently restricted form of CCS provides a trivial proof. In the same paper, Sangiorgi also touches on something which seems common within the literature (Fournet & Gonthier, 1996; Amadio, 1997; Wojciechowski, 2000; Stefani, 2003); while the expressiveness of the π calculus is interesting, it is necessary to restrict it in order to actually have something which is generally useful for reasoning over or using as the basis for a full programming language.

Another problem with the π calculus is that it carries with it a trait from CCS. Namely, it can't be used to model synchronisation with an arbitrary number of processes in a compositional way. This was considered earlier in 2.3 for CCS, and solved in 3.1 using the additions to the calculus given by TPL. While the π calculus has a notion of mobility and is thus more expressive than CCS, it still lacks an external entity with which to co-ordinate such a transaction.

A common motif reoccurs here, that was touched on earlier in the introduction to this review; even though something has a certain level of expressivity, it doesn't follow that it is the most appropriate mechanism for modelling a particular phenomenon. This also holds for the distributed calculi considered in 4.3. The π calculus may already model mobility, but these calculi do so in a different way, which may prove more suitable in a particular context.

[6]Such changes have also been made using the restrictions imposed by an appropriate type system (Sangiorgi, 1999).

4.3 Distribution and Migration

Allowing the scope of a name to change during execution is one possible way of modelling dynamic behaviour, but it isn't the only way. The concept of *mobility* naively implies the physical movement of processes, but, as shown above, this is not what actually happens in the π calculus. To do so requires some notion of *distribution*; this can be provided by *localities*, a term used to refer generally to a higher-level form of grouping, above that of processes. This concept has been applied to various calculi, in different forms, in order to model physical sites (Nomadic Pict, Wojciechowski, 2000), administrative or security domains (the Ambient Calculus and the Seal Calculus, Cardelli & Gordon, 1998; Vitek & Castagna, 1999) and biological cells (Brane Calculi, Cardelli, 2004), but can theoretically be applied in any context where the grouping of processes is useful. Localities can be used simply for observation or as a means to further control the behaviour of the processes encapsulated within them. They are generally named, so as to provide a communication target or a known destination for a migrating entity.

Originally, localities were used to distinguish between processes in order to provide further equivalence theories. Take the following simple CCS-based example process:

$$Spec \stackrel{\text{def}}{=} in.\tau.\overline{out}.Spec \tag{4.30}$$

which forms the *specification* for the behaviour of a system that receives an input, processes it and then returns the output. The actual *implementation* may differ from the specification by instead involving two processes:

$$Receiver \stackrel{\text{def}}{=} in.\overline{a}.Receiver \tag{4.31}$$

$$Sender \stackrel{\text{def}}{=} a.\tau.\overline{out}.Sender \tag{4.32}$$

which communicate over another channel, a. If these two processes are run concurrently:

$$(Receiver \mid Sender) \setminus a \tag{4.33}$$

with the scope of a restricted, they are *weakly bisimilar* (see 4.2.2) to one another. The specification performs the following derivations:

$$\xrightarrow{in}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{4.34}$$

prior to recursing and becoming *Spec* again, whereas the implementation produces:

$$\xrightarrow{in}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{4.35}$$

Table 4.1: LCCS Dynamic SOS Rules

$$\mathsf{Act1}\ \frac{-}{a.E \xrightarrow[l]{a} l :: E}\ for\ any\ l \in Loc \qquad \mathsf{Act2}\ \frac{E \xrightarrow[u]{a} E'}{l :: E \xrightarrow[lu]{a} l :: E'} \qquad \mathsf{Act3}\ \frac{-}{\tau.E \xrightarrow{\tau} E}$$

with the extra τ transition caused by the synchronisation on a. As weak bisimulation effectively ignores τ actions, the two are judged to be equivalent. If the specification was to include a further τ action, for an arbitrary reason, prior to the $\overline{out}$, then the two would also be strongly bisimilar. To summarise, the difference between the two sets of derivations is negligible, according to the bisimulation, yet the actual difference between the specification and its implementation is fairly significant. The specification effectively requests a monolithic solution, but weak bisimulation allows the final implementation to be distributed over multiple processes.

In most situations, this is beneficial. It means that the specification can be met by a concurrent system, composed of multiple processes running in parallel, superfluous τ transitions aside. When a distinction between the number of processes used is required, a finer equivalence is needed. *Location bisimulation* (Boudol, Castellani, Hennessy & Kiehn, 1993) provides exactly that, by assigning locations to processes and using them as part of the relation between processes.

Essentially, this means that each transition is annotated with a location name. In (Boudol et al., 1993), a located variant of CCS is defined, LCCS, which adds an additional piece of syntax, $l :: E$ to signify that a process E is located at l. This association is made within the operational semantics, of which there are two variants. The *static* approach allocates locations initially, while the *dynamic* method generates a new location for each non-silent transition. Here, the focus is on the latter, shown in Table 4.1, which essentially gives each process a *causal path*, by explicitly representing the number of transitions that have been performed.

The semantics, as with those for CaSE and TNT given in chapter 5, are based on a *labelled transition system*. The possible behaviour of a process is defined as a series of labelled transitions from one process to another, which are later used as the basis for the bisimulation-based equivalence theories shown earlier. The rules presented here are only a subset of those for LCCS, being those that are relevant to the use of locations. The remaining rules for summation, parallel composition and restriction are as for CCS itself, with the additional inclusion of the location on the transition. These are discussed informally in section 2.2, and also appear as part of the CaSE semantics.

The rule, Act1, handles the initial assignment of a location for any action, $a.E$,

where $a \in \mathcal{N} \cup \overline{\mathcal{N}}$ (i.e. $a \neq \tau$) and Loc is simply a set of location names. The rule states that the process may perform a transition to the process $l :: E$. The transition itself is annotated with both the action a and the new location, l, which causes the locations to appear in the sequence of transitions for each process (and, thus, the equivalence theory).

Act2 is a continuation of Act1, which handles processes that have already been assigned a location. If the process itself, E, can perform some action, a, with the location, u, to become E', then so can the located version of E. The interesting part of this rule is how the location is used in the new transition. The u from the new transition is concatenated with the l from the current location, so the transition depicts the specific route the process has taken through each location. The final rule, Act3, simply handles silent actions, which are unaltered from their behaviour in CCS, and have no association with locations.

How this actually works in practise is best shown by reconsidering the earlier CCS example. Recall the specification defined in 4.30. This is a process with essentially three actions, in, τ and $\overline{out}$, which may be localised via use of the LCCS semantics given above. As the process begins its life in an unlocated form, Act1 is applied to assign it a location:

$$in.\tau.\overline{out}.Spec \xrightarrow[l]{in} l :: \tau.\overline{out}.Spec \tag{4.36}$$

where l is an arbitrary location name[7]. The evolution of the resulting process, $l :: \tau.\overline{out}.Spec$ utilises both Act2 and Act3. Act2 provides the appropriate transition for such a located process, but its behaviour is based on that of the unlocated process, which in this case is $\tau.\overline{out}.Spec$. Thus, Act3 is used to yield:

$$\tau.\overline{out}.Spec \xrightarrow{\tau} \overline{out}.Spec \tag{4.37}$$

which is then applied as the precondition for Act2 to give:

$$l :: \tau.\overline{out}.Spec \xrightarrow[l]{\tau} l :: \overline{out}.Spec \tag{4.38}$$

As u is effectively the empty string, ϵ, in this case, due to the τ transition being unlocated, the result of the concatenation, ul, is simply l.

The final derivation again combines the use of Act2 with another rule. This time, the action is a member of $\overline{\mathcal{N}}$, so Act1 is used to give the derivation of the unlocated variant, $\overline{out}.Spec$:

$$\overline{out}.Spec \xrightarrow[k]{\overline{out}} k :: Spec \tag{4.39}$$

[7]The name is arbitrary in the sense that it doesn't matter what the name is, but, as the later discussion of bisimulation shows, the location names must be assigned in some kind of regular fashion to facilitate comparison.

where k is again an arbitrary location assigned to the new visible action. Merging this with the main process using Act2 gives:

$$l :: \overline{out}.Spec \xrightarrow[lk]{\overline{out}} l :: k :: Spec \tag{4.40}$$

resulting in a final process with a causal path of two locations, l and k.

But how does this help distinguish the specification from its dual process implementation shown previously? First, it is necessary to extend the definition of bisimulation given in 4.2.2 to incorporate the localised transitions of LCCS. Recall that a *bisimulation* is a symmetric binary relation between two processes, which exists if each process can simulate the behaviour of the other. $R \subseteq LCCS \times LCCS$ is a *dynamic location bisimulation* relation iff, $\forall (p,q) \in R \wedge a \in \mathcal{N} \cup \overline{\mathcal{N}} \wedge u \in Loc$:

1. $P \xrightarrow[u]{a} P' \implies \exists Q' \textit{ such that } Q \xrightarrow[u]{a} Q' \textit{ and } (P', Q') \in R$
2. $Q \xrightarrow[u]{a} Q' \implies \exists P' \textit{ such that } P \xrightarrow[u]{a} P' \textit{ and } (P', Q') \in R$
3. $P \xrightarrow{\tau} P' \implies \exists Q' \textit{ such that } Q \xrightarrow{\tau} Q' \textit{ and } (P', Q') \in R$
4. $Q \xrightarrow{\tau} Q' \implies \exists P' \textit{ such that } P \xrightarrow{\tau} P' \textit{ and } (P', Q') \in R$

This is the strong variant that observes τ transitions. A localised version of weak bisimulation merely requires satisfying the first two conditions. As the earlier comparison between the two processes was made using weak bisimulation, it is this weak variant of dynamic location bisimulation that will be used here.

The implementation with two processes, shown in 4.31, had the following transitions using plain CCS:

$$\xrightarrow{in} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\overline{out}} \tag{4.41}$$

whereas the specification exhibits the following behaviour in LCCS:

$$\xrightarrow[l]{in} \xrightarrow[l]{\tau} \xrightarrow[l]{\tau} \xrightarrow[lk]{\overline{out}} \tag{4.42}$$

To compare the two, it is necessary to give a similar localised treatment to the transitions for the implementation. Clearly, the τ transitions will be relatively unaffected, and, under a weak form of bisimulation, are irrelevant anyway. Essentially, the two sequences being compared are:

Specification (Localised)	Implementation
$\xrightarrow[l]{in} \xrightarrow[lk]{\overline{out}}$	$\xrightarrow{in} \xrightarrow{\overline{out}}$

when the τ transitions are ignored. To localise the latter of these, it is necessary to look back to the original two processes from which these transitions are derived. The first, $\xrightarrow{in}$, arises from the *Receiver* as follows:

$$in.\overline{a}.Receiver \xrightarrow{in} \overline{a}.Receiver \tag{4.43}$$

which, when localised, becomes:

$$in.\overline{a}.Receiver \xrightarrow[l]{in} l :: \overline{a}.Receiver \tag{4.44}$$

So, the first of the two transitions should be $\xrightarrow[l]{in}$ when LCCS is used.

However, the use of a makes things a little complicated. It appears in both the *Receiver* (as just shown) and the *Sender* as a visible action (a and $\overline{a}$ respectively), but these combine to become a τ action when the two are run in parallel. The above makes it appear that the *Receiver* will evolve to $l :: k :: Receiver$, by assigning a further location to a, but this doesn't match with the higher-level behaviour of the composed processes. Thus, to make assigning locations easier, it is better to look instead at the sequences of transitions from each process, rather than their explicit definitions:

$$\xrightarrow{in}\xrightarrow{\tau} \tag{Receiver}$$

$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{Sender}$$

where the τ transition arising from the synchronisation is given for both. From this, it is a simple matter of assigning a location to each observable action:

$$\xrightarrow[l]{in}\xrightarrow{\tau} \tag{Localised Receiver}$$

$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow[l]{\overline{out}} \tag{Localised Sender}$$

and merging the two to give a localised version of both the specification and its implementation:

Specification (Localised)	Implementation
$\xrightarrow[l]{in}\xrightarrow[lk]{\overline{out}}$	$\xrightarrow[l]{in}\xrightarrow[l]{\overline{out}}$

which illustrates a clear difference between the two.

For the first transition, both are capable of performing $\xrightarrow[l]{in}$ and can thus match each other. However, the relation breaks down on the second transition which

compares $\xrightarrow[lk]{\overline{out}}$ with $\xrightarrow[l]{\overline{out}}$. Under a normal weak bisimulation, these two transitions would be judged equivalent, as only the action is available for comparison; both perform an $\overline{out}$. However, a localised bisimulation requires the locations to also match, which fails here. The specification has a longer causal path, as its single process has performed two visible actions. In contrast, the two processes involved in the implementation have performed one action each, resulting in two separate paths with a length of one.

This shows that localities can be used to provide a stronger equivalence theory; a dynamic location bisimulation can distinguish more processes than a standard bisimulation. As stated earlier, localities are now more commonly used in calculi which exhibit mobility in the form of *migration*, where they are used to group arbitrary numbers of processes. The locality gives the grouping a context, which may change during execution of the system, via the movement of the locality or its constituent processes. What follows is a further examination of such distributed calculi, including those which have arisen from existing non-distributed formalisms, such as the Join calculus.

4.3.1 The Distributed Join Calculus

By adding localities, (Fournet, Gonthier, Lévy, Maranget & Rémy, 1996) defines a distributed variant of the Join calculus shown in 4.2.2. The extended syntax is as follows:

$$P, Q \ ::= \ 0 \mid \texttt{def}\ D\ \texttt{in}\ P \mid (P \mid Q) \mid x\langle\tilde{v}\rangle \mid go\langle b, \kappa\rangle \tag{4.45}$$

$$D, E \ ::= \ J \rhd P \mid D \wedge E \mid \mathbf{T} \mid a[D : P] \tag{4.46}$$

$$J, J' \ ::= \ x\langle\tilde{v}\rangle \mid (J \mid J') \tag{4.47}$$

with the additional syntax of $a[D : P]$ representing input channels located at a, the name of the locality. P is used to 'initialise' the locality. The names are globally scoped and unique to a particular definition, so:

$$\texttt{def}\ a[D : P] \wedge a[D' : Q] \rhd R\ \texttt{in}\ S \tag{4.48}$$

is disallowed. The syntax allows localities to be nested to form a hierarchical structure, with each node in the tree corresponding to a different location. All receptors for a channel must occur in the same location. The following is disallowed,

$$\texttt{def}\ a[x\langle y\rangle \rhd P : S] \wedge b[x\langle z\rangle \rhd Q : R]\ \texttt{in}\ T \tag{4.49}$$

as one receptor for x, P, is defined in location a and the other in location b. Instead,

$$\texttt{def}\ a[x\langle y\rangle \rhd P \wedge x\langle z\rangle \rhd Q : R]\ \texttt{in}\ T \tag{4.50}$$

may be used, where both P and Q are in location a.

Migration may occur using the new process construct, $go\langle b, \kappa\rangle$. Rather than the process itself migrating, this operator causes the surrounding location to migrate and become an immediate sub-location of b. Upon completion of the migration, an empty message is emitted on κ. This allows other processes to block until the migration is complete, by waiting for receipt of this completion message. For example,

$$\texttt{def}\ a[D : (P \mid go\langle b, \kappa\rangle)]\ \texttt{in}\ S \mid \texttt{def}\ b[E : Q]\texttt{in}\ T \tag{4.51}$$

reduces to:

$$\texttt{def}\ b[E : Q \mid (\texttt{def}\ a[D : (P \mid k\langle\rangle)]\ \texttt{in}\ S)]\texttt{in}\ T \tag{4.52}$$

when $go\langle b, \kappa\rangle$ is expanded, with a now a sub-location of b.

The distributed join calculus is an interesting example of how an existing calculus (the π calculus in this case) can be both adapted to suit a different purpose or remove perceived deficiencies (as shown in 4.2.2) and then later extended to incorporate mobility via distribution, via the simple addition of localities and a migration primitive. The advantage of this is that the new calculus can build on the established theory of the original calculus, instead of having to start from scratch. This differs from the approach taken by the ambient calculus, which instead begins again from first principles, in an attempt to formalise this more spatial form of mobility in a minimal fashion.

4.3.2 The Ambient Calculus

The ambients within the ambient calculus (Cardelli & Gordon, 1998) are a form of locality. Each ambient can contain processes and other ambients, allowing a nested structure of ambients to be formed. This topology is dynamic; new ambients may be created and existing ones moved or destroyed during execution. Within the formal syntax of the calculus,

$$E, F\ ::=\ 0 \mid M.E \mid (\nu n)E \mid (E \mid F) \mid n[E] \mid\ !E \tag{4.53}$$

the ambients are represented by the term $n[E]$, where n is an ambient name. In comparing this with the syntax given for CCS in Eqn. 2.1 and that of the π calculus from Eqn. 4.1, some apparent similarities can be seen, especially with regard to the latter. The same nil process, 0, is present, as is parallel composition and replication. $(\nu n)E$ looks similar to restriction[8]. Continuing on this presumption, $M.E$ may be

[8]This is the syntax used in versions of the π calculus later than (Milner, 1992).

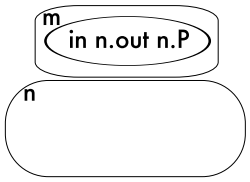


Figure 4.1: Spatial diagram of $m[in\ n.out\ n.P] \mid n[]$

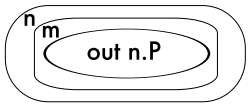


Figure 4.2: Spatial diagram of $n[m[out\ n.P]]$

considered to be the prefixing already seen in CCS and the π calculus. However, the syntax for M is

$$M \ ::= \ in\ n \mid out\ n \mid open\ n \tag{4.54}$$

which is quite different from that of action prefixing. The ambient calculus has no concept of channels; the only names present refer to ambients (so $(\nu n)E$ restricts these). What M provides is a set of mobility primitives, known as *capabilities*. Processes emit these in order to alter the structuring of the ambients, and thus perform the physical migration of ambients and the processes within them.

Perhaps the most confusing aspect of capabilities is that they are emitted by the process, but it is the ambient that actually moves. For example, if process P is defined as $in\ n.0$, then performing this action has the effect of moving the *ambient* in which P resides inside n, rather than just P. Likewise, $out\ n$ is the converse and moves the surrounding ambient outside n.

Such behaviour is best illustrated by an example. Suppose the process, $in\ n.out\ n.P$ begins its life in the ambient m (Fig. 4.1). Performing the first action, $in\ n$, moves its surrounding ambient, m, inside n (Fig. 4.2). The converse, $out\ n$, then moves m back outside n, resulting in a return to the original ambient structure (Fig. 4.3), but with the process having evolved into P.

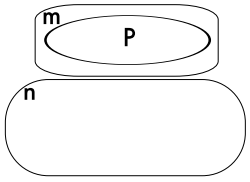


Figure 4.3: Spatial diagram of $m[P] \mid n[]$

open n is quite different. It alters the structure, just as *in* and *out* do, but rather than moving ambients, it destroys them. It is also applied to a child ambient rather than to the surrounding ambient, so $open\ m.P \mid m[Q]$ (as in (Cardelli & Gordon, 1998)) reduces to $P \mid Q$.

There are also issues with regard to the applicability of capabilities and the use of the names. A capability may only cause movement to occur when at least one applicable ambient is available. As such, movement is heavily dependent on context, and specifically the availability of an appropriately named ambient. Applicability is dependent upon the capability involved:

- For *in* m, there must be a sibling of the surrounding ambient named m.
- For *out* m, the parent of the surrounding ambient must be named m.
- For *open* m, there must be a child of the surrounding ambient named m.

All three capabilities are non-deterministic. The same ambient name may occur more than once, and each occurrence is regarded as being distinct. As a result, the reduction of a capability includes a choice if there is more than one applicable ambient present. For example, $open\ m.P \mid m[Q] \mid m[R]$ has two possible derivations,

1. $open\ m.P \mid m[Q] \mid m[R] \rightarrow P \mid Q \mid m[R]$
2. $open\ m.P \mid m[Q] \mid m[R] \rightarrow P \mid m[Q] \mid R$

The issue of non-determinism illustrates the behaviour that occurs when there is more than one applicable ambient. What about when there are none? The process stalls, and can not move on until such an ambient becomes available. This is akin to the situation in channel-based calculi, such as CCS or the π calculus, where a name is restricted, but the appropriate co-name is not available to provide synchronisation. For example,

$$(a.P)\backslash a \tag{4.55}$$

may never progress to become P as there is no $\overline{a}$ for a to synchronise with. This behaviour is particularly relevant with respect to *out* m, where the sole use of the name is to stop the surrounding ambient leaving its parent if the names don't match.

The restriction of ambient names, via $(\nu n)E$, combined with mobility means that scope extrusion is also present in the calculus. Just as the transmission of a name outside its scope causes extrusion in the π calculus, the restriction of ambient names may float outward as necessary. Scope intrusion is also possible in both calculi, as demonstrated by the presence of the structural congruence rule,

$$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \text{ if } n \notin fn(P) \qquad \text{(Struct Res Par)}$$

which allows the restriction of n to be removed from P if the name doesn't occur free within its body.

4.3.3 Variants of the Ambient Calculus

A general problem within concurrency is the possibility of *interference*. This was touched on briefly in the introduction to this review, where the value of x differed due to a race condition. In the ambient calculus, *redex interference* (Levi & Sangiorgi, 2003) is an issue, and is related to the non-determinism mentioned above.

Take the example process from (Levi & Sangiorgi, 2003).

$$n[in\ m.P] \mid m[Q] \mid m[R] \tag{4.56}$$

It is unclear what the environment of P will be, following the reduction of the capability, $in\ m$. There are two alternatives,

1. $n[in\ m.P] \mid m[Q] \mid m[R] \rightarrow m[n[P] \mid Q] \mid m[R]$
2. $n[in\ m.P] \mid m[Q] \mid m[R] \rightarrow m[Q] \mid m[n[P] \mid [R]]$

resulting from the two redexes formed between $n[in\ m.P]$ and $m[Q]$, and $n[in\ m.P]$ and $m[R]$. If one contracts, resulting in a reduction, the other is no longer possible. However, in this case, all three processes, P, Q and R, can still interact following either reduction.

In another example from the same paper,

$$open\ n.P \mid open\ n.Q \mid n[R] \tag{4.57}$$

again with two possible interactions

1. $open\ n.P \mid open\ n.Q \mid n[R] \rightarrow P \mid open\ n.Q \mid R$
2. $open\ n.P \mid open\ n.Q \mid n[R] \rightarrow open\ n.P \mid Q \mid R$

the resulting process includes a process, either $open\ n.Q$ or $open\ n.P$, which is stuck until such a time as another ambient named n appears as a child. This may never occur. These kinds of interference, referred to in (Levi & Sangiorgi, 2003) as *plain interferences*, may occur in other calculi. The equivalent in the π calculus would be:

$$\overline{x}z.P \mid x(y).Q \mid x(y).R \tag{4.58}$$

where again a reduction will occur between one of the two:

1. $\overline{x}z.P \mid x(y).Q \mid x(y).R \rightarrow P \mid Q\{z/y\} \mid x(y).R$
2. $\overline{x}z.P \mid x(y).Q \mid x(y).R \rightarrow P \mid x(y).Q \mid R\{z/y\}$

and the remaining process, either $x(y).Q$ or $x(y).R$, will be blocked.

Another more serious form of interference may occur in the ambient calculus, due to the provision of differing interactions ($in\ m, out\ m$ and $open\ m$). These *grave interferences* occur when an ambient is involved in two reductions occurring as the result of different types of capability. Take the example process,

$$open\ n.\mathbf{0} \mid n[in\ m.P] \mid m[Q] \tag{4.59}$$

in which two reductions can occur that are logically different. While the interferences described above are a representation of the kind of race conditions and non-determinism that would be expected in any concurrent model, for example, to represent competition for resources, grave interferences are usually unexpected and typically represent errors in the model. This process may perform two radically different reductions,

1. $open\ n.\mathbf{0} \mid n[in\ m.P] \mid m[Q] \rightarrow \mathbf{0} \mid in\ m.P \mid m[Q]$
2. $open\ n.\mathbf{0} \mid n[in\ m.P] \mid m[Q] \rightarrow open\ n.\mathbf{0} \mid m[n[P] \mid Q]$

where either n is destroyed, thus preventing the latter movement of P in to m as it has no surrounding ambient, or n moves inside m and is no longer available to be destroyed by $open\ n.\mathbf{0}$. Clearly, only one of these reductions is likely to be intentional.

Levi and Sangiorgi's calculus of Mobile Safe Ambients (Levi & Sangiorgi, 2000; Levi & Sangiorgi, 2003) presents a solution to this. It introduces a notion of co-capabilities, which enforce a pairing of mobility primitives before a reduction can be made. The result of this is that the ambient being entered, exited or opened is aware of what is taking place, and may react accordingly.

With these co-capabilities in place, the reduction rules for the calculus run as follows:

$$n[in\ m.P_1 \mid P_2] \mid m[\overline{in}\ m.Q_1 \mid Q2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2] \qquad \text{(SafeIn)}$$

$$m[n[out\ m.P_1 \mid P_2] \mid \overline{out}\ m.Q_2 \mid Q2] \rightarrow n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2] \qquad \text{(SafeOut)}$$

$$open\ n.P \mid n[\overline{open}\ n.Q_1 \mid Q_2] \rightarrow P \mid Q_1 \mid Q_2 \qquad \text{(SafeOpen)}$$

where, in each case, the capability must be able to synchronise with a co-capability in the relevant ambient for the reduction to take place. For example, in SafeIn, $in\ m.P_1$

must pair up with $\overline{in}\ m.Q_1$ in the ambient m. As a result, Q_1 can react appropriately to the change in structure, based on the fact that it knows the movement has occurred.

The changes in the calculus of safe ambients, though simple, have a dramatic effect on the ability to construct an algebraic theory for the calculus and prove properties, especially when coupled with an appropriate type system[9]. Essentially, they represent a move from asynchronous to synchronous mobility primitives. The calculus of controlled ambients (Teller, Zimmer & Hirschkoff, 2002) restricts behaviour further, by requiring that a co-capability must appear in both the source and the destination. Thus, an *in* m capability requires permission both to leave its current location and to enter the destination ambient. This is useful for the specific application of the calculus, controlling resources, but is excessive in most circumstances.

A further variant of the ambient calculus is the calculus of boxed ambients (Bugliesli, Castagna & Crafa, 2001). This removes the *open* capability altogether, replacing it with a form of directed communication inspired by the Seal calculus (Vitek & Castagna, 1999). Processes remain within their initial ambient permanently (hence the term 'boxed') and only the structure of the ambient topology changes via the *in* and *out* capabilities. Messages may be sent locally, upwards or downwards, but not to siblings.

An example process from (Bugliesli et al., 2001) is:

$$n[(x)^p P \mid p[\langle M\rangle \mid (x)Q \mid q[\langle N\rangle^{\uparrow}]]] \tag{4.60}$$

where n, p and q are ambients, both (x)s are inputs and $\langle M\rangle$ and $\langle N\rangle$ represent outputs. The use of the superscript on $(x)^p$ indicates a downward communication into the ambient p, while the use of $\uparrow$ in $\langle N\rangle^{\uparrow}$ indicates an upward communication directed at the parent ambient. Thus, $(x)Q$ may synchronise with either $\langle M\rangle$ locally or the upward communication from $\langle N\rangle$. $(x)^p P$ must synchronise with $\langle M\rangle$, as the only output in p.

The ideas behind the boxed ambients calculus result in a formalism which is more suited to communication-focused modelling, where the destruction of locations would be unnatural. Both it and the original ambient calculus have their own particular niche, being suited to particular applications. In contrast, the latter is clearly more suited to situations where the removal of a locality corresponds to a similar event in the real-world situation being modelled.

[9] In this case, the type system ensures single-threadedness, where only one process within an ambient may exercise a capability.

4.3.4 Advantages and Limitations of the Ambient Calculus

The most interesting aspect of the ambient calculus is that, while it includes no communication primitives, it can encode the asynchronous π calculus (see 4.2.2). This seems to imply that it is possible to model mobility in a more natural way without losing much of the expressivity of the π calculus. On consideration , this seems a little less surprising as ambient names exhibit the same scope extrusion seen with channel names in the π calculus. With this in mind, it is not too difficult to see that ambient names could be used to mimic channel names, with synchronisation being emulated by two processes performing some kind of interaction within the same ambient.

However, the representation of synchronisation illustrated in (Cardelli & Gordon, 1998) seems to suggest that the ambient calculus may still have problems dealing with the kind of global synchronisation needed for the compositional broadcast agent considered in 2.3. The operation is performed by destroying and recreating ambients, as a signal to the other process involved in the synchronisation. Extending this would seem to require using more ambients, which again leads to the problem of enumerating the number of entities who wish to synchronise. As before, this is possible but not compositional; every time synchronisation is performed with a different number of agents, the semantics of the process must be recreated.

Thus, the ambient calculus and the π calculus have more in common than is initially apparent, and the choice between the two seems to be largely based on the most natural formalism for a particular task.

4.3.5 P Systems

While providing a way of modelling concurrent spatially-oriented systems, P Systems (Păun, 2002; Păun, Rosenberg & Salomaa, 2009 (to appear)) arise from the area of formal language theory and re-writing rules rather than process calculi. They are considered here, as there exist a number of similarities between them and, for example, the ambient calculus both in providing a distributed model of computation and in finding applications in the area of biological modelling. Below, a basic P system with priorities is introduced.

A transition P system with priorities (Păun, 1998) of degree n, where $n \geq 1$ is represented as:

$$\Pi = (V, \mu, M_1, \ldots, M_n, (R_1, \rho_1), \ldots, (R_n, \rho_n), i_0) \tag{4.61}$$

where:

- V is an alphabet of objects.
- μ is the membrane structure, containing n membranes.

- M_i, where $1 \leq i \leq n$, is a multiset of objects from V which are contained in membrane i.
- R_i, where $1 \leq i \leq n$ is an evolution rule associated with one of the membranes, i. The corresponding ρ_i is a partial-order relation which determines the priority of the rule. The rules are rewriting rules of the form $a \rightarrow v$, which causes a to be replaced by v; where $a \in V$ and $v \in (V \times Tar)^*$, $Tar = \{here, out, in | 1 \leq i \leq n\}$. The set Tar of target regions gives all the possible derivations for symbols occurring on the right-hand side of each rule. Rules of the form $a \rightarrow v\delta$ indicate the membrane will disappear after its application.
- i_0 is a number between 1 and n which specifies the *output membrane* where the result of the computation should be found.

Any of the multisets, rules or priority relations may be empty. Evolution occurs in parallel, in a synchronous fashion involving all membranes (referred to as *maximal parallelism*). A universal clock is assumed to exist, which breaks the evolution of the system into cycles. Objects may move between membranes and membranes may be broken, causing their objects to flood into the membrane above and their rules to disappear. Such behaviour has echoes of the ambient calculus described in 4.3.2, where ambients may be destroyed by the *open* primitive and processes may move around the ambient hierarchy (but only within an ambient). The notion of synchronous clock cycles also recalls the discrete timed calculi of chapter 3, where evolution can also be bounded by clock cycles in a synchronous fashion. An interesting distinction is commonly made in P systems; the outer membrane or *skin membrane* is assumed to be special. For example, at least in a biological context, the system is assumed to terminate if the outer membrane is destroyed (biologically, the external membrane has been broken and thus the organism falls apart).

Consider the following example P system (Fig. 4.4),

$$
\begin{aligned}
\Pi_1 &= (V, \mu, M_1, M_2, M_3, M_4, (R_1, \rho_1), (R_2, \rho_2), (R_3, \rho_3), (R_4, \rho_4), 4) \\
V &= \{a, b, b', c, f\} \\
\mu &= [_1[_2[_3]_3[_4]_4]_2]_1 \\
M_1 &= \emptyset, R_1 = \emptyset, \rho_1 = \emptyset \\
M_2 &= \emptyset, R_2 = \{b' \rightarrow b, b \rightarrow b(c, in_4), r_1 : ff \rightarrow af, r_2 : f \rightarrow a\delta\}, \rho_2 = \{r_1 > r_2\} \\
M_3 &= \{af\}, R_3 = \{a \rightarrow ab', a \rightarrow b'\delta, f \rightarrow ff\}, \rho_3 = \emptyset \\
M_4 &= \emptyset, R_4 = \emptyset, \rho_4 = \emptyset
\end{aligned}
$$

where the only membrane that initially contains any objects is M_3. In M_3 are two objects, a and f. f only matches one rule, $f \rightarrow ff$, which causes the number of fs to double on each evolution. For a, there are two rules and one is chosen

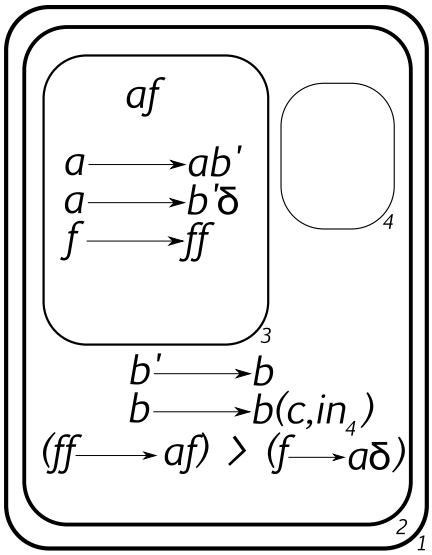


Figure 4.4: Example P System

non-deterministically. If the first, $a \rightarrow ab'$, is applied, then an additional object b' appears, and the rule may be applied again as an a is still present. If $a \rightarrow ab'$ and $f \rightarrow ff$ are applied for n steps, then n instances of b' and 2^n occurrences of f are present.

If the second a rule $a \rightarrow b'\delta$, is applied, the δ causes the membrane, M_3, to be dissolved. At this point, there will be one extra b' and one extra f resulting from the application of this rule and $f \rightarrow ff$, respectively, and no a. This changes the configuration of the system to become:

$$\begin{aligned}
\mu &= [_1[_2[_4]_4]_2]_1 \\
M_1 &= \emptyset, R_1 = \emptyset, \rho_1 = \emptyset \\
M_2 &= \{b'^{n+1}, f^{2n+1}\} \\
R_2 &= \{b' \rightarrow b, b \rightarrow b(c, in_4), r_1 : ff \rightarrow af, r_2 : f \rightarrow a\delta\}, \rho_2 = \{r_1 > r_2\} \\
M_4 &= \emptyset, R_4 = \emptyset, \rho_4 = \emptyset
\end{aligned}$$

The three rules that were present in M_3 are lost, while the objects float into the membrane above, M_2. In this configuration, n represents the number of times the pair of rules $a \rightarrow ab'$ and $f \rightarrow ff$ were applied prior to this, and is greater than or equal to zero.

In M_2, a priority relation exists that forces $ff \rightarrow af$ to be given precedence over $f \rightarrow a\delta$. As a result, whenever it is possible to apply $ff \rightarrow af$ (i.e. there are two f objects), it will be applied instead of $f \rightarrow a\delta$. The other two rules manipulate the b' objects. First, they are all converted in to b objects. This will always occur, as

there are at least two f objects in M_2 to begin with, which means $ff \rightarrow af$ will be applied rather than $f \rightarrow a\delta$ which destroys M_2. Each time $ff \rightarrow af$ is applied, the number of f objects halves.

The remaining rule, $b \rightarrow b(c, in_4)$, will evolve once for each occurrence of ff, of which there are n. M_2 contains $n+1$ b objects, all converted from the b' objects that were in M_3. As long as there is an even number of f objects, the two rules $b \rightarrow b(c, in_4)$ and $ff \rightarrow af$ will be applied, halving the number of f objects and creating $n+1$ c objects in M_4 (via (c, in_4)), while the number of b objects remains the same.

When only one f object is left, $f \rightarrow a\delta$ will be applied, resulting in M_2 being destroyed and the following configuration:

$$\begin{aligned} \mu &= [_1[_4]_4]_1 \\ M_1 &= \{a^{2n+1}, b^{n+1}\}, R_1 = \emptyset, \rho_1 = \emptyset \\ M_4 &= \{c^{(n+1)^2}\}, R_4 = \emptyset, \rho_4 = \emptyset \end{aligned}$$

No further evolution is possible, as there are no more rules. $c^{(n+1)^2}$ is the final output, as M_4 is the output membrane.

Further variants of P systems exist. Tissue P systems use a graph-based structure rather than the tree shown here, while population P systems also incorporate an environment and use a dynamic graph as an associated structure of it. There are also timed variants of P systems; rules are usually assumed to apply instantaneously, but in the timed variants they can take a specific duration.

4.4 Comparing Modelling Approaches

Biological systems are inherently concurrent, being focused on the behaviour of multiple entities from low-level molecules, through bacteria and other bodies, to full cellular structures and beyond. Models which incorporate spatial distribution, such as the ambient calculus (4.3.2) and P systems (4.3.5) are especially useful for representing the structure of real-world biological entities.

Such modelling is becoming commonplace within the literature (Regev, Silverman & Shapiro, 2001; Regev, Panina, Silverman, Cardelli & Shapiro, 2004; Pérez-Jiménez & Romero-Campero, 2006), where concurrent models represent an alternative to the use of ordinary differential equations (ODEs). The usual approach is to create a model of the system within the formalism and then perform simulations. Such simulations rely on reducing the non-determinism within the model by introducing a stochastic semantics. In each of the biochemical stochastic π calculus (Regev et al., 2001), the BioAmbient variant (Regev et al., 2004) and P systems

(Pérez-Jiménez & Romero-Campero, 2006), these are based on Gillespie's algorithm (Gillespie, 1977).

The algorithm selects which reaction occurs next and the necessary advancement of the system's 'clock' (a real time value in this context, rather than some discrete notion). A probability is associated with each reaction, so that the algorithm basically runs as follows:

1. a_0 is calculated as the sum of the probabilities.
2. Two random numbers, r_1 and r_2, are generated from a uniform distribution over the unit interval 0 to 1.
3. Calculate the waiting time for the next reaction, $\tau_i = \frac{1}{a_0}\ln(\frac{1}{r_1})$
4. Take the index, j, of the reaction such that $\sum_{k=1}^{j-1} p_k < r_2 a_0 \leq \sum_{k=1}^{j} p_k$ where p_k is the kth probability.
5. Return the pair (τ_i, j)

determining which one occurs. Slight alterations are made in distributed models to handle the rules arising from different localities. For example, the P systems model (Pérez-Jiménez & Romero-Campero, 2006) adapts the algorithm to form a multi-compartmental variant, which treats each membrane separately, to a degree, while also taking into account that activity in one membrane may affect others.

Clearly, different formalisms offer different approaches. In the original π calculus approach of (Regev et al., 2001), the focus was solely on communication with biological compartments abstracted as private channels. The model given for BioAmbients (Regev et al., 2004) is more natural due to the explicit realisation of these compartments.

Take the following example from (Regev et al., 2004),

$$\begin{aligned} \texttt{System} &::= \texttt{molecule}[\texttt{Mol}] \mid \ldots \mid \texttt{molecule}[\texttt{Mol}] \mid \texttt{cell}[\texttt{Porin}] \\ \texttt{Mol} &::= \texttt{enter cell1.Mol} + \texttt{exit cell2.Mol} \\ \texttt{Porin} &::= \texttt{accept cell1.Porin} + \texttt{expel cell2.Porin} \end{aligned} \tag{4.62}$$

which demonstrates a membranal pore, which molecules use to pass through a membrane. Both the cell and the molecules are represented by ambients. Each molecule is controlled by a process, Mol, which, at any time, has the option of performing either an `enter` or an `exit`. Similarly, the $Porin$ process, which represents the membranal pore, may `accept` or `expel`.

Within the BioAmbient calculus, movement is synchronous and takes place by the pairing of an `enter` and `accept` action (the equivalent of *in*) or an `exit` and `expel` action (equivalent to *out*). The first action in each case is used by the moving process. Both must also mention the same channel name (`cell1` and `cell2` here). In the case of the system shown above, both `Mol` and `Porin` permanently offer their halves of this pairing. However, the spatial context makes one of them inapplicable. Initially, `exit` and `expel` won't synchronise, as `Mol` is not inside the ambient from which it is being expelled. Likewise, once it has entered, it can't do so again, even though the actions make this possible.

Models such as this seem a little unnatural as molecules are modelled as both an ambient and a process. This is because only ambients may move but only processes can emit the necessary mobility primitives to do so. The notions of mobility present in the ambient calculus, including this idea, have been carried across, even though it doesn't directly adopt the primitives of the ambient calculus; the style is still more akin to the π calculus.

In contrast, (Pérez-Jiménez & Romero-Campero, 2006) takes a different approach using P systems, representing signals and proteins directly as objects in the membranes. One particular application of this technique is *quorum sensing*. This is a gene regulation system where a population of bacterial cells communicate in order to regulate the expression of certain genes in a co-ordinated way which is dependent on the size of the population. (Pérez-Jiménez & Romero-Campero, 2006) presents a model of this phenomenon in *vibrio fischeri*, a marine bacterium, using a P system[10]:

$$\begin{aligned}
\Pi_{vf} &= (O, \{e, b\}, \mu, (w_1, e), (w_2, b), \ldots, (w_{n+1}, b), \mathcal{R}_b, \mathcal{R}_e) \\
O &= \{OHHL, LuxR, LuxR\text{-}OHHL, LuxBox, LuxBox\text{-}LuxR\text{-}OHHL\} \\
w_1 &= \emptyset \\
w_i &= \{LuxBox\} \text{ where } 2 \leq i \leq n+1
\end{aligned}$$

where each bacteria is represented as a membrane, b, within an environment membrane, e. The alphabet, O, contains the signal, *OHHL*, the protein, *LuxR* and the regulatory region, *LuxBox*, in addition to the protein-signal complex (*LuxR-OHHL*) formed and its regulatory region, *LuxBox-LuxR-OHHL*. The initial configuration shown above leaves the environment empty and places just the genome, *LuxBox*, inside each bacteria membrane to start production of the signal and the protein. $\mathcal{R}_b$ and $\mathcal{R}_e$ contain the rules which affect the bacteria and the environment respectively. The reader is referred to the full paper for full details of these.

[10]This method of defining the configuration differs slightly from that in 4.3.5, as it also includes a set of labels, rather than assuming that the natural numbers are used.

There are similarities between process calculi and P systems, even though they arose from two different communities. Many of the same techniques and properties have been proven to apply to both, and it is possible to translate a P system into a process calculus form to take advantage of model checking techniques originally designed for a process calculus. The difference in their backgrounds also results in some interesting differences in their design and the way they are used; P systems arise from language generation and thus the rules are applied in parallel, while on the other hand, process calculi are usually based around transitions and communication. The choice of which one is best to use really depends upon the use to which it is being put; P systems make sense when there is structure inherent in the model and the various process algebraic techniques lend themselves to situations with a large amount of communication that needs to be monitored.

4.5 Bigraphs

Bigraphs (Jensen & Milner, 2004; Milner, 2005) are an attempt at providing a unifying framework, able to represent both spatial relationships (*locality*) in the style of the ambient calculus (see 4.3.2) and link-based-relationships (*connectivity*) seen in the π calculus (see 4.2.1). Their particular application area is within pervasive computing, where a mixture of both concepts is needed to represent both movement through space and the change in relationships between agents.

The nodes in a bigraph support a dual structure, hence the name. On one level, there are nodes nested within nodes, representing locality. This is called the *place graph*. These nodes have *ports* which are connected via links to form a *link graph*. Each node has a *control* with an arity that defines the number of ports. The two graphs share nodes, but are otherwise independent. Nesting can only occur in nodes with a *non-atomic* control. These can also be *active* or *passive*. The former allows reactions to occur within the node. *Holes* may occur in bigraphs, where other bigraphs can appear.

Within this model, it is possible to encode both the π calculus and the ambient calculus. Take the following rule from the asynchronous π calculus without summation,

$$\overline{x}y \mid x(z).P \rightarrow P\{y/z\} \tag{4.63}$$

which represents synchronisation. In (Jensen & Milner, 2004), Milner encodes this as a bigraph with two controls, *send* and *get*, both of which have an arity of two. To represent the fact that the output prefix has no continuation in the asynchronous π calculus, *send* is declared atomic. *get* is non-atomic but inactive.

The node *get* includes a nested hole with the port z. This represents the continuation P, with z being the name bound on input. The port z is linked from the hole

to *get* itself. *send* has two ports: x, which is also connected to *get*, and y. With these concepts in place, the reaction may be represented as:

$$send_{xy} \mid get_{x(z)}\Box \rightarrow x \mid y/(z)\Box \tag{4.64}$$

the *send* node disappearing afterwards, leaving y connected to z and x unused.

Similarly, (Jensen & Milner, 2004) shows how the *in* capability from the ambient calculus:

$$n[in\ m.P_1 \mid P_2] \mid m[Q] \rightarrow m[n[P_1 \mid P_2] \mid Q] \tag{4.65}$$

may be encoded using two controls, *amb* and *in*, both with an arity of one. The two ambients involved are represented by instances of *amb*, while *in* is an atomic control representing the process that emits the capability. The *amb* control is non-atomic and active, each ambient containing a hole which represents their continued behaviour,

The ambient names are represented as the node's single port. In the case of the ambient named in the capability, this is also linked to the *in* instance. To model the reaction above, n is connected to the port of one *amb*, while m is connected to both the other *amb* and *in*. The reaction is then encoded as:

$$amb_n(in_m \mid \Box_0) \mid amb_m\Box_1 \rightarrow amb_m(amb_n\Box_0 \mid \Box_1) \tag{4.66}$$

where the similarities between the two are clear.

Bigraphs provide an interesting framework for unifying the two disparate concepts outlined above in 4.2 and 4.3. It will be exciting to see how this theory develops, and whether it can also be used to encode the discrete time notions described in chapter 3.

4.6 Conclusion

The π calculus seems to provide the best of both worlds, being able to model concurrent systems and still retain the expressiveness of the λ calculus. However, a key limitation was identified which reinforced the idea that expressivity only makes a model capable, and not suitable, for simulating any recursive function: modelling global synchronisation via a broadcasting agent. This limitation seems to hold for both CCS and the π calculus, and it is also likely that it applies to many other process calculi, such as the ambient calculus, a formalism that provides a more natural form of mobility via structural changes.

Biology was also considered briefly (see 4.4), as a motivating paradigm. P systems seem the most natural formalism in this area and there are clear parallels between P systems and ambient calculi, especially in their use of structure. In

the next chapter, we take the general concepts of structural change and migrating objects present in both, and combine them with CaSE to create a new calculus; Nomadic Time.

Chapter 5

Nomadic Time

5.1 Introduction

This chapter introduces the process calculus used as the basis for DynamiTE, Nomadic Time, in which timed processes are mobile. In the preceding chapters, we have looked at a number of existing calculi and the features of the main ones[1] are summarised in Table 5.1. *Synchronisation* refers to the single transition arising from the pairing of a name and a co-name running in parallel, which is fundamental to CCS (see 2.2) and is retained in TPL, CaSE, the π calculus and our own Nomadic Time. It is usually used to represent sender-receiver communication between two processes. *Timeouts* were introduced in our discussion of TPL in 3.2 and solved the problem of providing a compositional broadcast agent. *Multiple clocks* and *clock hiding* allow this feature to scale up to larger systems.

Scope mobility is a concept introduced by the π calculus (see 4.2.1), allowing processes to gain knowledge of communication channels during execution, thus increasing their scope. *Localities* were discussed in 4.3 and allow distribution to be modelled. *Location mobility* refers to the movement of localities, including all contents, as provided by the *in* m and *out* m operations of the ambient calculus from 4.3.2. *Destruction* refers to the ability to destroy a locality. This can be seen in the use of *open* m in the ambient calculus or the dissolution of a membrane in P Systems (see 4.3.5). *Co-migration* refers to the need for a mobility primitive to synchronise with a corresponding co-mobility primitive, as in the *in* m and $\overline{in}$ m pairs found in the safe ambients calculus (see 4.3.3). Finally, *process mobility* refers to the ability to directly move a process from one locality to another, without changing the

[1] We miss out some covered earlier due to space restrictions, but it should be clear how these fit into the matrix. CSA bridges the gap between TPL and CaSE with timeouts and multiple clocks, but no clock hiding. The join calculus has the same features as the π calculus, while the distributed join calculus adds localities to the feature set. The boxed ambients calculus is the same as the ambient calculus, but without destruction; it has directed communication instead.

Table 5.1: Feature Summary from the Literature Survey

	CCS	TPL	CaSE	π	Amb.	Safe Amb.	P Systems	**NT**
Synchronisation	√	√	√	√				√
Timeouts		√	√					√
Multiple Clocks			√					√
Clock Hiding			√					√
Scope Mobility				√				
Localities					√	√	√	√
Location Mobility					√	√		√
Destruction					√	√	√	√
Co-Migration						√		√
Process Mobility							√	√

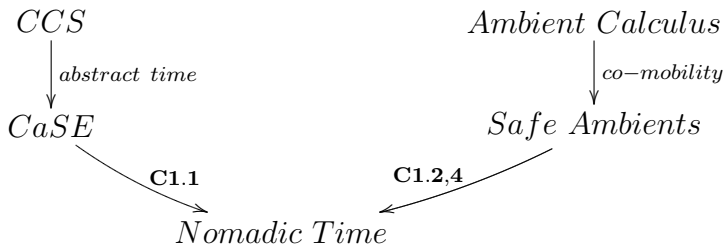


Figure 5.1: Derivation of Nomadic Time

structure of the localities. P Systems have something close to this in that an object can move between membranes, but these objects don't have any behaviour; the Dπ calculus (Riely & Hennessy, 1998) and the M^3 calculus (Coppo, Dezani-Ciancaglini, Giovannetti & Salvo, 2003), derivatives of the π calculus and the ambient calculus respectively, provide examples of the same idea being applied to processes.

As can be seen from the table, Nomadic Time (**NT**) has the features of both CaSE and the safe ambients calculus, from which it draws its inspiration[2]. This is illustrated in Figure 5.1, which also shows how our creation of Nomadic Time fits in with our contributions to knowledge. Over the first half of this chapter, we go through this process in detail. CaSE (Norton et al., 2003) (itself a timed

[2]The only missing feature of Nomadic Time, according to the table, is scope mobility; this could be added in just the same way CCS was extended to create the π calculus but it would be superfluous, given we already have a form of mobility based on localities.

derivative of CCS) is first extended with localities (see 5.2), merging them with clock hiding to give contribution **C1.1**. We then introduce location mobility in the style of the ambient calculus (see 5.3.1), as well as a form of objective process mobility which reuses the concept of name-based synchronisation from CCS and CaSE; this gives contributions **C1.2** and **C1.3**. At this stage, we have a calculus which provides both the ability to migrate processes during execution and to perform global synchronisation in a compositional manner. Finally, we introduce the concept of co-mobility primitives from the safe ambients calculus, but rather than applying these to normal processes, we introduce 'bouncers' (see 5.4), which guard a particular locality (an *environ* in Nomadic Time). This effectively gives each environ a security policy, allowing the migration primitives which effect it to be restricted; the bouncers define which mobility actions may be performed and how many times. This addition forms contribution **C1.4**.

The second half of this chapter contains the operational semantics of the calculus, including a set of structural congruence rules (see 5.5); these form contributions **C1.5** and **C1.6**. We also demonstrate how the properties of prioritisation and time determinacy are provided by the semantics; this is contribution **C1.7**. Finally, we close with a comprehensive example (see 5.6) demonstrating both the mobility and global synchronisation features of the calculus, and the application of the calculus to our prototypical application (see 5.7).

5.2 Localising the Calculus

Localisation, discussed in detail in 4.3, effectively adds another level of grouping to the calculus. A set of composed processes may be contained within one *locality*, a notion which is often used in the modelling of *distribution*. This idea, which can be taken to its logical conclusion by forming a hierarchy of such localities, has echoes of the notion of *clock hiding* within CaSE, as described in 3.4.

Thus, the first step in the evolution towards NT is to combine these two hierarchical concepts by effectively localising CaSE. The notion of components and encapsulation is explicitly realised by an *environ*, which also handles the hiding of clocks. As a result, the clock hiding operator from CaSE disappears, being replaced by a new operator which allows the creation of environs. The bounds of the environ define both a new group and the scope of the clock hiding. The syntax for localised CaSE is thus:

$$\begin{aligned} \mathcal{E}, \mathcal{F} \ ::= \ & \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ & \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid m[\mathcal{E}]_{\vec{\sigma}} \end{aligned} \tag{5.1}$$

where $\vec{\sigma}$ is a set of clocks and m represents an arbitrary environ name[3]. In particular,

[3]Note that although names are added to the environs here, this is not really necessary at this

m may be equal to the empty string, ϵ, thus facilitating the use of anonymous environs. This allows the semantics of CaSE's clock hiding to be encoded:

$$[\![E/\sigma]\!] \stackrel{\text{def}}{=} [E]_{\{\sigma\}} \tag{5.2}$$

thus making localised CaSE a conservative extension. The environs form a forest structure, due to the ability to nest environs to an arbitrary depth and the possibility of multiple environs occurring at the top level.

Recall the example of clock hiding given in (3.11):

$$(P/\sigma) \mid Q \tag{5.3}$$

This becomes:

$$[P]_{\{\sigma\}} \mid Q \tag{5.4}$$

in localised CaSE, or:

$$m[P]_{\{\sigma\}} \mid Q \tag{5.5}$$

if an arbitrary name, m, is assigned to the environ. Just as with the clock hiding operator, the clock σ is hidden outside the environ, m, causing its ticks to be visible only to P.

With this extension the set of visible clocks for a particular environ may be obtained by finding the set difference between $\mathcal{T}$ and the union of the sets of clocks of its child environs. For example, consider the more complex scenario:

$$n[E \mid m[F \mid k[G]_{\{\sigma\}}]_{\{\rho\}}]_{\{\gamma\}} \tag{5.6}$$

where the top-level environ, n, contains a process E and a further sub-environ, m. Likewise, m contains both a process, F, and the sub-environ, k. Finally, k contains just the single process, G. The set of clocks for the environ k is $\{\sigma\}$ and its parents are m (with the set $\{\rho\}$) and n (with $\{\gamma\}$). Thus, the set of visible clocks for k is $\mathcal{T}\backslash\emptyset$, as it has no child environs. which means that G, located in k, can see the ticks of all clocks.

F, by comparison, can only see the ticks of the clocks in $\mathcal{T}\backslash\{\sigma\}$ as σ is hidden outside k. E, in the top-level environ, n, can only observe silent actions resulting from the two hidden clocks, ρ and σ, but can see the ticks of γ and any other clocks in $\mathcal{T}$, its set of visible clocks being $\mathcal{T}\backslash\{\sigma, \rho\}$.

This is illustrated in Figure 5.2 which shows the scope of each clock. The smallest circle represent the scope of σ, and contains just the process G. The middle circle represents the scope of ρ and incorporates F and the environ k in which G resides. The largest circle represents the scope of γ, which covers the entire system. Within each circle, the ticks of the respective clock are visible. Outside it, they become silent actions.

---

stage; they provide nothing more than a way to refer to environs in talking about a system. However, they are necessary for providing migration as discussed in 4.3

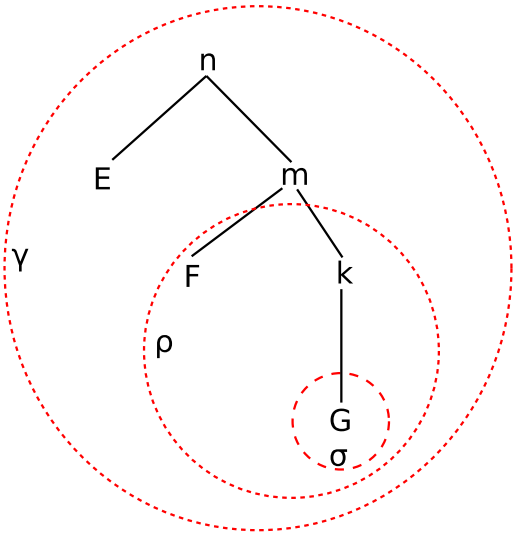


Figure 5.2: Clock Hiding in $n[E \mid m[F \mid k[G]_{\{\sigma\}}]_{\{\rho\}}]_{\{\gamma\}}$

5.3 Adding Mobility

Localised CaSE makes the notion of components and encapsulation clearer than in the original calculus, by allowing them to be given explicit names. However, it doesn't provide a great deal of extra functionality[4]. The most natural progression from this stage is to add mobility. For this, the primitives of the ambient calculus are adopted, as they provide a very natural and simplistic formalism, which builds on the component-oriented nature of the calculus, now explicitly realised by environs. This is shown in more detail in 5.3.1.

In addition, NT allows the movement of individual processes. In the ambient calculus, only ambients can move, which restricts the separability of processes. For a given group of processes, the size of the group may only change by:

1. One of the processes becoming **0**. Both NT and the ambient calculus include a structural congruence law,
$$E \mid \mathbf{0} \equiv E \tag{5.7}$$
which allows such processes to be removed.

2. The process splitting into two or more processes via parallel composition. For example, *in* $m.(E|F)$ enters the ambient, m, and then splits into two separate processes, E and F.

[4]Although the semantics could be adapted so as to use the environs for bisimulation, as in 4.3.

3. Another process *open*ing the ambient, causing the set of processes to merge with those in the parent.

What the ambient calculus doesn't allow is for a selected process or group of processes to be moved from one ambient to another. That process or group must be in its own ambient for this to happen.

Take the example process,

$$m[E \mid F \mid G] \mid n[\mathbf{0}] \mid H \tag{5.8}$$

where E, F, G and H are all processes and m and n are ambients. The topology of this process may change in several ways, as outlined above. Any of the four processes might evolve to $\mathbf{0}$, or fork into two or more processes. In addition, E, F or G may emit an *in* n capability, causing the ambient m to move inside n. Similarly, H may perform an *open* m, causing m to be removed and the top-level to include all four processes.

So, several events may occur but there are also some that are intuitive, but difficult to achieve. For instance, all three processes in m must move as a unit, whether this is to the top-level due to an *open* capability or as a result of m moving in to n. Moving one process, E for example, requires the interaction of both E itself and another process at the final destination.

To move E to the top level on its own requires converting it to the form,

$$Emov \stackrel{\text{def}}{=} z[out\ m.E] \tag{5.9}$$

where z is a new name, which doesn't occur free in either E, F, G or H. The effect is clearer when this is placed in context,

$$m[z[out\ m.E] \mid F \mid G] \mid n[\mathbf{0}] \mid H \tag{5.10}$$

where it can be clearly seen that the new capability prefixed on E will cause the new surrounding ambient, z, to move outside of m. To actually have E at the top-level, and not E nested in an ambient, requires the presence of a top-level process to open the z ambient. This results in something along the lines of:

$$m[z[out\ m.E] \mid F \mid G] \mid n[\mathbf{0}] \mid H \mid open\ z.\mathbf{0} \tag{5.11}$$

to truly encode the movement of E alone. Moving just E into n is even more convoluted:

$$m[z[out\ m.in\ n.E] \mid F \mid G] \mid n[open\ z.\mathbf{0}] \mid H \tag{5.12}$$

and neither are particularly natural. NT instead provides this functionality as a base part of the syntax, which will be explored in 5.3.2.

Finally, it should be noted that the scope of an action is implicitly restricted to the bounds of an environ within NT. For instance, in the following process:

$$a.P \mid m[\overline{a}.Q]_{\{\sigma\}} \tag{5.13}$$

synchronisation between the two processes is not permitted as they lie on either side of an environ boundary. This is not an issue, as the presence of mobility allows processes to move into a situation where the co-action is in scope. In addition, NT (at present) does not incorporate the scoping of environ names; see 8.2.1.

5.3.1 Location Mobility

To add an ambient calculus style of mobility, the existing syntax of localised CaSE is extended with a mobility prefix, $\mathcal{M}.\mathcal{E}$, to give:

$$\begin{array}{c}\mathcal{E},\mathcal{F} ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E}+\mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor\mathcal{E}\rfloor\sigma(\mathcal{F}) \mid \\ \lceil\mathcal{E}\rceil\sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid m[\mathcal{E}]_{\vec{\sigma}} \mid \mathcal{M}.\mathcal{E}\end{array} \tag{5.14}$$

where $\mathcal{M}$ is further defined as:

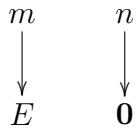
$$\mathcal{M} ::= \circledv m \mid \circledwedge m \mid \circledast m \tag{5.15}$$

with m again representing the name of an environ. The behaviour of these primitives is identical to the behaviour of their equivalents in the ambient calculus ($\circledv m$ being *in* m, $\circledwedge m$ being *out* m and $\circledast m$ being *open* m)[5], so just a short recap of section 4.3.2 is given here, using the syntax above.

When a process emits an $\circledv m$ capability, the surrounding environ may move into a sibling environ with the name, m. Given the context,

$$m[E]_\emptyset \mid n[\mathbf{0}]_\emptyset \tag{5.16}$$

or, diagrammatically,



if E is defined as

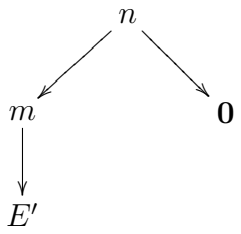
$$E \stackrel{\text{def}}{=} \circledv n.E' \tag{5.17}$$

then this will allow the derivation

$$m[\circledv n.E' \mid n[\mathbf{0}]_\emptyset]_\emptyset \stackrel{\circledv n}{\rightarrow} n[m[E']_\emptyset \mid \mathbf{0}]_\emptyset \tag{5.18}$$

[5]The mnemonics $\circledv m$, $\circledwedge m$ and $\circledast m$ are used to prevent confusion with the names of actions.

to occur, giving



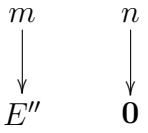
Similarly, defining E' to be

$$E' \stackrel{\text{def}}{=} \oslash n.E'' \tag{5.19}$$

allows the converse

$$n[m[\oslash n.E'' \mid \mathbf{0}]_\emptyset]_\emptyset \stackrel{\oslash n}{\rightarrow} m[E'']_\emptyset \mid n[\mathbf{0}]_\emptyset \tag{5.20}$$

to take place, returning us to

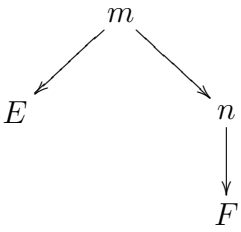


as $\oslash n$ allows the surrounding environ to move outside a parent environ named n. As noted above, these are fairly dull, both being identical to the same primitives in the ambient calculus. The behaviour of $\circledast m$ is more interesting, due to its interaction with the environ's clock environment.

Take the example context,

$$m[E \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \tag{5.21}$$

or, diagrammatically,



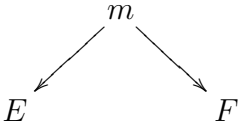
where E is defined as

$$E \stackrel{\text{def}}{=} \circledast n.E' \tag{5.22}$$

and thus may cause the environ, n, to be destroyed

$$m[\circledast n.E' \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \stackrel{\circledast n}{\rightarrow} m[E' \mid F]_{\{\sigma,\rho\}} \tag{5.23}$$

giving

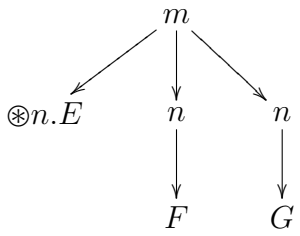


and causing the two clock environments to merge. As a result, not only does the context of F change with respect to nearby processes, as in the ambient calculus, but now E is also affected. Prior to the emission of $\circledast n$, E could only see ticks from the clock ρ. The ticks of σ were converted to silent actions by the environ barrier. Following the dissolution of the environ, n, these ticks become visible to E. So, the *open* capability in NT not only changes the environ hierarchy, but also the clock context within the parent environ.

Just as in the ambient calculus, the reduction of capabilities is subject to the availability of applicable environs, thus allowing for stalled capabilities (when there are none) and non-determinism (when there are several). For example, the process

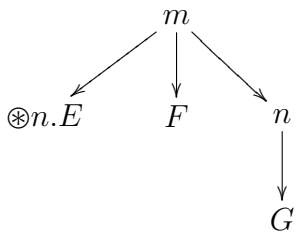
$$m[\circledast n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \tag{5.24}$$

or, diagrammatically,

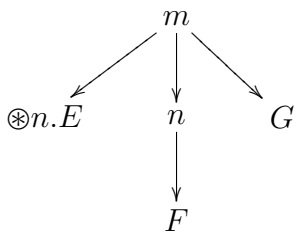


has two possible derivations

1. $m[\circledast n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \overset{\circledast n}{\rightarrow} m[E \mid F \mid n[G]_{\{\gamma\}}]_{\{\sigma,\rho\}}$



2. $m[\circledast n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \overset{\circledast n}{\rightarrow} m[E \mid n[F]_{\{\rho\}} \mid G]_{\{\gamma,\rho\}}$



and, as a result, two different resulting clock contexts. In the full calculus, this non-determinism is restricted by the notion of *bouncers*, introduced in section 5.4, which reduce the possibility of *grave interferences* (see 4.3.3).

5.3.2 Process Mobility

In NT, the mobility prefix is further extended as follows:

$$\mathcal{M} ::= \circledcirc m \mid \circledast m \mid \circledast m \mid on\ \beta \circledcirc m \mid on\ \beta \circledast m \tag{5.25}$$

where $\beta \in \mathcal{N}$ refers to an action. While the location mobility described above is *subjective* (the process requesting the move does the move), process mobility, in this form, is *objective*. The process which emits one of the two new capabilities synchronises with a partner process on the given action, and it is this partner which actually moves. The partner will be a process in the same environ, due to the scoping of actions described above.

Such behaviour is initially difficult to understand, but can be made clearer with a simple example. Take the process,

$$on\ go \circledcirc m.E \mid go.F \mid m[\mathbf{0}]_{\{\sigma\}} \tag{5.26}$$

where E is emitting the capability $on\ go \circledcirc m$, but it is $go.F$ that will actually move,

$$on\ go \circledcirc m.E \mid go.F \mid m[\mathbf{0}]_{\{\sigma\}} \overset{on\ go \circledcirc m}{\rightarrow} E \mid m[F \mid \mathbf{0}]_{\{\sigma\}} \tag{5.27}$$

with the continuation, F, continuing to evolve in the environ m.

Encoding process mobility in this objective form doesn't prevent it from being used to perform subjective movement. As processes can fork, a process that wishes to move can evolve into a situation where it is composed in parallel with a new process that exhibits the required capability. To demonstrate the converse action, *out*, in the scenario above, F can be defined as

$$F \overset{\text{def}}{=} leave.F' \mid on\ leave \circledast m \tag{5.28}$$

where the process on the right moves the one on the left outside m. In context, this performs as follows:

$$E \mid m[leave.F' \mid on\ leave \circledast m.\mathbf{0}]_{\{\sigma\}} \overset{on\ leave \circledast m}{\rightarrow} E \mid F' \mid m[\mathbf{0}]_{\{\sigma\}} \tag{5.29}$$

to give a final process which is very similar to the original.

More generally, a subjective process movement may be encoded as

$$[\![\circledcirc m\ E.F]\!] \overset{\text{def}}{=} z.E \mid on\ z \circledcirc m.F \tag{5.30}$$

where e is the process that will move in to m, F is the continuation and z is a new name. The converse is pretty much the same:

$$[\![\circledast m\ E.F]\!] \overset{\text{def}}{=} z.E \mid on\ z \circledast m.F \tag{5.31}$$

5.4 Bouncers

This description of NT is concluded by the addition of the final element, the *bouncers*. Named after the staff who restrict access to a night club[6], the bouncer is an additional property of an environ which appears in the top right of the expression. It has no real behaviour of its own, but instead performs the job of protecting the environ, being a process with a limited choice of available constructs[7]. The bouncer provides a structured selection of co-primitives ($\overline{\circledcirc}$, $\overline{\oslash}$ and $\overline{\circledast}$), similar to those in (Levi & Sangiorgi, 2003) (see section 4.3.3) and dictates which mobility transitions may occur, and when.

The full syntax of NT may now be given as:

$$\begin{array}{rcl} \mathcal{E}, \mathcal{F} & ::= & \mathbf{0} \mid \Omega \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid \mathcal{E} \,|\, \mathcal{F} \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ & & \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus A \mid m[\mathcal{E}]^{\mathcal{F}}_{\sigma} \mid \mathcal{M}.\mathcal{E} \\ \mathcal{M} & ::= & \circledcirc\, m \mid \oslash m \mid \circledast m \mid on\ \beta\, \circledcirc\, m \mid on\ \beta\, \oslash\, m \mid \overline{\circledcirc} \mid \overline{\oslash} \mid \overline{\circledast} \end{array} \tag{5.32}$$

with Ω representing the bouncer with no behaviour (the equivalent of $\mathbf{0}$). For a process or environ to enter another environ, its bouncer must allow this to occur by providing the corresponding $\overline{\circledcirc}$ co-capability. Likewise, it must provide $\overline{\oslash}$ to allow a process or environ to leave. With regard to the destruction of an environ, the environ's bouncer must allow it to be removed by providing a $\overline{\circledast}$ co-capability.

Recall the example given in 5.3.1.

$$m[\circledcirc n.E'] \mid n[\mathbf{0}] \tag{5.33}$$

With the addition of bouncers, this becomes:

$$m[\circledcirc n.E']^{\Omega} \mid n[\mathbf{0}]^{\overline{\circledcirc}.\Omega} \tag{5.34}$$

where, again, a syntactic abbreviation of $m[E]^F$ for $m[E]^F_{\{\}}$ is used when the clock context is empty. The environ m has Ω as its bouncer, as no movement affects it[8]. The bouncer for n is defined as $\overline{\circledcirc}.\Omega$, which allows the movement of m into n to occur:

$$m[\circledcirc n.E']^{\Omega} \mid n[\mathbf{0}]^{\overline{\circledcirc}.\Omega} \xrightarrow{\circledcirc} n[m[E']^{\Omega} \mid \mathbf{0}]^{\Omega} \tag{5.35}$$

but any subsequent behaviour is disallowed, as the bouncer of n has now evolved to also be Ω. Note that the transition is labelled with a $\circledcirc$ to represent the synchronisation. This is a member of a set of *high priority transitions* (denoted $\mathcal{H}$),

[6] American usage: doorman/woman.

[7] This limited choice is only explicitly imposed by the type system. There is no restriction in the abstract syntax.

[8] In contrast, the most permissive bouncer is $\mu X.(\overline{\circledcirc}.X + \overline{\oslash}.X + \overline{\circledast}.X)$ which always allows any movement to occur.

which includes τ and the mobility transitions, $\circledcirc$, $\circledcirc$ and $\circledast$. If a process may emit a transition in $\mathcal{H}$, then low-priority transitions are prevented from occurring. This also applies in CaSE, where $\mathcal{H}$ is simply $\{\tau\}$.

There is a distinct advantage to using a high priority label here. It allows movements to be treated in the same way as synchronisations (which emit τ), so that they also form part of the synchronous clock cycles, via *maximal progress*, allowing them to be used for broadcasting in the same compositional style demonstrated in chapter 3 for actions. This notion is central to the example presented in section 5.6. In addition, generalising to a set of such labels rather than simply using τ throughout means will can still distinguish between synchronisations and movements.

Using bouncers, it becomes possible to specify how many entities (processes or environs) may enter or leave an environ. As bouncers can recurse, the number can be unlimited. For example, the bouncer:

$$\mu X.\overline{\circledcirc}.\overline{\circledcirc}.\overline{\circledcirc}.\overline{\circledcirc}.X \tag{5.36}$$

allows two entities to enter, but two must then leave before another can enter. On the subject of exiting an environ, the synchronisation with $\overline{\circledcirc}$ works in the same way as $\overline{\circledcirc}$:

$$n[m[\circledcirc n.E'']^{\Omega} \mid \mathbf{0}]^{\overline{\circledcirc}.\Omega} \xrightarrow{\circledcirc} m[E'']^{\Omega} \mid n[\mathbf{0}]^{\Omega} \tag{5.37}$$

Finally, the destruction of an environ is probably the easiest of the three to understand. Again, using an example from 5.3.1,

$$m[\circledast n.E' \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \tag{5.38}$$

it may be endowed with bouncers to give:

$$m[\circledast n.E' \mid n[F]^{\overline{\circledast}.\Omega}_{\{\sigma\}}]^{\Omega}_{\{\rho\}} \tag{5.39}$$

This allows the following synchronisation to occur:

$$m[\circledast n.E' \mid n[F]^{\overline{\circledast}.\Omega}_{\{\sigma\}}]^{\Omega}_{\{\rho\}} \xrightarrow{\circledast} m[E' \mid F]^{\Omega}_{\{\sigma,\rho\}} \tag{5.40}$$

in which the clock contexts merge, the actions of F become available to E and the bouncer of n disappears along with n itself.

5.5 The Semantics

This section gives NT an operational semantics in terms of a labelled transition system, $(\mathcal{P}, \mathcal{L}, \rightarrow)$, defined up to structural congruence. $\mathcal{P}$ is the set of NT expressions; $\mathcal{L}$ the alphabet comprising actions, clocks and mobility primitives; and $\rightarrow$ the transition relation. Transitions with labels in $\mathcal{A}$ are known as *action transitions*, those in

$\mathcal{T}$ as *clock transitions* and those in $\{\circledcirc, \circledast, \circledast\}$ as *mobility transitions*. The transition relation, $\rightarrow \subseteq \mathcal{P} \times \mathcal{L} \times \mathcal{P}$ is defined in Tables 5.2 and 5.3. We use E, F and G to range over process terms; σ and ρ over the set of clocks ($\mathcal{T}$); α over the set of actions ($\mathcal{A}$); h over $\mathcal{H}$; a and b over $\mathcal{S} \stackrel{\text{def}}{=} (\mathcal{L} \setminus \{\tau\}) \cup \{\circledcirc m, \circledast m, \circledast m\} \cup \{on\ \beta \circledcirc m, on\ \beta \circledast m\}$; κ over $\mathcal{A} \cup \{\circledcirc, \circledast, \circledast\}$; γ over $\mathcal{A} \cup \mathcal{T} \cup \{\circledcirc, \circledast, \circledast\}$ and A ranges over a subset of $\mathcal{S}$.

Structural congruence is the least congruence relation that satisfies the laws given in Table 5.4, allowing structural rearrangement and simplification of process terms. A, B and C range over subsets of $\mathcal{S}$. Notably, the rules allow multiple restriction operators to be combined into a single set (StrResRes).

Table 5.2 shows the core operational semantics, some of which are derived from those of CaSE given in Table 3.1; the derivation is summarised in Table 5.5. *Idle* and *Patient* represent the progress of time over **0** and action prefixes respectively. *Act* allows an action to be performed, with an appropriately labelled transition, with the process continuing as E. *Stall* represents the stopping of a specific clock, σ, allowing transitions to occur for any other clock, ρ. All of these are taken directly from CaSE.

New to Nomadic Time is the rule $SCong$, which links the structural congruence rules to the labelled transition system, and the rules $Cap1$ and $Cap2$ which allow the mobility prefix, $\mathcal{M}$, to evolve. The former allows commutativity to be implied by the presence of structural congruence, so $Sum1$ and $Par1$ from CaSE are sufficient to describe the behaviour of the summation and parallel composition operators for single actions. $Sum2$ and $Par3$ represent the passage of time over these two operators. Note that time must be able to pass on both sides, and that the restriction $E \mid F \stackrel{h}{\nrightarrow}$ enforces prioritisation. Again, these are taken directly from CaSE but this time with minor changes to include the mobility primitives (by replacing α with κ) and to prevent the progress of time in the presence of any high priority transitions (h), not just τ ones.

$Par2$ encapsulates synchronisation; when one of the processes can perform an action and the other can perform the matching co-action, a silent action is performed and both evolve. $FTO1$ and $STO1$ are essentially identical, allowing the dissolution of the timeout via a tick of the associated clock, σ, on the provision that $E \stackrel{h}{\nrightarrow}$. The difference between the two timeouts is shown by $FTO2$, $STO2$ and $STO3$. $FTO2$ is a general rule for the fragile timeout, which allows E to be performed and the timeout removed on the occurrence of any transition other than the clock tick. For the stable timeout, the effect of clocks and actions are separated. According to $STO3$, clocks other than σ may tick, but the timeout stays in place. $STO2$ handles the removal of the stable timeout, due to an action performed by E. Again, these expand on the CaSE equivalents by including mobility primitives and high priority transitions as above.

Recursion is provided by *Rec*, which performs substitution of X for the body of

Table 5.2: Core Semantics

$$\text{Idle}\ \frac{-}{\mathbf{0} \xrightarrow{\sigma} \mathbf{0}} \qquad \text{Act}\ \frac{-}{\alpha.E \xrightarrow{\alpha} E}$$

$$\text{Patient}\ \frac{-}{a.E \xrightarrow{\sigma} a.E} \qquad \text{Stall}\ \frac{-}{\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma}\ \rho \neq \sigma$$

$$\text{Sum1}\ \frac{E \xrightarrow{\kappa} E'}{E + F \xrightarrow{\kappa} E'} \qquad \text{Par1}\ \frac{E \xrightarrow{\kappa} E'}{E \mid F \xrightarrow{\kappa} E' \mid F}$$

$$\text{Sum2}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F'}{E + F \xrightarrow{\sigma} E' + F'} \qquad \text{Par2}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\text{Par3}\ \frac{E \xrightarrow{\sigma} E', F \xrightarrow{\sigma} F', E \mid F \overset{h}{\not\to}}{E \mid F \xrightarrow{\sigma} E' \mid F'} \qquad \text{FTO1}\frac{E \overset{h}{\not\to}}{\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F}$$

$$\text{FTO2}\frac{E \xrightarrow{\gamma} E'}{\lfloor E \rfloor \sigma(F) \xrightarrow{\gamma} E'}\ \gamma \neq \sigma \qquad \text{STO1}\frac{E \overset{h}{\not\to}}{\lceil E \rceil \sigma(F) \xrightarrow{\sigma} F}$$

$$\text{STO2}\frac{E \xrightarrow{\kappa} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\kappa} E'} \qquad \text{STO3}\frac{E \xrightarrow{\rho} E', E \overset{h}{\not\to}}{\lceil E \rceil \sigma(F) \xrightarrow{\rho} \lceil E' \rceil \sigma(F)}\ \rho \neq \sigma$$

$$\text{Rec}\ \frac{E \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'\{\mu X.E/X\}} \qquad \text{Res}\ \frac{E \xrightarrow{\gamma} E'}{E \setminus A \xrightarrow{\gamma} E' \setminus A}\ \gamma \notin A$$

$$\text{LHd1}\frac{E \xrightarrow{\sigma} E'}{m[E]^B_{\vec{\sigma}} \xrightarrow{\tau} m[E']^B_{\vec{\sigma}}}\ \sigma \in \vec{\sigma} \qquad \text{LHd2}\frac{E \xrightarrow{h} E'}{m[E]^B_{\vec{\sigma}} \xrightarrow{h} m[E']^B_{\vec{\sigma}}}$$

$$\text{LHd3}\frac{E \xrightarrow{\rho} E', E \xrightarrow{\sigma}, E \overset{h}{\not\to}}{m[E]^B_{\vec{\sigma}} \xrightarrow{\rho} m[E']^B_{\vec{\sigma}}}\ \rho \notin \vec{\sigma}, \sigma \in \vec{\sigma} \qquad \text{Cap1}\ \frac{-}{\mathcal{M}.E \xrightarrow{\mathcal{M}} E}$$

$$\text{Cap2}\ \frac{-}{\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E} \qquad \text{SCong}\frac{E \equiv E', E' \xrightarrow{\gamma} F', F' \equiv F}{E \xrightarrow{\gamma} F}$$

Table 5.3: Mobility Semantics

$$\text{InEnv}\quad \frac{E \overset{@m}{\rightarrow} E', B_1 \overset{\overline{@}}{\rightarrow} B_1'}{n[E]^{B_2}_{\vec{\sigma}} \mid m[G]^{B_1}_{\vec{\rho}} \overset{@}{\rightarrow} m[G \mid n[E']^{B_2}_{\vec{\sigma}}]^{B_1'}_{\vec{\rho}}}$$

$$\text{OutEnv}\quad \frac{E \overset{@m}{\rightarrow} E', B_1 \overset{\overline{@}}{\rightarrow} B_1'}{m[G \mid n[E]^{B_2}_{\vec{\sigma}}]^{B_1}_{\vec{\rho}} \overset{@}{\rightarrow} n[E']^{B_2}_{\vec{\sigma}} \mid m[G]^{B_1'}_{\vec{\rho}}}$$

$$\text{Open}\quad \frac{E \overset{\circledast m}{\rightarrow} E', B_1 \overset{\overline{\circledast}}{\rightarrow} B_1'}{n[E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\gamma}} \overset{\circledast}{\rightarrow} n[E' \mid F]^{B_2}_{\vec{\gamma} \cup \vec{\sigma}}}$$

$$\text{ProcIn}\quad \frac{E \overset{a}{\rightarrow} E', F \xrightarrow{on\ a@m} F', B_1 \overset{\overline{@}}{\rightarrow} B_1'}{((E \mid G) \setminus A) \mid F \mid m[H]^{B_1}_{\vec{\sigma}} \overset{@}{\rightarrow} (G \setminus A) \mid F' \mid m[H \mid E']^{B_1'}_{\vec{\sigma}}}$$

$$\text{ProcOut}\quad \frac{E \overset{a}{\rightarrow} E', F \xrightarrow{on\ a@m} F', B_1 \overset{\overline{@}}{\rightarrow} B_1'}{m[((E \mid G) \setminus A) \mid F]^{B_1}_{\vec{\sigma}} \overset{@}{\rightarrow} E' \mid m[(G \setminus A) \mid F']^{B_1'}_{\vec{\sigma}}}$$

Table 5.4: Structural Congruence Laws

StrSum1	$E + F$	$\equiv$	$F + E$
StrSum2	$E + (F + G)$	$\equiv$	$(E + F) + G$
StrPar1	$E \mid F$	$\equiv$	$F \mid E$
StrPar2	$E \mid (F \mid G)$	$\equiv$	$(E \mid F) \mid G$
StrIdent	$E \mid \mathbf{0}$	$\equiv$	E
StrResRem	$\mathbf{0} \setminus A$	$\equiv$	$\mathbf{0}$
StrResRes	$E \setminus A \setminus B$	$\equiv$	$E \setminus A \cup B$

Table 5.5: Derivation of Nomadic Time from CaSE

Rule in CaSE	Use in Nomadic Time
$Idle$	As in CaSE
Act	As in CCS and CaSE
$Patient$	As in CaSE
$Stall$	As in CaSE
$Sum1$	Now includes mobility actions, κ replaces α
$Sum2$	Redundant due to $SCong$
$Sum3$	$Sum2$; As in CaSE
$Par1$	Now includes mobility actions, κ replaces α
$Par2$	Redundant due to $SCong$
$Par3$	$Par2$; As in CCS and CaSE
$Par4$	$Par3$; with τ generalised to h
$FTO1$	τ generalised to h
$FTO2$	As in CaSE
$STO1$	τ generalised to h
$STO2$	Now includes mobility actions, κ replaces α
$STO3$	τ generalised to h
Rec	As in CaSE
Res	As in CaSE
$Hid1$	$LHd1$; Clock hiding changed to environ, now uses sets of clocks
New	$LHd2$; allows high-priority transitions
$Hid2$	$LHd3$; Clock hiding changed to environ, now uses sets of clocks
New	$Cap1$; allows mobility primitives to be performed
New	$Cap2$; allows time to run over mobility primitives
New	$SCong$; integrates the structural congruence laws

the recursion as soon as any transition, γ, occurs. The *Res* rule defines restriction, which disallows any transitions for the given name. Both rules are taken directly from CaSE.

The rules $LHd1$ and $LHd3$ are based on $Hid1$ and $Hid2$ in CaSE, but with the clock hiding operator replaced by the new syntax for an environ. With this syntactic change also comes a change from the hiding of a single clock, σ, in CaSE, to the hiding of a set of clocks, $\vec{\sigma}$. For $LHd1$, the Nomadic Time equivalent of $Hid1$, this adds a side condition whereby the σ used in the transition must now be a member of the set $\vec{\sigma}$; in CaSE, the two are simply equal. Likewise, $LHd3$, the equivalent of $Hid2$, defines σ as a member of $\vec{\sigma}$ and ρ as not being a member. The effect of the two rules is that clock transitions for clocks which are members of $\vec{\sigma}$ become τ transitions, while the others are retained as is. The latter only occurs on the provision that there are no transitions from clocks in $\vec{\sigma}$ as these result in τ transitions being created which have a higher priority than clock transitions.

The three remaining rules, $LHd2$, $Cap1$ and $Cap2$ are new to Nomadic Time. The rule $LHd2$ simply allows high priority transitions (h) to be visible outside the environ. Note that we don't allow actions (i.e. members of $\mathcal{N}$ and $\overline{\mathcal{N}}$) to cross the boundaries of an environ so communication can not take place between a process on the outside and one on the inside. The $Cap1$ and $Cap2$ rules are simply the equivalents of *Act* and *Patient* but for the mobility primitives.

Consequently,

Proposition 1 *The semantics exhibit the following properties:*

1. *prioritisation;* $E \xrightarrow{\sigma}$ *implies* $E \overset{h}{\not\rightarrow}$
2. *time determinacy;* $E \xrightarrow{\sigma} E'$ *and* $E \xrightarrow{\sigma} E''$ *implies* $E' = E''$. □

These properties can be observed directly from the rules. The transition $E \xrightarrow{\sigma}$ is produced by the rules *Idle*, *Patient* and $Cap2$, and retained over summation ($Sum2$), parallel composition ($Par3$), timeouts ($FTO1$, $FTO2$, $STO1$ and $STO3$), recursion (*Rec*), restriction (*Res*) and environs ($LHd3$) where $E \overset{h}{\not\rightarrow}$. The rule $LHd1$ provides the conversion of ticks emitted by the hidden clocks to silent actions: if E can perform a σ transition, then it performs a τ transition in any context where σ is hidden. As τ is one of the possible values of h, the presence of transitions for the clocks being hidden prevents transitions being created for those that are not hidden in $LHd3$. The property is implicit for summation, recursion and restriction as, if the composed processes can perform a σ transition, then it must not be blocked by a h transition. The presence of time determinacy is obvious from the *Idle*, *Patient* and $Cap2$ rules which generate σ transitions as the process ($\mathbf{0}$, $\alpha.E$ or $\mathcal{M}.E$) remains unchanged as a result of performing $\xrightarrow{\sigma}$.

The semantics in Table 5.3 focus on mobility and are completely new to Nomadic Time. $InEnv$ allows a $\circledcirc$ transition to occur and n to move into m if matching $\circledcirc m$ and $\overline{\circledcirc}$ transitions are available from the process $\circledcirc m.E$ and bouncer, B_1. Conversely, $OutEnv$ concerns the interaction between $\circledcirc m.E$ and $\overline{\circledcirc}$, allowing a $\circledcirc$ transition to occur and n to move outside m. Likewise, $\circledast m$ causes a $\circledast$ transition to occur when both an $\circledast m$ and an $\overline{\circledast}$ transition are available. The named environ, m, is destroyed along with its bouncer, and the two clock contexts are unified.

Finally, $ProcIn$ and $ProcOut$ describe the movement of processes between environs. In both rules, E moves due to a mobility primitive which is part of F. This occurs if an a transition takes place in the presence of matching *on* $a \circledcirc m$ and $\overline{\circledcirc}$, or *on* $a \circledcirc m$ and $\overline{\circledcirc}$, actions. An appropriate mobility transition ($\circledcirc$ or $\circledcirc$) is emitted as a result of this three-way synchronisation. In both, we include the use of restriction on $E \mid G$ so as to show how E escapes its scope. Process mobility, in this form, is *objective*. The process which emits the mobility primitive synchronises with a partner process, and it is this partner which actually moves. The partner is necessarily a process in the same environ, due to the scoping of actions described above.

5.6 A Simple Example

Consider the familiar children's game of musical chairs. The conduct of the game can be divided into the following stages:

1. The players begin the game standing. The number of players is initially equal to the number of chairs.
2. The music starts.
3. A chair is removed from the game.
4. The music stops.
5. Each player attempts to obtain a chair.
6. Players that fail to obtain a chair are out of the game.
7. The music restarts. Any players who are still in the game leave their chairs and the next round begins (from stage three).

The winner is the last player left in the game. A model of this game can be created using the NT process calculus.

The game environment is represented using environs. In the musical chairs scenario, each chair is represented by an environ, as is the 'sin bin', to which players are moved when they are no longer in the game. These environs are all nested inside a

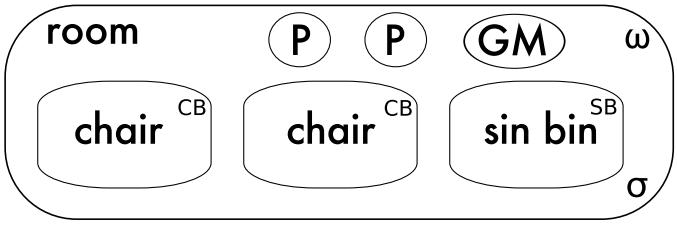


Figure 5.3: The Musical Chairs Environment

further environ which represents the room itself. This is not a necessity, but makes for a cleaner solution; it allows multiple instances of the system to be nested inside some context, each performing its own internal interactions and entering into the synchronisation cycle of the larger system.

The environ structure is represented graphically by Fig. 5.3 and in the calculus by the equation shown below.

$$room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid sinbin[\mathbf{0}]^{SB} \mid P \mid P \mid GM]^{\Omega}_{\{\sigma\}} \tag{5.41}$$

where $m[E]^F$ is abbreviated from $m[E]^F_{\{\}}$. The players themselves are represented by *processes*. This allows them both to interact and to move between environs. A gamesmaster process is also introduced. This doesn't play an active role in the game itself, but is instead responsible for performing the administrative duties of removing chairs from the game and controlling player movement. The process definitions are summarised in Table 5.6, and make use of the derived syntax for a clock prefix, $\sigma.P$, shown in 3.4.2. Their names are shorthand for Chair Bouncer (CB), Sinbin Bouncer (SB), Games Master x (GMx), Player (P), Moving Player (MP), Player in Chair (PiC), Player Leaving Chair (PLC) and Leaver (L).

The presence of music is signified by the ticks of a clock, σ. A tick from σ is also used to represent the implicit acknowledgement that everyone who can obtain a chair has done so, and that the remaining player left in the room has lost. With regard to the bouncers of the environs, the room environ is not prone to either destruction or the entry or exit of other environs, having a bouncer simply equal to Ω. This retains the encapsulation of the model as a single room environ, and prevents other processes or environs from interfering with its behaviour.

The definition of appropriate bouncers is essential for the chairs (5.42) and the *sinbin* (5.43). It is the chair bouncer that enforces the implicit predicate that only one player may inhabit a chair at any one time, while the *sinbin* bouncer prevents players leaving the *sinbin* once they have entered.

To model stage one of the game, n player processes and n chair environs are placed in the room. The advantage of using NT for this model is that the actual number of players or chairs is irrelevant. They need not even be equal. The calculus

Table 5.6: Summary of Processes and Derived Syntax for Musical Chairs

$$CB \stackrel{\text{def}}{=} \mu X.(\overline{\circledcirc}.\overline{\oslash}.X + \overline{\circledast}.\Omega) \tag{5.42}$$

$$SB \stackrel{\text{def}}{=} \mu X.\overline{\circledcirc}.X \tag{5.43}$$

$$GM1 \stackrel{\text{def}}{=} \sigma.GM2 \tag{5.44}$$

$$GM2 \stackrel{\text{def}}{=} \circledast chair.GM3 \tag{5.45}$$

$$GM3 \stackrel{\text{def}}{=} \sigma.GM4 \tag{5.46}$$

$$GM4 \stackrel{\text{def}}{=} \mu X.(\lceil on\ sit \circledcirc chair.X \rceil \sigma(GM5)) \tag{5.47}$$

$$GM5 \stackrel{\text{def}}{=} \mu X.(\lceil on\ leave \circledcirc sinbin.X \rceil \sigma(GM1)) \tag{5.48}$$

$$P \stackrel{\text{def}}{=} \sigma.\sigma.MP \tag{5.49}$$

$$MP \stackrel{\text{def}}{=} \lceil sit.PiC \rceil \sigma(L) \tag{5.50}$$

$$PiC \stackrel{\text{def}}{=} \sigma.\sigma.PLC \tag{5.51}$$

$$PLC \stackrel{\text{def}}{=} on\ stand \oslash chair.\mathbf{0} | stand.P \tag{5.52}$$

$$L \stackrel{\text{def}}{=} leave.\mathbf{0} \tag{5.53}$$

allows the creation of a compositional semantics, as discussed in chapter 1, which works with any n.

For the purposes of demonstration, n is assumed to be two to give the following starting state:

$$room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid P \mid P \mid GM1]^{\Omega}_{\{\sigma\}}. \tag{5.54}$$

The room and chairs appear as shown earlier. The player processes (5.49) simply wait until two clock cycles have passed, the end of each being signalled by a tick from σ. The intermittent period between the ticks (the second clock cycle) represents the playing of the music.

Stage two, where the music is started, is thus represented simply by the first tick of σ,

$$\begin{aligned} & room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid P \mid P \mid GM1]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} & room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM2]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{5.55}$$

which the gamesmaster ($GM1$ (5.44)) also waits for, before evolving into $GM2$ (5.45). The second cycle, prior to the music stopping, is used to remove a chair

from the game. Maximal progress, as explained in section 1, ensures that this occurs before the next clock tick, as the removal emits a high priority action, $\circledast$. The transition from stage three to stage four is thus as follows:

$$\begin{aligned} & room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM2]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\circledast} & room[chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM3]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{5.56}$$

with one of the two chairs being chosen non-deterministically. The second tick then occurs, leading in to stage five and the most interesting part of the model.

$$\begin{aligned} & room[chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM3]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} & room[chair[\mathbf{0}]^{CB} \mid MP \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{5.57}$$

The aim of stage five is to get as many player processes as possible inside chair environs. This is handled by again relying on maximal progress to perform a form of broadcast that centres on mobile actions, as briefly mentioned in 5.3.2. Rather than sending a signal to a number of recipients, a request to move into a chair (see (5.47) and (5.50)) is delivered instead.

If a chair is available, then a player process will enter it (the choice of chair and player is non-deterministic). This will cause a high priority action to occur, which takes precedence over the clock tick. Thus, when the clock eventually does tick, it is clear that no more players can enter chairs. Using clocks in this manner makes the system *compositional*; in contrast to other models, players and chairs can be added without requiring changes to the process definitions. In this running example, there are two players, but only one chair, which results in a single $\oslash$ transition:

$$\begin{aligned} & room[chair[\mathbf{0}]^{CB} \mid MP \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\oslash} & room[chair[PiC]^{\overline{\oslash}.CB} \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{5.58}$$

that causes one of the MP processes to move in to a chair, and become a PiC process. This is followed by the σ transition, which marks the move to stage six.

$$\begin{aligned} & room[chair[PiC]^{\overline{\oslash}.CB} \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} & room[chair[\sigma.PLC]^{\overline{\oslash}.CB} \mid L \mid GM5]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{5.59}$$

Both stage six and seven proceed in a similar way. Stage six sees essentially the same broadcasting behaviour applied to the losing players (see (5.48) and (5.53)). The difference is that stage six demonstrates something which wouldn't be possible without mobility: the broadcast is limited to those player processes which remain in the room. As communication between processes in different environs is disallowed

in NT, an implicit scoping of the broadcast occurs. In the example, stage six again sees just one $\circledast$ transition:

$$\begin{array}{r} room[chair[\sigma.PLC]^{\overline{\emptyset}.CB} \mid L \mid GM5]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\circledast} room[chair[\sigma.PLC]^{\overline{\emptyset}.CB} \mid GM5]^{\Omega}_{\{\sigma\}} \end{array} \tag{5.60}$$

which results in the remaining MP (now a losing process, L) moving to the sinbin. Due to space constraints, the sinbin environ is not shown in the above derivations. It may be factored in to the above as follows:

$$\begin{array}{r} sinbin[\mathbf{0}]^{SB} \mid L \mid GM5 \\ \xrightarrow{\emptyset} sinbin[\mathbf{0}]^{SB} \mid GM5 \end{array} \tag{5.61}$$

where the L process evolves to become a simple $\mathbf{0}$ process. The broadcast is again terminated by a tick from σ,

$$\begin{array}{r} room[chair[\sigma.PLC]^{\overline{\emptyset}.CB} \mid GM5]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} room[chair[PLC]^{\overline{\emptyset}.CB} \mid GM1]^{\Omega}_{\{\sigma\}} \end{array} \tag{5.62}$$

which, in this case, also signifies the music starting up again. The remaining players leave their chairs:

$$\begin{array}{r} room[chair[PLC]^{\overline{\emptyset}.CB} \mid GM1]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\emptyset} room[chair[\mathbf{0}]^{CB} \mid GM1 \mid P]^{\Omega}_{\{\sigma\}} \end{array} \tag{5.63}$$

and the system essentially returns to the beginning, with $n-1$ chairs and $n-1$ players.

5.7 A Prototypical Application in NT

Recall that in 1.3.1 we specified a series of requirements for a music player application:

- The application should provide some form of interface with which the user can interact.
- It should be able to take a wave file and return a sequence of sound data for playback.
- It should be able to output the sound data through the speakers.
- It should be able to generate a spectral analysis of the sound data as a form of visual feedback.

Now that we have our process calculus to work with, we can provide a formal design for this application, which can then be converted directly into a real-world application using DynamiTE in the next chapter.

First, we will consider the reading of the wave file. The simplest solution is:

$$In \stackrel{\text{def}}{=} i.\mu X.\tau.\overline{o}.X \tag{5.64}$$

where the filename is read in on i and processed to produce some sound data in the τ action. The data is then output on o. We then recurse, continuing to produce more sound data and output it on o[9].

From this, we can already determine some things about the other processes in the system:

- The interface ($Intf$) must output on $\overline{i}$ to trigger In into starting to produce sound.
- The speaker output (Out) must read on o to obtain the sound data and send it to the speakers.
- The spectral analyser ($Analy$) must read on o to obtain the sound data and produce the visual feedback.

and definitions for Out and $Analy$ follow fairly trivially[10]:

$$\begin{aligned} Out &\stackrel{\text{def}}{=} o.\tau.\mathbf{0} \\ Analy &\stackrel{\text{def}}{=} o.\tau.\mathbf{0} \end{aligned} \tag{5.65}$$

This already highlights one problem with the current In process. Both Out and $Analy$ need to synchronise with it on the o channel, but it currently only performs one $\overline{o}$ action. Thus, on each loop within In, one of the two will synchronise and the other will miss out.

From 3.2 and 5.6, we already know the best way to solve this; with a timeout. In needs to recurse over the $\overline{o}$ action until it can no longer synchronise with a recipient, at which point it reads the next piece of sound data; this is exactly the same logic

[9] At some point, the τ process will reach the end of the file; there isn't an obvious way of representing this in the design so we just have to assume that, in the implementation, the internal τ process will terminate the thread running In. We could use $X + \mathbf{0}$ at the end, but then we are representing the end of the process as being non-deterministic, when it is in fact determined by the file.

[10] We could represent these processes outputting on channels representing the speakers and display respectively, but in reality these are going to be system calls in the τ process, just as the τ in In performs reading and decoding operations on the file to produce sound data

as we employed for the compositional broadcast agent in 3.2. As a result, *In* now looks like this:

$$In \stackrel{\text{def}}{=} i.\mu X.\tau.\mu W.\lceil \overline{o}.W \rceil \sigma(X) \tag{5.66}$$

We bind W to $\lfloor \overline{o}.W \rfloor \sigma(X)$ so that each time $\overline{o}$ is performed, we return to our original state. When *In* is running in parallel with *Out* and *Analy*:

$$IntSys \stackrel{\text{def}}{=} i.\mu X.(\tau.\mu W.\lceil \overline{o}.W \rceil \sigma(X) \mid o.\tau.\mathbf{0} \mid o.\tau.\mathbf{0}) \tag{5.67}$$

the presence of both o and $\overline{o}$ will allow a τ transition to occur (rule $Par2$ in the semantics), which in turn inhibits σ. Once both have occurred, σ transitions can occur and will cause recursion to occur via the expansion of X. Our structural congruence rules ($SCong$) mean that we can simply discard the two $\mathbf{0}$ processes left behind by *Out* and *Analy*.

There is one remaining issue with this construction; the internal τ actions of *Out* and *Analy* will also cause *In* to continue to recurse on W rather than reading the next piece of input. The solution to this is to also synchronise these two processes on σ:

$$Out \stackrel{\text{def}}{=} o.\lceil \Delta \rceil \sigma(\tau.\mathbf{0}) \tag{5.68}$$

so that now, once input has been received on o, *Out* waits until σ becomes unimpeded and is able to tick; Δ produces no transitions, so neither $STO2$ or $STO3$ can be applied, while $STO1$ requires the absence of any high priority transitions, which includes τ transitions. This should allow all three processes to continue with their internal processing. The same definition can be applied to *Analy*.

Of course, we could provide a much simpler solution by just performing $\overline{o}$ twice. The advantage of this solution, as we have discussed before, is that we can add any number of other processes that need to synchronise on o without having to alter our definition of *In*.

All that remains to complete our definition of this system is to define the interface. This is simply a means of translating user actions into calls to our internal system. This can be as simple as:

$$Intf \stackrel{\text{def}}{=} useri.(\overline{i}.\mathbf{0} | IntSys) \tag{5.69}$$

But what does *useri* synchronise with? This comes from the user and is outside the system itself:

$$on\ begin \between player.\mathbf{0} \mid begin.\overline{useri}.\mathbf{0} \mid player[Intf]^{\overline{\between}.\Omega}_{\{\sigma\}} \tag{5.70}$$

Our system is now encapsulated in an environ, *player*. To start the player, a client must enter *player* and synchronise on *useri*. The bouncer of the *player* environ

allows only one process to enter, by providing only one $\overline{\textcircled{\scriptsize V}}$ with which to synchronise. The clock σ is hidden outside the environ so all the ticks from within *player* appear as silent actions to those outside (see $LHd1$). As all the other transitions performed by *Intf* will also be silent actions, being a mix of internal τ actions and synchronisations between i and o, processes outside *player* can use the presence of these transitions to determine whether or not *Intf* is active or not.

We have deliberately kept the design as simple as possible to make it easier to digest and to reduce the complexity of the corresponding implementation we will cover in 6.6 as part of our discussion of DynamiTE. There are many more things that could be represented, not the least being a way of stopping playback once begun!

5.8 Conclusion

In this chapter, we introduced our process calculus, Nomadic Time, giving the first set of novel contributions (**C1.1** through **C1.7**) in this thesis. Nomadic Time extends the CaSE process calculus described in 3.4 with the notions of localities (see 4.3) (**C1.1**) and migration (see 4.3.2) (**C1.2** and **C1.3**). We also introduced the notion of 'bouncers' (**C1.4**), a security mechanism which allows the number and type of mobility operations which may be performed on an environ (our term for a locality in NT) to be defined. The first half of the chapter (5.2 to 5.4) demonstrated how each of these features was layered onto the calculus from the stage before, with CaSE as our starting point.

In 5.5, we presented the operational semantics for Nomadic Time (**C1.6**). With respect to the core semantics, although they are partially derived from CaSE, there was still a significant amount of novel work involved in ensuring that the derived rules correctly covered the new possibilities introduced by the expanded syntax and transitions available in Nomadic Time. For example, $Cap1$ and $Cap2$ were not present in initial revisions, but without these, the transitions upon which the mobility semantics (Table 5.3) rely would simply not exist. We believe our formulation of the mobility semantics to be novel; while the general concepts embodied within them are present in the Ambient Calculus and its derivatives, the semantics of these calculi are not constructed as structural operational semantics. We thus believe those presented here to be fairly unique, especially with regard to the presence of co-mobility primitives from the bouncers.

Unlike CaSE, Nomadic Time also has a set of structural congruence rules (**C1.5**). These allow two of the rules from CaSE ($Sum2$ and $Par2$) by defining the summation and parallel composition operators to be commutative. They also allow redundant **0** processes and restriction operators to be removed, and for a pair of restriction operators to be unified into one. Section 5.5 also demonstrates how the properties of prioritisation and time determinacy hold for our new calculus (**C1.7**).

The final sections demonstrated how the calculus may be used via two motivating examples; the first (5.6) aimed to demonstrate each of the features of the calculus in action, while the second (5.7) expanded on the application specified in 1.3.1 from a more real-world perspective.

In the next chapter, we show how Nomadic Time may be used to construct a concurrent programming framework in the Java programming language. We refer to this framework as the DynamiTE (Dynamic Theory Execution) framework, and this forms the second set of novel contributions (**C2.1** to **C2.4**) in this thesis. The first half of the chapter focuses on different elements of DynamiTE; the implementation of the Nomadic Time syntax and semantics as Java objects (6.3; **C2.1** and **C2.2**), the plugin framework (6.4; **C2.3**) and the provision of execution semantics via the evolver framework (6.5; **C2.4**). We then use DynamiTE to implement a prototypical application, using the design from 5.7 to construct corresponding Java classes which meet the requirements in 1.3.1. Finally, we cover some other existing attempts to provide implementations of process calculi in 6.7.

Chapter 6

The DynamiTE Framework

6.1 Introduction

In this chapter, we introduce DynamiTE, the Dynamic Theory Execution framework. This provides the solution we first proposed in 1.3, using the Nomadic Time (NT) calculus introduced in chapter 5 as a foundation for application development. When using DynamiTE, programmers compose NT processes, realised as Java objects, to create a working system. The framework handles running these processes, in parallel if necessary, and negotiates the communication between them. Both features are provided by leveraging existing facilities in the underlying Java virtual machine and class library.

The first half of this chapter explores the design of the framework and its initial implementation in the Java programming language. In 6.3, we describe how Nomadic Time processes are mapped onto Java objects (see 6.3) with the provision of a translation schema (**C2.1**) and examples of how the semantics are implemented (**C2.2**). The next section, 6.4, deals with practicalities beyond simply implementing the syntax of the calculus; we describe how the *context* of the calculus is represented and also how we attach additional side effects to transitions via the plugin framework (**C2.3**). In 6.5, we look at the final piece of the jigsaw; executing the processes in order to produce behaviour. The execution semantics are embodied in the evolver framework (**C2.4**), which allows the programmer to swap in different implementations, depending on whether they want to simply examine all the possible transitions of a process or to follow a single deterministic route with side effects.

Finally, we show how DynamiTE can be used to create an implementation of the application we introduced in 1.3.1 (see 6.6), before closing with a brief look at other similar concurrent frameworks in 6.7. But first, we discuss why we chose Java as the host language for DynamiTE and what advantages and disadvantages this brings to its implementation.

6.2 Why Java?

The first implementation of the Java programming language was released in 1995 by Sun Microsystems. It takes the form of a block structured language with a syntax akin to C or C++. However, unlike programs written in those languages, Java applications tend to be compiled to platform-independent Java bytecodes which are then executed by a Java Virtual Machine or JVM. This allows the same Java program to be executed on multiple platforms without the need for recompilation. With this new operating environment comes the removal of a number of features found in Java's predecessors and the restriction of others, with the aim of creating a safer and more portable language:

- **No pointer manipulation**. All primitive types in Java (integer, floating point numbers, booleans and single characters) are passed by value. All objects are stored and passed as pointers or references to their location in memory. These pointers are immutable, removing the ability to perform pointer arithmetic (e.g. for iterating over arrays) and with it, a host of problems inherent with inappropriate memory access. For example, attempts to use a `null` pointer are caught by the virtual machine and produce a checked exception, rather than causing a segmentation fault which brings down the entire process.

- **All arrays are bounds checked**. A major cause of errors and security issues in C and C++ programs is the possibility of buffer overflows, where programs write to memory beyond the end of an array. In Java, such errors are prevented by the virtual machine; any attempt to access an index outside the bounds of an array causes a checked exception to be thrown and direct access to the array's memory is forbidden by the lack of pointer manipulation.

- **All memory management is performed by a garbage collector**. While allowing manual memory management allows the programmer greater control, it leads to an equivalent to the issue we saw with semaphores in 1.2.3; every allocation must be paired with a later deallocation to avoid the possibility of an application leaking memory. The problem is even more pronounced with regard to memory management as, while the acquisition and release of a lock tend to occur in close proximity to one another, allocation and deallocation can occur in quite disparate parts of the application. Java instead makes use of existing research into *garbage collectors* which allocate memory as needed and periodically reclaim it. The downside of this is that the garbage collector has to use processor time to perform its scans which would otherwise be used by the application. However, as new garbage collection techniques, such as concurrent and generational collectors, become prevalent, this disadvantage is further outweighed by the prospect of chasing memory leaks.

- **Lack of unsigned types**. All integer types in Java use a bit to store the sign of the value, with no equivalent unsigned types that instead use this bit to store larger values. This makes bitwise operations (and (`&`), or(`|`) and not(`~`)) more inefficient as they need to operate on the type one size above (bytes (2^8) on shorts (2^{16}), shorts on ints (2^{32}), etc.). Indeed, section 15.22.1 of the Java language specification (Gosling, Joy, Steele & Bracha, 2005) states that *binary numeric promotion* (as defined in 5.6.2) should be applied to the operands, causing them to be converted to integer or long integer levels of precision before the operation is performed. Thus, it logically follows that it is impossible to work with unsigned long integers (2^64) without resorting to the overhead of a class which implements arbitrary precision integers, such as `java.lang.BigInteger`. Unsigned types continue to be proposed for addition to the Java language, but no such extension is scheduled for the next release (Java 7) in early 2010.

Although these changes are made at the expense of flexibility for the programmer and possible efficiency gains, they save time overall in chasing bugs caused by memory allocation errors, buffer overflows or leaks. Besides, the Java Native Interface (JNI) can be used to implement certain methods in C, should the need arise. Many of the methods provided by the Java class library do just that, usually to make use of a platform-specific application programming interface (API). Doing so has some overhead and means losing the safety and memory management benefits of Java, but is possible when a native library needs to be used to facilitate reuse.

Performance has been a common criticism of Java, not just because of these features but also because the Java bytecodes it uses must be either interpreted or compiled into native code at run time. This is much less of an issue than it once was, due to advances in virtual machine design and Just-In-Time (JIT) compilation techniques. Theoretically, JIT compilation should eventually exceed the performance of code compiled Ahead-Of-Time (AOT) as it can take advantage of information only available at runtime. This includes knowing the exact platform on which the code will execute and being able to make better optimisations based on statistics gathered through execution (e.g. better branch prediction). For example, HotSpot, the virtual machine used by Sun's implementation of Java, only uses the JIT compiler to create native code when it believes the code is used enough ('hot' enough) to make doing so worthwhile.

Of these changes, the absence of unsigned types is the only one that seems to have no advantage, other than simplifying the language. Many file formats and network protocols include unsigned types, so working with them in Java becomes harder than is necessary. Although their absence may have made sense in earlier versions of the language, the complexity of understanding unsigned arithmetic now seems trivial when compared with the existential type system and its lack of reification

which was introduced by the addition of '*generics*' in Java 5. We thus hope that they may make an appearance in Java 8.

6.2.1 Concurrency Provision

From the perspective of implementing DynamiTE, one advantage of Java is its broad support for concurrency. Java is one of the few languages to have an implementation of *monitors*, a feature we demonstrated in 1.2.3. It has also had support for threads from the very beginning, with support as a core part of the virtual machine rather than as an auxiliary library (the approach used for C). Java's platform independence means that the same threading constructs and semantics, as mandated by the VM specification (Lindholm & Yellin, 1999), can be used across all operating systems supported by a Java virtual machine. The actual implementation is provided by the virtual machine and class library, which may either map them on to native threads or provide *green* threads, where the virtual machine itself creates and schedules threads. The main disadvantage of the latter is that blocking calls to the operating system performed by one thread will cause the virtual machine and all its threads to be blocked; as the system is unaware of the presence of the threads, its only option is to block the entire VM process. Green threads are however much faster to create and synchronise, as everything takes place within the VM. They can also match the required thread semantics exactly, rather than having to map those provided by the operating system's threads. While earlier versions of Sun's implementation used green threads, native threads are now used on all supported platforms.

The result of this early adoption of multithreading is that its implementation in Java is reasonably mature and well-tested. With Java 5, this support was greatly expanded as a result of research led by Doug Lea and incorporated into the Java platform via JSR166 and the `java.util.concurrent` packages (Goetz, Peierls, Bloch, Bowbeer, Holmes & Lea, 2006; Lea, 2009).

The extensions provided by JSR166 take the form of a host of new classes, backed by support in the virtual machine. The Java language itself is not altered. It provides support for:

- **Atomic Variables**. These provide replacements for integer, long integer and reference fields which can be updated in an atomic fashion, which are safer than `volatile` variables and more efficient than locking. The Java memory model allows operations which alter the value of normal fields to be reordered by the VM as a form of optimisation, as long as this reordering is not visible from within the same thread. However, this means that other threads may see the changes in the wrong order or not at all. Marking a field as `volatile` makes the VM aware that it may be accessed by multiple threads, causing updates to be made visible to all threads immediately. However, `volatile`

fields are still prone to race conditions when used in non-atomic operations such as incrementing a value or performing a conditional update[1]. The usual solution is to obtain a lock on the class every time the variable is altered; this provides both the update guarantees of a `volatile` variable and blocks other threads trying to obtain the same lock. Atomic variables provide an alternate solution by allowing the processor's CAS operation (see 1.2.3) to be used. This is usually more efficient than locking the entire class, which will involve the VM performing a CAS operation on the lock at some point anyway. While locking takes a pessimistic approach to thread safety by blocking all other threads, CAS operations are optimistic; the update is attempted, and if it fails, we try again until it succeeds. Implementing such a check successfully is even more prone to error than locking, as the programmer has to ensure they check the result of the CAS and loop accordingly, but it is usually much more efficient when contention is low. With the addition of atomic variables to Java, programmers now have the choice of using either.

- **Explicit Locks**. As we saw in 1.2.3, Java has implicit reentrant locking via the `synchronised` keyword. Their use, however, is limited; there is only one lock per class, so all its variables must be protected by the same lock, and threads are always blocked until they either acquire the lock or the thread is interrupted by `Thread.interrupt()`. The `ReentrantLock` class provides a more advanced version with the following additional features:

  - `tryLock()` can be called to perform a non-blocking acquisition of the lock. It immediately returns with `true` if the lock was acquired, and `false` if it wasn't.
  - `tryLock(long, TimeUnit)` can be called to perform a timed acquisition. If the lock is available, it acquires it and returns immediately. Otherwise, it blocks. However, unlike the implicit lock provision and the `lock()` method, it will become unblocked after the given timeout and return `false`.
  - The lock can operate in a fair mode, where threads acquire the lock in the order they requested it. Both implicit and explicit locks default to unfair behaviour, which permits *barging* (jumping the queue) if a new thread happens to request a lock when it is unheld. Unfair locks are much faster[2], but fairness is sometimes needed to ensure correctness.

[1] e.g. in `if (x == 4) x = 5`, it is possible for `x`'s value to have been changed by another thread before the assignment but after the comparison

[2] If threads are not allowed to jump the queue, then we end up blocking and descheduling a thread which could have quite happily acquired the lock but isn't allowed to do so because of the fairness policy

A class can have multiple instances of an explicit lock, just like any other variable, and this benefit is utilised by `ReentrantReadWriteLock`. This class provides both a shared (**read**) lock and an exclusive (**write**) lock. Multiple threads can acquire the read lock, but to acquire the write lock, both locks must be unheld. This can be used to make classes more efficient when compared with the brute force approach of enforcing mutual exclusion for all operations. For example, a collection class can allow multiple threads to read values as long as there is no thread altering the collection. Locks, and other implementations such as `Semaphore`, are based on `AbstractQueuedSynchronizer` (Lea, 2004) which provides a common framework thread queues.

- **Explicit Condition Queues**. As in the case of locks, Java already has its own implicit condition queues, accessible via the `wait`, `notify` and `notifyAll` methods. These also have similar limitations to the implicit locks; only one queue is available per class and either one or all threads must be notified. With only one condition queue, the usability of `notify` to alert a single thread is extremely limited; using it is dangerous if there is more than one condition as the wrong thread may be awoken, and it is inefficient unless a change in the condition means that one and only one thread may proceed. The former can be observed in the buffer example of 1.2.3 where there are two conditions: `used == BUFFER_SIZE` and `used == 0`. The latter is observable in a 'gate' scenario where multiple threads queue up waiting for a condition to hold, and then all proceed when it does. Explicit condition queues address these issues by allowing a class to have multiple condition queues. Each `Condition` is obtained from a `Lock` by a call to `Lock.newCondition` and that same lock must be held when calling its methods. In the buffer example, the synchronized blocks would be replaced by the use of explicit locks and the calls to `wait` and `notifyAll` by `await` and `signal` calls on one of two `Condition` objects. The `signal` method can now be used rather than `signalAll`, awakening just one thread, as we know the thread will be waiting for the condition whose state has changed and no other. This avoids waking all threads and having all but one go back to sleep.

- **Executors and Thread Pools**. The new classes provide a framework for executing tasks in the form of the `Executor` interface. This decouples the process of submitting a task from how it is executed. Tasks (in the form of an object which implements the `Runnable` or `Callable` interface) are submitted to an `Executor` instance, and then performed in a manner determined by the `Executor` implementation. The `Executors` class provides a number of pre-defined instances:

  – A single thread executor, which performs tasks sequentially.

  - An executor with a fixed size pool of threads.
  - An executor with an unbounded pool that grows and shrinks as demand allows.
  - An executor with a fixed size pool of threads and delayed or periodic task execution.

  The programmer is also, of course, free to define their own implementation. This feature is very useful for implementing parallel composition in DynamiTE as each process may be submitted to an executor, the choice of which is left up to the user of the framework.

- **New Collections**. The standard Java collections apply an all-or-nothing approach to thread safety; either the instance is unsafe for multithreaded use (as with instances of the Java 1.2 classes – `HashMap`, `ArrayList`, etc.) or every method call locks the class (as with the legacy classes such as `Vector` and `Hashtable` or the 1.2 classes when wrapped by the `synchronizedX` methods in `Collections`). The JSR166 extensions provide a new set of collections which utilise the features listed above. For example, `ConcurrentHashMap` provides a hash map which utilises *lock striping*; the map is protected by multiple read and write locks which protect only a segment of the whole map each. Thus, not only can multiple readers access the map concurrently, but it may be possible to perform multiple writes concurrently if they effect different areas of the map. The new collections also include various `BlockingQueue` implementations, which implement the producer-consumer model we demonstrated with the buffer example in 1.2.3. One such implementation is `SynchronousQueue` which closely matches the semantics of synchronous channels in Nomadic Time; it has no storage so a thread performing a `put` blocks until a receiving thread calls `take`.

With these additions, the programmer is given a lot of control and flexibility when implementing concurrent programs in Java, and we will leverage many of these features when implementing DynamiTE. Having essential components such as locks and concurrent collections already available and well tested makes it much easier to meet the requirements of the framework.

Other languages are not so lucky. In C and C++, threads are provided by an operating system library and thus vary depending on platform. The POSIX standard for threads attempts to overcome this by providing a standard threading interface and semantics for POSIX systems. While POSIX-based systems including GNU/Linux, Solaris, FreeBSD and Mac OS X all provide implementations. the problem remains with systems that do not provide such by default, notably Microsoft Windows.

Haskell has been slow to introduce threading support. Although the Concurrent Haskell (Jones, Gordon & Finne, 1996) extension was originally proposed in 1996, it does not form part of the Haskell 98 standard and the GHC documentation still lists it as experimental. Both Hugs and the Glasgow Haskell Compiler (GHC), the two main implementations of Haskell, provide an implementation of Concurrent Haskell's `Control.Concurrent` module, they do so using green threads. As mentioned above, while these are faster than native threads, blocking calls to the operating system, such as I/O, will cause all threads to be blocked. A workaround is provided in GHC when it is built with the `-threaded` option; it uses a pool of worker threads to execute Haskell code and switches to a new one when a `safe` foreign call is made. It also allows native threads via `forkOS` when built in this manner. As with C, this makes Haskell's thread behaviour dependent on the underlying system as opposed to providing a standard set of operations and semantics; whether threads are provided and how well they perform depends entirely on which implementation of Haskell is being used.

However, functional languages in general should be a good basis for concurrency. They already operate in a task-oriented manner through *pure functions*; data is fed in, manipulated as desired and the result output without altering memory. Those that do alter memory, and thus could lead to concurrency issues, are clearly denoted (e.g. by monads in Haskell), reducing the amount of code that has to be checked for race conditions.

It is thus a pity that they are not more widely used and their concurrency facilities not more well developed. This is changing, however. GHC has recently been extended with support for Software Transactional Memory (STM) (Harris, Marlow, Jones & Herlihy, 2005), which provides a new `atomic` function and `STM` monad for implementing transactions. The STM logs all actions and then performs a single atomic commit, provided there are no conflicts with other updates. This allows Haskell programmers to compose new atomic transactions from others, and moves the need to ensure atomicity away from each individual function to the caller, who can only invoke them from within an atomic environment.

Erlang (Armstrong, Virding, Wikström & Williams, 1996) is another interesting case, as both it and DynamiTE focus on message passing between processes as opposed to shared data and locking. Erlang differs in that it operates asynchronously, collecting messages in a mailbox on a per-process basis and filtering which ones are received in any one operation. However, synchronous delivery can be implemented by requiring messages to be acknowledged. The main limitation of current Erlang implementations is that they use green processes; unlike green threads, these don't share state but they do have the same downside that a blocking system call from one will cause them all to become blocked by the system.

Both Erlang and Haskell provide an interesting environment in which to imple-

ment a framework like DynamiTE. Indeed, we hope that the majority of the design explained here in 6.3 can be applied to most languages with sufficient threading support. However, there is another reason for our choice of Java as the initial prototype language.

6.2.2 Popularity

Popularity is rarely a good reason to do anything but, in combating developer inertia, it is a good weapon to have. The simple fact is that most of today's developers know Java and sometimes little else; it (or its close relative, C#) is taught as part of most computer science degrees and is used as the language of choice for many applications, especially in the area of enterprise web applications.

As we discussed in 1.3, easing the barriers for adoption is an essential aspect in influencing developers to try something new. With DynamiTE, we are already advocating the idea of using message passing rather than state manipulation to Java developers, a body of programmers who will generally be more familiar with object-oriented design techniques which focus on manipulating data. Adding the prospect of learning an entirely new language is not going to help our case, and we believe this to be the main reason other solutions have not moved far beyond their academic roots. Instead, DynamiTE is developed as a Java class library like any other, which leverages standard features of the Java platform and which can be further developed by the very people that use it.

We will be the first to admit that Java has issues; its age means that with hindsight many design decisions can now be seen as flawed and attempting to change this leads us to consider the bane of all programming languages – backwards compatibility. Most features, good or bad, are now enshrined in the language and further development rightly takes a conservative attitude to avoid breaking the huge body of existing code already in use. This means that APIs are deprecated rather than removed, causing the class library to become more bloated than ever, and new language features such as generics take years to appear and even then have to be limited. No consensus has yet been reached on how closures should be implemented, so they will not appear in Java 7 either. These issues are here to stay; Java is unlikely to ever have a type system as advanced as that of most functional languages or a separation between pure and impure functions. But with these come maturity and a vast body of developers which we believe to be far more useful in achieving our goal than the possibilities of a perfect but niche language. Java is also the language with which we are most familiar, and thus it makes sense to experiment first using Java and then turn to other languages.

6.3 Mapping Theory to Practicality

In this section, we show how the syntactic constructs of NT introduced in chapter 5 are mapped on to Java classes by the DynamiTE framework. Within DynamiTE, developers can create concurrent applications simply by implementing the specific behaviour they require in appropriate subclasses. Recall the syntax of NT from 5.32:

$$\begin{array}{rcl} \mathcal{E},\mathcal{F} & ::= & \mathbf{0} \mid \Omega \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E}+\mathcal{F} \mid \mathcal{E}\,|\,\mathcal{F} \mid \lfloor\mathcal{E}\rfloor\sigma(\mathcal{F}) \mid \\ & & \lceil\mathcal{E}\rceil\sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E}\setminus A \mid m[\mathcal{E}]^{\mathcal{F}}_{\vec{\sigma}} \mid \mathcal{M}.\mathcal{E} \\ \mathcal{M} & ::= & \circledcirc\, m \mid \oslash m \mid \circledast m \mid on\ \beta\ \circledcirc\, m \mid on\ \beta\ \oslash\, m \mid \overline{\circledcirc} \mid \overline{\oslash} \mid \overline{\circledast} \end{array} \tag{6.1}$$

Each process term $(\mathcal{E},\mathcal{F})$ above becomes a class that implements `Process`:

```
public interface Process
  extends State
{
  Set<Transition> getPossibleTransitions();
  Process substitute(String var, Process proc);
}
```

A rough translation schema is given in 6.1, where `sigma` is an instance of `Clock`, `clockSet` is a set of instances of `Clock`, `alpha` is an instance of a subclass of `Action`, `beta` is an instance of `Name`, `mobprim` is an instance of a subclass of `MobPrim` and `proc` and `proc2` are further instances of subclasses of `Process`. Note that `Name` is a subclass of `Action` and `EnvIn`, `EnvOut`, `Open`, `ProcIn` and `ProcOut` are all subclasses of `MobPrim` while the `NIL`, `OMEGA`, `DELTA` constants are instances of the class to which they belong. Similarly, `BOUNCER_IN`, `BOUNCER_OUT` and `BOUNCER_OPEN` are instances of anonymous subclasses of `MobPrim`. Operation follows a top-down approach; the complete system is represented by a single instance of one of these classes which, in most cases, will be an operator that composes together further instances as appropriate. Table 6.2 supplements the schema, by showing how names, co-names, silent actions and clocks are converted into objects. The first three are all subclasses of `Action`, more details of which can be found in 6.5.

The `Process` interface itself extends the marker interface, `State`. This, along with `Transition`, forms part of our implementation of a labelled transition system, found under the `lts` subpackage. By making implementations of `Process` also implement `State`, they can be used as the start and finish state in the transitions represented by the `Transition` class. The label used by the transition is provided by a subclass of `Action`, which also allows for the possibility of side effects, which we cover in 6.5.

Each `Process` is required to implement `getPossibleTransitions()` and it is in this method that the operational semantics found in tables 5.2 and 5.3 are realised

Table 6.1: Translation Schema from NT to DynamiTE `Process` Subclasses

$[\![\mathbf{0}]\!]$	=	`Nil.NIL`
$[\![\Omega]\!]$	=	`Omega.OMEGA`
$[\![\Delta]\!]$	=	`Delta.DELTA`
$[\![\Delta_\sigma]\!]$	=	`new Stall(sigma)`
$[\![\alpha.\mathcal{E}]\!]$	=	`new Prefix(alpha, proc)`
$[\![\mathcal{E}+\mathcal{F}]\!]$	=	`new Sum(proc, proc2)`
$[\![\mathcal{E}\mid\mathcal{F}]\!]$	=	`new Par(proc, proc2)`
$[\![\lfloor\mathcal{E}\rfloor\sigma(\mathcal{F})]\!]$	=	`new To(proc,sigma,proc2)`
$[\![\lceil\mathcal{E}\rceil\sigma(\mathcal{F})]\!]$	=	`new STo(proc,sigma,proc2)`
$[\![\mu X.\mathcal{E}]\!]$	=	`new Rec("X", proc)`
$[\![X]\!]$	=	`new Var("X")`
$[\![\mathcal{E}\setminus A]\!]$	=	`new Res(proc, "X")`
$[\![m[\mathcal{E}]^{\mathcal{F}}_{\vec{\sigma}}]\!]$	=	`new Env("m",proc,proc2,clockSet)`
$[\![\mathcal{M}.\mathcal{E}]\!]$	=	`new MobPrefix(mobprim, proc)`
$[\![\circledv m]\!]$	=	`new EnvIn("m")`
$[\![\circledwedge m]\!]$	=	`new EnvOut("m")`
$[\![\circledast m]\!]$	=	`new Open("m")`
$[\![on\ \beta\circledv m]\!]$	=	`new ProcIn(beta, "m")`
$[\![on\ \beta\circledwedge m]\!]$	=	`new ProcOut(beta, "m")`
$[\![\overline{\circledv}]\!]$	=	`MobPrim.BOUNCER_IN`
$[\![\overline{\circledwedge}]\!]$	=	`MobPrim.BOUNCER_OUT`
$[\![\overline{\circledast}]\!]$	=	`MobPrim.BOUNCER_OPEN`

Table 6.2: Translation Schema from NT to DynamiTE Classes

$[\![a]\!]$	=	`new Name("a")`
$[\![\overline{a}]\!]$	=	`new Coname("a")`
$[\![\tau]\!]$	=	*Subclass of* ***`Tau`***
$[\![\sigma]\!]$	=	`new Clock("sigma")`

in Java code. The simplest implementation is found in the representation of Δ, realised as the class `Delta`, as it has no transitions.

```
public Set<Transition> getPossibleTransitions()
{
  return Collections.emptySet();
}
```

The other method in `Process`, `substitute(String,Process)` is primarily used to implement recursion. The arguments passed to `substitute` are the variable name and the process bound to that name respectively, and the implementation is expected to return the same process with this substitution applied.

We implement substitution in this manner so that it is independent of the syntax of the calculus. Nomadic Time is implemented in DynamiTE by deriving from classes which implement CaSE, which in turn derive from those implementing CCS. Of these, only CCS has an implementation of recursion:

```
public Set<Transition> getPossibleTransitions()
{
  Set<Transition> trans = new HashSet<Transition>();
  for (Transition t : proc.getPossibleTransitions())
  {
    Process end = (Process) t.getFinish();
    trans.add(new Transition(this,
                             end.substitute(var, this),
                             t.getAction()));
  }
  return trans;
}
```

as the same rule is used in all three calculi. Because the new constructs in CaSE and Nomadic Time all provide an implementation of `substitute`, the call to `substitute` in the CCS implementation of recursion (provided by a class called `Rec` with variables `proc` and `var`) will still work, even when one of these forms the final state, `end`.

The above implementation of recursion highlights a common pattern in the semantics, which is visible both in their formal representation and the Java version; the rules reference one or more component processes, and create new transitions based on the transitions of these processes. As a result, most of the implementations of `getPossibleTransitions()` in DynamiTE operate by looping over the set

of transitions from each component process, checking if they meet the prerequisites for one of the rules and then creating new transitions[3].

Recursion is probably one of the simplest examples of this. From its semantics,

$$\text{Rec}\ \frac{E \xrightarrow{\gamma} E'}{\mu X.E \xrightarrow{\gamma} E'\{\mu X.E/X\}}$$

we can see that it applies its transformation to all transitions (γ ranges over all possible labels), and the only change it makes is to apply substitution to E', represented in Java as the final state of the `Transition` object (`t.getFinish()`). Thus, all the new transitions returned perform the same action (`t.getAction()`), have the current instance of `Rec` as the start state and a final state derived from the original via substitution.

Implementing the | operator, via the `Par` class, is a more involved task. In CCS alone, | features in three of its operational rules: $Par1$, $Par2$ and $Par3$ (see table 2.1). CaSE adds a further rule, $Par4$, to deal with the passage of time over the operator (see table 3.1). With Nomadic Time, the first two rules are combined due to structural congruence[4], but a further five are introduced ($InEnv$, $OutEnv$, $Open$, $ProcIn$ and $ProcOut$) to handle mobility. All of these are handled in much the same way as $Par3$ (inspect the composed processes and their transitions, and apply as required) so we will just look at the CCS implementation here for brevity:

```
public Set<Transition> getPossibleTransitions()
{
  Set<Transition> trans = new HashSet<Transition>();
  // Par1
  for (Transition t : left.getPossibleTransitions())
  {
    Process nextLeft = (Process) t.getFinish();
    trans.add(new Transition(this,
                             new Par(nextLeft, right),
                             t.getAction()));
  }
  // Par2
  for (Transition t : right.getPossibleTransitions())
  {
```

[3]It is possible to make the implementations shown here more efficient, firstly by retrieving the transitions of the subprocesses simultaneously using separate threads and secondly by caching the result so that future calls don't recompute the transitions.

[4]This makes no difference to the implementation; it merely cuts down on the number of rules that need to be listed in the semantics. The case of $F \xrightarrow{\alpha}$, which is missing in the rules for Nomadic Time, is handled by a combination of $StrPar1$ ($E \mid F \equiv F \mid E$) and $SCong$

```
    Process nextRight = (Process) t.getFinish();
    trans.add(new Transition(this,
                             new Par(left, nextRight),
                             t.getAction()));
  }
  // Now find pairs for synchronisation (Par3)
  Set<Transition> syncTrans = new HashSet<Transition>();
  for (Transition t : trans)
  {
    String label = t.getAction().getLabel().getText();
    if (Context.getContext().isRegisteredName(label))
    {
      for (Transition t2 : trans)
      {
        String label2 = t2.getAction().getLabel().getText();
        if (Context.isConame(label2) &&
            label.equals(Context.convertLabelToName(label2)))
        {
          Par finish1 = (Par) t.getFinish();
          Par finish2 = (Par) t2.getFinish();
          Action sync = new Sync(t, t2);
          if (!finish1.left.equals(left) &&
              !finish2.right.equals(right))
            syncTrans.add(new Transition(this,
                          new Par(finish1.left, finish2.right),
                          sync));
          else if (!finish1.right.equals(right) &&
                   !finish2.left.equals(left))
            syncTrans.add(new Transition(this,
                          new Par(finish2.left, finish1.right),
                          sync));
        }
      }
    }
  }
  trans.addAll(syncTrans);
  return trans;
}
```

The class `Par` maintains references to the two composed processes as `left` and `right`. Thus, $Par1$ and $Par2$ are implemented by iterating over the transitions of

these processes, as with recursion. For each original transition, each iteration produces a new transition, with the `Par` instance as the start state, the same transition action as the original (`t.getAction()` and a new `Par` instance as the final state, where one argument is the unchanged process (either `left` or `right`) and the other is the final state of the original transition. Looking at $Par1$,

$$\text{Par1}\ \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F}$$

it should be clear how this corresponds to the behaviour described there, if E is `left` and F is `right`.

The majority of the method is spent handling $Par3$:

$$\text{Par3}\ \frac{E \xrightarrow{a} E', F \xrightarrow{\overline{a}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

which represents synchronisation. The implementation loops over the set of new transitions[5], searching for a process which performs a name ($E \xrightarrow{a} E'$). For each one it finds, it iterates over the transitions again, this time in search of the corresponding coname. If it finds a match, and both transitions originate from different processes, then it creates a new τ transition, using this instance of `Par` as the start state, an instance of `Sync` as the action and a final state created by joining together the two final states from each transition in a new `Par` instance. The two transitions from which the transition was derived, `t` and `t2`, are stored by the `Sync` action.

The `Par` implementations for CaSE and Nomadic Time are created by extending this class and creating further transitions, based on their additional rules. The other constructs, including +, the timeout operators and clock hiding are also implemented in much the same way. DynamiTE itself is concerned with more than just generating the transitions of a process from its semantics, however, and in the next section we see how this is handled by considering a class introduced in the implementation of `Par` above: `Context`.

6.4 The Context of the Calculus

Beyond the operational semantics, there are two important issues involved in representing a process calculus programatically:

[5]We could equally loop over the transitions of `left` and `right`, but we'd then need to store these together in another new set. It seems slightly more efficient to iterate over the new transitions and decompose them as needed, adding any resulting transitions to a new set which is then added to the other new transitions at the end.

1. We need to know the *context* in which algebraic constructions in the calculus will operate. This includes the sets of names and co-names (from CCS), the set of clocks (from CaSE) and the environ names (from Nomadic Time); the foundations on which our semantic rules are built. Although this is not essential for implementing CCS, where the names and co-names appear as part of the prefix construct $\alpha.E$, it is a necessary part of both CaSE and Nomadic Time. Both these calculi have rules defined with respect to the set of clocks, but this set is not defined by other constructs in the calculus. Instead, time is always present and we need to know the set of clocks to derive even the transitions for the **0** process using the *Idle* rule. The `Context` class maintains these, and more, in DynamiTE and we will look at this in more depth in the remainder of this section.

2. We need to know the *execution semantics* for each process. These answer questions such as: what happens when there are multiple transitions from a particular process? And are there any side-effects to performing a transition? In DynamiTE, these semantics are encoded using the `Evolver` framework, which we cover in 6.5.

In the implementation of `Par` in the previous section, we saw how the current `Context` instance could be used to find out whether or not a label referred to a name, using `isRegisteredName(String)`. In our implementation of CCS, all names and co-names are registered with the `Context`. This happens automatically on the user's behalf as part of the construction of a `Name` or `Coname` instance, or as part of generating the transitions for the renaming operation, $E[f]$. This centralised checking of names and co-names gives two primary advantages over just allowing the use of any arbitrary string:

1. We can prevent the silent action, τ, being used as a name or co-name. In CaSE and Nomadic Time, we can also prevent conflicts with clock names, environ names and the new mobility primitives such as $\circledcirc$.

2. We can enforce registration (and thus existence of the name or co-name) as a pre-requisite for other methods. This is especially useful in working with channels (see 6.4.1).

As a repository for names and co-names, `Context` becomes an appropriate place for other methods related to their use. As a result, it also includes a number of static utility methods which are used with co-names; these are `convertLabelToName`, `convertConameToLabel` and `isConame`, each of which take a single `String` argument. In our implementation, we differentiate names from co-names by marking each letter with a combining macron (so a becomes $\overline{a}$). `convertConameToLabel(String)`

creates these labels from the original name, and `convertLabelToName(String)` returns the original name by removing the macrons. The `isConame(String)` method is a simple method which just checks to see if any macrons are present in the name. The benefit of abstracting all these methods out into the `Context` class is that we can later change the way co-names are represented by modifying just one class.

The other main use for the `Context` is as the user's interface to the plugin framework. We have already touched on how DynamiTE supplies implementations for multiple calculi: CCS, CaSE and Nomadic Time. One way this is made possible is by making the implementation as generic as possible; we avoid relying on the specific structure or types in a particular calculus, separating them out into methods which can be overridden by other implementations as with `substitute(String,Process)`. Another aspect of this is allowing the user to select which calculus they want to work with at run-time. This is made possible by the plugin framework.

6.4.1 The Plugin Abstraction

In the `Par` implementation in 6.3, we obtain an instance of `Context` not by calling a construct but using `Context.getContext()`. When using DynamiTE, only one global instance of `Context` exists[6], which is created at startup by a `ContextFactory`. As with `Context`, an instance of `ContextFactory` is obtained using a static method rather than a constructor. The user calls `ContextFactory.getInstance`, supplying three `String` arguments; the name of a process calculus, a *channel implementation* and a *locality implementation*. The two latter arguments are used to determine the execution semantics for synchronisation and movement respectively. The returned `ContextFactory` will be able to supply a `Context` instance for the specified process calculus which uses the given channel and locality implementations. If one can not be found, an `UnsupportedContextException` is thrown.

DynamiTE supplies an implementation of `ContextFactory` in the form of the `DynamiTEContextFactory`. It in turn probes for classes which implement `Calculus`, `ChannelFactory` and `LocalityFactory` and an instance of it is returned to the user if it finds an implementation of each which meets the user's requirements. Once the user has obtained an instance of `ContextFactory`, they can call `getContext` on it to return a `Context` instance. This is stored using the static method `Context.setContext(Context)`, and later retrieved by `Context.getContext()`.

Although this may sound convoluted and unnecessary, it makes the framework much more flexible and ready for future extension, while giving the user greater freedom of choice. A new calculus can be implemented simply by creating an implementation of `Process` for each construct and an instance of `Calculus`. The

[6]Should DynamiTE be used across multiple host virtual machines, then there may be multiple instances, but these communicate between each other to provide one central store.

same goes for new channel and locality implementations, and the option is there for the entire `ContextFactory` to be replaced if needed. From the user's perspective, everything can be handled in a single line of code:

```
Context.setContext(ContextFactory.getInstance("CCS",
  "threaded", "dummy").getContext());
```

which supplies an implementation of CCS with threaded channels and a dummy locality implementation.

One advantage of using Java is that support for dynamically probing for implementations at runtime is built into its class library. The library itself already includes a number of frameworks which work in this fashion (including image I/O, sound and XML support) and the `java.util.ServiceLoader` API provided in 1.6 makes it easy for developers to define new ones. In the `plugin` subpackage, DynamiTE provides a means of using this API to support plugins:

```
public static <T extends Probeable> Map<String,T>
  probePlugins(ServiceLoader<T> sl)
{
  Map<String,T> map = new HashMap<String,T>();
  for (T probeable : sl)
  {
    Config.logger.config(String.format("Loaded plugin: " +
      "%s %d.%d.%d%s", probeable.getName(),
      probeable.getMajorVersion(), probeable.getMinorVersion(),
      probeable.getMicroVersion(), probeable.getAdditionalInfo()));
    map.put(probeable.getName(), probeable);
  }
  return Collections.unmodifiableMap(map);
}
```

Plugins are required to implement the interface `Probeable`, so that the name and version information can be obtained programatically. In DynamiTE, the `Calculus`, `ChannelFactory` and `LocalityFactory` interfaces all extend `Probeable` so `DynamiTEContextFactory` need only call the above method with an appropriate service loader for the interface and it will receive back a `Map` linking names to instances of implementations of that interface. Most of the actual work is done by `ServiceLoader` which reads from a text file named after the interface, which contains a list of implementing classes. The DynamiTE framework supplies such text files for its implementations of `ContextFactory`, `Calculus`, `ChannelFactory` and `LocalityFactory`. The `ServiceLoader` loads a listed class each time its `next()` method is called[7], and returns an instance of it, which `probePlugins` then stores in

[7]This happens indirectly in `probePlugins` via each iteration of the for-each loop.

the map.

The `Context` class stores and provides indirect access to the implementations chosen via the `ContextFactory`. For instance, calling `getSyntax()` on the current `Context` instance will return the set of syntactic constructs which form the calculus currently in use. The `Calculus` instance is also used in the process of registering a name; the `Context` calls the `Calculus` implementation to obtain the transition label for the name, which gives the `Calculus` a chance to veto the choice. This is used by the `CCS` class to prevent τ being used as a name, by returning an instance of `CCSLabel`, the constructor of which contains the following check:

```
if (label.equals(TAU))
  throw new IllegalArgumentException(TAU +
                                     " is a reserved label.");
```

where `TAU` is a unique instance of `CCSLabel`. The other instances maintained by `Context` are used in the execution semantics which we will cover next.

6.5 The Evolver Framework

For DynamiTE to actually be useful to users for building concurrent applications, it needs to do something more than just evaluating an algebraic construct and providing the possible transitions according to the semantics of the calculus. For an application, such as our music player example (see 1.3.1 and 5.7), to actually work, the user needs to be able to define their own internal behaviour and share the results.

The evolver and channel frameworks provide this facility. An implementation of the `Evolver` interface implements the method `evolve(Process)` according to its own particular execution semantics. Through this method, it is the `Evolver` instance that makes decisions such as which transition to follow to find the next state and also whether to process any side effects. Side effects take the form of additional methods which may optionally be called after a transition has been followed. As we saw in 6.3, each transition references an `Action`; this is an abstract class which provides a method `perform()` for the purpose of implementing side effects.

DynamiTE provides a simple implementation of `Evolver` called `Simulator` which ignores side effects. While this is of little use for applications, it is useful for testing design constructs as it allows the possible transitions from a process to be visualised. All `Simulator` does is take a `Process` and loop over its transitions, calling itself recursively with each final state. In this way, it explores the possible transitions in a depth-first manner, until it reaches a process with either no transitions or where all transitions have equal start and end states. The latter condition prevents it looping

forever over states with just clock transitions or simple forms of infinite recursion, such as $\mu X.a.X$.

```
public void evolve(Process p)
{
  System.out.println("Evolving process: " + p);
  Set<Transition> trans = p.getPossibleTransitions();
  System.out.println("Possible transitions: " + trans);
  for (Transition t : trans)
  {
    State f = t.getFinish();
    if (f instanceof Process)
    {
      if (f.equals(p))
        System.out.println("Not following transition " + t);
      else
      {
        System.out.println("Following transition " + t);
        evolve((Process) f);
      }
    }
  }
}
```

A more practical `Evolver` is a more complex undertaking; further research in this area is suggested in 8.2.2. It has to make choices as to which transition to pick when several are presented; although CaSE and Nomadic Time have a notion of priority in maximal progress, choices must still be made between the ticks of different clocks, or between a clock tick and an action. These choices form the execution semantics of an `Evolver` implementation, and there is plenty of room for further experimentation in this area.

Returning to the notion of side effects, the channel framework is accessed through four subclasses of `Action`, three of which are also used in `Prefix`, the implementation of $\alpha.E$:

1. The class `Name` is used to represent the use of a name as part of the process $\alpha.E$. It is also a subclass of `Action` and implements `perform()` by reading from an `InputChannel` and storing the result.

2. Likewise, the class `Coname` represents the use of a co-name in $\alpha.E$ and implements `perform()` by retrieving a value from storage and transmitting it over an `OutputChannel`.

3. The `Tau` class is the last of the three classes used in $\alpha.E$ and is used for the internal action, τ. Implementing `perform()` for `Tau` is left to the user, who can use it to implement arbitrary sequential behaviour as required.

4. The `Sync` class is created through `Par` (see 6.3) and is a subclass of `Tau` which implements `perform()` using the two synchronising transitions provided to it on construction.

The `InputChannel` and `OutputChannel` instances are obtained from the `ChannelFactory`, via the `Context`, and provide `read()` and `write(Object)` methods respectively. Although there is currently no realisation of data within the formal layer of the calculus, this only matters to the extent that we wish transmitted data to alter the constructs themselves via substitution[8]. Data can be transferred between processes and used within internal actions without having to be explicitly realised at the formal level. The operation of these I/O operations, and the creation of a suitable environment in which this may happen, is left to the implementation of the `ChannelFactory` and there are a multitude of ways of doing so. These range from simple mechanisms like files and sockets to complex interprocess communication protocols such as Java's Remote Method Invocation (RMI), the Common Object Request Broker Architecture (CORBA) and web services. The plugin nature of the channel architecture means that any of these possibilities may be used and more besides.

DynamiTE provides a sample implementation, `ThreadedChannelFactory`, built on a `SynchronousQueue`; the `read` and `write` operations are performed by synchronising two different threads and transferring the data directly. Thus, if `read` is called, and another thread is not already waiting in the `write` method, it will block until this is the case. The same is true for `write`. As noted above, the `ChannelFactory` is also responsible for creating the necessary environment for these operations, so the `ThreadedChannelFactory` has to ensure that appropriate threads are created and used. A hook, `runInParallel(Process, Process)` is used for this purpose and is called indirectly by the constructor of `Par`.

Data storage is also provided by the `ChannelFactory`. The channel I/O operations are automated side-effects of following a transition labelled with a name or co-name, so user code (implemented in a `Tau` subclass) must be able to access any values retrieved and store new ones when it is itself performed. The `ChannelFactory` provides storage repositories, keyed by the name of the channel, for this purpose. When a user wishes to transmit a value, they call `store(String,Object)`, where

[8]The π calculus (see 4.2.1) is an obvious example of such behaviour, which goes to the extreme of not only allowing data to be transferred but also references to channels which can then later be used in the language constructs. This, in essence, provides the form of mobility present in the π calculus.

the first argument is the channel name and the second the data to store. Later, the performance of a `Coname` with that channel name will lookup the data and transmit it to the corresponding `Name`. User code in a further `Tau` implementation can then retrieve this using `retrieve(String)`, passing to it the name of the channel and receiving back the data. The `ThreadedChannelFactory` implements this by storing data in a `ThreadLocal`, so that it is only retrievable by the same thread that stored it:

```
public void store(String name, Object data)
{
  ThreadLocal<Object> store = repositories.get(name);
  if (store == null)
  {
    ThreadLocal<Object> newStore = new ThreadLocal<Object>();
    store = repositories.putIfAbsent(name, newStore);
    if (store == null)
      store = newStore;
  }
  store.set(data);
}
```

The `repositories` variable stores an instance of `ConcurrentHashMap`, which guarantees that retrieval operations will always return results which reflect the most recently completed operations. As we described in 6.2.1, `ConcurrentHashMap` is thread-safe while also being much more efficient than `Hashtable` which locks the entire collection on any operation. These guarantees do not, however, help in performing a sequence of operations on the map; specifically, the map may still be changed by another thread in the interim period between performing a `get` and a `put`.

This applies to the method above, as we need to check whether or not a `ThreadLocal` instance has been created for a particular channel name, and if not, create one. The initial `get` operation will return `null` if we have not yet created a mapping between that channel name and a `ThreadLocal`. However, in the time it takes for this thread to check the return value and create a `ThreadLocal`, another thread may have added such a mapping. The `ConcurrentMap` interface (which `ConcurrentHashMap` implements) provides an additional operation for just this issue: `putIfAbsent`. This provides an atomic version of the following code:

```
if (map.containsKey(key))
  return map.get(key);
else
  return map.put(key, value);
```

If, as in the most likely case, a mapping has not been added since our initial `get`, then the `putIfAbsent` call folds down to a normal `put` call. The `put` method returns either the previous value of the mapping or `null` if there wasn't one. In this case, it will always return `null` as the `containsKey` call has already ensured no mapping exists. Thus, if `null` is returned, we know that our new `ThreadLocal` was used for the mapping. If the return value is not `null`, then another thread managed to add a mapping before us, and the `ThreadLocal` we created is simply discarded when we leave the scope of the `if` block in which it was created.

The complexity of this operation again shows how difficult it can be to make operations involving shared objects thread-safe. The advantage of using DynamiTE is that the user doesn't have to come into contact with this. The user merely store a value and the framework handles the thread safety issues involved. Providing these kind of reusable generic constructs makes it much easier to build concurrent applications that are thread-safe.

From this discussion, it should now be clear how concurrent applications are created in DynamiTE; the user constructs `Process` instances which match their design in the process calculus, and provides `Tau` subclasses for the actual work performed by the application. Data is passed between `Tau` subclasses via the channel framework.

DynamiTE also provides hooks for a locality framework, which allows side effects to take place on the performance of mobility actions, just as the channel framework provides side effects for synchronisation. At present, DynamiTE simply has a dummy implementation for this, but this has great scope for being used to implement process migration. Migrating an active process is not a simple operation – not only must the continuation of the code be transferred, but any local data must also migrate. Using the Nomadic Time process calculus as our basis allow us to achieve a significant amount of simplification here; the transferred process is already separated from other code within the system by virtue of the moving process being in the form of a `Prefix` instance. When the action is matched to the one used for the mobility operation, the `Process` instance is transferred to its new location. There is no need to deal with code that is currently being executed. We also know exactly what data to be transferred, as this is centrally managed by the channel framework. Again, this is an interesting area for future work (see 8.2.2).

Now we have explored DynamiTE, the next section provides a walkthrough example of creating an application using the framework, following on from 1.3.1 and 5.7. We also compare this implementation with a standard implementation using low-level concurrency primitives, thus allowing us to evaluate whether our claims made in 1.3 hold true.

6.6 A Prototypical Application in DynamiTE

In 5.7, we created a design for the music player application using the Nomadic Time calculus. This resulted in a system defined by the following equations:

$$\begin{aligned} Out &\stackrel{\text{def}}{=} o.\lceil\Delta\rceil\sigma(\tau.\mathbf{0}) \\ Analy &\stackrel{\text{def}}{=} o.\lceil\Delta\rceil\sigma(\tau.\mathbf{0}) \\ IntSys &\stackrel{\text{def}}{=} i.\mu X.(\tau.\mu W.\lceil\overline{o}.W\rceil\sigma(X) \mid Out \mid Analy) \\ Intf &\stackrel{\text{def}}{=} useri.(\overline{i}.\mathbf{0}|IntSys) \\ App &\stackrel{\text{def}}{=} player[Intf]^{\overline{\emptyset}.\Omega}_{\{\sigma\}} \end{aligned} \tag{6.2}$$

In this section, we will see how these constructions can be turned into Java objects. The process is fairly simple, and results in a system which leverages the existing concurrency work performed in the development of the DynamiTE framework. We then look at how the same application could be written without the framework, using objects in shared memory.

The operators in the above all map to Java classes as we saw in 6.3; Δ is implemented by `Delta`, the stable timeout by `STo`, recursion by `Rec` and the environ by `Env`. The channels, i, o and $useri$ are represented using instances of `Name` and `Coname`. Thus, all that remains is the internal silent actions τ.

Each silent action is represented by a different subclass of `Tau` and its `perform()` method is implemented so as to do the actual work required of the application. The implementation for the silent action used in Out looks something like this:

```
public class Out
  extends Tau
{
  public void perform()
  {
    Context ctx = Context.getContext();
    Object soundData = ctx.retrieve("o");
    if (soundData != null && soundData instanceof byte[])
    {
      play((byte[]) soundData);
    }
    else
    {
      throw new InternalError("Didn't receive sound data.");
    }
  }
```

```
}
```

The `perform()` method first retrieves the data from the `"o"` repository, where it should have been placed earlier by the o action. If it is successfully, the data is played out on the speakers. In the unlikely case that something went wrong, and the expected data is not held in the repository, we throw an error.

The implementation for the silent action in *Analy* is identical, except that the sound data retrieved is visualised rather than played. The one for In is slightly more complicated, as we have to both store and retrieve from different repositories:

```
public class In
  extends Tau
{

  private InputStream is;

  public void perform()
    throws IOException
  {
    if (is == null)
    {
      Context ctx = Context.getContext();
      Object fileName = ctx.retrieve("i");
      is = new FileInputStream((String) fileName);
    }
    byte[] soundData = readAndProcessData(is);
    ctx.store("o", soundData);
  }
}
```

This time we allow the use of the retrieved value by the file routines throw up any issues. If this is the first time the τ action is encountered, then the filename is retrieved from the `"i"` repository where it was stored by the earlier i action and an `InputStream` created so that data may be read from the file. After this, and on each subsequent invocation, data is read from the stream, processed and store in a `byte` array in the `"o"` repository.

Now we have our silent actions, we can construct our processes:

```
Context.setContext(ContextFactory.getInstance("NT",
  "threaded", "dummy").getContext());
Context ctx = Context.getContext();
```

```
Clock sigma = new Clock("\u03C3"); \\ u03C3 = sigma in Unicode
Name output = new Name("o");
Process out = new Prefix(output,
  new STo(Delta.DELTA, sigma, new Prefix(new Out(), Nil.NIL));
Process analy = new Prefix(output,
  new STo(Delta.DELTA, sigma, new Prefix(new Analy(), Nil.NIL));
Process inLoop = new Prefix(new In(), new Rec("W",
  new STo(new Prefix(new Coname("o"), new Var("W")), sigma,
          new Var("X"))));
Process intSys = new Prefix(new Name("i"),
  new Rec("X", new Par(new Par(inLoop, out), analy)));
Process intf = new Prefix(new Name("useri",
  new Par(new Prefix(new Coname("i"), Nil.NIL), intSys)));
Set<Clock> hiddenClocks = new HashSet<Clock>();
hiddenClocks.add(sigma);
Process app = new Env("player", intf,
  new MobPrefix(MobPrim.BOUNCER_IN, Omega.OMEGA),
  hiddenClocks);
```

If the above is compared with Eqn. 6.2, it should be clear how each term is turned into a instance of a Java class. The references to `Delta.DELTA`, `Nil.NIL`, `MobPrim.BOUNCER_IN` and `Omega.OMEGA` refer to singleton instances, as none of these have any variable attributes, and the constructors of `Name`, `Coname`, `Clock` and `Env` register the new entity with the `Context`. At the end of running this code, we are left with a `Context` containing the names `i`,`o` and `useri`, the conames `i` and `o`, the clock σ, the environ `player`, and an instance `app` which can be passed to an `Evolver` instance to run the application.

The same application can be implemented without DynamiTE in innumerable ways, but one thing holds for all of them; they must either include provisions to ensure thread safety or be purely single threaded. Writing such an application using a single thread produces an unworkable result; while the sound data is being output, nothing else can be done so any visualiser will be out of sync with the sound being played. Additionally, no data will be being read while this is happening, so the application has to rely on there being enough time between the data has been sent to the speakers and the actual sound finishing for it to read more data and send it. This becomes even more unlikely when the visualiser is factored in.

Thus, we will assume that any sensible implementation, like our DynamiTE-based application, will use a thread for each of the input, output and visualisation processes. Unlike with DynamiTE, we now have to consider how the sound data will be stored and how it will be shared between threads. For simplicity, we start by just considering the input and output threads:

```
public class Player
{

  private BlockingQueue<byte[]> queue;

  private String fileName;

  public void input()
  {
    InputStream is = new FileInputStream((String) fileName);
    while (true)
    {
      byte[] soundData = readAndProcessData(is);
      queue.put(soundData);
    }
  }

  public void output()
  {
    while (true)
    {
       byte[] soundData = queue.take();
       play(soundData);
    }
  }

  public static void main(String[] args)
  {
    fileName = args[0];
    queue = new LinkedBlockingQueue();
    Thread input = new Thread(new Runnable()
    {
      public void run() { input(); }
    }, "input");
    Thread output = new Thread(new Runnable()
    {
      public void run() { output(); }
    }, "output");
    input.join();
    output.join();
```

```
  }
}
```

We've kept the implementation as close as possible to that for `In` and `Out` above, including not handling the end of the file. Realistically, the thread should terminate when this happens. Although the similarities between the two should be clear, so should the differences. In this example, we have had to consider both how data is stored and how the methods are run; these are handled by the channel and evolver frameworks respectively in DynamiTE. The `BlockingQueue` is thread-safe, so the threads can not corrupt the data structure if two or more happen to try and perform an operation on the queue at the same time. This is the same as the locks used in 1.2.3 with the locking occurring inside the collection rather than visibly in the surrounding code. The `take` operation also blocks if the queue is empty until an item is added[9]; again the behaviour inside the queue is akin to what we saw with signalling in 1.2.3.

This works fine for this example, but there is an immediate problem if we want to introduce the visualiser into the mix; the `take` operation performed on the queue by both `output` and the new `visualise` method will remove the item from the queue so one thread will get a particular value and the other one won't. There are a number of possible solutions to this. For example:

1. We can replace the queue with an indexed collection and remember which index was last used. The new collection also has to be thread-safe and we put ourselves at risk of running out of memory as the queue will only ever increase in size.

2. We add each item to the queue twice. This makes the `input` method dependent on the number of consuming threads. It also means we have to add our own external locking to ensure that one of the consumers does not perform a `take` while the items are being added, and we have to check on each iteration that we are getting a new value and not a copy of the previous one still waiting to be taken by the other thread.

Below we provide an implementation of the second solution:

```
public class Player
{

  private static final int NUMBER_OF_CONSUMERS = 2;
```

[9]The `put` operation also blocks if a capacity is given for the list on creation; we don't do so here.

```
private Queue<byte[]> queue;

private String fileName;

public void input()
{
  InputStream is = new FileInputStream((String) fileName);
  while (true)
  {
    byte[] soundData = readAndProcessData(is);
    synchronized (this)
    {
      for (int a = 0; a < NUMBER_OF_CONSUMERS; ++a)
        queue.offer(soundData);
      notifyAll();
    }
  }
}

public void output()
{
  byte[] soundData = null;
  while (true)
  {
     synchronized (this)
     {
       while (queue.peek() == soundData ||
              queue.peek() == null)
       {
         wait();
       }
       soundData = queue.poll();
       notifyAll();
     }
     play(soundData);
  }
}

public void visualise()
```

```
  {
    byte[] soundData = null;
    while (true)
    {
       synchronized (this)
       {
         while (queue.peek() == soundData ||
                queue.peek() == null)
         {
           wait();
         }
         soundData = queue.poll();
         notifyAll();
       }
       visualise(soundData);
    }
  }

  public static void main(String[] args)
  {
    fileName = args[0];
    queue = new LinkedList();
    Thread input = new Thread(new Runnable()
    {
      public void run() { input(); }
    }, "input");
    Thread output = new Thread(new Runnable()
    {
      public void run() { output(); }
    }, "output");
    Thread analy = new Thread(new Runnable()
    {
      public void run() { visualise(); }
    }, "analy");
    input.join();
    output.join();
  }
}
```

This is clearly more complicated than the example with just input and output; although it still decomposes nicely, we have to handle synchronisation ourselves.

We dispense with the `BlockingQueue` as we need to obtain a lock anyway to ensure the atomicity of the multiple `offer` calls in `input` and the checks in `output` and `visualise`. Both the `output` and `visualise` methods are very similar and it would be a very good idea to generalise these in a `Consumer` superclass; as we noted in 1.2.3, the correct placement of locking and signalling constructs is prone to simple errors so anything we can do to simplify this and minimise the risk is advantageous.

The consumer methods now loop until `peek` returns a new unseen value. When we reach this loop, `peek` may return one of three things:

1. It may be `null` if nothing has yet been added or both values from the last run have been retrieved.

2. It may equal the previous value, indicating that the other consumer has not yet read its data.

3. It may be a new unseen value which is neither `null` or equal to `soundData`. In this case, we exit the loop, remove the value from the queue and notify any waiting threads that the thread has changed.

If either of the first two conditions hold, `wait` is called, causing the thread to relinquish its lock and sleep until notified by a call to `notifyAll`. This solution should be thread safe as presented, but it should also be obvious how a simple misplaced call could break this; thread safety is a very fragile property.

It should be clear from this example that DynamiTE makes implementing this a lot simpler; not only do we not have to worry about locking the data structure and signalling other processes at the correct points, we have a solution which works for any number of consumers without changing `input`. The only way that would be possible here is by a hack to use the names of threads as identifiers whether they are consumers or not; not a very elegant solution. The abstraction in DynamiTE also means that a simple one line change to the channel factory being used can turn an application communicating between threads as above, to one communicating over a network with no other changes to the application.

In summary, we think DynamiTE has achieved its goals; it abstracts away many complex and fragile pieces of code which ensure thread safety, in much the same way as the concurrency collection classes do in the class library. They are replaced by simple abstract concepts such as `store` and `retrieve` calls to the repositories. In implementing DynamiTE, we believe we have also made it approachable for existing Java programmers, both for building their own applications and for contributing to DynamiTE itself. By learning the meaning of a small number of algebraic concepts, they can leverage the power of DynamiTE to create applications limited only by the plugin factories being used.

6.7 Related Work

In this final section, we look briefly at the existing body of research concerned with providing concurrent frameworks, including those based on process calculi.

The π calculus has been the subject of much of this work, primarily due to its status as the most prevalent mobile process calculus. Obliq (Cardelli, 1995) and Pict (Turner, 1996) are both programming languages with semantics founded in the π calculus, while Nomadic Pict (Wojciechowski, 2000) extends Pict by introducing distribution, a feature not usually present in the π calculus. The Seal calculus (Vitek & Castagna, 1999) is an example of yet another distributed variant of the π calculus, and this has also lead to an implementation in the form of JavaSeal (Vitek & Bryce, 2001). However, the ambient calculus has not been neglected, and an implementation exists in the form of the safe ambients abstract machine (Giannini, Sangiorgi & Valente, 2006). In the remainder of this chapter, we look at these implementations in more detail.

6.7.1 Obliq

Obliq was originally developed by Luca Cardelli in 1995, prior to his work on the ambient calculus. While being an object-oriented language, it has no notion of classes in the same way that languages like C++ and Java do. Objects are instead constructed directly and assigned to variables:

```
let o =
  {
    x => 3,
    inc => meth(s,y) s.x := s.x+y; s end
    next => meth(s) s.inc(1).x end
  }
```

In the above example, an object is created with two methods, `inc` and `next`. The first argument of a method is explicitly named (`s` in the above) but always contains a reference to the object itself (`this` in Java), rather than some argument passed by the method call. The object instance is assigned to `o`, so the only way of executing its methods or manipulating its values is via either `o` itself or a clone of it (created by `clone(o)`. The method `inc` increases the value of `x` by `y`, while the method `next` uses this method to give the next value of `x`. Methods implicitly return the value computed by their body; `s` and `s.inc(1).x` respectively in this case.

Objects in Obliq contain only fields, but these fields can contain *methods*, *aliases* and *values*. The latter including procedures, which differ from methods in not having a reference to the object as their first argument. Fields are dynamically typed, so,

for example, even if `x` is given the value `3` initially, it can later be assigned a method. An alias allows an operation to be redirected; for example `a.x := alias y of b end` makes any attempt to access `a.x` equivalent to accessing `b.y`.

The main feature of Obliq is that it contains concurrency and remote invocation primitives. The following implements the producer/consumer queue we saw in 1.2.3 in Obliq:

```
let queue =
  (let nonEmpty = condition();
   var q = [];

   { protected, serialized
     write =>
       meth(s,elem)
         q := q @ [elem];
         signal(nonEmpty);
       end,
     read =>
       meth(s)
         watch nonEmpty until #(q)>0 end;
         let q0 = q[0];
         q := q[1 for #(q)-1];
         q0;
       end;
   }
  );
```

The field `nonEmpty` is assigned a condition queue, like the implicit one provided to all objects in Java, while `q` stores the data. The `protected` modifier prevents external modification to the fields (the equivalent of declaring them all `private` in Java), while `serialized` is akin to acquiring a mutex at the beginning of each method and releasing it at the end. The latter thus ensures that the queue is only manipulated by one thread at a time, and also works in tandem with the condition queue. The `watch` statement is equivalent to `wait`, except that it is slightly safer as the need to loop over a condition is tied to the waiting process by the use of unique syntax as opposed to a method call. The statement translates to `while (q.length() == 0) { wait(); }` in Java. Likewise, `signal` is equivalent to `notifyAll`.

The following code:

```
let t = fork(proc() queue.read() end, 0);
queue.write(3);
let result = join(t);
```

shows how the queue can be used. A separate thread, `t`, is forked to read from the queue. This is necessary as the call will block until the queue's length becomes greater than zero. We then write the value 3 to the queue, and wait using a `join` call for `t`. Once `t` has awoken and read from the queue, it should return a result of 3 which ends up in `result`.

Remote invocation works in the same style as protocols such as CORBA (OMG, 2009) and Java RMI (Sun, 2008); a name server allows objects to be registered with a name, allowing them to be looked up and retrieved. The following Obliq code implements object migration:

```
let migrateProc =
  proc(obj, engineName)
    let engine = net_importEngine(engineName, namer);
    let remoteObj = engine(proc(arg) clone(obj) end);
    redirect obj to remoteObj end;
    remoteObj;
  end;
```

The procedure takes two arguments: the object to migrate (`obj`) and the remote site to migrate it to (`engineName`). The invocation of `net_importEngine` obtains a reference to an *execution engine* from the name server, `namer`. Execution engines accept a procedure as an argument, which is then run at the remote site; thus `clone(obj)` is run not locally but at the site of `engineName` so that `remoteObj` ends up being a reference to the remote clone. The `redirect` statement then acts as a shorthand for aliasing each field in the local `obj` to point to `remoteObj` so that future invocations on `obj` access the remote clone.

Obliq has two advantages over traditional languages:

1. Threads, condition queues and remote invocation are built into the language as primitives, so developers do not have to depend on external libraries.

2. It can be given a semantics using the π calculus (Merro, Kleist & Nestmann, 2002), which allow its form of migration, known as *object surrogation*, described above to be proved correct.

However, on the downside, Obliq requires the user to learn the syntax and semantics of a completely new language, as well as giving up any existing libraries that may be available for development in the language they traditionally work in. It also doesn't particularly simplify anything; locking still has to be performed at the same intricate level of granularity as in a language like Java, while foregoing the much greater base of experience and libraries available in that language. This is unlike our framework, DynamiTE, where locking and transmission of data is abstracted

away via the use of storage repositories, and the user can work in the familiar Java programming language, thus having only to learn a fairly small API in order to use the framework. In all, Obliq is interesting as a piece of early research in this area, but we don't see it being of much practical use.

6.7.2 Nomadic Pict

Pict is a programming language based on the asynchronous π calculus (see 4.2.2). Nomadic Pict extends this by adding in a notion of distribution; users can create *agents* which migrate between *sites*, and channels are located at a particular site. An example Nomadic Pict program looks like this:

```
new answer : ^String
  def spawn [s:Site prompt:String] =
    (agent b =
      (migrate to s
       <a@s'>answer!(sys.read prompt))
     in
      ()
    )
  (spawn ! [s1 "How are you? - "]
  |spawn ! [s2 "When does the meeting start? - "]
  |answer ?* s = print!s)
```

The code runs inside an agent `a` located at the site `s'`. It defines a function `spawn` which, when called, creates an agent `b` which migrates to the supplied site `s`. Once at `s`, the agent attempts to output on the channel `answer` which is located back at site `s'` and attached to agent `a`, having being created by the `new` statement. The value transmitted is first input by the user, who is first prompted with the string supplied to the `spawn` function.

The agent `a` itself spawns three processes, two of which call `spawn`. The third forms the other end of the communication with agent `b` by waiting for input on `answer`. The use of `?*` for input as opposed to just `?` makes the input replicated using the ! operator from the π calculus, so it is always available. When the program is run, the first two processes will migrate to `s1` and `s2` respectively. They then prompt the user with their respective messages, and return the user's input to agent `a` via the `answer` channel. The third process in `a` prints whatever is received.

Perhaps the clearest thing about Nomadic Pict from this example is that it is not the most readable of languages. Being designed primarily as a way of programming in the π calculus rather than as a usable language means that the syntax leads something to be desired and is fairly inaccessible for those who don't know the calculus in

detail. On the positive side, this does give the language a strong formal background and semantics, and unlike Obliq, it has a rich type system with polymorphism and subtyping. This can be seen above where `answer` is declared as `^String`, a channel type which both inputs and outputs values of type `String`. This type itself is a subtype of both `!String` (an output channel which sends a `String`) and `?String` (an input channel expecting a `String`).

Most of the criticisms we had of Obliq do apply to Nomadic Pict however. While it represents interesting research, and could prove very useful in some areas, it is not suitable as a general purpose language for the masses.

6.7.3 The Safe Ambients Abstract Machine

Something closer to our work on DynamiTE is the abstract machine PAN for safe ambients (Giannini, Sangiorgi & Valente, 2006). Safe ambients were discussed back in 4.3.3; they provide a form of the ambient calculus which is protected from *grave interferences* by requiring each mobility action (*in*, *out* and *open*) to be matched by a corresponding co-action. We adopt the same idea ourselves in Nomadic Time with bouncers, but unlike the co-actions in the safe ambient calculus, ours must only be used by the bouncer which is located on each environ.

There are also similarities between our implementation, DynamiTE, and PAN, as both are implemented in Java. They differ, however, in that with PAN, the implementation takes the form of an abstract machine which is first formally specified and then implemented in Java. Rather than attaching Java code to certain terms in the calculus, as in DynamiTE, PAN users write their programs in the abstract machine, and the code is then compiled and executed by the Java implementation, which maintains a 1:1 relationship between ambients and threads. As this means the input to PAN is a slightly extended form of the safe ambient calculus, the present implementation described in (Giannini et al., 2006) would need to be extended further 'to embed this core language into a real language'. Interestingly, the authors also mention that an 'orthogonal direction would be to make the ambient constructs into a framework'; this sounds very similar to what we have now with DynamiTE. Sadly, we have been unable to find any details of further work beyond this paper.

The majority of the paper is concerned with the design of the abstract machine itself, rather than the implementation. The main focus is on clearly defining the separation between the logical distribution of the ambients derived from terms in the safe ambient calculus, and the physical distribution used by the machine. This work is fairly generic, and could also be applied as a means to formalise and clarify the use of environs within DynamiTE; these also have a logical distribution given by Nomadic Time and a physical distribution as defined by the `LocalityFactory`.

A term in the safe ambient calculus is mapped onto a flat physical topology of *located ambients*. For example, given the following safe ambient construct:

$$n[P_1 \mid P_2 \mid m_1[Q_1] \mid m_2[Q_2]] \tag{6.3}$$

A located ambient has the form $h : n[P]_k$, where h is the location of the ambient and k is the location of the parent. For the above, we end up with three locations: h, k_1 and k_2, all of which co-exist at the same level; there is no hierarchical topology to the locations as there is for the ambients. This gives us $h : n[P_1 \mid P_2]_{root}$, $k_1 : m_1[Q_1]_h$ and $k_2 : m_2[Q_2]_h$, where the subambient relationship between n and m_1, and likewise n and m_2, is represented by the use of h in the latter two terms rather than by physical placement.

This changes how the mobility operations proceed as well; *in* and *out* have no effect on the physical structure, as they simply change the parent location. The *open* operation does cause a physical move, as the processes within the destroyed ambient are moved into the parent. For example, if $P_1 = open\ k_1.P_1'$ and $P_2 = \overline{open}\ k_1.P_2'$, then k_1 would be destroyed and Q_1 would move into h, giving $h : n[P_1' \mid P_2' \mid Q_1]_{root}$.

This representation makes a lot of sense, and relates closely to how we foresee a full `LocalityFactory` implementation operation mapping the logical to the physical. Thus, although work on the implementation of PAN seems to have come to a halt, we can make use of the research the project produced by feeding it into the development of DynamiTE.

6.7.4 JavaSeal

The final piece of related work we will consider is the Seal calculus (Vitek & Castagna, 1999). This is probably the closest of those covered here to DynamiTE; it was designed with implementation in mind (Vitek & Bryce, 2001), specifically 'secure distributed applications', and uses a similar technique of 'cherry-picking' some of the best features from other calculi to create a new one that best fits the proposed goal of the project.

The implementation of the Seal calculus, the JavaSeal Mobile Agent Kernel, also uses a similar approach to DynamiTE, with the constructs of the calculus represented as objects:

```
public abstract class Seal
  implements Runnable, Serializable
{
  public static Seal currentSeal();
  public static void dispose(Name subseal);
  public static void rename(Name subseal, Name subseal);
  public static SAF wrap(Name subseal);
  public void run();
}
```

A `Seal` instance is *wrapped* for migration by stopping its threads and serialising its contents into a byte array. Both when a seal is created and when it is *unwrapped* following migration, a `Strand` is created for it and the `run` method is invoked. Each `Strand` instance is bound to the particular `Seal` instance that created it, and provides the necessary mapping on to Java threads.

Formally, just as Nomadic Time combines the ambient calculus with CaSE, the Seal calculus takes the synchronous polyadic π calculus, and adds localities to it, in the form of *seals*. One of the main differences between Nomadic Time and the Seal calculus is that the latter uses channels for migration, rather than applying the ambient set of mobility primitives (*in*, *out* and *open*). Seal migration takes place objectively over channels, with the local process as the sender and the remote process as the recipient.

The action prefix, $\alpha.E$, in the Seal calculus has four forms:

1. $\overline{x}^{\eta}(\vec{y})$
2. $x^{\eta}(\lambda\vec{y})$
3. $\overline{x}^{\eta}\{y\}$
4. $x^{\eta}\{\vec{y}\}$

The first two handle the communication of names, as in the π calculus, and the second two are for the transmission of seals. Channels exist within a certain location, and η is used to direct the communication. It takes one of three values: $\eta ::= \star | \uparrow | n$, where $\star$ refers to a local channel, $\uparrow$ to a channel in the parent seal and n to a channel in a child seal. Communication occurs between either a pair of corresponding local prefixes, or between a local prefix and a remote prefix such as $x^{\star}(\lambda y)$ and $\overline{x}^{n}(z)$, or $x^{\star}(\lambda z)$ and $\overline{x}^{\uparrow}(y)$. Access restriction can be enforced via the use of *portals* and the $open_{\eta}x$ syntax; for remote interaction to take place, the corresponding *open* action must be provided by a process running in parallel with the process offering the local prefix, in the same way that a co-capability must be provided in the Safe Ambients calculus or a bouncer in Nomadic Time. For example, in

$$open_{n}\overline{x}.S_1 \mid x^{\star}(\lambda z).\overline{z}^{n}().S_2 \mid n[\overline{x}^{\uparrow}(y).open_{\uparrow}\overline{y}.P_1 \mid y^{\star}().P2] \tag{6.4}$$

the provision of $open_{n}\overline{x}$ allows communication to occur between $x^{\star}(\lambda z)$ and $\overline{x} \uparrow (y)$, and $open_{\uparrow}\overline{y}$ does likewise for $\overline{z}^{n}()$ and $y^{\star}()$.

As with PAN, it seems that development on the Seal calculus has stopped, especially as regards the implementation work. Further research into the calculus itself has been performed (Castagna, Ghelli & Nardelli, 2001; Castagna & Nardelli, 2002) since the initial publication of the Seal calculus, but this now also seems to have

finished. Unfortunately, it seems common in the academic community for implementations to be written as a proof of concept, but then not taken any further. Outside academia, even the newest approaches are still relatively low-level; the latest offering is CUDA (NVIDIA, 2009), a C-style parallel processing language for programming NVIDIA graphics processing units (GPUs). The development direction with frameworks and languages like CUDA is usually the inverse of the academic case; little research goes into formalising it but it does get heavily used in real-world situations to produce results, some of which may be true.

6.8 Conclusion

As far as implementations go, we believe our work to be novel in approaching the task of translating a calculus with both global discrete time and mobility into a usable programming framework. We also seem to be one among only a few research projects to create an implementation in an existing programming language and allow users to work with it in that language; of the above, this only applies to JavaSeal. This is a shame, as it means otherwise good ideas are let down by the implementation being simply unapproachable for the majority of software developers.

Through our example in 6.6, we saw how DynamiTE achieves its goal of simplifying the creation of concurrent applications by abstracting away many complex and fragile pieces of code which ensure thread safety, replacing them with simple abstract concepts such as `store` and `retrieve` calls to the repositories. By learning the meaning of a small number of algebraic concepts, programmers can leverage the power of DynamiTE to create applications limited only by the plugin factories being used.

In this chapter, we have presented the overall structure of the DynamiTE framework for concurrent systems, giving the second set of novel contributions (**C2.1** through **C2.4**) in this thesis. This includes the translation from Nomadic Time to a process framework (**C2.1**) with an implementation of the operational semantics (**C2.2**), the plugin framework for introducing side effects (**C2.3**) and the evolver framework for working with execution semantics (**C2.4**). We believe that the framework provides a unique way of developing concurrent systems. It provides features which have already proved advantageous in a theoretical setting, such as the n-ary process synchronisation mechanism described in chapter 3 and mobility. The existence of a formal theory for DynamiTE's behaviour gives many advantages over more ad-hoc approaches, potentially allowing the underlying design to be rigorously examined before being applied to the implementation. The behaviour of the system may be established clearly and unambiguously in the underlying process calculus before implementation even begins, giving a solid grounding on top of which the individual tasks may be developed.

In implementing DynamiTE, we believe we have also made it approachable for existing Java programmers, both for building their own applications and for contributing to DynamiTE itself. The actual implementation of the DynamiTE framework is still in heavy development; the code is available at:

`https://savannah.nongnu.org/projects/dynamite/`

and patch submissions are welcome. At its lowest level, it provides a means of simulating the operations of the Nomadic Time process calculus, allowing them to be more clearly understood. In application, it can provide a useful mechanism for structuring concurrent programs, clearly dividing internal behaviour and interprocess communication. The possibility to add further implementations of the channel and locality factories, via the plugin mechanism, also means that fairly complex concepts can then be leveraged by the programmer in the same simple manner provided by the framework.

In the next chapter, we look at how Nomadic Time, and DynamiTE in turn, can be extended with a type system, providing the final set (**CC3.1** to **CC3.4**) of novel contributions. We begin the chapter with a look at existing work on type systems for process calculi in 7.2, before moving onto the design of our typing rules (**C3.2**) using a group type for processes (**C3.1**) to define further restrictions on migration. The standard type safety proofs of progress and progression are given (**C3.3**), before we again return to DynamiTE in 7.4 to show how it may be extended to support this new type system (**C3.4**).

Chapter 7

Typed Nomadic Time

7.1 Introduction

A type system is a common addition to a process calculus. This is especially true, when the intended use of the calculus is as the basis for a programming language or a distributed system, which is the case here. In this final chapter of original research, we demonstrate how Nomadic Time may be extended with a type system based on the notion of groups (see 7.3). In Typed Nomadic Time, each process is assigned to a group (**C3.1**), which then determines which environs it may *reside* in, *open*, *leave* or *enter*. This can be used to restrict movement based on *which* process is attempting to do so, rather than by enumeration of possible actions, the mechanism employed by our bouncer construct. These restrictions are enforced by the typing rules for Nomadic Time (**C3.2**), for which we prove type safety (**C3.3**).

Section 7.4 looks at how this typed form of Nomadic Time may be used in the DynamiTE framework described in 6. We extend the process framework from 6.3 using a new `TypedProcess` interface, which allows the objects from our translation schema (Table 6.1) to provide typing rules in addition to operational semantics (**C3.4**). We close the chapter in 7.5 by returning to the musical chairs example from 5.6, showing how the type system may be applied to it and the result implemented using DynamiTE.

But before we enter into the technicalities of how this is all implemented, we first present some existing type systems used in other process calculi, including the origins of this group-based system.

7.2 Existing Typed Calculi

Type systems can be used to restrict the calculus in ways that aren't always possible via mere manipulation of the syntax and semantics. Adding a type system can be

as simple as formalising implicit notions, such as the use of $in\ m$ as a capability and not as part of a path (Cardelli, Ghelli & Gordon, 2002) or the fact that the x in $x(y)$ should represent a link and not a mere value (Sangiorgi, 2002). It may also provide more complex intuitions, by distinguishing individual entities, controlling mobility (Cardelli, Ghelli & Gordon, 2002; Levi & Sangiorgi, 2003) or resources (Riely & Hennessy, 1998) or even providing a full subtyping relation (Pierce & Sangiorgi, 1996; Merro & Sassone, 2002). This section considers a few examples of such type systems for both the π calculus (7.2.1) and the ambient calculus (7.2.2).

7.2.1 Type Systems for the π Calculus

Various type systems have been introduced for the π calculus in the literature, ranging from the simple notion of sorts introduced by Milner (Milner, 1999) to those introduced for a specific purpose (Sangiorgi, 2002) and more complex systems involving subtyping (Pierce & Sangiorgi, 1996). Here, sorts are considered followed by a brief look at the distinction between values and links made by Sangiorgi (Sangiorgi, 2002) for the purpose of proving termination.

Sorts

The earliest notion of types was introduced by Milner in (Milner, 1993b; Milner, 1999). The discipline of *sorts* is simply a way of representing 'the length and nature of the vector of names a name may carry in communication' (Milner, 1993b). Formally, a sort is a partial function,

$$ob : \Sigma \rightarrow \Sigma^* \tag{7.1}$$

mapping a name to a vector of names. From this, it is simple to define a sort for all communications in CCS and CaSE as $\{NAME \mapsto ()\}$ (as nothing is passed) and the monadic π calculus as $\{NAME \mapsto (NAME)\}$.

Take the simple example of a buffer,

$$Buf \stackrel{\text{def}}{=} (in, out)(in(x).\overline{out}x.Buf\langle in, out\rangle \tag{7.2}$$

which simply receives a value on in and transmits it on out. x may be assigned the sort $s_1 \mapsto S$, where S is the unknown sort of the buffered value and s_1 is an arbitrary name for the new sort. From this, it follows that both the in and out channels have the sort $s_2 \mapsto (s_1)$, as they both receive or transmit x.

The purpose behind introducing sorts is to make explicit the need to match the number of values being received with the number being sent. Matching the length of these vectors becomes a necessity when dealing with the polyadic π calculus, which

doesn't have the same uniform sort for all channels as is present in CCS, CaSE or the monadic π calculus.

Consider the example from (Milner, 1999) of two processes, P and Q:

$$P \stackrel{\text{def}}{=} x(y).\overline{y}uv.\mathbf{0} \tag{7.3}$$

$$Q \stackrel{\text{def}}{=} \overline{x}y'.y'(w).Q' \tag{7.4}$$

where the parallel composition of these two processes should be disallowed. This is made clear following the first reduction that would result from such a composition:

$$P \mid Q \rightarrow \overline{y'}uv.\mathbf{0} \mid y'(w).Q' \tag{7.5}$$

where Q transmits y' to P. P then tries to use y' to transmit two values, u and v, whereas y' is only used with one, w, in the input of Q. Applying an appropriate sort discipline,

$$\begin{aligned} u &: s_1 \mapsto S \\ v &: s_2 \mapsto T \\ w &: s_3 \mapsto (s_1) \\ y &: s_4 \mapsto (s_1, s_2) \\ y' &: s_5 \mapsto (s_1) \end{aligned} \tag{7.6}$$

allows the typing of x to be prevented by distinguishing between types based on the length of the sort. In P, x must have a sort of length two, while in Q, its sort would only be of length one. This kind of type system formalises an intuition already adopted implicitly (that the length of the input vector should equal that of the output vector), which is a common methodology for type systems.

Typing for Termination

A similar realisation of implicit assumptions is made by Sangiorgi (Sangiorgi, 2002) and is used to prove termination for a subset of possible π calculus processes. The type system is used to explicitly realise the *order* of a name. The types use the simple grammar,

$$T ::= \#T \mid unit \tag{7.7}$$

where *unit* represents a value and a series of # symbols is used to represent the level of indirection which exists between the value and the current name. For example, $\#unit$ is the type of a *first-order link*, representing a name which is used to pass

Table 7.1: Typing Rules from (Sangiorgi, 2002)

$$\text{T-Out}\,\frac{\vdash v : \#T, \vdash w : T, \vdash M}{\vdash \overline{v}w.M}$$

$$\text{T-Inp}\,\frac{\vdash v : \#T, x \in T, \vdash M}{\vdash v(x).M}$$

$$\text{T-Res}\,\frac{x_i \in \#T_i \text{ for some } T_i (1 \leq i \leq n), \vdash M}{\vdash (x_1 \ldots x_n)M}$$

values between processes. A type with more than one # represents a *higher-order link*, which is used to pass links between processes.

This notion is used within the fragment of the type system shown in Table 7.1 to restrict the possible types used in input and output prefixing, and restriction. The rule T-Out ensures that an output prefix, $\overline{v}w.M$, is only typeable if:

- v is at least a first-order link (it has one or more #s)
- w has a type, T
- The continuation, M, is typeable

which prevents v from being a simple value. Similarly, T-In restricts v to being at least a first-order link in $v(x).M$ and T-Res ensures that each restricted name is a link.

These are all ideas that are adopted implicitly in using the π calculus to model systems, but, when not enforced by a type system, these properties can not be included in proofs. The type system in Sangiorgi's paper, although simple, allows a set of processes which are syntactically correct, but logically flawed, to be excluded by only considering processes which are typeable.

7.2.2 Type Systems for the Ambient Calculus

Early work (Cardelli & Gordon, 1999) on providing a type system for the ambient calculus focused on typing the derived communication primitives and specifically the values being exchanged. While interesting, this doesn't really relate to the focus of the calculus, spatial mobility. In (Cardelli, Ghelli & Gordon, 1999; Gordon & Cardelli, 1999), a first attempt is made at providing types for mobility, via mobility and locking annotations. Mobility annotations are used to mark an ambient as

mobile ($\underline{\vee}$) or immobile ($\curvearrowright$), where mobile ambients may be involved in movement operations using the capabilities *in* and *out*. Locking annotations control the use of *open*; locked ambients ($\bullet$) may not be the target of an *open* capability, while unlocked ambients ($\circ$) may.

A more general theory is given in (Cardelli et al., 2002) with the introduction of *groups*. Rather than simply specifying whether or not an ambient can move or be destroyed, the type system is more specific as to which ambients may effect others. To avoid dependent types (Coquand & Huet, 1988), where the types are dependent on the values being typed, an intermediary notion of a group is introduced. This is also advantageous in that it allows a series of ambients to have the same typing, while typing in relation to a single ambient is still possible by having a group with only one member.

For example, given two ambients m and n, the types should express that n can enter m. A dependent formalisation would say that n has the type $CanEnter(m)$, while, using groups, m is given the type G (where G is a group) and n is typed as $CanEnter(G)$. Within the type system itself, ambients are allocated to groups via the use of a group binder, (νG). Just like the ambient binder, (νn), the scope of this may extrude outwards. However, the type system prevents it from ever encapsulating ambients which did not form part of its initial scope (i.e. it only tracks the movements of ambients that are a member of that group). Within (Cardelli et al., 2002), groups are used to assign properties to its members, such as the type of communication possible and the control of crossing or opening ambients.

The types of messages or *exchanges* may specify either no communication (Shh) or a tuple of partners for the communication:

$$S, T ::= Shh \mid W_1 \times \cdots \times W_k \tag{7.8}$$

For example, in the simplest form of the calculus, $Agent[Shh]$ represents a group called $Agent$, the members of which may not exchange values. Nesting is possible, so $Place[Agent[Shh]]$ represents a $Place$ where groups of $Agent$s may stay and continue to be silent.

The full type system, given in (Cardelli et al., 2002), includes these exchange types along with types to control the opening and crossing of ambients. Groups are parametrised over F,

$$F ::=^{\curvearrowright} \mathbf{G},^{\circ} \mathbf{H}, T \tag{7.9}$$

with the final form of ambient type being $G^{\curvearrowright}\mathbf{G}'[F]$. $\mathbf{G}'$ represents the groups that the ambient may cross via objective moves (introduced in the same paper), while $\mathbf{G}$ includes the groups that the ambient may cross via standard subjective movement. Finally, $\mathbf{H}$ distinguishes the groups whose ambients may be *open* e, while T is as defined above.

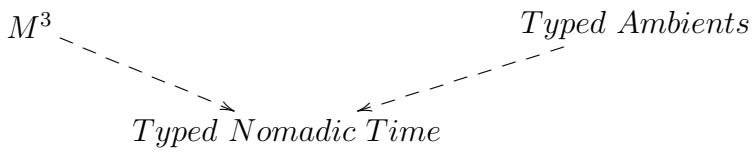


Figure 7.1: Derivation of Typed Nomadic Time

A similar system is adopted by the type system of the M^3 calculus (Coppo, Dezani-Ciancaglini, Giovannetti & Salvo, 2003), but, as this is based on boxed ambients (see 4.3.3), no control of *open* is required. It does introduce a new set of groups, however, to handle the lightweight process mobility presented. In both cases, the type system has a positive effect on the calculus. Not only does it alleviate some of the syntax ambiguity, but it also allows a more fine-grained notion of mobility, where specific ambients can be made immobile or unable to cross a particular ambient.

Type systems were also briefly considered as a way of restricting the behaviour of a process algebraic model. These tend to explicitly reduce the expressivity of the formalism in order to ensure that unwanted constructs can not be created by making them untypeable. This also makes it easier to prove properties of the calculus.

7.3 Mobility Types for Nomadic Time

In this section, we consider the specification of a simple type system for Nomadic Time, which fulfils two main goals:

1. It ensures the sanity of a given syntactic construction, which is implicit in the earlier examples. This is primarily achieved by ensuring that normal process primitives and the primitives used by bouncers remain distinct. For example, $\circledcirc n.\overline{\circledcirc}.\mathbf{0}$ should not be a valid bouncer, especially as $\circledcirc n$ suggests that the bouncer (and its environ) should move inside n.

2. It extends and refines our control over mobility by adding a secondary mechanism orthogonal to the use of bouncers.

Our type system is inspired in part by those given for the ambient calculus (see 7.2.2), specifically the notion of groups presented in (Cardelli et al., 2002) and (Coppo et al., 2003); see Figure 7.1. However, the structure of the groups and the typing rules are novel. Each process is assigned a group type, which determines the

use of the mobility primitives. Each group is a tuple comprising four sets of environ names[1]:

- $\mathscr{R}$ – Environs in which the process may *reside*
- $\mathscr{O}$ – Environs which it may *open*
- $\mathscr{L}$ – Environs which it may *leave*
- $\mathscr{E}$ – Environs which it may *enter*

$\mathscr{L}$ and $\mathscr{E}$ form subsets of $\mathscr{R}$, as clearly, if a process may enter or leave an environ, it must also be able to reside within it. As an example, consider the group $(\{n\}, \emptyset, \emptyset, \{n\})$. Processes of this type may enter and reside in n, but, once there, they may not leave. They also lack the ability to destroy n. We write $g(\mathscr{R})$, $g(\mathscr{O})$, $g(\mathscr{L})$ and $g(\mathscr{E})$ for the components of the group g.

The type system is presented in Tables 7.2, 7.3, 7.4 and 7.5; the general syntax for a type T is given by

$$\begin{aligned} T ::= &\quad G \mid \mathit{Bouncer} \\ G ::= &\quad g \mid G \oplus G \mid G \otimes G \\ &\quad \text{where } g\colon \mathit{Group} \text{ ranges over group types.} \end{aligned}$$

The rule T-Env states that if ξ of type T is a member of Γ, then a typing derivation $\vdash \xi : T$ may be made in the context of Γ. This forms the basis of all later rules. Notice that our system is naturally polymorphic; $\mathbf{0}$, Δ and Δ_σ can have any group type g. In contrast, Ω can only be typed as a *Bouncer* (Table 7.4), and any variable X can take any type prior to being bound, thus distinguishing them from the behaviourally equivalent process, Δ.

The remaining rules in Table 7.2 allow types to be applied in accordance with the various operators present in the calculus. When handling the binary operators, we use the rules T-Sum and T-Par to construct appropriate composite types that maintain the groups used on either side. The rules for timeouts (Table 7.3) follow much the same design as the rules for the summation operator.

In the rules for the bouncers (Table 7.4), *BRec* allows recursive bouncers to be defined, while *BIn*, *BOut* and *BOpen* allow an existing bouncer, B, to be prefixed with one of the three bouncer primitives ($\overline{\textcircled{\scriptsize\triangledown}}$, $\overline{\textcircled{\scriptsize\triangle}}$ and $\overline{\textcircled{\scriptsize\ast}}$). *BSum* simply allows the result of composing two bouncers with the summation operator, $+$ to be typeable as well.

The mobility types (Table 7.5) form the remaining focus of our type system; the type g of an environ $m[E]^B_\sigma$ is that of its encapsulated process E, subject to the constraint that $m \in g(\mathscr{R})$ and B is a *Bouncer*. Consequently, if $m[E]^B_\sigma$ is of type $g : \mathit{Group}$, this implies that $m \in g(\mathscr{R})$. Similar sanity checks are performed in the other rules. For T-EnvIn, we check that the group of n allows it to enter m (and also reside there, given $\mathscr{E} \subseteq \mathscr{R}$ as discussed earlier). The T-EnvOut rule is similar, but

[1]Each group g is defined abstractly to be of kind *Group*.

Table 7.2: Types

$$\text{T-Env}\ \frac{\xi : T \in \Gamma}{\Gamma \vdash \xi : T} \qquad \text{T-Nil}\ \frac{\Gamma \vdash g : \mathit{Group}}{\Gamma \vdash \mathbf{0} : g}$$

$$\text{T-Stop}\ \frac{\Gamma \vdash g : \mathit{Group}}{\Gamma \vdash \Delta : g} \qquad \text{T-Stall}\ \frac{\Gamma \vdash g : \mathit{Group}}{\Gamma \vdash \Delta_\sigma : g}$$

$$\text{T-Var}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash X : t} \qquad \text{T-Act}\ \frac{\Gamma \vdash E : g : \mathit{Group}}{\Gamma \vdash \alpha.E : g}$$

$$\text{T-Rec}\ \frac{\Gamma \vdash E : g : \mathit{Group}}{\Gamma, X : g \vdash \mu X.E : g} \qquad \text{T-Res}\ \frac{\Gamma \vdash E : g : \mathit{Group}}{\Gamma \vdash E \setminus a : g}$$

$$\text{T-Sum}\ \frac{\Gamma \vdash E : g : \mathit{Group}, F : g' : \mathit{Group}}{\Gamma \vdash E + F : g \oplus g'} \qquad \text{T-Par}\ \frac{\Gamma \vdash E : g : \mathit{Group}, F : g' : \mathit{Group}}{\Gamma \vdash E \mid F : g \otimes g'}$$

Table 7.3: Timeout Types

$$\text{T-FTO}\ \frac{\Gamma \vdash E : g : \mathit{Group}, F : g' : \mathit{Group}}{\Gamma \vdash \lfloor E \rfloor \sigma(F) : g \oplus g'}$$

$$\text{T-STO}\ \frac{\Gamma \vdash E : g : \mathit{Group}, F : g' : \mathit{Group}}{\Gamma \vdash \lceil E \rceil \sigma(F) : g \oplus g'}$$

Table 7.4: The Bouncer Type

$$\text{BNil}\ \frac{-}{\Gamma \vdash \Omega : \mathit{Bouncer}} \qquad \text{BRec}\ \frac{\Gamma \vdash B : \mathit{Bouncer}}{\Gamma, X : \mathit{Bouncer} \vdash \mu X.B : \mathit{Bouncer}}$$

$$\text{BIn}\ \frac{\Gamma \vdash B : \mathit{Bouncer}}{\Gamma \vdash \overline{\textcircled{\scriptsize\rm V}}.B : \mathit{Bouncer}} \qquad \text{BOut}\ \frac{\Gamma \vdash B : \mathit{Bouncer}}{\Gamma \vdash \overline{\textcircled{\scriptsize\rm \Lambda}}.B : \mathit{Bouncer}}$$

$$\text{BOpen}\ \frac{\Gamma \vdash B : \mathit{Bouncer}}{\Gamma \vdash \overline{\circledast}.B : \mathit{Bouncer}} \qquad \text{BSum}\ \frac{\Gamma \vdash B, B' : \mathit{Bouncer}}{\Gamma \vdash B + B' : \mathit{Bouncer}}$$

Table 7.5: Mobility Types

$$\text{T-Environ}\quad \frac{\Gamma \vdash E : g : \mathit{Group}, B : \mathit{Bouncer}, m \in g(\mathscr{R})}{\Gamma \vdash m[E]^{B}_{\vec{\sigma}} : g}$$

$$\text{T-EnvIn}\quad \frac{\Gamma \vdash n[E]^{B}_{\vec{\sigma}} : g : \mathit{Group}, m \in g(\mathscr{E})}{\Gamma \vdash n[\circledcirc m.E]^{B}_{\vec{\sigma}} : g}$$

$$\text{T-EnvOut}\quad \frac{\Gamma \vdash k[E]^{B_1}_{\vec{\sigma}} : g : \mathit{Group}, m \in g(\mathscr{L}), n \in g(\mathscr{E})}{\Gamma \vdash n[m[k[\circledcirc m.E]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}]^{B_3}_{\vec{\gamma}} : g}$$

$$\text{T-Open}\quad \frac{\Gamma \vdash E : g : \mathit{Group}, F : h : \mathit{Group}, m \in g(\mathscr{O}), n \in h(\mathscr{E})}{\Gamma \vdash n[\circledast m.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}} : g}$$

$$\text{T-ProcIn}\quad \frac{\Gamma \vdash E \mid F : g \otimes g' : \mathit{Group}, m \in g(\mathscr{E})}{a.E \mid \mathit{on}\ a\ \circledcirc\ m.F : g \otimes g' : \mathit{Group}}$$

$$\text{T-ProcOut}\quad \frac{\Gamma \vdash E \mid F : g \otimes g' : \mathit{Group}, m \in g(\mathscr{L}), n \in g(\mathscr{E})}{\Gamma \vdash n[m[a.E \mid \mathit{on}\ a\ \circledcirc\ m.F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}} : g \otimes g' : \mathit{Group}}$$

we must also check that k can enter n as well as being able to leave m. In T-Open, E is the process that performs the mobility primitive, $\circledast m$ (subject to the constraint $m \in g(\mathscr{O})$). However, the destruction of m also has an effect on its process (F). As a result, F must have an appropriate type, h, such that F can reside in the parent environ, n, after m is removed. The final two rules T-ProcIn and T-ProcOut are the equivalents of $EnvIn$ and $EnvOut$ for processes; the group g concerned is that of E, while the group g' of F can be anything. Note that deciding whether an environ name occurs in a group that is a composite type ($g \oplus g'$ or $g \otimes g'$) requires matching the group to the appropriate term it was connected with prior to composition. For example, if $\circledcirc m.E \mid F$ has the type $g \otimes g'$, then, by Par, $\circledcirc m.E$ has type g and F has type g'. Thus, it is $g(\mathscr{E})$ that must contain m. In general use, the environ name must appear in either of the composed groups for $g \oplus g'$, and in both for $g \otimes g'$, the latter being the Cartesian product of the two sets.

For clarity, we also show how the structural congruence laws of 5.4 interact with the type system in Table 7.6. Notably, the decomposition of $g \oplus g'$ and $g \otimes g'$ depend on the ordering correspondence between the composed processes and the composed types, so the types are also swapped over in $StrSum1$ and $StrPar1$. In $StrIdent$, the type corresponding to the **0** process is also removed. For the other rules, the types are identical on both sides.

Our type system also exhibits the standard properties of *type safety*:

Table 7.6: Structural Congruence Laws with Types

StrSum1	$E + F : g \oplus g'$	$\equiv$	$F + E : g' \oplus g$
StrSum2	$E + (F + G) : g \oplus (g' \oplus g'')$	$\equiv$	$(E + F) + G : (g \oplus g') \oplus g''$
StrPar1	$E \mid F : g \otimes g'$	$\equiv$	$F \mid E : g' \otimes g$
StrPar2	$E \mid (F \mid G) : g \otimes (g' \otimes g'')$	$\equiv$	$(E \mid F) \mid G : (g \otimes g') \otimes g''$
StrIdent	$E \mid \mathbf{0} : g \otimes g'$	$\equiv$	$E : g$
StrResRem	$\mathbf{0} \setminus A : g$	$\equiv$	$\mathbf{0} : g$
StrResRes	$E \setminus A \setminus B : g$	$\equiv$	$E \setminus A \cup B : g$

1. *Progress*: A well-typed term is not stuck. It can either take a step according to the operational semantics or is one of Δ, Ω or X.

2. *Preservation*: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

Proofs of these are given in appendices A and B respectively.

7.4 DynamiTE and the Type System

Extending DynamiTE to support the type system of Typed Nomadic Time (TNT), and type systems in general is relatively simple. We first extend the `Process` interface to create a new interface, `TypedProcess`:

```
public interface TypedProcess
  extends Process
{
  Type getType()
    throws UntypeableProcessException;
}
```

With `TypedProcess`, we extend the contract for implementing a process by one method: `getType`. This either returns the type of the process, or throws an exception if the process is untypeable. The type of a process is represented by a new hierarchy of classes, all of which implement the `Type` marker interface. Thus, in the same way the syntax of a calculus is represented as a set of classes which implement `Process`, so its type system is represented by a set of classes which implement `Type`. The `Calculus` interface (introduced in 6.4.1) becomes:

```
public interface Calculus
```

```
  extends Probeable
{
  public Collection<Class<? extends Process>> getSyntax();
  public Collection<Class<? extends Type>> getTypeSystem();
  public Label getLabel(String label);
}
```

The methods `getSyntax` and `getLabel(String)` were both provided before, and return the syntax and transition labels for the calculus respectively, the latter also acting as a validation mechanism. The new method is `getTypeSystem` which returns the classes that represents valid types; for TNT, `getTypeSystem` represents T (see 7.3) just as `getSyntax` represents the process terms E and F (see 5.32). We implement the method for TNT as follows:

```
  public Collection<Class<? extends Type>> getTypeSystem()
  {
    Set<Class<? extends Type>> types =
      new HashSet<Class<? extends Type>>();
    types.add(ProcessType.class);
    types.add(BouncerType.class);
    return types;
  }
```

The classes have a common suffix of 'Type' to avoid conflicts with the syntax classes. The `BouncerType` class is simple, providing a singleton instance to represent the type. To represent P, there are three further subclasses of `ProcessType`: `Group`, `SumType` and `ProdType`. To handle the constraints imposed by group membership, such as $m \in g(\mathscr{E})$, `ProcessType` requires its subclasses to implement the follow methods:

```
public interface ProcessType
  extends Type
{
  boolean canResideIn(String environ);
  boolean canOpen(String environ);
  boolean canLeave(String environ);
  boolean canEnter(String environ);
}
```

These are implemented in `Group` using sets provided by the user. For `SumType` and `ProdType`, the check is performed on the appropriate constituent type. The implementations of `getType` for each process term follow fairly straightforwardly

from the type system given in 7.2, 7.3, 7.4 and 7.5. For instance, implementing $BNil$ is just a matter of adding the following method to the `Omega` class:

```
public Type getType()
{
  return BouncerType.BOUNCER;
}
```

Those which contain the $g : Group$ prerequisite are more complicated. Just as with names, clocks and environs, the available groups are created by the user and registered with the `Context`. To create the association between a process and a group to begin with, the classes `Nil`, `Delta`, `Stall` and `Var` (those that don't contain an instance of `Process`) gain an additional constructor which records the group of the process. This has the effect of making `Nil` and `Delta` no longer singletons. Instead, the class is implemented as follows:

```
public class Delta
  extends Process
{
  public static final Delta DELTA = new Delta();

  private Delta()
  {
    this(null);
  }

  public Delta(Group g)
  {
    this.g = g;
  }

  public Type getType()
  {
    return g;
  }
}
```

We still allow instances of these classes to be constructed without a group (`getType()` will return `null`) so as to still support Nomadic Time. The other classes likewise implement `getType()` in accordance with the typing rules, so that the group of the overall process is derived from that of its constituents. For example, the implementation for `Sum` looks like this:

```
public Type getType()
{
  return new SumType(left.getType(), right.getType());
}
```

where `left` and `right` are the processes composed by the + operator.

All the implementations of `getType()` we have shown so far always succeed. This is not the case with the mobility typing rules implemented in subclasses of `ModPrefix`. For example, we implement `getType()` in `InEnv` as follows:

```
public Type getType()
{
  ProcessType procType = proc.getType();
  if (procType != null &&
      procType.canEnter(env))
  {
    return procType;
  }
  else
  {
    throw new UntypeableProcessException(this);
  }
}
```

The call to `procType.canEnter(String)` ensures that $m \in g(\mathscr{E})$ holds. The actual code executed by the call depends on the type of `procType`; for a `Group`, the check only searches the single $\mathcal{E}$ set belonging to that group, but for `ProdType`, it must ensure that m is in the $\mathcal{E}$ set of both composed groups.

Finally, the type checks may be performed at runtime by an evolver that supports them. We add an additional interface, `TypedEvolver`, to evolve a `TypedProcess`:

```
public interface TypedEvolver
{
  void evolve(TypedProcess p);
}
```

When the checks are made and how the evolver reacts to them are left up to each implementation.

7.5 Typed Musical Chairs

With our typed variant of Nomadic Time, TNT, we can control the movement of processes in a much more fine grained manner using groups. This allows us to enhance the musical chairs example from 5.6 so that only processes with a group type where $chair \in Chair(\mathscr{R}) \cap Chair(\mathscr{E})$ are allowed to enter the *chair* environ. This is a stronger requirement than that exhibited by the bouncers; the chair bouncer $CB = \mu X.(\overline{\circledcirc}.\overline{\circledcirc}.X + \overline{\circledast}.\Omega)$ dictates that a process must leave the chair before another may enter it, but it does not say anything about *which* processes can occupy the chair in the first place. In the untyped version, it is perfectly possible for a rogue process to enter a chair and not leave. If that process is the only inhabitant (which will be the case, given that the chairs are initially empty), it can prevent a chair from being used properly.

We can see this by considering a simplified version of the full musical chairs system, concentrating solely on the movement of the player into the chair:

$$\begin{aligned} GM4 &\stackrel{\text{def}}{=} \mu X.(\lceil on\ sit \circledcirc chair.X \rceil \sigma(\mathbf{0})) \\ MP &\stackrel{\text{def}}{=} \lceil sit.\mathbf{0} \rceil \sigma(\mathbf{0}) \end{aligned} \tag{7.10}$$

We replace the processes PiC, L and $GM5$ from the definitions in 5.6 with $\mathbf{0}$ so as to keep the example here simple and focus on the use of the type system to restrict movement. If we consider the type rule T-ProcIn, we can see that the final type of $sit.\mathbf{0} \mid on\ sit \circledcirc chair.X$ will be $g \otimes g'$, and thus that $\mathbf{0}$ will be of type g and X of type g'. We also know that $chair \in g(\mathscr{E})$ must hold, so we can derive that the type of $\mathbf{0}$ in $sit.\mathbf{0}$ must be at least $(\{chair\}, \emptyset, \emptyset, \{chair\})$.

The process MP as a whole has a type $g \oplus g'$ (from STO), where g corresponds to $sit.\mathbf{0}$ and thus has to meet the requirement $chair \in g(\mathscr{E})$. The constraint $g \oplus g'(\mathscr{E})$ is satisfied if only one of the two groups meets the constraints; thus having g do so is enough. The $GM4$ process has a type of the same form, $g'' \oplus g'''$, again due to the stable timeout. This is maintained over recursion by T-Rec, so when we compose $GM4$ and MP to form $MP \mid GM4$, we end up with the type $(g \oplus g') \otimes (g'' \oplus g''')$ through the rule T-Par. From this, only $g \oplus g'$ is constrained, as this maps to g in the rule T-ProcIn. As we have already seen, it is sufficient for one of these to meet the requirements for the composite process to do so. Thus, we need only an appropriate type for the $\mathbf{0}$ in $sit.\mathbf{0}$ for the entire process to be typeable.

Why is this useful? As implied above, this makes the requirements to enter the *chair* environ stricter; if we have a process *Rogue* that has the same definition as MP but has the type $(\{chair\}, \emptyset, \emptyset, \emptyset)$, then this process will be blocked from entering the chair in the typed variant of the calculus, but not in the untyped form. Placing this form of MP, which ends in $\mathbf{0}$ rather than becoming PiC, in the original untyped game will prevent the chair it enters from being used subsequently, as the

initial move will use the $\overline{\textcircled{\scriptsize V}}$ offered by the bouncer and there is nothing in the *chair* environ to synchronise with the $\overline{\textcircled{\scriptsize \Lambda}}$ now offered.

Just as we did in 6.6, we can convert this directly from the formal notation into objects in DynamiTE:

```
Name sit = new Name("sit");
Group g = new Group(new String[]{"chair"},null,null,
                    new String[]{"chair"});
Prefix moving = new Prefix(sit, new Nil(g));
Clock sigma = new Clock("\u03C3");
MobPrefix sitAction = new MobPrefix(new ProcIn(sit, "chair"),
                                    new Var("X"));
Process gm4 = new Rec("X", new STo(sitAction, sigma, Nil.NIL));
Process mp = new STo(moving, sigma, Nil.NIL);
Process chairMovement = new Par(mp, gm4);
Process bInOut = new MobPrefix(MobPrim.BOUNCER_IN,
  new MobPrefix(MobPrim.BOUNCER_OUT, new Var("Y")));
Process bOpen = new MobPrefix(MobPrim.BOUNCER_OPEN, Omega.OMEGA);
Env chair = new Env("chair", Nil.NIL,
  new Rec("Y", new Sum(bInOut, bOpen)), null);
Process app = new Par(chair, chairMovement);
```

We create a new instance of `Group` which is allow to reside in and enter an environ named `"chair"`. This group is then assigned to an instance of `Nil` prefixed by the name `sit`. The type is passed through the `Process` instances according to the typing rules and will eventually be checked to ensure that `"chair"` is a member of both the $\mathscr{R}$ and $\mathscr{E}$ sets when the `ProcIn` instance is reached. The group instance above will pass this type check. Were one of the arguments to be replaced with `null`, then it would fail with a `UntypeableProcessException`.

7.6 Conclusion

In this chapter, we presented the typed variant of our calculus, Typed Nomadic Time or TNT, giving the third and final set of novel contributions (**C3.1** through **C3.4**) in this thesis. Following consideration of a number of type systems from the literature, including the use of sorts for typing communication (see 7.2.1) and the use of groups to restrict ambient movement (see 7.2.2), we created our own group type (**C3.1**) and formed a set of typing rules using it (**C3.2**) in 7.3. While the general idea of group-based restriction is taken from existing ambient type systems, we believe our construction here and its associated type system to be novel. As is

standard in the literature, we proved type safety for TNT by showing that typed processes may progress, and that types are preserved correctly in doing so (**C3.3**).

We closed the chapter by showing how DynamiTE could be modified to support TNT in 7.4 (**C3.4**), and put this into practise in 7.5 by adapting the musical chairs example. Just as DynamiTE is not intended to be strongly tied to Nomadic Time as its process calculus, the same is true of the typed variant of DynamiTE and TNT; we expect that other type systems could be implemented in the same manner for other process calculi.

To conclude, we believe that our type system provides a useful optional addition to Nomadic Time. It can be employed, when needed, to provide a finer level of access control than the simple approach of enumerating possible actions used by bouncers. There are possibilities for extending the type system further, which we will discuss in 8.2.3 as part of the next and final chapter on future work.

Chapter 8

Contributions and Future Work

This final chapter summarises our contributions to knowledge from 1.4, before turning to the discussion of possible future work in 8.2.

8.1 Our Contributions

We have presented the following contributions to knowledge which we believe to be novel:

C1. Nomadic Time, an algebraic process calculus with *compositional global synchronisation*, *mobility* and security provision via the notion of '*bouncers*' (see chapter 5). This includes:

- **C1.1** The merging of clock hiding from the CaSE process calculus (Norton et al., 2003) with the notion of distribution, so that the CaSE term E/σ becomes $m[E]^B_{\{\sigma\}}$ in Nomadic Time. In CaSE, clock hiding is used as a form of encapsulation; hiding a clock makes its ticks appear as silent actions to processes outside the scope of the clock hiding operator. We can think of the processes encapsulated by the scope of the clock hiding operator as forming a component, and so it seems natural to name it. The localised form of clock hiding present in Nomadic Time allows this through the use of named *environs* (such as m in the previous example). Note that we use a set of clocks in Nomadic Time, so a CaSE term like $E/\sigma/\rho$ can be simplified to just $m[E]^B_{\{\sigma,\rho\}}$.
- **C1.2** The combining of this localised form of CaSE with structural mobility primitives from the ambient calculus (Cardelli & Gordon, 1998) to give a new mobile calculus with the local and global synchronisation properties of CaSE. This adds a new process form, $\mathcal{M}.\mathcal{E}$, to

the existing syntax of CaSE, where $\mathcal{M}$ is one of $\circledcirc m$, $\circledcirc m$ and $\circledast m$, each in turn exhibiting the behaviour of *in* m, *out* m and *open* m from the ambient calculus. We change the name of the operators in Nomadic Time to make them more distinct from our existing names and co-names, which aren't present in the ambient calculus.

C1.3 The addition of a pair of process mobility primitives which allow direct process movement by synchronising on a name. We add two new alternatives to $\mathcal{M}$, namely *on* $a \circledcirc m$ and *on* $a \circledcirc m$. Just as a process, $a.P$, can synchronise with $\overline{a}.Q$ in CCS or CaSE, it can also synchronise with *on* $a \circledcirc m.Q$ or *on* $a \circledcirc m.Q$ in Nomadic Time. If such a synchronisation happens, not only does $a.P$ becomes P, but it is also moved inside or outside of m. This kind of direct process migration appears in few process calculi, and we believe this particular form, which leverages the existing synchronisation process, to be unique.

C1.4 The introduction of 'bouncers', which add a security mechanism to the calculus. The range of $\mathcal{M}$ is again extended to include $\overline{\circledcirc}$, $\overline{\circledcirc}$ and $\overline{\circledast}$, which correspond to $\overline{in}$ m, $\overline{out}$ m and $\overline{open}$ m respectively from the safe ambient calculus. The difference between their use in the safe ambient calculus and their use in Nomadic Time is that these co-actions must come from a process which is attached to the top-right of an environ such as B in $m[E]^B_{\{\sigma\}}$. This is why the operators themselves do not need to name an environ.

C1.5 The creation of a set of structural congruence laws for Nomadic Time (Table 5.4), allowing process terms to be simplified and the number of semantic rules to be reduced. With these rules in place, we were able to drop the $Sum2$ and $Par2$ rules taken from CaSE's semantics. They are also useful in cases where, for example, a number of processes have evolved to $\mathbf{0}$ e.g. $\mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$ can be rewritten as $\mathbf{0}$ using $StrIdent$.

C1.6 The provision of structured operational semantics for Nomadic Time (Tables 5.2 and 5.3). These extend those from CaSE as demonstrated in Table 5.5, extending the notion of prioritisation and adding new rules to handle the introduction of environs and mobility primitives. The main contributions in the semantics come from formulating appropriate structural operational semantics for the new mobility primitives, especially with regard to the three-way synchronisation present in the process mobility operators, and the generalisation of ideas from CaSE to incorporate mobility. The latter includes the creation of a notion of *high priority transitions*; this includes τ transitions and the transitions which occur when a mobility synchronisation (e.g. $\circledcirc m$ and $\overline{\circledcirc} m$) takes place. So, although a number of the rules are derived

from CaSE, they have had to be extensively re-examined to ensure that they work correctly in the context of the new mobility primitives.

C1.7 The demonstration of the properties of *prioritisation* and *time determinacy* inherent in the new calculus. This follows on from the semantics and explicitly shows how the high priority transitions defined there prevent any clock transitions from taking place.

C2. The realisation of the aforementioned calculus as a *design framework*, DynamiTE, through the implementation of its constructs as programmatic elements in the Java programming language (see chapter 6). This allows the specification of system interactions to be shifted directly from the theoretical domain into an implementation backed by a formal methodology, with the intention of improving industrial adoption of concurrent techniques. This includes:

C2.1 The creation of a translation schema (Table 6.1), mapping process terms in Nomadic Time to Java objects. This was largely a mechanical process of turning the syntax into objects, but there are a few areas that required more finesse. In particular, it was necessary to create $\mathbf{0}$, Ω, Δ and the bouncer primitives as singletons in order to avoid there being multiple distinct instances of them. Having only a single instance of each reduces memory usage and means that two separate references to them can be judged equivalent by comparing just the references and not the objects referred to. The `Action` framework is also notable, particularly as it is the abstract notion of `Tau` which connects the intercommunication mechanism provided by DynamITE to the user's own code.

C2.2 The implementation of the operational semantics as methods in the appropriate Java objects defined in the schema. This was again a largely mechanical process, but did require significant testing to get right. It also had the additional benefit of showing whether the semantics performed as expected in a practical situation.

C2.3 The design and implementation of a plugin framework, allowing the use of different process calculi and different *side effects* as the result of performing a transition. As with any plugin framework, the full merits of this remain to be discovered, as other alternate implementations are developed and used with it. It does provide an important separation between the calculus, the framework and the side-effecting mechanism which will hopefully mean that others can use DynamiTE even if they do not wish to use Nomadic Time. We believe DynamiTE to be novel

in this respect, allowing other calculi to be used and also for existing implementations to be extended; so calculi that extend each other in the theoretical domain can also do so at the implementation level. This is the case with CCS, CaSE and Nomadic Time, which, as noted earlier, share the same implementation of recursion (`Rec`).

C2.4 The design and implementation of the *evolver* framework, allowing the execution semantics to be both clearly denoted and interchangeable. By providing just two implementations in DynamiTE – the `Simulator`, which explores all transitions in a depth-first fashion, and the `RandomExecutor`, which follows one transition at random – it is already possible to evaluate the communication semantics with the former, change a single line of code, and execute the process (including side effects) with the latter.

C3. The optional addition of a *type system* to Nomadic Time in order to allow movement restriction to be based on the group membership of processes (see chapter 7); we refer to this extended version as Typed Nomadic Time (TNT). This includes:

C3.1 The design of a group type which can be applied to a Nomadic Time process to restrict movement. Although the idea of a type system based on groups is derived from existing work on type systems for ambient calculi (see 7.2.2), our particular form of group, using the sets $\mathscr{R}$ (*resides in*), $\mathscr{O}$ (*opens*), $\mathscr{L}$ (*leaves*) and $\mathscr{E}$ (*enters*) is believed to be novel.

C3.2 The provision of typing rules (Tables 7.2, 7.3, 7.4 and 7.5) for the new typed form of Nomadic Time. Again, we believe these to be novel. In a number of cases, the rules are obvious, with the majority of work focusing on handling what happens when two groups converge (T-Sum, T-Par, T-FTO and T-STO) and the actual use of the groups in the mobility rules (T-EnvIn, T-EnvOut, T-Open, T-ProcIn and T-ProcOut). We believe the rules to be useful in providing an additional layer of security, above and beyond the bouncer mechanism, when required.

C3.3 Proofs of type safety for the type system. These were largely mechanical but areas such as handling substitution required significant thought, and they did expose some holes in the type system which led to it being further revised and rectified.

C3.4 The extension of DynamiTE to handle type systems in general, and specifically TNT. This has many similarities with **C2.1** and **C2.2**,

with the processes in the translation schema from **C2.1** now implementing `TypedProcess` and the typing rules being implemented in a similar manner to the way the operational semantics were in **C2.2**. In addition, we had to decide how to best represent the group type, leading to the `ProcessType` interface and its subclasses, and how to signal a failure to type a process in an appropriate way, for which a specific exception type was used.

8.2 Future Work

In this final section, we look at a number of ideas with regard to further developing the process calculus Nomadic Time (8.2.1), the DynamiTE framework (8.2.2) and the type system (8.2.3). We also touch on other possible applications for Nomadic Time and its typed derivative, TNT (8.2.4).

8.2.1 Nomadic Time

Separation of Syntax

At present, the syntax represents bouncers as a form of process for simplicity. This has the advantage of being minimal, but also means that there are number of possible constructs that we would prefer to deem illegal (e.g. the use of bouncers with the timeout operators or Ω being used as a 'normal' process term). In chapter 7, we showed how the type system could be used to make such terms invalid, but in hindsight it would better to have a more verbose syntax which rules these out from the start.

Equivalence

The main element lacking in the current version of Nomadic Time is some notion of equivalence. This is necessary to be able to compare processes, and brings the additional benefit of being able to reduce them to a simpler form. This would also ripple through to DynamiTE in the implementation of an `equals` method for the Nomadic Time implementations of `Process`, which currently only return true where both objects are the same instance.

Any bisimulation theory for NT would be based on the labelled transition system defined by the semantics. In particular, the semantics share a lot in common with those of CaSE, for which a form of bisimulation-based equivalence (*temporal observation congruence*) already exists. Nomadic Time introduces a number of new transitions, including the three *mobility transitions* ($\circledcirc$, $\circledcirc$ and $\circledast$) and the component transitions formed from terms in $\mathcal{M}$ which pair up to generate $\circledcirc$, $\circledcirc$ and $\circledast$.

Thus, any equivalence theory would need to consider issues such as whether $\circledcirc m$ is equivalent to $\circledcirc n$; do we consider only that an 'in' action is being performed or do we also take into account the specific environ involved?

Mobility affects the topology of the process, and thus the difference between two processes may not be fully realised simply by looking at transitions. For example, $m[n[\circledcirc m.\mathbf{0}]^{\Omega}_{\emptyset}]^{\Omega}_{\emptyset}$ and $m[\circledcirc m.\mathbf{0}]^{\Omega}_{\emptyset}$ both produce a $\circledcirc m$ transition along with one for each member of $\mathcal{T}$. Assuming $\mathcal{T}$ is the same for both, the difference in topology is not discernable from merely the set of possible transitions.

Scoping of Environ Names

Back in 5.3, we noted that Nomadic Time does not yet support the scoping of names. In the ambient calculus, $(\nu n)E$ is used to restrict the scope of the ambient name, n, to within E, thus allowing only mobility operations which form part of E to use the name n. Instead, Nomadic Time currently assumes that all environ names are available globally. While in most cases this isn't a problem, as references to environs relate to siblings or parents, it becomes an issue if a process moves location and now refers to an environ it wouldn't have before.

For example, in

$$n[m[\circledast n.P \mid \ \circledcirc k.Q]^{B_1}_{\sigma}]^{B_2}_{\rho} \mid k[R]^{B_3}_{\gamma} \tag{8.1}$$

the environ m may move outside n[1], due to $\circledast n.P$, giving:

$$n[\mathbf{0}]^{B_2}_{\rho} \mid k[R]^{B_3}_{\gamma} \mid m[P \mid \ \circledcirc k.Q]^{B_1}_{\sigma} \tag{8.2}$$

In such a situation, $\circledcirc k.Q$ may now cause m to move inside k, but, prior to the move, it couldn't as k wasn't a sibling of m. It is, of course, possible that such behaviour was intentional and such changes in context are, after all, the point of having migration in the calculus to begin with. However, the current calculus does not give the user the option of preventing such situations from occurring. Adding a scoping operator, in a similar manner to the ambient calculus, would allow the two references to k in the above to remain distinct from one another and prevent the $\circledcirc k$ from operating on the k environ.

Note that the above assumes that the bouncer B_3 will allow m to enter. Similarly, the type of the processes could also be used to control access to k with TNT. Both these existing mechanisms reduce the damage inherent in the use of global names, but they don't remove it altogether.

[1] Assuming appropriate bouncers are available.

Bigraphs

Back in 4.5, we look at Milner's unifying framework of bigraphs (Jensen & Milner, 2004) and saw how these could be used to represent both features of both the π calculus and the ambient calculus. It would be interesting to see if Nomadic Time could also be represented in the same framework. The biggest challenge here is likely to be incorporating the notion of time, and so it may be best to first try and represent a simpler calculus such as TPL.

8.2.2 DynamiTE

As we hinted at in chapter 6, there are still a number of areas we wish to explore in the future. These include:

- The development of further `Evolver` implementations which implement a variety of execution semantics. This gives the possibility of experimenting with additional constraints and levels of priority, which may later be formally adopted in the calculus.
- Expanding the locality framework so that we actually move code and data between processes or even hosts via a network.
- The addition of data to the clock signals, allowing them not only to act as phasing signals but also as a mechanism for broadcast data.
- The development of a parser and/or graphical design tool so that processes can be constructed from algebraic terms or diagrams, rather than Java code.

There are also plenty of possibilities for further work in providing more complex implementations of the plugin services such as interprocess communication via web services and supporting other process calculi. Hopefully, third parties may also wish to get involved in this area.

The Abstract Nomadic Time Machine

In 6.7.3, we looked at the abstract machine PAN which is used to provide an implementation of the safe ambients calculus (Giannini et al., 2006). Our existing translation schema from Nomadic Time to DynamiTE, given in Table 6.1, goes directly from the calculus to a programming language, Java. With an abstract machine in the same vein as PAN, but for Nomadic Time, we could instead express the translation in more general terms. The syntactic constructs of Nomadic Time would be translated into operations using the abstract machine, while DynamiTE would

no longer implement the calculus directly, but instead provide an implementation of the abstract machine.

By providing this additional layer of abstraction, we have an implementation of Nomadic Time that exists at a more formal level. This allows us to prove properties of this implementation, without having to contend with the complicated semantics of a programming language such as Java. It may also be possible to express other calculi using the same abstract machine, thus giving a general framework for implementations of process calculi.

Implement Bigger Applications with DynamiTE

Through the course of this thesis, we've shown how DynamiTE and its associated development process can be used to take an application through the usual stages of requirements analysis (1.3.1), design (5.7) and implementation (6.6). For this, we used the example of a music player with a fairly limited set of requirements. While this is fine as an easily digestable example here, to really prove the utility of DynamiTE, we need to develop some big examples using the framework and prove that it really helps with concurrent applications on a scale where tracking down race conditions and deadlocks starts to become intractable.

8.2.3 The Type System

More Advanced Types

There are a number of ideas that could be further investigated:

- The addition of types to actions and/or clocks. In 8.2.3, we explore one possibility for the former, but there may be more.
- Bouncers only have a single rudimentary type; this could be extended. Alternatively, the syntactic form of bouncers could be removed altogether, with the same functionality instead being provided by the type system.
- It may be worth implementing the types in a theorem prover and deducing results, such as proofs of consistency.

Nomadic Time Extended With Data

At present, the calculus does not explicitly represent the passing of data in any form. When synchronisation occurs on a channel, a, it simply produces a τ transition. Within DynamiTE, the transfer of a single `Object` from sender to recipient takes place. This may be inefficient if many objects need to be sent, and it would be preferable to transfer multiple items in a single synchronisation. It would also be

preferable if the type of the data was represented, so that the type expected by the receiver matches that sent by the sender.

Both of these could be implemented via the type system. The matching of the number of items being sent with those received could also be performed through additions to the syntax and semantics, but also has the downside of introducing a new set of names; we already have distinct sets of names ($\mathcal{N}$), co-names ($\overline{\mathcal{N}}$) and clocks ($\mathcal{T}$) along with the names used by environs.

Such an extension would draw on Milner's work on sorts (see 7.2.1) and allow synchronisation to proceed only when the following holds for $a : \vec{x}$ and $\overline{a} : \vec{y}$:

1. The length n of $\vec{x}$ ($\#x$) must equal the length of $\vec{y}$ ($\#y$).
2. For each element x_i in $\vec{x}$, the corresponding element in $\vec{y}$, y_i, must have the same type t_i.

The types used in $\vec{x}$ and $\vec{y}$ are not members of T (see 7.3), but are user-defined. In DynamiTE, for example, t_1 may be `String`. Thus if a is defined as having the type (t_1, t_2, t_3), it can only synchronise with $\overline{a}$ if it also has the type (t_1, t_2, t_3). Clearly, the second rule could be widened to allow a subtyping relation between the two types, rather than equivalence.

Explicit Type Assignment

At present, there is no explicit assignment of types to processes in the type system. We assume that the user either adds their own axioms to fix the type of a process, or that the type of a process is derived; for example, we can derive a suitable type for $\circledcirc m.\mathbf{0}$ through the knowledge that $m \in g(\mathscr{R})$ must hold. In the DynamiTE implementation of the type system (see 7.4), types are assigned when constructing `Nil`, `Delta`, `Stall` and `Var` instances. This could also be added to the formal version by modifying the syntax and semantics to use $\mathbf{0} : T$, $\Delta : T$ and $\Delta_\sigma : T$.

8.2.4 Other Applications

There may be other domains in which our calculus will find applicability. Pervasive computing may be one such area, as may web services and biological modelling. In addition, its relation to P systems (4.3.5) and the ability to encode it using a bigraph (4.5) would form interesting areas of study.

Bibliography

Aceto, L. & Murphy, D. (1996). Timing and Causality in Process Algebra, *Acta Informatica* **33**(4): 317–350.

Amadio, R. (1997). An Asynchronous Model of Locality, Failure and Process Mobility, *Proc. of the Second International Conference on Coordination Languages and Models (COORDINATION '97)*, number 1282 in *Lecture Notes in Computer Science*, Springer, pp. 374–391.

Amazon (2009). Amazon Web Services. Last checked: 10/08/2009.
URL: *http://aws.amazon.com/*

Andersen, H. R. & Mendler, M. (1994). An Asynchronous Process Algebra with Multiple Clocks, *Proc. of the 5th European Symposium on Programming (ESOP '94)*, number 788 in *Lecture Notes in Computer Science*, Springer, pp. 58–73.

Apple (2009). Grand Central Dispatch: A Better Way to Do Multicore. Last checked: 15/09/2009.
URL: *http://www.apple.com/macosx/technology/*

Armstrong, J., Virding, R., Wikström, C. & Williams, M. (1996). *Concurrent Programming in Erlang*, Prentice Hall.

Beaten, J. C. M. & Middelburg, C. A. (2001). Process Algebra with Timing: Real Time and Discrete Time, *in* J. A. Bergstra, A. Ponse & S. A. Smolka (eds), *Handbook of Process Algebra*, North-Holland, London; Amsterdam, chapter 10, pp. 627–684.

Bergstra, J. A. & Klop, J. W. (1984). Process Algebra for Synchronous Communication, *Information and Control* **60**: 109–137.

Berry, G. & Boudol, G. (1992). The Chemical Abstract Machine, *Theoretical Computer Science* **96**: 217–248.

Boudol, G. (1992). Asynchrony and the π-Calculus (note), *Technical Report RR-1702*, INRIA-Sophia Antipolis.

Boudol, G., Castellani, I., Hennessy, M. & Kiehn, A. (1993). Observing Localities, *Theoretical Computer Science* **114**: 31–61.

Brinkmann, M. & Walfield, N. H. (2007). A Critique of the GNU Hurd Multi-Server Operating System, *ACM SIGOPS Operating Systems Review* .
URL: *http://www.gnu.org/software/hurd/hurd/critique.html*

Bugliesli, M., Castagna, G. & Crafa, S. (2001). Boxed Ambients, *in* N. Kobayashi & B.C.Pierce (eds), *Theoretical Aspects of Computer Software: 4th International Symposium (TACS '01)*, number 2215 in *Lecture Notes in Computer Science*, Springer, pp. 38–63.

Bushnell, T. (1994). Towards a New Strategy of OS Design, *GNU's Bulletin* **1**.
URL: *http://www.gnu.org/software/hurd/hurd-paper.html*

Cardelli, L. (1995). Obliq: A Language with Distributed Scope, *Computing Systems* **8**: 27–59.

Cardelli, L. (2004). Brane Calculi - Interactions of Biological Membranes, *in* V. V. Danos (ed.), *Proc. of the International Conference on Computational Methods in Systems Biology (CMSB 2004)*, number 3082 in *Lecture Notes in Computer Science*, Springer, pp. 257–280.

Cardelli, L., Ghelli, G. & Gordon, A. D. (1999). Mobility Types for Mobile Ambients, *Proc. of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)*, number 1644 in *Lecture Notes in Computer Science*, Springer, pp. 230–239.

Cardelli, L., Ghelli, G. & Gordon, A. D. (2002). Types for the Ambient Calculus, *Information and Computation* **177**: 160–194.

Cardelli, L. & Gordon, A. D. (1998). Mobile Ambients, *Proc. of the 1st Intl. Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)*, number 1378 in *Lecture Notes in Computer Science*, Springer, pp. 140–155.

Cardelli, L. & Gordon, A. D. (1999). Types for Mobile Ambients, *Proc. of the 26th. Annual Symposium on Principles of Programming Languages (POPL '99)*, ACM Press, pp. 79–92.

Carriero, N. & Gelernter, D. (1989). Linda in Context, *Communications of the ACM* **32**(4): 444–458.

Castagna, G., Ghelli, G. & Nardelli, F. Z. (2001). Typing Mobility in the Seal Calculus, *in* K. Larsen & M. Nielsen (eds), *Proc. of the 12th International Conference on Concurrency Theory (CONCUR '01)*, number 2154 in *Lecture Notes in Computer Science*, Springer, pp. 82–101.

Castagna, G. & Nardelli, F. Z. (2002). The Seal Calculus Revisited: Contextual Equivalence and Bisimilarity, *Proc. of the 5th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '02)*, number 2556 in *Lecture Notes in Computer Science*, Springer, pp. 85–96.

Cleaveland, R., Lüttgen, G. & Mendler, M. (1997). An Algebraic Theory of Multiple Clocks, *Proc. of the 8th International Conference on Concurrency Theory (CONCUR '97)*, number 1243 in *Lecture Notes in Computer Science*, Springer, pp. 166–180.

Coppo, M., Dezani-Ciancaglini, M., Giovannetti, E. & Salvo, I. (2003). M^3: Mobility Types for Mobile Processes in Mobile Ambients, *Proc. of Computing: the Australasian Theory Symposium (CATS '03)*, Vol. 78 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 1–34.

Coquand, T. & Huet, G. (1988). The Calculus of Constructions, *Information and Computation* **76**: 95–120.

Dahl, O.-J., Myhrhaug, B. & Nygaard, K. (1968). *SIMULA 67—Common Base Language*, Norwegian Computing Center.

Dijkstra, E. W. (1968). Co-operating Sequential Processes, *Programming Languages* pp. 43–112.

Dijkstra, E. W. (1971). Hierarchical Ordering of Sequential Processes, *Acta Informatica* **1**(2): 115–138.

Fournet, C. & Gonthier, G. (1996). The Reflexive Chemical Abstract Machine and the Join-Calculus, *Proc. of the 23rd. Annual Symposium on Principles of Programming Languages (POPL '96)*, pp. 372–385.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L. & Rémy, D. (1996). A Calculus of Mobile Agents, *Proc. of the 7th International Conference on Concurrency Theory (CONCUR '96)*, number 1119 in *Lecture Notes in Computer Science*, Springer, pp. 406–421.

Giannini, P., Sangiorgi, D. & Valente, A. (2006). Safe Ambients: Abstract Machine and Distributed Implementation, *Science of Computer Programming* **59**(3): 209–249.

Gillespie, D. T. (1977). Exact Stochastic Simulation of Coupled Chemical Reactions, *Journal of Physical Chemistry* **81**(25): 2340–2361.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. (2006). *Java: Concurrency in Practice*, Addison Wesley.

Google (2009). Google Apps. Last checked: 10/08/2009.
URL: *http://www.google.com/apps/*

Gordon, A. D. & Cardelli, L. (1999). Mobility Types for Mobile Ambients, *Technical Report MSR-TR-99-32*, Microsoft Research.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005). *The Java Language Specification, Third Edition*, Addison Wesley.

Hansen, P. B. (1973). Shared Classes, *Operating System Principles* pp. 226–232.

Harris, T., Marlow, S., Jones, S. P. & Herlihy, M. (2005). Composable Memory Transactions, *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP '05)*.

Hennessy, M. & Regan, T. (1995). A Process Algebra for Timed Systems, *Information and Computation* **117**(2): 221–239.

Hewitt, C., Bishop, P. & Steiger, R. (1973). A Universal Modular Actor Formalism for Artificial Intelligence, *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, pp. 235–245.

Hoare, C. A. R. (1974). Monitors, An Operating System Structuring Concept, *Communications of the ACM* **17**: 549–557.

Hoare, C. A. R. (1978). Communicating Sequential Processes, *Communications of the ACM* **21**(8): 666–677.

Honda, K. & Tokoro, M. (1991). An Object Calculus for Asynchronous Communication, *in* M. Tokoro, O. Nierstrasz, P. Wegner & A. Yonezawa (eds), *Proc. of the European Conference on Object-Oriented Programming (ECOOP '91)*, number 512 in *Lecture Notes in Computer Science*, Springer, pp. 133–147.

Hughes, A. (2006). Nomadic Time (extended abstract), *in* R. Schmidt & G. Struth (eds), *Proc. of the PhD Programme at Relational Methods in Computer Science/Applications of Kleene Algebra (RelMiCS/AKA) 2006*, number CS-06-09 in *University of Sheffield Technical Reports*, pp. 60–64.
URL: *http://www.dcs.shef.ac.uk/~andrew/papers/relmics06.pdf*

Hughes, A. (2007). A Framework for Mobile Java Applications, *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ '07)*, pp. 243–248.
URL: *http://www.dcs.shef.ac.uk/~andrew/papers/framework.pdf*

Jensen, O. H. & Milner, R. (2004). Bigraphs and Mobile Processes (revised), *Technical Report UCAM-CL-TR-580*, University of Cambridge, Computer Laboratory.

Jones, S. P., Gordon, A. & Finne, S. (1996). Concurrent Haskell, *Proceedings of the Symposium on Principles of Programming Languages (POPL '96)*.

Last.fm (2009). API – Last.fm. Last checked: 10/08/2009.
URL: *http://www.last.fm/api*

Lea, D. (2004). The java.util.concurrent Synchronizer Framework, *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP '04)*.

Lea, D. (2009). Concurrency JSR-166 Interest Site. Last checked: 11/08/2009.
URL: *http://g.oswego.edu/dl/concurrency-interest/*

Lee, I., Philippou, A. & Sogolsky, O. (2005). A Family of Resource-Bound Real-Time Process Algebras, *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond*, Vol. NS-05-03 of *BRICS Notes*, pp. 151–154.

Lee, J. Y. & Zic, J. (2002). On Modeling Real-time Mobile Processes, *Proc. of the 25th Australasian Conference on Computer Science*, Vol. 17 of *Conferences in Research and Practice in Information Technology Series*, pp. 139–147.

Levi, F. & Sangiorgi, D. (2000). Controlling Interference in Ambients, *Proc. of the 27th. Annual Symposium on Principles of Programming Languages (POPL '00)*, ACM Press, pp. 352–364.

Levi, F. & Sangiorgi, D. (2003). Mobile Safe Ambients, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **25**(1): 1–69.

Lévy, J.-J. (1997). Some Results in the Join-Calculus, *Proc. of the Third International Symposium on the Theoretical Aspects of Computer Software (TACS '97)*, number 1281 in *Lecture Notes in Computer Science*, Springer.

Lindholm, T. & Yellin, F. (1999). *The Java Virtual Machine Specification, Second Edition*, Prentice Hall.

Mazurkiewicz, A. (1977). Concurrent Program Schemes and their Interpretations, *Technical Report AIM PB-78*, Computer Science Department, University of Aarhus.

Merro, M. (2001). *Locality in the π-Calculus and Applications to Object-Oriented Languages*, PhD thesis, Ecoles des Mines de Paris.

Merro, M., Kleist, J. & Nestmann, U. (2002). Mobile Objects as Mobile Processes, *Information and Computation* **177**: 195–241.

Merro, M. & Sassone, V. (2002). Typing and Subtyping Mobility in Boxed Ambients, *in* L. Brim, P. Janar, M. Ketinsky & A. Kuera (eds), *Proc. of the 13th International Conference on Concurrency Theory (CONCUR '02)*, number 2421 in *Lecture Notes in Computer Science*, Springer, pp. 304–320.

Milner, R. (1989a). A Complete Axiomatisation for Observation Congruence of Finite-State Behaviours, *Information and Computation* **81**(2): 227–247.

Milner, R. (1989b). *Communication and Concurrency*, Prentice-Hall, London.

Milner, R. (1992). Functions As Processes, *Mathematical Structures in Computer Science* **2**(2): 119–141.

Milner, R. (1993a). Elements of Interaction: Turing Award Lecture, *Communications of the ACM* **36**(1): 78–89.

Milner, R. (1993b). The Polyadic π-Calculus: a Tutorial, *in* F. L. Bauer, W. Brauer & H. Schwichtenberg (eds), *Proc. of the NATO Advanced Study Institute on Logic and Algebra of Specification*, Springer, pp. 203–246.

Milner, R. (1999). *Communicating and Mobile Systems: The π Calculus*, Cambridge University Press.

Milner, R. (2005). Pervasive Process Calculus, *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond*, Vol. NS-05-03 of *BRICS Notes*, pp. 180–183.

Milner, R., Parrow, J. & Walker, D. (1989). A Calculus of Mobile Processes, Parts I and II, *Technical Report ECS-LFCS-89-86*, University of Edinburgh.

Moller, F. & Tofts, C. (1989). A Temporal Calculus of Communicating Systems, *Technical Report ECS-LFCS-89-104*, University of Edinburgh.

Norton, B. (2005). Behavioural Types for Synchronous Software Composition, *Proc. of the Workshop on Foundations of Interface Technologies (FIT '05)*, Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier.
URL: *http://www.dcs.shef.ac.uk/~barry/RealType/FIT05.pdf*

Norton, B. & Fairtlough, M. (2004). Reactive Types for Dataflow-Oriented Software Architectures, *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA '04)*, Vol. P2172, IEEE Computer Society Press, pp. 211–220.
URL: *http://www.dcs.shef.ac.uk/~barry/RealType/WICSA04.pdf*

Norton, B., Foster, S. & Hughes, A. (2005). A Compositional Operational Semantics for OWL-S, *Proc. of the 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005)*, number 3670 in *Lecture Notes in Computer Science*, Springer, pp. 303–317.
URL: *http://www.dcs.shef.ac.uk/~andrew/papers/WSFM05.pdf*

Norton, B., Lüttgen, G. & Mendler, M. (2003). A Compositional Semantic Theory for Synchronous Component-Based Design, *Proc. of the 14th Intl. Conference on Concurrency Theory (CONCUR '03)*, number 2761 in *Lecture Notes in Computer Science*, Springer, pp. 461–476.
URL: *http://www.dcs.shef.ac.uk/~barry/RealType/CONCUR03.pdf*

NVIDIA (2009). CUDA Zone. Last checked: 09/09/2009.
URL: *http://www.nvidia.com/object/cuda_home.html*

OMG (2009). Document Access Page. Last checked: 27/10/2009.
URL: *http://www.omg.org/technology/documents*

Pérez-Jiménez, M. J. & Romero-Campero, F. J. (2006). P Systems, a New Computational Modelling Tool for Systems Biology, *Transactions on Computational Systems Biology* **VI**: 176–197.

Petri, C. A. (1962). *Kommunikation mit Automaten*, PhD thesis, Institut für Instrumentelle Mathematik.

Pierce, B. C. & Sangiorgi, D. (1996). Typing and Subtyping for Mobile Processes, *Mathematical Structures in Computer Science* **6**(5): 409–454.

Păun, G. (1998). Computing with Membranes, *Technical Report 208*, Institute of Mathematics of the Romanian Academy.

Păun, G. (2002). *Membrane Computing: An Introduction*, Springer-Verlang.

Păun, G., Rosenberg, G. & Salomaa, A. (eds) (2009 (to appear)). *The Handbook of Membrane Computing*, Oxford University Press.

Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. & Jones, M. (1989). Mach: A System Software kernel, *Proceedings of the 34th Computer Society International Conference (COMPCON '89)*.

Regev, A., Panina, E. M., Silverman, W., Cardelli, L. & Shapiro, E. (2004). BioAmbients: An Abstraction for Biological Compartments, *Theoretical Computer Science* **325**(1): 141–167.

Regev, A., Silverman, W. & Shapiro, E. (2001). Representation and Simulation of Biochemical Processes Using the Pi-Calculus Process Algebra, *in* R. B. Altman, A. K. Dunker & T. E. Klein (eds), *Proc. of the Pacific Symposium on Biocomputing*, Vol. 6, pp. 459–470.

Riely, J. & Hennessy, M. (1998). A Typed Language for Distributed Mobile Processes, *Proc. of the 25th. Annual Symposium on Principles of Programming Languages (POPL '98)*, ACM Press, pp. 378–390.

Sangiorgi, D. (1993). *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*, PhD thesis, The University of Edinburgh.

Sangiorgi, D. (1999). The Name Discipline of Uniform Receptiveness, *Theoretical Computer Science* **221**(2): 457–493.

Sangiorgi, D. (2001). Asynchronous Process Calculi: the First- and Higher-Order Paradigms, *Theoretical Computer Science* **253**(2): 311–350.

Sangiorgi, D. (2002). Types, or: Where's the Difference Between CCS and π?, *Proc. of the 13th International Conference on Concurrency Theory (CONCUR '02)*, number 2421 in *Lecture Notes in Computer Science*, Springer, pp. 76–97.

Satoh, I. (1996). *Time and Asynchrony in Distributed Computing*, PhD thesis, Keio University.

Satoh, I. & Tokoro, M. (1993). A Timed Calculus for Distributed Objects with Clocks, *Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, number 707 in *Lecture Notes in Computer Science*, Springer, pp. 326–345.

Stefani, J.-B. (2003). A Calculus of Kells, *Proc. of the 2nd European Association for Theoretical Computer Science International Workshop on Foundations of Global Computing (FGC '03)*, Vol. 85.1 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 40–60.

Sun (2008). Trail: RMI. Last checked: 17/09/2009.
URL: *http://java.sun.com/docs/books/tutorial/rmi/index.html*

Tanenbaum, A. S. & Woodhull, A. S. (2006). *Operating Systems: Design and Implementation (3rd Edition)*, Prentice Hall.

Teller, D., Zimmer, P. & Hirschkoff, D. (2002). Using Ambients to Control Resources, *in* L. Brim, P. Janar, M. Ketinsky & A. Kuera (eds), *Proc. of the 13th International Conference on Concurrency Theory (CONCUR '02)*, number 2421 in *Lecture Notes in Computer Science*, Springer, pp. 288–303.

Thomsen, B. (1989). A Calculus of Higher Order Communicating Systems, *Proc. of the 16th. Annual Symposium on Principles of Programming Languages (POPL '89)*, ACM Press, pp. 143–154.

Turing, A. M. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem, *Proc. of the London Mathematical Society, Series 2*, Vol. 42, pp. 230–265.

Turner, D. N. (1996). *The Polymorphic Pi-calculus: Theory and Implementation*, PhD thesis, The University of Edinburgh.

Various (2009a). API – Facebook Developer Wiki. Last checked: 10/08/2009.
URL: *http://wiki.developers.facebook.com/index.php/API*

Various (2009b). Twitter API Documentation. Last checked: 10/08/2009.
URL: *http://apiwiki.twitter.com/Twitter-API-Documentation*

Vitek, J. & Bryce, C. (2001). The JavaSeal Mobile Agent Kernel, *Autonomous Agents and Multi-Agent Systems* **4**(4): 359–384.

Vitek, J. & Castagna, G. (1999). Seal: A Framework for Secure Mobile Computations, *Proc. of the ICCL '98 Workshop on Internet Programming Languages*, number 1686 in *Lecture Notes in Computer Science*, Springer, pp. 47–77.

Wojciechowski, P. T. (2000). *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*, PhD thesis, The University of Cambridge.

Appendix A

Progress

A well-typed term is not stuck. It can either take a step according to the operational semantics or is one of Δ, Ω or X.

Theorem 1 *If* $\Gamma \vdash P : t : T$ *then either* $P \to P'$, $P = \Delta$, $P = \Omega$ *or* $P = X$.

Proof. By induction on a derivation of $P : t$. At each step, we assume that the desired property holds for all subderivations and proceed by case analysis on the final rule in the derivation. We assume that $\mathcal{T}$ is non-empty; if instead $\mathcal{T} = \emptyset$ then Δ_σ and $\mathbf{0}$ are equivalent to Δ and are possible values for P.

Case T-Nil:
If the last rule in the derivation is T-Nil, then we know from the form of the rule that P must be the process term $\mathbf{0}$. From the *Idle* rule in the semantics, $\mathbf{0}$ has a transition for each clock $\sigma \in \mathcal{T}$ and so is not stuck.

Case T-Stop:
If the last rule in the derivation is T-Stop, then we know from the form of the rule that P must be the term Δ.

Case T-Stall:
If the last rule in the derivation is T-Stall, then we know from the form of the rule that P must be the process term Δ_σ. From the *Stall* rule in the operational semantics, Δ_σ has a transition for each clock ρ where $\rho \neq \sigma$ and so is not stuck.

Case T-Var:
If the last rule in the derivation is T-Var, then we know from the form of the rule that P must be the process term X.

Case T-Act:
If the last rule in the derivation is T-Act, then we know from the form of the rule that P must be the process term $\alpha.E$. The operational semantics provide two rules, *Act* and *Patient*, so that $\alpha.E$ has a transition $\alpha.E \xrightarrow{\alpha} E$ and additional transitions $\alpha.E \xrightarrow{\sigma} \alpha.E$ for each clock $\sigma \in \mathcal{T}$. So the term is not stuck.

Case T-Rec:
If the last rule in the derivation is T-Rec, then we know from the form of the rule that P must be the process term $\mu X.E$. From the rule *Rec* in the operational semantics, $\mu X.E$ has a transition $\xrightarrow{\gamma}$ for each such transition that occurs from E. By the induction hypothesis, we know E is not stuck so $\mu X.E$ is also not stuck as it either has the same transitions, or $E = \Delta$, $E = \Omega$ or $E = X$.

Case T-Res:
If the last rule in the derivation is T-Res, then we know from the form of the rule that P must be the process term $E \setminus a$. From the rule *Res* in the semantics, $E \setminus a$ has a transition $\xrightarrow{\gamma}$ for each such transition from E, minus those where $\gamma = a$. We know by the induction hypothesis that E is not stuck. If E has transitions, then $E \setminus a$ also has transitions as a will never match cases where $\gamma \in \mathcal{T}$ (produced by the base rules *Idle* and *Patient*) or $\gamma \in \mathcal{H}$ (produced by $Par3$, $LHid1$, *InEnv*, *OutEnv*, *Open*, *ProcIn* and *ProcOut*). Otherwise, $E = \Delta$, $E = \Omega$ or $E = X$. In either case, $E \setminus a$ is not stuck.

Case T-Sum:
If the last rule in the derivation is T-Sum, then we know from the form of the rule that P must be the process term $E + F$. By the induction hypothesis, neither E or F are stuck and the rules $Sum1$ and $Sum2$ cause any transitions from the composed processes to also hold for $E + F$. Thus either $E + F$ has transitions, or both E and F are one of Ω, X or Δ. For both, $E + F$ is not stuck.

Case T-Par:
If the last rule in the derivation is T-Par, then we know from the form of the rule that P must be the process term $E \mid F$. The same logic applies as for the summation case using the rules $Par1$ and $Par2$. Using the rules $Par3$, *InEnv*, *OutEnv*, *Open*, *ProcIn* and *ProcOut*, $E \mid F$ can also produce further transitions. So $E \mid F$ is not stuck.

Case T-FTO:
If the last rule in the derivation is T-FTO, then we know from the form of the rule that P must be the process term $\lfloor E \rfloor \sigma(F)$. Again, we know E and F are not stuck from the induction hypothesis. All transitions $\xrightarrow{\gamma}$ produced by E hold for $\lfloor E \rfloor \sigma(F)$ with the exception of the case where $\gamma = \sigma$. In this case, there is a transition $\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F$. If $E = X$ or $E = \Delta$, then the transition $\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F$ is still present, as E has no transitions and thus $E \overset{h}{\nrightarrow}$. Therefore, $\lfloor E \rfloor \sigma(F)$ is not stuck as it always has transitions.

Case T-STO:
If the last rule in the derivation is T-STO, then we know from the form of the rule that P must be the process term $\lceil E \rceil \sigma(F)$. The proof here is the same as for T-FTO, the only difference between the two being that $FTO2$ is split into the two rules $STO2$ and $STO3$, so that the timeout is preserved for $\xrightarrow{\rho}$ where $\rho \neq \sigma$.

Case BNIL:
If the last rule in the derivation is BNil, then we know from the form of the rule that P must be the process term Ω.

Case BREC:
If the last rule in the derivation is BRec, then we know from the form of the rule that P must be the process term $\mu X.B$ and the same proof holds as for T-Rec using the *Rec* rule from the operational semantics.

Case BIN:
If the last rule in the derivation is BIn, then we know from the form of the rule that P must be the process term $\overline{\circledcirc}.B$. The operational semantics provide two rules, $Cap1$ and $Cap2$, so that $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$ and additional transitions $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock $\sigma \in \mathcal{T}$. So the term is not stuck.

Case BOUT:
If the last rule in the derivation is BOut, then we know from the form of the rule that P must be the process term $\overline{\circledcirc}.B$. The same proof applies as for BIn.

Case BOPEN:
If the last rule in the derivation is BOpen, then we know from the form of the rule that P must be the process term $\overline{\circledast}.B$. The same proof applies as for BOpen.

Case BSUM:
If the last rule in the derivation is BSum, then we know from the form of the rule that P must be the process term $B + B'$. The same proof applies as for T-Sum, using the operational semantics rule Sum.

Case T-ENVIRON:
If the last rule in the derivation is Environ, then we know from the form of the rule that P must be the process term $m[E]^B_{\vec{\sigma}}$. The operational semantics provide three rules, $LHd1$, $LHd2$ and $LHd3$, so that $m[E]^B_{\vec{\sigma}}$ has a transition $m[E]^B_{\vec{\sigma}} \xrightarrow{h} E$ for each transition $\xrightarrow{h}$ produced by E and additional transitions $\mathcal{M}.E \xrightarrow{\rho} \mathcal{M}.E$ for each clock $\rho \in \mathcal{T}$ where $\rho \notin \vec{\sigma}$. Any transitions $E \xrightarrow{\sigma} E'$, where $\sigma \in \vec{\sigma}$ convert to transitions of the form $m[E]^B_{\vec{\sigma}} \xrightarrow{\tau} m[E']^B_{\vec{\sigma}}$. Transitions produced by the rules Act and $Cap1$ are not propagated, but the assumed presence of clocks means that at least one transition will always apply to $m[E]^B_{\vec{\sigma}}$, so the term is not stuck.

Case T-ENVIN:
If the last rule in the derivation is T-EnvIn, then we know from the form of the rule that P must be the process term $n[\circledcirc m.E]^B_{\vec{\sigma}}$. As with the cases BIn, BOut and BOpen, transitions arise from the rules $Cap1$ and $Cap2$, so the term is not stuck.

Case T-ENVOUT:
If the last rule in the derivation is T-EnvOut, then we know from the form of the rule that P must be the process term $n[m[k[\circledcirc m.E]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}]^{B_3}_{\vec{\gamma}}$. The proof is the same as for T-EnvIn, BIn, BOut and BOpen.

Case T-OPEN:

If the last rule in the derivation is T-Open, then we know from the form of the rule that P must be the process term $n[\circledast m.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$. The same proof applies as for T-EnvIn and T-EnvOut, with the addition of the transition $P \xrightarrow{\circledast} n[E \mid F]^{B_2}_{\vec{\rho}}$ resulting from the *Open* rule which causes the environ m to be destroyed.

Case T-ProcIn:
If the last rule in the derivation is ProcIn, then we know from the form of the rule that P must be the process term $a.E \mid on\ a \circledcirc m.F$. In a similar manner to T-Open, transitions are produced by the $Cap1$ and $Cap2$ rules in addition to the possibility of process movement arising from the $ProcIn$ rule, $P \xrightarrow{\circledcirc} m[E]^{B}_{\vec{\sigma}} \mid F$ so P is not stuck.

Case T-ProcOut:
If the last rule in the derivation is T-ProcOut, then we know from the form of the rule that P must be the process term $n[m[a.E \mid on\ a \circledcirc m.F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$. As with T-ProcIn, transitions are produced by $Cap1$ and $Cap2$ rules in addition to the possibility of process movement arising from the $ProcOut$ rules, $P \xrightarrow{\circledcirc} n[a.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$, so P is not stuck. □

Appendix B

Preservation

If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

Theorem 2 *If* $\Gamma \vdash P : t : T$ *and* $P \rightarrow P'$*, then there exists* $t' : T$ *such that* $\Gamma \vdash P' : t'$*.*

Proof. By induction on a derivation of $P : t$. At each step, we assume that the desired property holds for all subderivations and proceed by case analysis on the final rule in the derivation.

Case T-Nil:
If the last rule in the derivation is T-Nil, then we know from the form of the rule that P must be the process term $\mathbf{0}$ and t must be the type g where $g : \mathit{Group}$. From the semantics (Idle), $\mathbf{0}$ has a transition $\mathbf{0} \xrightarrow{\sigma} \mathbf{0}$ for each clock σ in $\mathcal{T}$. In each case, $P' = \mathbf{0}$ which, by the T-Nil rule, has type g so our proposition is satisfied.

Case T-Stop:
If the last rule in the derivation is T-Stop, then we know from the form of the rule that P must be the process term Δ and t must be the type g where $g : \mathit{Group}$. From the semantics, there are no transitions for Δ so there is nothing to prove.

Case T-Stall:
If the last rule in the derivation is T-Stall, then we know from the form of the rule that P must be the process term Δ_σ and t must be the type g where $g : \mathit{Group}$. From the semantics (Stall), Δ_σ has a transition $\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma$ for each clock ρ in $\mathcal{T}$ where $\rho \neq \sigma$. In each case, $P' = \Delta_\sigma$ which, by the T-Stall rule, has type g so our proposition is satisfied.

Case T-Var:
If the last rule in the derivation is T-Var, then we know from the form of the rule that P must be the process term X and t must be some type t. From the semantics, there are no transitions for X so there is nothing to prove.

Case T-Act:
If the last rule in the derivation is T-Act, then we know from the form of the rule that P must be the process term $\alpha.E$ and t must be the type g where $g : Group$. From the semantics, there are two subcases:

Subcase Act:
The *Act* rule from the semantics provides the transition $\alpha.E \xrightarrow{\alpha} E$, so P' is E. From the subderivations of the T-Act typing rule, we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $P' : g$.

Subcase Patient:
From the *Patient* rule of the semantics, $\alpha.E$ has a transition $\alpha.E \xrightarrow{\sigma} \alpha.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = \alpha.E$ which, by the T-Act typing rule, has type g so our proposition is satisfied.

Case T-Rec:
If the last rule in the derivation is T-Rec, then we know from the form of the rule that P must be the process term $\mu X.E$ and t must be the type g where $g : Group$. From the *Rec* rule of the semantics, $\mu X.E$ has a transition to $E'\{\mu X.E/X\}$ for any transition γ which can be performed by E. In each case, $P' = E'\{\mu X.E/X\}$ so we need to show that this is well-typed. From the subderivations of the T-Rec typing rule, we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $E' : g : Group$. So, we just need to show that the well-typedness of E' is preserved when the substitution $(\{\mu X.E/X\}$ is performed.

Lemma 1 *If* $\Gamma, x : S \vdash P : t$ *and* $\Gamma \vdash x : S$ *then* $\Gamma \vdash P\{x/X\} : t$

Proof. By induction on a derivation of the statement $\Gamma, x : S \vdash P : t$. For a given derivation, we proceed by cases on the final typing rule. There is only one case where X appears and this is Var, where $P = X$ and $t = g$. There are two possible subcases:

Subcase Match: $x = X$
If X matches the bound variable being substituted, x, then it becomes x, which we know is well-typed from the precondition.

Subcase NoMatch: $x \neq X$
If X does not match the bound variable being substituted, x, then it remains as X, which is typeable using the T-Var rule.

□

Case T-Res:
If the last rule in the derivation is T-Res, then we know from the form of the rule that P must be the process term $E \setminus a$ and t must be the type g where $g : Group$. From the *Res* rule of the semantics, $E \setminus a$ has a transition to $E' \setminus a$ for all transitions (γ) which can be performed by E where $\gamma \neq a$. In each case, $P' = E'$ so we need to show that this is well-typed. From the subderivations of the T-Res typing rule,

we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $E' : g : Group$.

Case T-Sum:

If the last rule in the derivation is T-Sum, then we know from the form of the rule that P must be the process term $E + F$ and t must be the type $g \oplus g'$ where $g, g' : Group$. From the semantics, there are two subcases:

Subcase Sum1:

The $Sum1$ rule from the semantics provides the transition $E + F \xrightarrow{\kappa} E'$, so P' is E'. From the subderivations of the T-Sum typing rule, we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $P' : g$.

Subcase Sum2:

From the $Sum2$ rule of the semantics, $E + F$ has a transition $E + F \xrightarrow{\sigma} E' + F'$ for each clock σ in $\mathcal{T}$. In each case, $P' = E' + F'$. From the subderivations of the T-Sum typing rule, we know that $E : g : Group$ and $F : g : Group$ and by applying the induction hypothesis, we know both E' and F' are well typed. As a result, we can apply T-Sum to give $P' : g \oplus g'$.

Case T-Par:

If the last rule in the derivation is T-Par, then we know from the form of the rule that P must be the process term $E \mid F$ and t must be the type $g \otimes g'$ where $g, g' : Group$. From the semantics, there are eight subcases:

Subcase Par1:

The $Par1$ rule from the semantics provides the transition $E \mid F \xrightarrow{\kappa} E' \mid F$, so P' is $E' \mid F$. From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ so we can apply the induction hypothesis to obtain $E' : g$ and the T-Par typing rule to give $E' \mid F : g \otimes g'$.

Subcase Par2:

From the $Par2$ rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\tau} E' \mid F'$ when E and F synchronise. In this case, $P' = E' + F'$. From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both E' and F' are well typed. As a result, we can apply T-Par to give $P' : g \oplus g'$.

Subcase Par3:

From the $Par3$ rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\sigma} E' \mid F'$ for each clock σ in $\mathcal{T}$, as long as $E \mid F$ does not contain a high priority transition ($\overset{h}{\nrightarrow}$). In each case, $P' = E' + F'$. From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g : Group$ and by applying the induction hypothesis, we know both E' and F' are well typed. As a result, we can apply T-Par to give $P' : g \oplus g'$.

Subcase InEnv:

From the *InEnv* rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\textcircled{v}} E' \mid F'$ where E takes the form $n[G]^{B_2}_{\vec{\sigma}}$ and F is $m[H]^{B_1}_{\vec{\rho}}$. Following the transition, $P' = m[H \mid n[G']^{B_2}_{\vec{\sigma}}]^{B'_1}_{\vec{\rho}}$. If this is well-typed, then, by using T-Environ, we know $H \mid n[G']^{B_2}_{\vec{\sigma}}$ has type $g : Group$, $B'_1 : Bouncer$ and $m \in g(\mathscr{R})$. According to T-Par, $g = g' \otimes g''$, so H has type g' and $locvnG'B_2\vec{sigma}$ has type g''. Again, T-Environ gives $G' : g''' : Group$, $B_2 : Bouncer$ and $n \in g'''(\mathscr{R})$.

From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both G' and B'_1 are well typed. As F is well-typed, then H must be well-typed by the rule T-Environ. Thus P' is well-typed.

Subcase OutEnv:
From the *OutEnv* rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\textcircled{\scriptsize v}} E' \mid F'$ where E takes the form H and F is $n[G]^{B_2}_{\vec{\sigma}}$. Following the transition, $P' = n[G']^{B_2}_{\vec{\sigma}} \mid m[H]^{B'_1}_{\vec{\rho}}$. Using T-Par, we know that $P' : g \otimes g'$, $n[G']^{B_2}_{\vec{\sigma}}$ is of type g and $m[H]^{B'_1}_{\vec{\rho}}$ is of type g'.

From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both G' and B'_1 are well typed. By T-Environ, $n[G']^{B_2}_{\vec{\sigma}}$ is well-typed as G' and B_2 (from the original F) are well typed. Similarly, $m[H]^{B'_1}_{\vec{\rho}}$ is well-typed as H is well-typed (from the original E) and B'_1 is well-typed. Thus P' is well-typed.

Subcase Open:
From the *Open* rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\circledast} E' \mid F'$ where E takes the form G and F is $m[H]^{B_1}_{\vec{\sigma}}$. Following the transition, $P' = n[G' \mid H]^{B_2}_{\vec{\sigma}\cup\vec{\rho}}$. Using T-Par, we know that $P' : g \otimes g'$, G' is of type g and H is of type g'.

From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both G' and B'_1 are well typed. By T-Environ, as $m[H]^{B_1}_{\vec{\sigma}}$ is well-typed, H (from the original F) is also well typed. Thus P' is well-typed.

Subcase ProcIn:
From the *ProcIn* rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\textcircled{v}} E' \mid F'$. Following the transition, $P' = m[E']^{B'_1}_{\vec{\sigma}} \mid F'$. Using T-Par, we know that $P' : g \otimes g'$, $m[E']^{B'_1}_{\vec{\sigma}}$ is of type g and F' is of type g'.

From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both E', F' and B'_1 are well typed. By T-Environ, $m[E']^{B'_1}_{\vec{\sigma}}$ is well-typed as both E' and B'_1 are. Thus P' is well-typed.

Subcase ProcOut:
From the *ProcOut* rule of the semantics, $E \mid F$ has a transition $E \mid F \xrightarrow{\textcircled{v}} E' \mid F'$. Following the transition, $P' = E' \mid m[F']^{B'_1}_{\vec{\sigma}}$. Using T-Par, we know that $P' : g \otimes g'$,

E' is of type g and $m[F']^{B'_1}_{\vec{\sigma}}$ is of type g'.

From the subderivations of the T-Par typing rule, we know that $E : g : Group$ and $F : g' : Group$ and by applying the induction hypothesis, we know both E', F' and B'_1 are well typed. By T-Environ, $m[F']^{B'_1}_{\vec{\sigma}}$ is well-typed as both F' and B'_1 are. Thus P' is well-typed.

Case T-FTO:

If the last rule in the derivation is T-FTO, then we know from the form of the rule that P must be the process term $\lfloor E \rfloor \sigma(F)$ and t must be the type $g \oplus g'$ where $g, g' : Group$. From the semantics, there are two subcases:

Subcase FTO1:

From the $FTO1$ rule of the semantics, $\lfloor E \rfloor \sigma(F)$ has a transition $\lfloor E \rfloor \sigma(F) \xrightarrow{\sigma} F$ for each clock σ in $\mathcal{T}$ as long as $E \overset{h}{\not\rightarrow}$. In each case, $P' = F$. From the subderivations of the T-FTO typing rule, we know that $F : g : Group$ so P' is thus well-typed.

Subcase FTO2:

The $FTO2$ rule from the semantics provides the transition $\lfloor E \rfloor \sigma(F) \xrightarrow{\gamma} E'$, so P' is E' where $\gamma \neq \sigma$. From the subderivations of the T-FTO typing rule, we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $P' : g$.

Case T-STO:

If the last rule in the derivation is T-STO, then we know from the form of the rule that P must be the process term $\lceil E \rceil \sigma(F)$ and t must be the type $g \oplus g'$ where $g, g' : Group$. From the semantics, there are three subcases:

Subcase STO1:

From the $STO1$ rule of the semantics, $\lceil E \rceil \sigma(F)$ has a transition $\lceil E \rceil \sigma(F) \xrightarrow{\sigma} F$ as long as $E \overset{h}{\not\rightarrow}$. So, $P' = F$. From the subderivations of the T-STO typing rule, we know that $F : g : Group$ so P' is thus well-typed.

Subcase STO2:

The $STO2$ rule from the semantics provides the transition $\lceil E \rceil \sigma(F) \xrightarrow{\kappa} E'$, so P' is E'. From the subderivations of the T-STO typing rule, we know that $E : g : Group$ so we can apply the induction hypothesis to obtain $P' : g$.

Subcase STO3:

From the $STO3$ rule of the semantics, $\lceil E \rceil \sigma(F)$ has a transition $\lceil E \rceil \sigma(F) \xrightarrow{\rho} F$ for each clock ρ in $\mathcal{T}$ as long as $E \overset{h}{\not\rightarrow}$ and $\rho \neq \sigma$. In each case, $P' = \lceil E' \rceil \sigma(F)$. From the subderivations of the T-STO typing rule, we know that $E : g : Group$ and $F : g : Group$ so we can apply the induction hypothesis to ensure E' and F' are well typed. By then applying T-STO, we know that P' is well-typed.

Case BNil:

If the last rule in the derivation is BNil, then we know from the form of the rule that P must be the process term Ω and t must be the type $Bouncer$. From the semantics, there are no transitions for Ω so there is nothing to prove.

Case BRec:

If the last rule in the derivation is BRec, then we know from the form of the rule that P must be the process term $\mu X.B$ and t must be the type $Bouncer$. From the Rec rule of the semantics, $\mu X.B$ has a transition to $B'\{\mu X.B/X\}$ for any transition γ which can be performed by B. In each case, $P' = B'\{\mu X.B/X\}$ so we need to show that this is well-typed. From the subderivations of the T-Rec typing rule, we know that $B : Bouncer$ so we can apply the induction hypothesis to obtain $B' : Bouncer$. We know from our previous lemma in the T-Rec case that the well-typedness of B' is preserved when the substitution ($\{\mu X.B/X\}$ is performed, so P' is well-typed.

Case BIn:

If the last rule in the derivation is BIn, then we know from the form of the rule that P must be the process term $\overline{\textcircled{\scriptsize\rm V}}.B$ and t must be the type $Bouncer$. From the semantics, there are two subcases:

Subcase Cap1:

The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is E. From the subderivations of the BIn typing rule, we know that $B : Bouncer$ so $P' : Bouncer$.

Subcase Cap2:

From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = \overline{\textcircled{\scriptsize\rm V}}.E$ which is the same as P so our proposition is satisfied.

Case BOut:

If the last rule in the derivation is BOut, then we know from the form of the rule that P must be the process term $\overline{\textcircled{\scriptsize\rm A}}.B$ and t must be the type $Bouncer$. From the semantics, there are two subcases:

Subcase Cap1:

The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is E. From the subderivations of the BOut typing rule, we know that $B : Bouncer$ so $P' : Bouncer$.

Subcase Cap2:

From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = \overline{\textcircled{\scriptsize\rm A}}.E$ which is the same as P so our proposition is satisfied.

Case BOpen:

If the last rule in the derivation is BOpen, then we know from the form of the rule that P must be the process term $\overline{\circledast}.B$ and t must be the type $Bouncer$. From the semantics, there are two subcases:

Subcase Cap1:

The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is E. From the subderivations of the BOpen typing rule, we know that $B : Bouncer$ so $P' : Bouncer$.

Subcase CAP2:
From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = \overline{\circledast}.E$ which is the same as P so our proposition is satisfied.

Case BSUM:
If the last rule in the derivation is BSum, then we know from the form of the rule that P must be the process term $B + B'$ and t must be the type $Bouncer$. From the semantics, there are two subcases:

Subcase SUM1:
The $Sum1$ rule from the semantics provides the transition $E + F \xrightarrow{\kappa} E'$, so P' is E'. From the subderivations of the BSum typing rule, we know that $B : Bouncer$ so we can apply the induction hypothesis to obtain $P' : Bouncer$.

Subcase SUM2:
From the $Sum2$ rule of the semantics, $E + F$ has a transition $E + F \xrightarrow{\sigma} E' + F'$ for each clock σ in $\mathcal{T}$. In each case, $P' = B'' + B'''$. From the subderivations of the BSum typing rule, we know that $B : Bouncer$ and $B' : Bouncer$ and by applying the induction hypothesis, we know both B'' and B''' are well typed. As a result, we can apply BSum to give $P' : Bouncer$.

Case T-ENVIRON:
If the last rule in the derivation is Environ, then we know from the form of the rule that P must be the process term $m[E]^B_{\vec{\sigma}}$ and t must be the type $g : Group$. From the semantics, there are three subcases:

Subcase LHD1:
From the $LHd1$ rule of the semantics, $m[E]^B_{\vec{\sigma}}$ has a transition $m[E]^B_{\vec{\sigma}} \xrightarrow{\tau} m[E']^B_{\vec{\sigma}}$ as long as $E \xrightarrow{\sigma} E'$ and $\sigma \in \vec{\sigma}$. So, $P' = m[E']^B_{\vec{\sigma}}$. From the subderivations of the T-Environ typing rule, we know that $E : g : Group$ and, by the induction hypothesis, E' is also well-typed. By then applying T-Environ to the well-typed terms E' and B, we can see that P' is well-typed.

Subcase LHD2:
From the $LHd2$ rule of the semantics, $m[E]^B_{\vec{\sigma}}$ has a transition $m[E]^B_{\vec{\sigma}} \xrightarrow{h} m[E']^B_{\vec{\sigma}}$. So, $P' = m[E']^B_{\vec{\sigma}}$. From the subderivations of the T-Environ typing rule, we know that $E : g : Group$ and, by the induction hypothesis, E' is also well-typed. By then applying T-Environ to the well-typed terms E' and B, we can see that P' is well-typed.

Subcase LHD3:
From the $LHd3$ rule of the semantics, $m[E]^B_{\vec{\sigma}}$ has a transition $m[E]^B_{\vec{\sigma}} \xrightarrow{\rho} m[E']^B_{\vec{\sigma}}$ for each clock ρ in $\mathcal{T}$ as long as $\rho \notin \vec{\sigma}$ and $E \xrightarrow{\sigma}\!\!\!\!\!\rightarrow$ where $\sigma \in \vec{\sigma}$. In each case, $P' = m[E']^B_{\vec{\sigma}}$. From the subderivations of the T-Environ typing rule, we know that $E : g : Group$ and, by the induction hypothesis, E' is also well-typed. By then applying T-Environ to the well-typed terms E' and B, we can see that P' is well-typed.

Case T-EnvIn:
If the last rule in the derivation is T-EnvIn, then we know from the form of the rule that P must be the process term $n[\circledcirc m.E]^B_{\vec{\sigma}}$ and t must be the type $g : Group$. From the semantics, there are two subcases:

Subcase Cap1:
The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is $n[E]^B_{\vec{\sigma}}$. From the subderivations of the T-EnvIn typing rule, we know that $E : g : Group$ so by the T-Environ rule P' is well-typed.

Subcase Cap2:
From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = n[\circledcirc m.E]^B_{\vec{\sigma}}$ which is the same as P and thus well-typed.

Case T-EnvOut:
If the last rule in the derivation is T-EnvOut, then we know from the form of the rule that P must be the process term $n[m[k[\circledcirc m.E]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}]^{B_3}_{\vec{\gamma}}$ and t must be the type $g : Group$. From the semantics, there are two subcases:

Subcase Cap1:
The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is $n[m[k[E]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}]^{B_3}_{\vec{\gamma}}$. From the subderivations of the T-EnvOut typing rule, we know that $E : g : Group$ so by multiple application of the T-Environ rule P' is well-typed.

Subcase Cap2:
From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = n[m[k[\circledcirc m.E]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}]^{B_3}_{\vec{\gamma}}$ which is the same as P and thus well-typed.

Case T-Open:
If the last rule in the derivation is T-Open, then we know from the form of the rule that P must be the process term $n[\circledast m.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$ and t must be the type $g : Group$. For the case of F, we use the induction hypothesis to retain well-typedness. For $\circledast m.E$, there are three subcases in the semantics:

Subcase Cap1:
The $Cap1$ rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is $n[E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$. From the subderivations of the T-Open typing rule, we know that $E : g : Group$ so by application of the T-Par and T-Environ rules P' is well-typed.

Subcase Cap2:
From the $Cap2$ rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = n[\circledast m.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$ which is the same as P and thus well-typed.

Subcase Open:
The $Open$ rule from the semantics allows $P \xrightarrow{\circledast} P'$, so $P' = n[E \mid F]^{B_2}_{\vec{\rho}}$. From the subderivations of the T-Open typing rule, we know that $E : g : Group$ and

$F : g : Group$ so by application of the T-Par rule P' is well-typed.

Case T-ProcIn:

If the last rule in the derivation is T-ProcIn, then we know from the form of the rule that P must be the process term $a.E \mid on\ a \circledcirc m.F$ and t must be the type $g \otimes g' : Group$. There are five subcases in the semantics, two of which we have already proved for the case T-Act; for $a.E$, we know that it can progress by the *Act* or *Patient* rule and still be typeable, and in these cases P' is also well-typed via the T-Par rule.

Subcase Cap1:

The *Cap*1 rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is $a.E \mid F$. From the subderivations of the T-ProcIn typing rule, we know that $F : g : Group$ so by application of the T-Par rule P' is well-typed.

Subcase Cap2:

From the *Cap*2 rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = a.E \mid on\ a \circledcirc m.F$ which is the same as P and thus well-typed.

Subcase ProcIn:

The *ProcIn* rule from the semantics allows $P \xrightarrow{\circledcirc} P'$, so $P' = m[E]^{B}_{\vec{\sigma}} \mid F$. From the subderivations of the T-ProcIn typing rule, we know that $E : g : Group$ and $F : g : Group$ so by application of the T-Environ and T-Par rules P' is well-typed.

Case T-ProcOut:

If the last rule in the derivation is T-ProcOut, then we know from the form of the rule that P must be the process term $n[m[a.E \mid on\ a \circledcirc m.F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$ and t must be the type $g \otimes g' : Group$. There are five subcases in the semantics, two of which we have already proved for the case T-Act; for $a.E$, we know that it can progress by the *Act* or *Patient* rule and still be typeable, and in these cases P' is also well-typed via the T-Par and T-Environ rules.

Subcase Cap1:

The *Cap*1 rule from the semantics provides the transition $\mathcal{M}.E \xrightarrow{\mathcal{M}} E$, so P' is $n[m[a.E \mid F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$. From the subderivations of the T-ProcIn typing rule, we know that $F : g : Group$ so by application of the T-Par and T-Environ rules P' is well-typed.

Subcase Cap2:

From the *Cap*2 rule of the semantics, $\mathcal{M}.E$ has a transition $\mathcal{M}.E \xrightarrow{\sigma} \mathcal{M}.E$ for each clock σ in $\mathcal{T}$. In each case, $P' = n[m[a.E \mid on\ a \circledcirc m.F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}} = P$ and is thus well-typed.

Subcase ProcOut:

The *ProcOut* rule from the semantics allows $P \xrightarrow{\circledcirc} P'$, so $P' = n[a.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\rho}}$. From the subderivations of the T-ProcOut typing rule, we know that $E : g : Group$ and $F : g : Group$ so by application of T-Environ and T-Par, P' is well-typed. □