Global Synchronization in a Mobile Context
Transfer Report

Andrew Hughes

October 20, 2006

Contents

1 Introduction **5**

2 Literature Review **7**

2.1 Algebraic Process Calculi . . . 9
2.1.1 CCS . . . 10
2.1.2 The Dining Philosophers in CCS . . . 12
2.1.3 Advantages and Limitations of CCS . . . 14
2.2 Timed Calculi . . . 16
2.2.1 Extending TPL . . . 17
2.2.2 Advantages and Disadvantages of Timed Calculi . . . 18
2.3 Mobility . . . 19
2.3.1 Scope Mobility . . . 19
2.3.2 Distribution and Migration . . . 28
2.4 Typed Calculi . . . 45
2.4.1 Type Systems for the π Calculus . . . 45
2.4.2 Type Systems for the Ambient Calculus . . . 48
2.5 Biological Applications . . . 50
2.6 Conclusion . . . 52

3 Current Work **54**

3.1 The Calculus of Synchronous Encapsulation (CaSE) . . . 54
3.1.1 Timeouts . . . 55
3.1.2 Clock Stopping and Insistency . . . 56
3.1.3 Encapsulation . . . 57
3.2 Localising the Calculus . . . 58
3.3 Adding Mobility . . . 59
3.3.1 Location Mobility . . . 61
3.3.2 Process Mobility . . . 63
3.3.3 Bouncers . . . 64
3.4 The Semantics . . . 66

3.5 A Simple Example 70
3.6 The Type System 74

4 Future Work **79**
4.1 The Type System 79
4.2 Equivalence 80
4.2.1 Structural Congruence 80
4.2.2 Bisimulation 81
4.3 Applications 82
Bibliography 83

List of Figures

2.1 Graph of $a.0 \mid \overline{a}.0$ 11
2.2 Graph of $a.0 + \overline{a}.0$ 12
2.3 The Dining Philosophers in CCS 13
2.4 Spatial diagram of $m[in\ n.out\ n.P] \mid n[]$ 35
2.5 Spatial diagram of $n[m[out\ n.P]]$ 36
2.6 Spatial diagram of $m[P] \mid n[]$ 36
2.7 Example P System 42

3.1 The Musical Chairs Environment 71

List of Tables

2.1 LCCS Dynamic SOS Rules 29
2.2 Typing Rules from [61] 48

3.1 Semantics: Common CaSE Subset 67
3.2 Semantics: Clock Hiding and Mobility 68
3.3 Semantics: Locality Mobility 69
3.4 Semantics: Open 69
3.5 Semantics: Process Mobility 70
3.6 Summary of Processes and Derived Syntax for Musical Chairs 72
3.7 Types: Basics 75
3.8 Types: Operators 76
3.9 Types: Mobility 77

4.1 Types: Linking Processes to Localities 79
4.2 Semantics: Summation 81
4.3 Semantics: Structural Congruence 81

Chapter 1

Introduction

CCS [38] is a process algebra commonly used for modelling synchronous communication between two processes, where one sends a signal and the other receives it at the same time (a concept referred to as *local synchronisation*). However, it cannot directly represent systems involving synchronisation of a sender with an arbitrary number of recipient processes (known as *global synchronisation*) in a *compositional* manner. Crucially, the semantics of a broadcast agent cannot suitably be represented using CCS. If the agent is defined as transmitting a signal to each of the recipients sequentially, through multiple local synchronisations, then its semantics will become non-compositional, because such behaviour depends upon the number of recipients. Each time a new recipient is introduced, or one of the existing ones is removed, the semantics will have to be changed.

A solution to this deficiency lies in determining when all possible synchronisations have taken place. With this facility available, the broadcast agent can recurse, transmitting signals, until this condition holds. The family of abstract timed process calculi (including TPL[24] and CaSE[50]) allow this by extending CCS with *abstract clocks*. These don't represent real time, with units such as minutes and seconds, but are instead used to form synchronous cycles of internal actions followed by clock ticks. A concept known as *maximal progress* enforces the precedence of internal actions over clock ticks, allowing the possible synchronisations to be monitored. When a synchronisation takes place, it appears to the system as an internal action. Thus, with maximal progress, synchronisations prevent the clock from ticking, and as a result, the occurrence of a clock tick also indicates that there are no possible synchronisations.

However, the timed calculi mentioned above lack any notion of spatial distribution or mobility. Thus, while they can adequately represent large static systems, involving both local and global synchronisation, they fail to model

more mobile systems, where the location of a process can change during execution. In contrast, the ambient calculus [13] includes both distribution (via structures known as *ambients*) and mobility (by allowing these structures to be moved, along with their constituent processes, during execution). But, it suffers from similar deficiencies to CCS when modelling global synchronisation.

This report presents the calculus of *Typed Nomadic Time* (TNT) [28], which combines the abstract timed calculus, CaSE, with notions of distribution and mobility from the ambient calculus and its variants ([33, 65]). This allows the creation of a compositional semantics for mobile component-based systems, which utilise the notion of communication between arbitrary numbers of processes within a mobile framework. To extend the example of a broadcast agent given above, this extension allow broadcasts to be localised to a particular group of processes, which can change during execution. Current work on TNT is discussed in chapter 3, while chapter 2 contains a review of the existing literature in this area. Finally, chapter 4 discusses the future development of the calculus, including possible case studies.

Chapter 2

Literature Review

Concurrency is an inherent part of the real world. Multiple events take place simultaneously, and each of these events can interact and affect others. Early computational models, however, take a simpler idealised view, where events occur sequentially and in isolation. Universal Turing machines [67] have proven to be computationally complete; they are capable of simulating all recursive functions. However, they do not directly model concurrent execution.

So, if these models can have this level of computational power without attempting to represent this particular aspect, 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.

To see the effect of concurrency on computation, consider a simple prototypical example, as demonstrated by Milner [41]. Observe the following programs,

$$\texttt{x}{=}\ 2; \tag{P1}$$

$$\begin{aligned}&\texttt{x}{=}\ \texttt{1};\\&\texttt{x}{=}\ \texttt{x} + \texttt{1};\end{aligned} \tag{P2}$$

where we assume that each line is an atomic action.

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 in the gap[1] between the assignment of the value 1 and the later summation which makes x 2.

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

$$\texttt{x} = 3; \tag{P3}$$

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). This is known as a *race condition*, as the final value of x depends on the timing of the various modifications of its value by the two programs. 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 interact. If each program exists in its own isolated environment, 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 a subset where this additional restriction of isolation applies.

Dijkstra's classic 'Dining Philosophers' problem [19] 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 a limited resource (the forks).

In cases where things go awry, *deadlock* or *starvation* may result. For example, if 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 on 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[2]. Alternatively, *starvation* may result if

[1] Assuming x is accessible by more than one process.

[2] The solution to breaking this deadlock is to break the symmetry; if the fifth philosopher

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.

As can be seen from these examples, concurrency raises issues outside the reach of traditional sequential models of computation. Thus, just as there is a requirement for models of sequential computation, models that can represent these phenomena are also necessary. This is even more relevant today, as hardware advances make more machines capable of true concurrency (via dual-core processors and beyond) and distributed computing paradigms, such as services, become more prevalent. To adequately work with these systems, appropriate formal models are needed to represent them and highlight their flaws. Many such models have been developed, and we will now consider a subset of these.

2.1 Algebraic Process Calculi

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 [35] or the graphical style associated with Petri nets [53] and Hewitt's Actor model [25]. Interaction between processes is via message-passing, rather than via shared memory[3] or a tuple space [15].

The foundational calculi in this field are Hoare's CSP [26], Milner's CCS [38] and Bergstra and Klop's ACP [5], 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, and later given a theoretical basis, following Milner's work on CCS. Both calculi have influenced each other, while starting out from different perspectives (Milner's being more of a theoretical one). ACP shares many of the ideas of CCS, and can be regarded as an 'alternative formulation' [5], using a similar set of operators to achieve a different goal.

Here, the focus is on CCS, as it forms the basis for most of the other calculi considered, including the π calculus [45] and CaSE [50]. Of the three, CCS has the most minimal 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, this is advantageous, as

---

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.

[3]Although 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 in shared memory.

it makes reasoning over the calculus a simpler process, and, as will be seen, adding further syntax to represent more features is a relatively simple process.

2.1.1 CCS

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

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

where α, a and f are explained below.

Communication between processes is via the sending and receiving of signals. The internal behaviour of the processes is abstracted, represented simply by the silent action τ. The full set of actions, $Act = \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, is used to describe the behaviour of the concurrent system, where $\mathcal{N}$ is an infinite set of names, and $\overline{\mathcal{N}}$ is the corresponding set of co-names, $\{\overline{a} | a \in \mathcal{N}\}$. These names are usually used to represent *channels*, which the processes use to communicate. Thus, $a.E$, 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 single process is thus defined as a sequence of inputs, outputs and silent actions. This can be seen in the above grammar, where 0 represents the empty process, which exhibits no behaviour, and $\alpha.E$ is the action prefix used for the limited sequential composition of actions, where $\alpha \in Act$.

For communication to actually take place, two processes must *synchronize*; they must emit corresponding actions on the same channel at the same time. For this to occur, the two processes must be running in parallel. Parallel composition in CCS is represented by the $|$ operator. When two processes are composed in this way, they may both perform their corresponding input and output actions simultaneously, resulting in a τ action being emitted.

For instance, if E is considered to be $a.0$ and F to be $\overline{a}.0$, 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. $E \mid F \xrightarrow{a} 0|F$
2. $E \mid F \xrightarrow{\overline{a}} E|0$
3. $E \mid F \xrightarrow{\tau} 0|0$

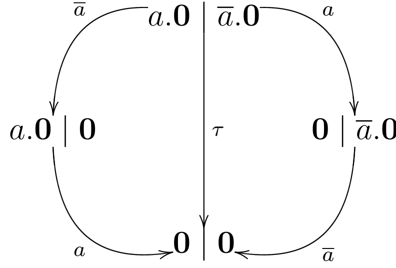

Figure 2.1: Graph of $a.0 \mid \overline{a}.0$

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 input or output can be paired with any corresponding action which is within the scope of the channel. To force the input of E to be paired with the output of F, the scope of a must be restricted so as to only include the two processes, E and F. This is handled by another operator in the core syntax, $\backslash$. The right operand of this is the name of a channel whose scope is restricted to that of the left operand. In this case, $(E|F)\backslash a$ appropriately limits the possible derivations to just $\stackrel{\tau}{\rightarrow}$.

The remaining binary CCS operator is $+$, which provides non-deterministic choice between two processes. Once a derivation is made from one process, the option of performing the actions of the other is lost. This contrasts with the parallel composition operator, where the other process remains running in parallel. Choice thus effectively corresponds to the familiar idea of branching found in sequential models. Using the same two exemplar processes again, $E + F$ may derive as follows:

1. $E \ + \ F \stackrel{a}{\rightarrow} 0$

2. $E \ + \ F \stackrel{\overline{a}}{\rightarrow} 0$

Again, this is illustrated in Fig. 2.2. There are clearly similarities between the two sets of possible derivations, but note that, with choice, there is no possibility of synchronisation.

The remaining operators in CCS handle recursion and relabelling. $\mu X.E$ binds X with the value of E, so that later occurrences of X are replaced with E. The function, f, in $E[f]$ has the type $Act \rightarrow Act$ and converts actions, while preserving τ and complementation.

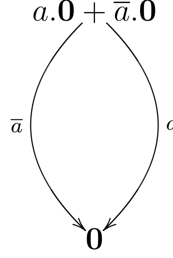

Figure 2.2: Graph of $a.0 + \overline{a}.0$

2.1.2 The Dining Philosophers in CCS

To fully appreciate CCS, it is necessary to see how it may be used to model an example scenario. Consider the dining philosopher's example illustrated above. Modelling this in CCS involves first ascertaining what processes form the basis of this 'system'. Clearly, each philosopher plays a part, so they should be represented by a process. Returning to the original definition of the problem, each philosopher may choose to eat or think. In CCS, this can be represented as:

$$Philosopher = EatingPhilosopher \ + \ ThinkingPhilosopher \tag{2.2}$$

where the philosopher is recursively defined as making the choice between becoming an $EatingPhilosopher$ process or a $ThinkingPhilosopher$ process. Defining the latter is simple; thinking is simply some internal process of the philosopher:

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

The focus of the model is on the eating process, which requires access to the system's shared resource: the forks. Modelling this necessitates defining a protocol whereby the philosopher may interact with the resource in order to obtain access to it. From this, it follows that the forks must also be represented as processes:

$$Fork = \mu X.takeFork.putDownFork.X \tag{2.4}$$

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

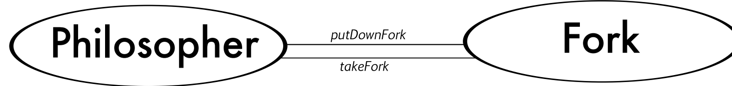

Figure 2.3: The Dining Philosophers in CCS

$$Fork' = putDownFork.X \tag{2.5}$$

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 $putDownFork$, which causes X to be expanded and the fork to wait for input on $takeFork$ again.

This interaction is further clarified by defining the final process, the $EatingPhilosopher$:

$$EatingPhilosopher = \overline{takeFork}.\overline{takeFork}.\tau.\overline{putDownFork}.\overline{putDownFork}.Philosopher \tag{2.6}$$

which needs to synchronize with two available $Fork$ processes to be able to eat (represented by τ) and then release the forks. The system as a whole is modelled by running a number of philosophers and forks in parallel (i.e. multiple copies of Fig. 2.3), 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 execution of the model.

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 $takeFork$ and $putDownFork$ 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 $takeFork_i$ (to take the left fork) and then $takeFork_{i-1}$ (with the exception that when $i - 1 = 0$, we use n) [4].

This model restricts which fork is taken by which philosopher (limiting the possible actions, and thus removing some non-determinism), but is still prone

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

to the effects 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, only one philosopher may end up eating). These may be regarded as implementational aspects of the model; all these phenomena could be represented, but a choice between these is not made at this level of abstraction.

2.1.3 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 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. However, the calculus is not Turing-complete[5]; there are limitations as to what may be expressed. As discussed earlier, Turing completeness does not necessarily guarantee the suitability of a model for a particular task. Likewise, the lack of such completeness doesn't imply that the model is unsuitable. As shown above, an appropriate model of the Dining Philosophers problem may be defined, without Turing completeness. The lack of this in CCS is not necessarily a problem. It may even be an advantage in some cases, where this lack of expressivity simplifies the formal reasoning over the model.

One fairly obvious limitation, and one that is relevant when discussing Turing completeness, 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 section 2.3.

CCS models are also relatively static; while processes may evolve (e.g. $a.P$ may become P) and the number of processes in the system may change (e.g. a process may branch using parallel composition), the communication

[5] A finite axiomatisation can be defined, if the simultaneous presence of parallel composition and recursion is avoided [39].

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 2.3.1.

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 multiple processes.

Consider the idea of 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 possible ways of defining semantics for the agent, but not one that provides a suitably compositional solution. Perhaps the most obvious of these is simply to extend the familiar synchronisation of two processes. An input and output pair can synchronize, 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

$$\mathbf{\bar{a}_1.\bar{a}_2.0} \mid a_1.P \mid a_2.Q \tag{2.7}$$

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?

$$\mathbf{\bar{a}_1.\bar{a}_2.\bar{a}_3.0} \mid a_1.P \mid a_2.Q \mid a_3.R \tag{2.8}$$

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 synchronize with it. Thus, the example for three inputs above becomes

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

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[6]. 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, disparate from the processes involved in the communication, which can be used to co-ordinate the synchronisation. In the next section, a branch of process calculi is considered which provides just such a facility.

2.2 Timed Calculi

Initially, the use of the word 'timed', within the context of the calculi considered here, is a bit of a misnomer. 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 [1, 4, 30, 31, 46, 62, 63], attempt to model this. Instead, this section focuses on a series of discrete timed calculi which focus on abstract time and the use of *clocks* for the primary purpose of global synchronisation (as described above).

Hennessy's Temporal Process Language (TPL) [24] extends the CCS language discussed above 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

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

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

$$\lfloor E \rfloor \sigma(F) \tag{2.10}$$

where E and F are processes and σ is the clock. In short, F acts if E *times out* on the clock, σ. This is similar to non-deterministic choice, in that only one of the two processes will ever act and the behaviour of the other is lost. 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).

With these additions, the problem of defining a suitable compositional broadcast agent, as mentioned above, 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

$$\mu X.\lfloor \bar{o}.X \rfloor \sigma(0) \mid o.P \mid o.Q \mid o.R \tag{2.11}$$

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 synchronizing 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.

2.2.1 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 synchronize. 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) [16] extends TPL with the idea of multiple clocks, drawn from PMC[7][3]. 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 pre-defined scope for each clock. A more elegant solution is provided in the Calculus for Synchrony and Encapsulation (CaSE) [50], 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' [50] of clocks at the level above. CaSE is further discussed in 3.1, where it forms the basis for the calculus of *Typed Nomadic Time* (TNT).

2.2.2 Advantages and Disadvantages of Timed Calculi

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 2.1.3, to be solved, but these timed calculi still retain the other problems with CCS we mentioned there. Neither TPL, PMC, CSA nor CaSE explicitly include data within the model. This is not necessarily a disadvantage; it is possible to model data implicitly, via the use of silent actions, and including data in the model complicates formal reasoning and equivalence theories.

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

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

static dataflow-oriented systems such as those in [48] and [49].

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

2.3 Mobility

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 [41], and is sufficiently expressive as to allow it to encode Church's λ calculus [40].

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 [13], which groups composed processes inside *ambients*. These ambients can be moved up and down a nested hierarchy of such objects, or 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.

2.3.1 Scope Mobility

The π Calculus

The π calculus follows on from Milner's earlier work on CCS discussed in 2.1.1. Essentially, it is a value-passing form of CCS with a generalisation from values and channels to simple *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 [40]

$$E, F \ ::= \ 0 \mid \overline{x}y.E \mid x(y).E \mid (a)E \mid (E \mid F) \mid !E \qquad (2.12)$$

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 [45], but is removed from the version given in [40] 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*[8] like restriction. x and y are both names, where 'x [is] the *subject* and y the *object*' [40]. 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,

$$x(y).\overline{y}x.0 \qquad (2.13)$$

becomes perfectly valid.

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:

$$(a)(a(x).\overline{x}a.0 \mid \overline{a}y.0) \mid y(z).P \qquad (2.14)$$

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 synchronize, the system evolves to:

[8]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.

$$(a)(\overline{y}a.0 \mid 0) \mid y(z).P \tag{2.15}$$

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*:

$$(a)(0 \mid 0 \mid P\{a/z\}) \tag{2.16}$$

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 [40]:

$$(x)(P \mid Q) \equiv P \mid (x)Q \text{ if x not free in P} \tag{2.17}$$

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

$$0 \mid 0 \mid (a)(P\{a/z\}) \tag{2.18}$$

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.

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 [40], which also defines a reduction-based semantics. The earlier tutorial papers [45] instead use recursion and a structured operational semantics, based on a labelled transition system.

The polyadic π calculus [42] 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[9]. Having this as a core part of the syntax provides advantages in representing abstractions and giving a natural sort discipline[10]. 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 [42]:

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

[10] Sorts are a way of applying typing to the π calculus, which will be covered further in section 2.4 on typed calculi.

$$x(yz) \mid \overline{x}y_1z_1 \mid \overline{x}y_2z_2 \tag{2.19}$$

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

$$[\![x(yz)]\!] \stackrel{\text{def}}{=} x(y).x(z) \tag{2.20}$$

$$[\![\overline{x}yz]\!] \stackrel{\text{def}}{=} \overline{x}y.\overline{x}z \tag{2.21}$$

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 synchronizes 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:

$$[\![x(yz)]\!] \stackrel{\text{def}}{=} x(w).w(y).w(z) \tag{2.22}$$

$$[\![\overline{x}yz]\!] \stackrel{\text{def}}{=} (w)(\overline{x}w.\overline{w}y.\overline{w}z) \tag{2.23}$$

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.

The asynchronous π calculus [7, 27, 60] 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 [9]:

$$[\![\bar{c}x.P]\!] \stackrel{\text{def}}{=} (r)(\bar{c}xr \mid r.P) \tag{2.24}$$

$$[\![cy.P]\!] \stackrel{\text{def}}{=} c(yr).(\bar{r} \mid P) \tag{2.25}$$

where r is not free in P. 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[11] before continuing with P. 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* [36], 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* [59], 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) [66], 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 [58]. It defines an extension to the π

[11] r is a syntactic abbreviation for $r()$ i.e. the input is an empty tuple.

calculus, HOπ, which not only allows the transmission of names (first-order) and processes (second-order), but also parameterised processes of arbitrarily high order (ω-order). This is best illustrated by some examples, drawn from [58]. 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,

$$\overline{x}P.Q \mid x(X).X \tag{2.26}$$

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:

$$Q \mid P \tag{2.27}$$

where the process P having being substituted for X.

A more complex example is given by considering Milner's encoding of the natural numbers [42]. 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:

$$[\![n]\!] \stackrel{\text{def}}{=} (y,z)\overline{y}^n.\overline{z} \tag{2.28}$$

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 parameterised 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:

$$Plus \stackrel{\text{def}}{=} (X,Y)(y,z)((x)(X\langle y,x\rangle \mid x.Y\langle y,z\rangle)) \tag{2.29}$$

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:

$$X(y,z) \stackrel{\text{def}}{=} \overline{y}.\overline{y}.\overline{z} \tag{2.30}$$

$$Y(y,z) \stackrel{\text{def}}{=} \overline{y}.\overline{y}.\overline{y}.\overline{z} \tag{2.31}$$

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:

$$\xrightarrow{y}\xrightarrow{y}\xrightarrow{\tau}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{z} \tag{2.32}$$

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

$$\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{y}\xrightarrow{z} \tag{2.33}$$

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[12]:

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

For a weak bisimulation, τ transitions are effectively ignored. A series of such transitions, $\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\ldots$ is abbreviated to $\xRightarrow{\tau}$ and $\xRightarrow{\tau}\xrightarrow{a}\xRightarrow{\tau}$ is deemed equivalent to $\xrightarrow{a}$. 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 [58] 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 [20] takes the asynchronous π calculus as its basis, and focuses on providing a formalism better suited as the basis for a distributed implementation.

[12]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.

Take the following example of a π calculus process given in [34]:

$$x(y).P \mid x(z).Q \mid \overline{x}a \tag{2.34}$$

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*[13] 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. 2.12 becomes:

$$P, Q \ ::= \ 0 \mid \texttt{def}\ D\ \texttt{in}\ P \mid (P \mid Q) \mid x\langle\tilde{v}\rangle \tag{2.35}$$

$$D, E \ ::= \ J \triangleright P \mid D \wedge E \mid \mathbf{T} \tag{2.36}$$

$$J, J' \ ::= \ x\langle\tilde{v}\rangle \mid (J \mid J') \tag{2.37}$$

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:

$$\texttt{def}\ (x\langle y\rangle \triangleright P) \wedge (x\langle z\rangle \triangleright Q)\ \texttt{in}\ x\langle a\rangle \tag{2.38}$$

which has essentially the same behaviour as the π calculus example presented earlier. $x\langle y\rangle \triangleright P$ 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:

$$(x)(!(x(y).P \mid x(z).Q) \mid \overline{x}a) \tag{2.39}$$

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 *localized* to a particular set of emitters and receptors.

[13]The join calculus uses an analogy with chemistry to describe its behaviour, based on the *CHemical Abstract Machine* (CHAM) [6].

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.[14]

Advantages and Disadvantages 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 [61], 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 [20, 64, 69, 2]; 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. Specifically, to prove termination for the π calculus, it is necessary to employ the asynchronous variant with uniform receptiveness and the input-guarded replication operator.

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 2.1.3 for CCS, and solved in 2.2 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 2.3.2. 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.

[14] Such changes have also been made using the restrictions imposed by an appropriate type system [59].

2.3.2 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 [69], administrative or security domains [13, 68] and biological cells [10], 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:

$$Spec \stackrel{\text{def}}{=} in.\tau.\overline{out}.Spec \tag{2.40}$$

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:

$$Receiver \stackrel{\text{def}}{=} in.\overline{a}.Receiver \tag{2.41}$$

$$Sender \stackrel{\text{def}}{=} a.\tau.\overline{out}.Sender \tag{2.42}$$

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

$$(Receiver \mid Sender) \setminus a \tag{2.43}$$

with the scope of a restricted, they are *weakly bisimilar* (see 2.3.1) to one another. The specification performs the following derivations:

$$\xrightarrow{in}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{2.44}$$

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

$$\xrightarrow{in}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{2.45}$$

Table 2.1: LCCS Dynamic SOS Rules

$$\textsf{Act1}\ \frac{-}{a.E \overset{a}{\underset{l}{\rightarrow}} l :: E}\ for\ any\ l \in Loc \quad \textsf{Act2}\ \frac{E \overset{a}{\underset{u}{\rightarrow}} E'}{l :: E \overset{a}{\underset{lu}{\rightarrow}} l :: E'} \quad \textsf{Act3}\ \frac{-}{\tau.E \overset{\tau}{\rightarrow} E}$$

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* [8] 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. A variant of CCS, LCCS, 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 2.1, 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 3, 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 2.1.1, 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 practice is best shown by reconsidering the earlier CCS example. Recall the specification defined in 2.40. 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:

$$in.\tau.\overline{out}.Spec \xrightarrow[l]{in} l :: \tau.\overline{out}.Spec \tag{2.46}$$

where l is an arbitrary location name[15]. 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:

$$\tau.\overline{out}.Spec \xrightarrow{\tau} \overline{out}.Spec \tag{2.47}$$

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

$$l :: \tau.\overline{out}.Spec \xrightarrow[l]{\tau} l :: \overline{out}.Spec \tag{2.48}$$

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.

[15]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.

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, $\overline{out}.Spec$:

$$\overline{out}.Spec \xrightarrow[k]{\overline{out}} k :: Spec \tag{2.49}$$

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

$$l :: \overline{out}.Spec \xrightarrow[lk]{\overline{out}} l :: k :: Spec \tag{2.50}$$

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 2.3.1 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 2.41, had the following transitions using plain CCS:

$$\xrightarrow{in}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\overline{out}} \tag{2.51}$$

whereas the specification exhibits the following behaviour in LCCS:

$$\xrightarrow[l]{in}\xrightarrow[l]{\tau}\xrightarrow[l]{\tau}\xrightarrow[lk]{\overline{out}} \tag{2.52}$$

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:

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

$$\xrightarrow{in}\xrightarrow{\overline{out}} \qquad \text{(Implementation)}$$

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:

$$in.\overline{a}.Receiver \xrightarrow{in} \overline{a}.Receiver \tag{2.53}$$

which, when localised, becomes:

$$in.\overline{a}.Receiver \xrightarrow[l]{in} l :: \overline{a}.Receiver \tag{2.54}$$

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:

$$\xrightarrow{in}\xrightarrow{\tau} \qquad \text{(Receiver)}$$

$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\overline{out}} \qquad \text{(Sender)}$$

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:

$$\xrightarrow[l]{in}\xrightarrow{\tau} \qquad \text{(Localised Receiver)}$$

$$\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow[l]{\overline{out}} \qquad \text{(Localised Sender)}$$

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

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

$$\xrightarrow[l]{in}\xrightarrow[l]{\overline{out}} \qquad \text{(Implementation (Localised))}$$

which illustrates a clear difference between the two.

For the first transition, the two can match each other, as both are capable of performing $\xrightarrow[l]{in}$. 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.

The Distributed Join Calculus

By adding localities, [21] defines a distributed variant of the Join calculus shown in 2.3.1. The extended syntax is as follows:

$$P, Q ::= 0 \mid \texttt{def}\ D\ \texttt{in}\ P \mid (P \mid Q) \mid x\langle\tilde{v}\rangle \mid go\langle b, \kappa\rangle \tag{2.55}$$

$$D, E ::= J \triangleright P \mid D \wedge E \mid \mathbf{T} \mid a[D : P] \tag{2.56}$$

$$J, J' ::= x\langle\tilde{v}\rangle \mid (J \mid J') \tag{2.57}$$

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:

$$\texttt{def}\ a[D:P] \wedge a[D':Q] \triangleright R\ \texttt{in}\ S \tag{2.58}$$

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,

$$\texttt{def}\ a[x\langle y\rangle \triangleright P : S] \wedge b[x\langle z\rangle \triangleright Q : R]\ \texttt{in}\ T \tag{2.59}$$

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

$$\texttt{def}\ a[x\langle y\rangle \triangleright P \wedge x\langle z\rangle \triangleright Q : R]\ \texttt{in}\ T \tag{2.60}$$

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,

$$\texttt{def}\ a[D : (P \mid go\langle b, \kappa\rangle)]\ \texttt{in}\ S \mid \texttt{def}\ b[E : Q]\texttt{in}\ T \tag{2.61}$$

reduces to:

$$\texttt{def}\ b[E : Q \mid (\texttt{def}\ a[D : (P \mid k\langle\rangle)]\ \texttt{in}\ S)]\texttt{in}\ T \tag{2.62}$$

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 2.3.1) 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.

The Ambient Calculus

The ambients within the ambient calculus [13] 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

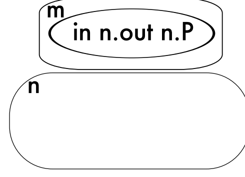

Figure 2.4: Spatial diagram of $m[in\ n.out\ n.P] \mid n[]$

be created and existing ones moved or destroyed during execution. Within the formal syntax of the calculus,

$$E, F \ ::= \ 0 \mid M.E \mid (\nu n)E \mid (E \mid F) \mid n[E] \mid !E \tag{2.63}$$

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. 2.12, 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[16]. Continuing on this presumption, $M.E$ may be considered to be the prefixing already seen in CCS and the π calculus. However, the syntax for M is

$$M \ ::= \ in\ n \mid out\ n \mid open\ n \tag{2.64}$$

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. 2.4). Performing the first action, $in\ n$, moves its surrounding ambient, m, inside n (Fig. 2.5). The converse, $out\ n$, then moves m back outside n, resulting in a return to the original ambient structure (Fig. 2.6), but with the process having evolved into P.

[16] This is the syntax used in versions of the π calculus later than [40].

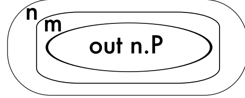

Figure 2.5: Spatial diagram of $n[m[out\ n.P]]$

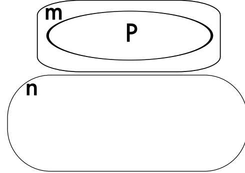

Figure 2.6: Spatial diagram of $m[P] \mid n[]$

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 [13]) 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,

$$(a.P)\backslash a \tag{2.65}$$

may never progress to become P as there is no $\overline{a}$ for a to synchronize 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.

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* [33] is an issue, and is related to the non-determinism mentioned above.

Take the example process from [33].

$$n[in\ m.P] \mid m[Q] \mid m[R] \tag{2.66}$$

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,

$$open\ n.P \mid open\ n.Q \mid n[R] \tag{2.67}$$

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 parent. This may never occur. These kinds of interference, referred to in [33] as *plain interferences*, may occur in other calculi. The equivalent in the π calculus would be:

$$\overline{x}z.P \mid x(y).Q \mid x(y).R \tag{2.68}$$

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,

$$open\ n.\mathbf{0} \mid n[in\ m.P] \mid m[Q] \tag{2.69}$$

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 [32, 33] 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:

$$n[in\ m.P_1 \mid P_2] \mid m[\overline{in}\ m.Q_1 \mid Q2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2] \quad \text{(SafeIn)}$$
$$m[n[out\ m.P_1 \mid P_2] \mid \overline{out}\ m.Q_2 \mid Q2] \rightarrow n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2] \quad \text{(SafeOut)}$$
$$open\ n.P \mid n[\overline{open}\ n.Q_1 \mid Q_2] \rightarrow P \mid Q_1 \mid Q_2 \quad \text{(SafeOpen)}$$

where, in each case, the capability must be able to synchronize 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[17]. Essentially, they represent a move from asynchronous to synchronous mobility primitives. The calculus of controlled ambients [65] 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 [9]. This removes the *open* capability altogether, replacing it with a form of directed communication inspired by [68]. 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 [9] is:

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

$$n[(x)^p P \mid p[\langle M \rangle \mid (x)Q \mid q[\langle N \rangle^{\uparrow}]]] \tag{2.70}$$

where n, p and q are ambients, (x) is an input and $\langle M \rangle$ and $\langle N \rangle$ represent outputs. $(x)Q$ may synchronize with either $\langle M \rangle$ locally or the upward communication from $\langle N \rangle$. $(x)^p P$ must synchronize 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.

Advantages and Disadvantages 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 2.3.1). 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 [13] 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 2.1.3. 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 synchronize. 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.

P Systems

While providing a way of modelling concurrent spatially-oriented systems, P Systems [51] 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.

A P system or *transition super-cell system* [51] of degree n, where $n \geq 1$ is represented as:

$$\Pi = (V, \mu, M_1, \ldots, M_n, (R_1, \rho_1), \ldots, (R_n, \rho_n), i_0) \tag{2.71}$$

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.
- 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 2.3.2, 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 2.2, 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

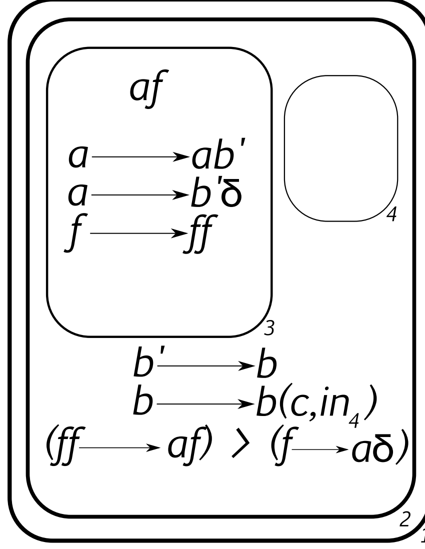

Figure 2.7: Example P System

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. 2.7),

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

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, $f \to ff$, which causes the number of fs to double on each evolution. For a, there are two rules and one is chosen non-deterministically. If the first, $a \to ab'$, is applied, then an additional object b' appears, and the rule may be applied again as an a is still present. If $a \to ab'$ and $f \to ff$ are applied for n steps, then n instances of b' and 2^n occurrences of f are present.

If the second a rule $a \to b'\delta$, 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 $f \to ff$, respectively, and no

a. This changes the configuration of the system to become:

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

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 $a \to ab'$ and $f \to ff$ were applied prior to this, and is greater than or equal to zero.

In M_2, a priority relation exists that forces $ff \to af$ to be given precedence over $f \to a\delta$. As a result, whenever it is possible to apply $ff \to af$ (i.e. there are two f objects), it will be applied instead of $f \to a\delta$. 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 $ff \to af$ will be applied rather than $f \to a\delta$ which destroys M_2. Each time $ff \to af$ is applied, the number of f objects halves.

The remaining rule, $b \to b(c, in_4)$, 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 $b \to b(c, in_4)$ and $ff \to af$ 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, $f \to a\delta$ 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. At present, the main flaw with modelling concurrent systems using this formalism is that the underlying theory is not as advanced as for those of process calculi, such as the π calculus. So far, P

systems research has focused on their power of expressivity, and their application within the field of biological modelling. In the latter, they provide a much more natural perspective than the channel-based operations of the π calculus, and this is something that will be considered further in section 2.5.

Bigraphs

Bigraphs [29, 44] are an attempt at providing a unifying framework, able to represent both spatial relationships (*locality*) in the style of the ambient calculus (see 2.3.2) and link-based-relationships (*connectivity*) seen in the π calculus (see 2.3.1). 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,

$$\overline{x}y \mid x(z).P \rightarrow P\{y/z\} \tag{2.72}$$

which represents synchronisation. In [29], 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:

$$send_{xy} \mid get_{x(z)}\square \rightarrow x \mid y/(z)\square \tag{2.73}$$

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

Similarly, [29] shows how the *in* capability from the ambient calculus:

$$n[in\ m.P_1 \mid P_2] \mid m[Q] \rightarrow m[n[P_1 \mid P_2] \mid Q] \quad (2.74)$$

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:

$$amb_n(in_m \mid \square_0) \mid amb_m\square_1 \rightarrow amb_m(amb_n\square_0 \mid \square_1) \quad (2.75)$$

where the similarities between the two are clear.

Bigraphs provide an interesting framework for unifying the two disparate concepts outlined above in 2.3.1 and 2.3.2. 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 2.2.

2.4 Typed Calculi

A common addition to a process calculus is a type system, especially in recent literature which attempts to use such calculi as the basis for a programming language or a distributed system. 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 [12] or the fact that the x in $x(y)$ should be represent a link and not a mere value [61]. It may also provide more complex intuitions, by distinguishing individual entities, controlling mobility [33, 12] or resources [57] or even providing a full subtyping relation [54, 37]. This section considers a few examples of such type systems for both the π calculus (2.4.1) and the ambient calculus (2.4.2).

2.4.1 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 [43] to those introduced for a specific purpose [61] and more complex systems involving

subtyping [54]. Here, sorts are considered followed by a brief look at the distinction between values and links made by Sangiorgi [61] for the purpose of proving termination.

Sorts

The earliest notion of types was introduced by Milner in [42, 43]. 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' [42]. Formally, a sort is a partial function,

$$ob : \Sigma \rightarrow \Sigma^* \tag{2.76}$$

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,

$$Buf \stackrel{\mathrm{def}}{=} (in, out)(in(x).\overline{out}x.Buf\langle in, out\rangle \tag{2.77}$$

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 π calculus.

Consider the example from [43] of two processes, P and Q:

$$P \stackrel{\mathrm{def}}{=} x(y).\overline{y}uv.\mathbf{0} \tag{2.78}$$

$$Q \stackrel{\mathrm{def}}{=} \overline{x}y'.y'(w).Q' \tag{2.79}$$

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:

$$P \mid Q \rightarrow \overline{y'}uv.\mathbf{0} \mid y'(w).Q' \tag{2.80}$$

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,

$$\begin{aligned} u &: s_1 \mapsto S \\ v &: s_2 \mapsto T \\ w &: s_3 \mapsto (s_1) \\ y &: s_4 \mapsto (s_1, s_2) \\ y' &: s_5 \mapsto (s_1) \end{aligned} \tag{2.81}$$

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 [61] 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,

$$T ::= \#T \mid unit \tag{2.82}$$

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 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 2.2 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

Table 2.2: Typing Rules from [61]

$$\text{T-OUT}\ \frac{\vdash v : \#T, \vdash w : T, \vdash M}{\vdash \overline{v}w.M}$$

$$\text{T-INP}\ \frac{\vdash v : \#T, x \in T, \vdash M}{\vdash v(x).M}$$

$$\text{T-RES}\ \frac{x_i \in \#T_i \text{ for some } T_i (1 \leq i \leq n), \vdash M}{\vdash (x_1 \dots x_n)M}$$

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.

2.4.2 Type Systems for the Ambient Calculus

Early work [14] 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 [11, 23], 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 [12] 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 [18], 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 the paper, 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:

$$S, T ::= Shh \mid W_1 \times \cdots \times W_k \tag{2.83}$$

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 [12], includes these exchange types along with types to control the opening and crossing of ambients. Groups are parameterised over F,

$$F ::= ^{\curvearrowright} \mathbf{G}, ^{\circ} \mathbf{H}, T \tag{2.84}$$

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.

A similar system is adopted in [17], but, as this refers to boxed ambients (see 2.3.2), 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.

2.5 Biological Applications

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 (2.3.2) and P systems (2.3.2) are especially useful for representing the structure of real-world biological entities.

Such modelling is becoming common place within the literature[56, 55, 52], 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 [56], the BioAmbient variant [55] and P systems [52], these are based on Gillespie's algorithm [22].

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 [52] 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 [56], the focus was solely on communication with biological compartments abstracted as private channels. The model given

for BioAmbients [55] is more natural due to the explicit realisation of these compartments.

Take the following example from [55],

$$\begin{aligned} \texttt{System} &::= \texttt{molecule}[\texttt{Mol}] \mid \ \ldots \ \mid \texttt{molecule}[\texttt{Mol}] \mid \texttt{cell}[\texttt{Porin}] \\ \texttt{Mol} &::= \texttt{enter cell1}.\texttt{Mol} + \texttt{exit cell2}.\texttt{Mol} \\ \texttt{Porin} &::= \texttt{accept cell1}.\texttt{Porin} + \texttt{expel cell2}.\texttt{Porin} \end{aligned} \tag{2.85}$$

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 synchronize, 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, [52] 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. [52] presents a model of this phenomenon in *vibrio*

fischeri, a marine bacterium, using a P system[18]:

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

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.

This model seems much more natural and has a clearer correspondence with the real-world representation. The main issue, as noted earlier in 2.3.2, is that the theory of P systems is not as well developed as that of the π calculus (upon which the BioAmbients calculus is essentially based). This can prove problematic, especially when model checking such models.

2.6 Conclusion

In conclusion, this review has taken a brief look at the field of concurrency, 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, is less expressive but can model this kind of behaviour.

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

[18]This method of defining the configuration differs slightly from that in 2.3.2, as it also includes a set of labels, rather than assuming that the natural numbers are used.

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.

Discrete timed calculi can overcome this. An example using TPL to model a compositional broadcasting agent, using semantics suitable for any arbitrary number of processes, is provided in 2.2. Extensions to TPL, such as CaSE, may scale even further using synchronous encapsulation to create systems of multiple components.

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. Biology was also considered briefly (see 2.5), as a potential application area. P systems seem the most natural formalism, but they lack some of the proven theoretical aspects of process calculi.

Following much consideration of the available literature, the concept of a calculus which combines both the mobility of the π and ambient calculi with the inherent scalability of a calculus like CaSE seems novel. This research hopes to provide just such a formalism.

Chapter 3

Current Work

The aim of this research is to construct a process calculus which combines the notions of discrete time and mobility. Earlier work during an undergraduate project focused on developing a semantics for the Cashews[1][49] language, using the CaSE process calculus (see section 2.2.1) and later, a conservative extension to it called Cashew-Nuts. It became clear during this project that it would be interesting to further extend CaSE with a notion of mobility, and this led to the development of the calculus of *Typed Nomadic Time* (TNT) discussed here.

3.1 The Calculus of Synchronous Encapsulation (CaSE)

The syntax for CaSE, given in [47], is as follows:

$$\begin{aligned} \mathcal{E}, \mathcal{F} \; ::= \; & \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ & \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid \mathcal{E} / \sigma \end{aligned} \tag{3.1}$$

where $\alpha \in Act$ as in the definition of CCS (see 2.1.1). $\mathbf{0}$, $\alpha.\mathcal{E}$, $\mathcal{E} + \mathcal{F}$, $(\mathcal{E} \mid \mathcal{F})$, $\mu X.\mathcal{E}$, X and $\mathcal{E} \setminus a$ retain their behaviour defined in CCS, but now exhibit additional actions due to the presence of clocks. These are drawn from a countably infinite set, $\mathcal{T}$, over which σ ranges.

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:

[1]Cashews is a language for web service composition, initially based on OWL-S.

$$a.0 \xrightarrow{\sigma} a.0 \tag{3.2}$$

assuming a clock context containing just the one clock, σ. Similarly, non-deterministic choice and parallel composition 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 synchronize, causing a τ transition, this takes precedence over the clock transitions, due to *maximal progress* (see 2.2) and the original set of derivations for parallel composition (see 2.1.1) 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.

3.1.1 Timeouts

Moving on to the new operators, CaSE, as presented in [47], includes two variants of the timeout operator, first seen in TPL. Recall from 2.2 that the operator essentially allows a decision to be made, based on the presence of a clock tick. In the general scenario,

$$\lfloor E \rfloor \sigma(F) \tag{3.3}$$

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 synchronization 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.

3.1.2 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,

$$\Delta_\sigma \xrightarrow{\rho} \Delta_\sigma \tag{3.4}$$

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

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:

$$[\![\underline{\alpha}.P]\!] \stackrel{\text{def}}{=} \alpha.P + \Delta \tag{3.5}$$

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.

$$[\![\underline{\alpha}_\sigma.P]\!] \stackrel{\text{def}}{=} \alpha.P + \Delta_\sigma \tag{3.6}$$

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

$$[\![\sigma.P]\!] \stackrel{\text{def}}{=} \lceil \mathbf{0} \rceil \sigma(P) \tag{3.7}$$

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:

$$[\![\underline{\sigma}.P]\!] \stackrel{\text{def}}{=} \lceil \Delta \rceil \sigma(P) \tag{3.8}$$

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.

3.1.3 Encapsulation

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

$$(P/\sigma) \mid Q \tag{3.9}$$

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.

3.2 Localising the Calculus

Localisation, discussed in detail in 2.3.2, 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 just described.

Thus, the first step in the evolution towards TNT is to combine these two hierarchical concepts by effectively localising CaSE. The notion of components and encapsulation is explicitly realised by a locality, 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 localities. The bounds of the locality define both a new group and the scope of the clock hiding. The syntax for localised CaSE is thus:

$$\begin{aligned}\mathcal{E},\mathcal{F} \ ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E}+\mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor\mathcal{E}\rfloor\sigma(\mathcal{F}) \mid \\ \lceil\mathcal{E}\rceil\sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E}\setminus a \mid m[\mathcal{E}]_{\{\vec{\sigma}\}}\end{aligned} \tag{3.10}$$

where m represents an arbitrary locality name[2]; the names of localities are not distinct, and hence do not form a set. In particular, m may be equal to the empty string, ϵ, thus facilitating the use of anonymous localities. This allows the semantics of CaSE's clock hiding to be encoded:

$$[\![E/\sigma]\!] \stackrel{\text{def}}{=} [E]_{\{\sigma\}} \tag{3.11}$$

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

Recall the example of clock hiding above (3.9). This becomes:

$$[P]_{\{\sigma\}} \mid Q \tag{3.12}$$

in localised CaSE, or:

[2]Note that although names are added to the localities here, this is not really necessary at this stage; they provide nothing more than a way to refer to localities in talking about a system. However, they are necessary for providing migration as discussed in 2.3.2

$$m[P]_{\{\sigma\}} \mid Q \tag{3.13}$$

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

With this extension the set of visible clocks for a particular locality may be obtained by taking the union of its set of clocks and the sets of the parent localities. For example, consider the more complex scenario:

$$n[E \mid m[F \mid k[G]_{\{\sigma\}}]_{\{\rho\}}]_{\{\gamma\}} \tag{3.14}$$

where the top-level locality, n, contains a process E and a further sub-locality, m. Likewise, m contains both a process, F, and the sub-locality, k. Finally, k contains just the single process, G. The set of clocks for the locality 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 $\{\sigma\} \cup \{\rho\} \cup \{\gamma\}$ or simply $\{\sigma, \rho, \gamma\}$, which means that G, located in k, can see the ticks of all three clocks.

F, by comparison, can only see the ticks of the clocks, ρ and γ, as σ is hidden outside k. E, in the top-level locality, n, can only observe silent actions resulting from the two hidden clocks, ρ and σ, but can see the ticks of γ. Taking this further, it is clear that the clock context, the set of clocks within the system, can be derived as the union of the sets of clocks associated with each locality ($\{\sigma, \rho, \gamma\}$ in this case).

3.3 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[3]. 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 localities. This is shown in more detail in 3.3.1.

In addition, TNT 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:

[3] Although the semantics could be adapted so as to use the localities for bisimulation, as in 2.3.2.

1. One of the processes becoming $\mathbf{0}$. The ambient calculus includes a structural congruence law,

$$E \mid \mathbf{0} \equiv E \tag{3.15}$$

which allows such processes to be removed. Note that this doesn't hold for TNT, due to the addition of clocks. $\mathbf{0}$ exists through time, and, as such, has transitions for each clock. Thus, if $\mathbf{0}$ is removed from the above equation, there will be fewer possible transitions and so it follows that the two should be regarded as different processes.

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.

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,

$$m[E \mid F \mid G] \mid n[\mathbf{0}] \mid H \tag{3.16}$$

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,

$$Emov \stackrel{\text{def}}{=} z[out\ m.E] \tag{3.17}$$

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,

$$m[z[out\ m.E] \mid F \mid G] \mid n[\mathbf{0}] \mid H \tag{3.18}$$

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:

$$m[z[out\ m.E] \mid F \mid G] \mid n[\mathbf{0}] \mid H \mid open\ z.\mathbf{0} \tag{3.19}$$

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

$$m[z[out\ m.in\ n.E] \mid F \mid G] \mid n[open\ z.\mathbf{0}] \mid H \tag{3.20}$$

and neither are particularly natural. TNT instead provides this functionality as a base part of the syntax, which will be explored in 3.3.2.

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

$$a.P \mid m[\overline{a}.Q]_{\{\sigma\}} \tag{3.21}$$

synchronization between the two processes is not permitted as they lie on either side of a locality 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, TNT (at present) does not incorporate the scoping of locality names.

3.3.1 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:

$$\begin{aligned} \mathcal{E}, \mathcal{F} \ ::= \mathbf{0} \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E} + \mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor \mathcal{E} \rfloor \sigma(\mathcal{F}) \mid \\ \lceil \mathcal{E} \rceil \sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E} \setminus a \mid m[\mathcal{E}]_{\{\vec{\sigma}\}} \mid \mathcal{M}.\mathcal{E} \end{aligned} \tag{3.22}$$

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

$$\mathcal{M} ::= in\ m \mid out\ m \mid open\ m \tag{3.23}$$

with m again representing the name of a locality. The behaviour of these primitives is identical to the behaviour of their equivalents in the ambient calculus, so just a short recap of section 2.3.2 is given here, using the syntax

above. Note that the syntactic abbreviation, $m[E]$, is used to represent $m[E]_{\{\}}$.

When a process emits an *in* m capability, the surrounding locality may move into a sibling locality with the name, m. Given the context,

$$m[E] \mid n[\mathbf{0}] \tag{3.24}$$

E may be defined as

$$E \stackrel{\text{def}}{=} in\ n.E' \tag{3.25}$$

allowing the derivation

$$m[E] \mid n[\mathbf{0}] \stackrel{in\ n}{\rightarrow} n[m[E'] \mid \mathbf{0}] \tag{3.26}$$

to occur. Similarly, defining E' to be

$$E' \stackrel{\text{def}}{=} out\ n.E'' \tag{3.27}$$

allows the converse

$$n[m[E'] \mid \mathbf{0}] \stackrel{out\ n}{\rightarrow} m[E''] \mid n[\mathbf{0}] \tag{3.28}$$

to take place, *out* m allowing the surrounding locality to move outside a parent locality named m. As noted above, these are fairly dull, both being identical to the same primitives in the ambient calculus. The behaviour of *open* m is more interesting, due to its interaction with the locality's clock environment.

Take the example context,

$$m[E \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \tag{3.29}$$

where E is defined as

$$E \stackrel{\text{def}}{=} open\ n.E' \tag{3.30}$$

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

$$m[E \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \stackrel{open\ n}{\rightarrow} m[E'' \mid F]_{\{\sigma,\rho\}} \tag{3.31}$$

and 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 *open* n, E could only see ticks from the clock ρ. The ticks of σ were converted to silent actions by the locality barrier. Following the dissolution of the locality, n, these ticks

become visible to E. So, the *open* capability in TNT not only changes the locality hierarchy, but also the clock context within the parent locality.

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

$$m[open\ n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \tag{3.32}$$

has two possible derivations

1. $m[open\ n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \xrightarrow{open\ n} m[E \mid F \mid n[G]_{\{\gamma\}}]_{\{\sigma,\rho\}}$
2. $m[open\ n.E \mid n[F]_{\{\sigma\}} \mid n[G]_{\{\gamma\}}]_{\{\rho\}} \xrightarrow{open\ n} m[E \mid n[F]_{\{\rho\}} \mid G]_{\{\gamma,\rho\}}$

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 3.3.3, which reduce the possibility of *grave interferences* (see 2.3.2).

3.3.2 Process Mobility

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

$$\mathcal{M} ::= in\ m \mid out\ m \mid open\ m \mid on\ \beta\ in\ m \mid on\ \beta\ out\ m \tag{3.33}$$

where $\beta \in \mathcal{N}$ and thus refers to an action. While the location mobility described above is *subjective* (the process who requests the move does the move), process mobility, in this form, is *objective*. The process which emits one of the two new capabilities synchronizes 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 locality, 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,

$$on\ go\ in\ m.E \mid go.F \mid m[\mathbf{0}]_{\{\sigma\}} \tag{3.34}$$

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

$$on\ go\ in\ m.E \mid go.F \mid m[\mathbf{0}]_{\{\sigma\}} \xrightarrow{\tau} E \mid m[F \mid \mathbf{0}]_{\{\sigma\}} \tag{3.35}$$

with the continuation, F, continuing to evolve in the locality m. Note that the transition is labelled with a silent τ action to represent the synchronization, rather than with a label to match the mobility primitive. There is a distinct advantage to this, in that movements are then treated in the same way as synchronizations. They form part of the synchronous clock cycles, via *maximal progress*, which allows them to be used for broadcasting in the same compositional style demonstrated in 2.2 for actions. In the next section, the addition of bouncers results in the location mobility primitives also emitting τ actions and thus also fitting in to this same structure.

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

$$F \stackrel{\text{def}}{=} leave.F' \mid on\ leave\ out\ m \tag{3.36}$$

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

$$E \mid m[leave.F' \mid on\ leave\ out\ m.\mathbf{0} \mid \mathbf{0}]_{\{\sigma\}} \xrightarrow{\tau} E \mid F' \mid m[\mathbf{0} \mid \mathbf{0}]_{\{\sigma\}} \tag{3.37}$$

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

More generally, a subjective process movement may be encoded as

$$[\![in\ m\ Q.E]\!] \stackrel{\text{def}}{=} z.Q \mid on\ z\ in\ m.E \tag{3.38}$$

where F is the process that will move in to m, E is the continuation and z is a new name that doesn't appear free in the surrounding context. The converse is pretty much the same:

$$[\![out\ m\ Q.E]\!] \stackrel{\text{def}}{=} z.Q \mid on\ z\ out\ m.E \tag{3.39}$$

with the most problematic issue with these definitions being the use of z. Subjective movement is safer on an ad-hoc basis where the surrounding context is known.

3.3.3 Bouncers

This description of TNT is concluded by the addition of the final element, the *bouncers*. Named after the person who stands outside a night club, the bouncer is an additional property of a locality which appears in the top right.

It has no real behaviour of its own, but instead performs the job of protecting the locality, essentially being a process with a more limited choice of available actions[4]. This is achieved by the bouncer dictating which capabilities may affect its locality, via the use of a series of co-capabilities, along the lines of those in [33] (see section 2.3.2).

The full syntax of TNT may now be given as:

$$\begin{aligned}\mathcal{E},\mathcal{F} \ ::= \mathbf{0} \mid \Omega \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid \mathcal{E}+\mathcal{F} \mid (\mathcal{E} \mid \mathcal{F}) \mid \lfloor\mathcal{E}\rfloor\sigma(\mathcal{F}) \mid \\ \lceil\mathcal{E}\rceil\sigma(\mathcal{F}) \mid \mu X.\mathcal{E} \mid X \mid \mathcal{E}\setminus a \mid m[\mathcal{E}]^{\mathcal{E}}_{\{\vec{\sigma}\}} \mid \mathcal{M}.\mathcal{E}\end{aligned} \tag{3.40}$$

where $\mathcal{M}$ is now

$$\begin{aligned}\mathcal{M} \ ::= in\ m \mid out\ m \mid open\ m \mid on\ \beta\ in\ m \mid on\ \beta\ out\ m \mid \\ \overline{in} \mid \overline{out} \mid \overline{open}\end{aligned} \tag{3.41}$$

and Ω represents the bouncer with no behaviour (the equivalent of $\mathbf{0}$). For a process or locality to enter another locality, the bouncer must allow this to occur by providing the corresponding $\overline{in}$ co-capability. Likewise, it must provide $\overline{out}$ to allow a process or locality to leave. With regard to the destruction of a locality, the locality's bouncer must allow it to be removed by providing a $\overline{open}$ co-capability.

Recall the example given in 3.3.1.

$$m[in\ n.E'] \mid n[\mathbf{0}] \tag{3.42}$$

With the addition of bouncers, this becomes:

$$m[in\ n.E']^{\Omega} \mid n[\mathbf{0}]^{\overline{in}.\Omega} \tag{3.43}$$

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

$$m[in\ n.E']^{\Omega} \mid n[\mathbf{0}]^{\overline{in}.\Omega} \xrightarrow{\tau} n[m[E']^{\Omega} \mid \mathbf{0}]^{\Omega} \tag{3.44}$$

but any subsequent behaviour is disallowed, as the bouncer of n has now evolved to also be Ω. Using this method, it becomes possible to specify how

[4]This limited choice is only explicitly imposed by the type system. Syntactically, there is no restriction.

many entities (processes or localities) may enter a locality. For example, the bouncer:

$$\mu X.\overline{in}.\overline{in}.\overline{out}.\overline{out}.X \tag{3.45}$$

allows two entities to enter, but two must then leave before another can enter. On the subject of exiting a locality, the synchronization with $\overline{out}$ works in the same way as $\overline{in}$:

$$n[m[out\ n.E'']^{\Omega} \mid \mathbf{0}]^{\overline{out}.\Omega} \xrightarrow{\tau} m[E'']^{\Omega} \mid n[\mathbf{0}]^{\Omega} \tag{3.46}$$

Finally, the destruction of a locality is probably the easiest of the three to understand. Again, using an example from 3.3.1,

$$m[open\ n.E' \mid n[F]_{\{\sigma\}}]_{\{\rho\}} \tag{3.47}$$

it may be endowed with bouncers to give:

$$m[open\ n.E' \mid n[F]^{\overline{open}.\Omega}_{\{\sigma\}}]^{\Omega}_{\{\rho\}} \tag{3.48}$$

This allows the following synchronization to occur:

$$m[open\ n.E' \mid n[F]^{\overline{open}.\Omega}_{\{\sigma\}}]^{\Omega}_{\{\rho\}} \xrightarrow{\tau} m[E' \mid F]^{\Omega}_{\{\sigma,\rho\}} \tag{3.49}$$

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

As mentioned in the previous section, all capabilities are now performed as a synchronization, following the introduction of bouncers. This means that any movement will cause an internal action, τ, to occur which fits in nicely with the synchronization cycles and maximal progress drawn from CaSE. This notion is central to the example presented in section 3.5.

3.4 The Semantics

This section gives TNT a structured operational semantics based on a labelled transition system, by extending the existing semantics of CaSE. Table 3.1 gives the subset of the semantics which are common to both TNT and CaSE, with σ and ρ ranging over the set of clocks, α over the set of actions, γ over both and a over the actions sans τ. *Idle* and *Patient* represent the progress of time over $\mathbf{0}$ 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.

Table 3.1: Semantics: Common CaSE Subset

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

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

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

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

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

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

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

$$\text{STO2}\ \frac{E \xrightarrow{\alpha} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\alpha} E'} \qquad \text{STO3}\ \frac{E \xrightarrow{\rho} E'}{\lceil E \rceil \sigma(F) \xrightarrow{\rho} \lceil E' \rceil \sigma(F)}\ \rho \neq \sigma$$

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

Table 3.2: Semantics: Clock Hiding and Mobility

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

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

*Sum*1 and *Sum*2 represent the performance of an action on either side of the summation operator, thus also implying the commutativity of the operator. *Par*1 and *Par*2 do the same for parallel composition. *Sum*3 and *Par*4 represent the passage of time over these two operators. Note that time must be able to pass on both sides, and that maximal progress is enforced by the restriction $E \mid F \not\xrightarrow{\tau}$ in *Par*4.

*Par*3 encapsulates synchronization; 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. *FTO*1 and *STO*1 are identical, allowing the dissolution of the timeout via a tick of the associated clock, σ, on the provision that $E \not\xrightarrow{\tau}$. The difference between the two timeouts is shown by *FTO*2, *STO*2 and *STO*3. *FTO*2 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 *STO*3, clocks other than σ may tick, but the timeout stays in place. *STO*2 handles the removal of the stable timeout, due to an action performed by E.

Finally, *Rec* provides recursion, performing substitution of X for the body of the recursion as soon as any transition, γ, occurs and *Res* defines restriction, which disallows any transitions for the given action.

The semantics given in Table 3.2 are similar to the hiding rules given for CaSE, but are instead applied to the new syntactic form used in TNT. Also included is the rule which allows the mobility prefix to evolve, thus completing the semantics for the syntax of $\mathcal{E}$ and $\mathcal{F}$.

The rules are quite simple. *LHd*1 provides the conversion of the ticks of the hidden clocks to silent actions; if E can perform a σ transition, then it performs a τ transition in a context where σ is one of the hidden clocks. *LHd*2 and *LHd*3 simply allow the remaining actions and clock transitions respectively, to occur normally. *Cap* allows the mobility capabilities and co-capabilities to emit a transition, but the process itself can only evolve in the

Table 3.3: Semantics: Locality Mobility

$$\text{InLoc}\ \frac{B_1 \stackrel{\overline{in}}{\rightarrow} B_1'}{n[in\ m.E \mid F]^{B_2}_{\vec{\sigma}} \mid m[G]^{B_1}_{\vec{\rho}} \stackrel{\tau}{\rightarrow} m[G \mid n[E \mid F]^{B_2}_{\vec{\sigma}}]^{B_1'}_{\vec{\rho}}}$$

$$\text{OutLoc}\ \frac{B_1 \stackrel{\overline{out}}{\rightarrow} B_1'}{m[G \mid n[out\ m.E \mid F]^{B_2}_{\vec{\sigma}}]^{B_1}_{\vec{\rho}} \stackrel{\tau}{\rightarrow} n[E \mid F]^{B_2}_{\vec{\sigma}} \mid m[G]^{B_1'}_{\vec{\rho}}}$$

Table 3.4: Semantics: Open

$$\text{Open}\ \frac{B_1 \stackrel{\overline{open}}{\rightarrow} B_1'}{n[open\ m.E \mid m[F]^{B_1}_{\vec{\sigma}}]^{B_2}_{\vec{\gamma}} \stackrel{\tau}{\rightarrow} n[E \mid F]^{B_2}_{\vec{\gamma}\cup\vec{\sigma}}}$$

context of movement.

On that subject, Table 3.3 gives the rules required for location mobility. *InLoc* allows a τ transition to occur and n to move into m if both an $in\ m$ and an $\overline{in}$ transition are available from the process $in\ m.E$ and the bouncer, B_1, respectively. *OutLoc* is basically the same thing, but for $out\ m.E$ and $\overline{out}$.

Table 3.4 depicts the behaviour of $open\ m$, which again causes a τ transition to occur when both an $open\ m$ and an $\overline{open}$ transition are available. The named locality is also destroyed and the two clock contexts unified.

Finally, Table 3.5 shows the semantics for the two process mobility capabilities. In both rules, E moves due to a mobility primitive which is part of F. This occurs if $on\ a\ in\ m$ or $on\ a\ out\ m$, a and $\overline{in}$ or $\overline{out}$ respectively, and a τ action is emitted as a result of this three-way synchronization.

Note that channels in E may become unrestricted due to its move to a different locality. This is illustrated in the rules by $((E \mid G) \setminus \vec{b})$, which becomes simply $G \setminus \vec{b}$ when E moves. As a result, any names that were members of $\vec{b}$ and thus restricted in E are no longer in this situation following the movement of the process. Actions are scoped to individual localities, and the names are unique, so E can neither see the name in the old locality, nor maintain its own copy.

Table 3.5: Semantics: Process Mobility

$$\text{ProcIn}\frac{E \xrightarrow{a} E', B_1 \xrightarrow{\overline{in}} B_1'}{((E \mid G) \setminus \vec{b}) \mid on\ a\ in\ m.F \mid m[H]_{\vec{\sigma}}^{B_1} \xrightarrow{\tau} (G \setminus \vec{b}) \mid F \mid m[H \mid E']_{\vec{\rho}}^{B_1'}}$$

$$\text{ProcOut}\frac{E \xrightarrow{a} E', on\ a\ out\ m.F \xrightarrow{on\ a\ \overline{out}\ m} on\ a\ out\ m.F, B_1 \xrightarrow{\overline{out}} B_1'}{H \mid m[((E \mid G) \setminus \vec{b}) \mid on\ a\ out\ m.F]_{\vec{\sigma}}^{B_1} \xrightarrow{\tau} H \mid E' \mid m[(G \setminus \vec{b}) \mid F]_{\vec{\sigma}}^{B_1'}}$$

3.5 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 TNT process calculus.

The game environment is represented using localities. In the musical chairs scenario, each chair is represented by a locality, as is the 'sin bin', to which players are moved when they are no longer in the game. These localities are all nested inside a further locality 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 larger system, each performing its own internal interactions and entering into the synchronization cycle of the larger system.

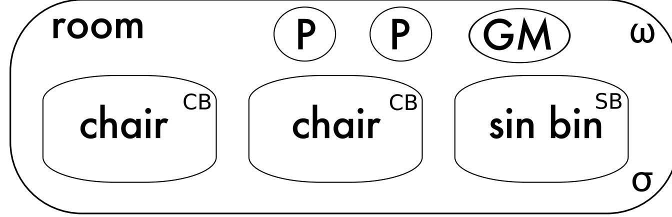

Figure 3.1: The Musical Chairs Environment

The locality structure is represented graphically by Fig. 3.1 and in the calculus by the equation shown below.

$$room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid sinbin[\mathbf{0}]^{SB} \mid P \mid P \mid GM]^{\Omega}_{\{\sigma\}} \tag{3.50}$$

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 localities. 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 3.6, and make use of the derived syntax for a clock prefix, $\sigma.P$, shown in 3.1.2.

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 localities, the room locality is not prone to either destruction or the entry or exit of other localities, having a bouncer simply equal to Ω. This retains the encapsulation of the model as a single room locality, and prevents other processes or localities from interfering with its behaviour.

The definition of appropriate bouncers is essential for the chairs (3.51) and the sin bin (3.52). It is the chair bouncer that enforces the implicit predicate that only one player may inhabit a chair at any one time, while the sin bin bouncer prevents players leaving the sin bin once they have entered.

To model stage one of the game, n player processes and n chair locations are placed in the room. The advantage of using TNT for this model is that the actual number of players or chairs is irrelevant. They only have to be equal at the start to accurately model the game. The calculus 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:

Table 3.6: Summary of Processes and Derived Syntax for Musical Chairs

$$CB \stackrel{\text{def}}{=} \mu X.(\overline{in}.\overline{out}.X + \overline{open}) \tag{3.51}$$

$$SB \stackrel{\text{def}}{=} \mu X.\overline{in}.X \tag{3.52}$$

$$GM1 \stackrel{\text{def}}{=} \sigma.GM2 \tag{3.53}$$

$$GM2 \stackrel{\text{def}}{=} open\ chair.GM3 \tag{3.54}$$

$$GM3 \stackrel{\text{def}}{=} \sigma.GM4 \tag{3.55}$$

$$GM4 \stackrel{\text{def}}{=} \mu X.(\lceil on\ sit\ in\ chair.X\rceil\sigma(GM5)) \tag{3.56}$$

$$GM5 \stackrel{\text{def}}{=} \mu X.(\lceil on\ leave\ in\ sinbin.X\rceil\sigma(GM1)) \tag{3.57}$$

$$P \stackrel{\text{def}}{=} \sigma.\sigma.MP \tag{3.58}$$

$$MP \stackrel{\text{def}}{=} \lceil sit.PIC\rceil\sigma(Loser) \tag{3.59}$$

$$PIC \stackrel{\text{def}}{=} \sigma.\sigma.PLC \tag{3.60}$$

$$PLC \stackrel{\text{def}}{=} on\ stand\ out\ chair.0|stand.P \tag{3.61}$$

$$L \stackrel{\text{def}}{=} leave.0 \tag{3.62}$$

$$room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid Pr \mid P \mid GM1]^{\Omega}_{\{\sigma\}}. \tag{3.63}$$

The room and chairs appear as shown earlier. The player processes (3.58) 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,

$$\begin{aligned} &room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid P \mid P \mid GM1]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma}\ &room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM2]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.64}$$

which the gamesmaster ($GM1$ (3.53)) also waits for, before evolving into $GM2$ (3.54). 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 silent action. The transition from stage three to stage four is thus as follows:

$$\begin{aligned} & room[chair[\mathbf{0}]^{CB} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM2]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\tau} \; & room[\mathbf{0} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM3]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.65}$$

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.

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0}]^{CB} \mid \sigma.MP \mid \sigma.MP \mid GM3]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} \; & room[\mathbf{0} \mid chair[\mathbf{0}]^{CB} \mid MP \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.66}$$

The aim of stage five is to get as many player processes as possible inside chair localities. This is handled by again relying on maximal progress to essentially perform a form of broadcast that centres on mobile actions, as briefly mentioned in 3.3.2. Rather than sending a signal to a number of recipients, a request to move into a chair (see (3.56) and (3.59)) is delivered instead.

If a chair is available, then a player process will enter it (the actual chair and player chosen is non-deterministic). This will cause an internal 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 τ transition:

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0}]^{CB} \mid MP \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\tau} \; & room[\mathbf{0} \mid chair[\mathbf{0} \mid PIC]^{\overline{out}.CB} \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.67}$$

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.

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0} \mid PIC]^{\overline{out}.CB} \mid MP \mid GM4]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} \; & room[\mathbf{0} \mid chair[\mathbf{0} \mid \sigma.PLC]^{\overline{out}.CB} \mid L \mid GM5]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.68}$$

Both stage six and seven proceed in a similar way. Stage six sees essentially the same broadcasting behaviour applied to the losing players (see (3.57) and (3.62)). 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 localities is disallowed in TNT, an implicit scoping of the broadcast occurs. In the example, stage six again sees just one τ transition:

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0} \mid \sigma.PLC]^{\overline{out}.CB} \mid L \mid GM5]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\tau} \; & room[\mathbf{0} \mid chair[\mathbf{0} \mid \sigma.PLC]^{\overline{out}.CB} \mid GM5]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.69}$$

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

$$\begin{aligned} & sinbin[\mathbf{0}]^{SB} \mid L \mid GM6 \\ \xrightarrow{\tau} & sinbin[\mathbf{0} \mid \mathbf{0}]^{SB} \mid GM6 \end{aligned} \tag{3.70}$$

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

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0} \mid \sigma.PLC]^{\overline{out}.CB} \mid GM5]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\sigma} \; & room[\mathbf{0} \mid chair[\mathbf{0} \mid PLC]^{\overline{out}.CB} \mid GM1]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.71}$$

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

$$\begin{aligned} & room[\mathbf{0} \mid chair[\mathbf{0} \mid PLC]^{\overline{out}.CB} \mid GM1]^{\Omega}_{\{\sigma\}} \\ \xrightarrow{\tau} \; & room[\mathbf{0} \mid chair[\mathbf{0} \mid \mathbf{0}]^{CB} \mid GM1 \mid P]^{\Omega}_{\{\sigma\}} \end{aligned} \tag{3.72}$$

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

3.6 The Type System

This final section focuses on the beginnings of a type system for the calculus. The concepts behind this are based on the type systems presented for the ambient calculus (see 2.4.2) and specifically the notion of groups presented in [12] and [17]. The current focus is on further restricting mobility, this time limiting which process may move rather than how many, as is implied by the bouncers of 3.3.3. The type system also provides the distinction between normal process primitives and the primitives used by bouncers, which is implicit in the examples above.

Table 3.7: Types: Basics

$$\text{Env}\ \frac{\xi : T \in \Gamma}{\Gamma \vdash \xi : T} \qquad \text{Nil}\ \frac{-}{\Gamma \vdash \mathbf{0} : Proc(g)}$$

$$\text{BNil}\ \frac{-}{\Gamma \vdash \Omega : BProc} \qquad \text{Stop}\ \frac{-}{\Gamma \vdash \Delta : Proc(g)}$$

$$\text{Stall}\ \frac{\Gamma \vdash \sigma : Clock}{\Gamma \vdash \Delta_\sigma : Proc(g)} \qquad \text{Act}\ \frac{\Gamma \vdash \alpha : Act, \Gamma \vdash P : Proc(g)}{\Gamma \vdash \alpha.P : Proc(g)}$$

$$\text{Rec}\ \frac{\Gamma \vdash P : Proc(g)}{\Gamma \vdash \mu X.P : Proc(g)} \qquad \text{Res}\ \frac{\Gamma \vdash a : Act, \Gamma \vdash P : Proc(g)}{\Gamma \vdash P \setminus a : Proc(g)}$$

Each process and locality is a member of a group, which determines the use of the mobility primitives. Each group has a type[5], $(\mathscr{C}, \mathscr{S}, \mathscr{O}, \mathscr{E})$, with each element being a set of group names. Entities in groups that are members of $\mathscr{C}$ are allowed to cross or pass through localities in the given group. For example, if g_1 has type G_1, where $g_2 \in \mathscr{C}(G_1)$, then localities or processes in group g_2 may cross through localities in g_1. In the same way, $\mathscr{S}$ is the set of groups that may stay in a locality of that group. This is implicitly a subset of $\mathscr{C}$ as, if an entity can reside permanently in a locality, it must also be able to simply pass through it as well. Finally, $\mathscr{O}$ contains the groups whose members may destroy the locality via the *open* primitive, and $\mathscr{E}$ those whose localities moving processes may enter.

Table 3.7 presents the basic rules and the rudimentary types used for the basic parts of the syntax, such as **0**. As is standard in the literature, the types are defined with respect to a type environment, Γ. On this note, the rule *Env* simply 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.

The remaining rules in Table 3.7 provide types for the processes. Via *Nil* and *Stop*, both **0** and Δ are given a type of $Proc(g)$, where g is a group. There are no preconditions for these derivations. Likewise, Ω can be typed as a $BProc$, a bouncer process, thus distinguishing it from the normal processes, such as **0**.

The other rules are also pretty simple. *Stall* simply says that Δ_σ may

[5] Or, more accurately, as groups are types themselves, it essentially has a type of a type or a *kind*

Table 3.8: Types: Operators

$$\text{Sum}\ \frac{\Gamma_1 \vdash P : Proc(g), \Gamma_2 \vdash Q : Proc(g), \Gamma_1 \# \Gamma_2}{\Gamma_1 \cup \Gamma_2 \vdash P + Q : Proc(g)}$$

$$\text{Par}\ \frac{\Gamma_1 \vdash P : Proc(g), \Gamma_2 \vdash Q : Proc(g), \Gamma_1 \# \Gamma_2}{\Gamma_1 \cup \Gamma_2 \vdash P \mid Q : Proc(g)}$$

$$\text{FTO}\ \frac{\Gamma_1 \vdash P : Proc(g), \Gamma_2 \vdash Q : Proc(g), \Gamma_1 \# \Gamma_2, \Gamma_1 \cup \Gamma_2 \vdash \sigma : Clock}{\Gamma_1 \cup \Gamma_2 \vdash \lfloor P \rfloor \sigma(Q) : Proc(g)}$$

$$\text{STO}\ \frac{\Gamma_1 \vdash P : Proc(g), \Gamma_2 \vdash Q : Proc(g), \Gamma_1 \# \Gamma_2, \Gamma_1 \cup \Gamma_2 \vdash \sigma : Clock}{\Gamma_1 \cup \Gamma_2 \vdash \lceil P \rceil \sigma(Q) : Proc(g)}$$

be typed as a process of group g if σ is a clock. *Act* states that $\alpha.P$ is a process in g if α is an action and P is also typeable as a process in the same group. In the same vein, *Rec* and *Res* type recursive and restricted processes respectively, if the constituent process, P, is already typeable as a process. In the case of *Res*, a must also be an action if the process is to be successfully typed.

In Table 3.8, types are given to the composition of processes using the binary operators for summation, parallel composition and timeout. All four are pretty much identical, providing a type for the process resulting from the combination of the operator with two other processes, P and Q. As each of these processes may be typed under a different type environment (represented by Γ_1 and Γ_2), the cumulative process is typed under the union of the two, on the condition that the two are compatible ($\Gamma_1 \# \Gamma_2$). Compatibility is possible if there is no overlap between the two environments. Such overlap occurs when the environments provide a different type for the same entity.

The only other issue worthy of note with respect to these rules is that *FTO* and *STO* also require that σ is typeable as a clock, another restriction which simply makes explicit a number of issues implied in the syntax.

The types in Tables 3.7 and 3.8 provide the basis for the mobility types presented in Table 3.9, which form the focus of the type system. This table is so far incomplete, as it lacks typing for process mobility and the rule to link processes and localities. The latter exists, but may need further work. These issues are discussed in 4.1.

Of the ones presented here, *LocIn*, *LocOut* and *Open* are fairly similar, all relating to whether a particular location movement is typeable, based on

Table 3.9: Types: Mobility

$$\text{Cap}\ \frac{\Gamma \vdash \mathcal{M} : Proc(g_1) \to Proc(g_2), \Gamma \vdash P : Proc(g_1)}{\Gamma \vdash \mathcal{M}.P : Proc(g_2)}$$

$$\text{LocIn}\frac{\Gamma \vdash g_2 : G_2, \Gamma \vdash m : Loc(g_1), g_1 \in \mathscr{C}(G_2)}{\Gamma \vdash in\ m : Proc(g_2) \to Proc(g_2)}$$

$$\text{LocOut}\frac{\Gamma \vdash g_1 : G_1, \Gamma \vdash g_2 : G_2, \Gamma \vdash m : Loc(g_1), g_1 \in \mathscr{C}(G_2), \mathscr{S}(G_1) \subseteq \mathscr{S}(G_2)}{\Gamma \vdash out\ m : Proc(g_2) \to Proc(g_2)}$$

$$\text{Open}\frac{\Gamma \vdash g_1 : G_1, \Gamma \vdash g_2 : G_2, \Gamma \vdash m : Loc(g_1), g_1 \in \mathscr{O}(G_2), g_2 \in \mathscr{S}(G_1)}{\Gamma \vdash open\ m : Proc(g_2) \to Proc(g_2)}$$

the groups of the localities and the processes within them. *Cap* differs in that it provides the actual change in type that occurs when the movement takes place. Its behaviour is akin to function composition.

Within the type system, mobility primitives are given function types, to represent the fact that they may cause a transition from one group to another[6]. The rule itself simply states that if $\mathcal{M}$ has a function type, transforming processes of the group g_1 to a process in the group g_2, and P is a process in group g_1, then $\mathcal{M}.P$ is typed as the result of applying $\mathcal{M}$ to P, to give a process in group g_2.

The function types used for this are generated by rules like *LocIn*, *LocOut* and *Open*, which are specific to each mobility primitive. *LocIn* states that if m is a locality of group g_1, then *in* m is typeable as $Proc(g_2) \to Proc(g_2)$ if the group g_1 is one of the members of the set of crossable localities maintained by G_2, the type of the group, g_2, used by the process emitting the capability.

The other two rules run along the same lines. *LocOut* is nearly the same, except that the set of locality groups where members of g_1 can stay must be a subset of the set in which members of g_2 can stay. This is because the moving locality, in g_1, must be able to stay in the locality in which m (a member of g_2) is currently situated, when it moves outside it. Such a restriction is unnecessary for *LocIn* as it is implied by the fact that $\mathscr{C}(G) \subseteq \mathscr{S}(G)$, as mentioned earlier.

Finally, the rule for *Open* states that g_1, the group of the locality being

[6] This is not evident in the rules given, as the processes move as part of the locality, and thus stay in the same group. Such changes occur in process movement, where the locality of a process changes, and thus its group.

opened, must be a member of the set of groups that are openable by members of g_2 and that g_2 must be a member of the set of groups that can stay in localities of group g_1. The latter condition is necessary to ensure that the contents of the destroyed locality are allowed to enter the parent locality.

Chapter 4

Future Work

This chapter briefly discusses future work on the calculus, with regard to both the theoretical aspects (sections 4.1 and 4.2) and its applications (section 4.3).

4.1 The Type System

The type system presented in 3.6 is still incomplete. Specifically, as previously mentioned, it lacks rules for handling process mobility and the association between localities and processes.

A possible formalisation of the latter is given in Table 4.1. The locality is a member of group g, while the process is a member of group g'. The two are unified by the requirement that g' be a member of the set of groups that can stay in localities of group g, giving a final typing of $Loc(g')$.

The rule also requires that B is a bouncer process. Absent from the rule is the handling of the set of clocks. Ideally, the restriction should be something along the lines of $\{\Gamma \vdash \sigma : Clock \mid \sigma \in \vec{\sigma}\}$, stating that every element of the vector, $\vec{\sigma}$, is a clock. But it is presently unclear whether this is an appropriate way of handling this requirement.

The task of handling process mobility is more tricky. Objective mobility is the issue, as the process which emits the mobility primitive is not the process

Table 4.1: Types: Linking Processes to Localities

$$\text{Loc} \quad \frac{\Gamma \vdash m : Loc(g), \Gamma \vdash P : Proc(g'), \Gamma \vdash B : BProc, \Gamma \vdash g : G, g' \in \mathscr{S}(G)}{\Gamma \vdash m[P]^{B}_{\{\vec{\sigma}\}} : Loc(g')}$$

which moves and thus its type doesn't change either. Instead, the process which does move is only recognisable by its provision of a corresponding action. This seems to imply that a more complex way of handling actions is required. Another issue is how groups are to be allocated to processes and localities. It may be necessary to add a form of type annotation or a group binder (e.g. (νG); see 2.4.2), as in the type systems based on the ambient calculus.

On a more general note, it would be interesting to consider further the typing of both actions and clocks. At present, they just have the simplest of types, but this could be extended to provide more interesting behaviour. Similarly, bouncers only have rudimentary types and may possibly be changed to include more detail.

4.2 Equivalence

The main element lacking in the current version of TNT is some notion of equivalence. An equivalence notion is necessary to be able to compare processes, with the additional benefit of being able to reduce them to a simpler form.

Equivalence falls under two areas. The first is structural congruence (4.2.1), which is present in many distributed calculi such as the π calculus and the ambient calculus. The second area is bisimulation (4.2.2) and specifically, the extension of the temporal observation bisimulation congruence given for CaSE.

4.2.1 Structural Congruence

The semantics of TNT, specifically those in Table 3.1 in section 3.4, may benefit from a notion of structural congruence. A structural congruence relationship allows the rearrangement of the structure of a process by deeming the two to be equivalent. A clear example of where this would be advantageous with the current semantics is the rules for summation given in Table 4.2.

The only difference between the two rules is the side of the summation on which the α transition occurs. A structural congruence rule along the lines of:

$$E + F \equiv E + F \tag{4.1}$$

would remove the need for one of these rules and make clear the commutative nature of the operator. The same is true of the parallel composition operator,

Table 4.2: Semantics: Summation

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

Table 4.3: Semantics: Structural Congruence

$$\text{SCong}\frac{E \equiv E', E' \xrightarrow{\gamma} F', F' \equiv F}{E \xrightarrow{\gamma} F}$$

which again has two rules, $Par1$ and $Par2$.

Another common role for structural congruence is to remove unneeded elements. In many calculi, **0** is removed in this way. For example, the ambient calculus has a structural congruence rule of the form

$$E \mid \mathbf{0} \equiv E \tag{4.2}$$

which removes the superfluous **0**. However, as was discussed in 3.3, this is more problematic in CaSE and TNT, where **0** is not without behaviour (or, more specifically, transitions). Such a rule would however be very useful, as many of the examples in the previous chapter run in to problems with superfluous **0** processes appearing.

Any structural congruence relationship defined needs to integrate with the existing semantics to be useful. Table 4.3 gives an additional rule which shows how the structural congruence interacts with transitions. If a process E' can perform any action, γ, and become F', while both E' and F' are structurally congruent to E and F respectively, then the same transition may occur from E to F.

4.2.2 Bisimulation

Any bisimulation theory for TNT will 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. With respect to the changes involved in TNT, the additional transitions provided by the mobility

primitives are all silent actions, which are discarded by an observation-based congruence. As a result, it is unclear what changes will actually be necessary to provide the same behavioural theory for TNT as is defined by this congruence for CaSE.

Mobility affects the topology of the process, and thus the difference between two processes which exhibit mobility is much clearer via a structural comparison. Hence, structural congruence may end up being more important with respect to the addition of mobility.

4.3 Applications

The primary application of TNT will be biological modelling. This is an area where physical location is important and mobility is realised as shifts in topological composition. Nested structures are important, being clearly present in, for example, cell membranes. Within the literature, models of a similar style have already been applied, including Cardelli's brane calculi [10] and the membrane computing of P systems (see section 2.3.2). This will also be an interesting area to consider in terms of diversity, providing a pleasant contrast to the more theoretical notions implicit in designing the calculus itself.

One particular case study that is already being considered is that of quorum sensing (see section 2.5). In brief, this focuses on the reactive behaviour of bacteria in relation to the current level of concentration of a particular gene. The multi-party synchronisation implicit in TNT is likely to be useful in this area.

In modelling this, it is expected that additions will be made to the calculus to allow the use of probabilistic derivation, akin to the use of the stochastic π calculus and the Gillespie algorithm in similar studies (see section 2.5). In particular, this will hopefully allow simulations to be run and the behaviour of the model to be analysed.

In the longer term, probably outside the scope of this thesis, there may be other domains in which TNT will find applicability. Pervasive computing is thought to be one such area, and this may make more effective use of the type system than biological modelling. In addition, its relation to P systems (2.3.2) and the ability to encode it using a bigraph (2.3.2) would form interesting areas of study.

Bibliography

[1] Aceto, L., and Murphy, D. Timing and causality in process algebra. *Acta Informatica 33*, 4 (June 1996), 317–350.

[2] Amadio, R. An asynchronous model of locality, failure and process mobility. In *Proceedings of the Second International Conference on Coordination Languages and Models (COORDINATION '97)* (September 1997), no. 1282 in Lecture Notes in Computer Science, Springer-Verlag, pp. 374–391.

[3] Andersen, H. R., and Mendler, M. An asynchronous process algebra with multiple clocks. In *Proceedings of the 5th European Symposium on Programming (ESOP '94)* (April 1994), no. 788 in Lecture Notes in Computer Science, Springer-Verlag, pp. 58–73.

[4] Beaten, J. C. M., and Middelburg, C. A. Process algebra with timing: Real time and discrete time. In *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds. North-Holland, London; Amsterdam, 2001, ch. 10, pp. 627–684.

[5] Bergstra, J. A., and Klop, J. W. Process algebra for synchronous communication. *Information and Control 60* (1984), 109–137.

[6] Berry, G., and Boudol, G. The chemical abstract machine. *Theoretical Computer Science 96* (1992), 217–248.

[7] Boudol, G. Asynchrony and the π-calculus (note). Tech. Rep. RR-1702, INRIA-Sophia Antipolis, 1992.

[8] Boudol, G., Castellani, I., Hennessy, M., and Kiehn, A. Observing localities. *Theoretical Computer Science 114* (1993), 31–61.

[9] Bugliesli, M., Castagna, G., and Crafa, S. Boxed ambients. In *Theoretical Aspects of Computer Software: 4th International Symposium (TACS '01)* (2001), N. Kobayashi and B.C.Pierce, Eds., vol. 2215 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 38–63.

[10] Cardelli, L. Brane calculi - interactions of biological membranes. In *Proceedings of the International Conference on Computational Methods in Systems Biology (CMSB 2004)* (May 2004), V. V. Danos, Ed., no. 3082 in Lecture Notes in Computer Science, Springer-Verlag, pp. 257–280.

[11] Cardelli, L., Ghelli, G., and Gordon, A. D. Mobility types for mobile ambients. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP '99)* (1999), no. 1644 in Lecture Notes in Computer Science, Springer-Verlag, pp. 230–239.

[12] Cardelli, L., Ghelli, G., and Gordon, A. D. Types for the ambient calculus. *Information and Computation 177* (2002), 160–194.

[13] Cardelli, L., and Gordon, A. D. Mobile ambients. In *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98)* (1998), vol. 1378 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 140–155.

[14] Cardelli, L., and Gordon, A. D. Types for mobile ambients. In *Proceedings of the 26th. Annual Symposium on Principles of Programming Languages (POPL '99)* (January 1999), ACM Press, pp. 79–92.

[15] Carriero, N., and Gelernter, D. Linda in context. *Communications of the ACM 32*, 4 (April 1989), 444–458.

[16] Cleaveland, R., Lüttgen, G., and Mendler, M. An algebraic theory of multiple clocks. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR '97)* (1997), no. 1243 in Lecture Notes in Computer Science, Springer-Verlag, pp. 166–180.

[17] Coppo, M., Dezani-Ciancaglini, M., Giovannetti, E., and Salvo, I. m^3: Mobility types for mobile processes in mobile ambients. In *Proceedings of Computing: the Australasian Theory Symposium (CATS '03)* (April 2003), vol. 78 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 1–34.

[18] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76* (1988), 95–120.

[19] Dijkstra, E. W. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 2 (June 1971), 115–138.

[20] Fournet, C., and Gonthier, G. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd. Annual Symposium on Principles of Programming Languages (POPL '96)* (1996), pp. 372–385.

[21] Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and D.Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)* (1996), no. 1119 in Lecture Notes in Computer Science, Springer-Verlag, pp. 406–421.

[22] Gillespie, D. T. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry 81*, 25, 2340–2361.

[23] Gordon, A. D., and Cardelli, L. Mobility types for mobile ambients. Tech. Rep. MSR-TR-99-32, Microsoft Research, June 1999.

[24] Hennessy, M., and Regan, T. A process algebra for timed systems. *Information and Computation 117*, 2 (March 1995), 221–239.

[25] Hewitt, C., Bishop, P., and Steiger, R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (1973), pp. 235–245.

[26] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (August 1978), 666–677.

[27] Honda, K., and Tokoro, M. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)* (1991), M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, Eds., no. 512 in Lecture Notes in Computer Science, Springer-Verlag, pp. 133–147.

[28] Hughes, A. Nomadic time (extended abstract). In *Proceedings of the PhD Programme at Relational Methods in Computer Science/Applications of Kleene Algebra (RelMiCS/AKA) 2006* (2006), R. Schmidt and G. Struth, Eds., no. CS-06-09 in University of Sheffield Technical Reports, pp. 60–64.

[29] Jensen, O. H., and Milner, R. Bigraphs and mobile processes (revised). Tech. Rep. UCAM-CL-TR-580, University of Cambridge, Computer Laboratory, February 2004.

[30] Lee, I., Philippou, A., and Sogolsky, O. A family of resource-bound real-time process algebras. In *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond* (2005), vol. NS-05-03 of *BRICS Notes*, pp. 151–154.

[31] Lee, J. Y., and Zic, J. On modeling real-time mobile processes. In *Proceedings of the 25th Australasian Conference on Computer Science* (2002), vol. 17 of *Conferences in Research and Practice in Information Technology Series*, pp. 139–147.

[32] Levi, F., and Sangiorgi, D. Controlling interference in ambients. In *Proceedings of the 27th. Annual Symposium on Principles of Programming Languages (POPL '00)* (2000), ACM Press, pp. 352–364.

[33] Levi, F., and Sangiorgi, D. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems (TOPLAS) 25*, 1 (January 2003), 1–69.

[34] Lévy, J.-J. Some results in the join-calculus. In *Proceedings of the Third International Symposium on the Theoretical Aspects of Computer Software (TACS '97)*, no. 1281 in Lecture Notes in Computer Science, Springer-Verlag.

[35] Mazurkiewicz, A. Concurrent program schemes and their interpretations. Tech. Rep. AIM PB-78, Computer Science Department, University of Aarhus, 1977.

[36] Merro, M. *Locality in the π-calculus and applications to object-oriented languages.* PhD thesis, Ecoles des Mines de Paris, 2001.

[37] Merro, M., and Sassone, V. Typing and subtyping mobility in boxed ambients. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)* (2002), L. Brim, P. Janar, M. Ketinsky, and A. Kuera, Eds., no. 2421 in Lecture Notes in Computer Science, Springer-Verlag, pp. 304–320.

[38] Milner, R. *Communication and Concurrency.* Prentice-Hall, London; New York, 1989.

[39] Milner, R. A complete axiomatisation for observation congruence of finite-state behaviours. *Information and Computation 81*, 2 (1989), 227–247.

[40] Milner, R. Functions as processes. *Mathematical Structures in Computer Science 2*, 2 (1992), 119–141.

[41] Milner, R. Elements of interaction: Turing award lecture. *Communications of the ACM 36*, 1 (January 1993), 78–89.

[42] Milner, R. The polyadic π-calculus: a tutorial. In *Proceedings of the NATO Advanced Study Institute on Logic and Algebra of Specification* (July 1993), F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds., Springer-Verlag, pp. 203–246.

[43] Milner, R. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, 1999.

[44] Milner, R. Pervasive process calculus. In *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond* (2005), vol. NS-05-03 of *BRICS Notes*, pp. 180–183.

[45] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, parts I and II. Tech. Rep. ECS-LFCS-89-86, University of Edinburgh, June 1989.

[46] Moller, F., and Tofts, C. A temporal calculus of communicating systems. Tech. Rep. ECS-LFCS-89-104, University of Edinburgh, December 1989.

[47] Norton, B. Behavioural types for synchronous software composition. In *Proceedings of the Workshop on Foundations of Interface Technologies (FIT '05)* (August 2005), Electronic Notes in Theoretical Computer Science (ENTCS) (to appear), Elsevier.

[48] Norton, B., and Fairtlough, M. Reactive types for dataflow-oriented software architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA '04)* (2004), vol. P2172, IEEE Computer Society Press, pp. 211–220.

[49] Norton, B., Foster, S., and Hughes, A. A compositional operational semantics for OWL-S. In *Proceedings of the 2nd International Workshop on Web Services and Formal Methods (WS-FM 2005)* (September 2005), no. 3670 in Lecture Notes in Computer Science, Springer-Verlag, pp. 303–317.

[50] Norton, B., Lüttgen, G., and Mendler, M. A compositional semantic theory for synchronous component-based design. In *Proceedings*

of the 14th International Conference on Concurrency Theory (CONCUR '03) (2003), no. 2761 in Lecture Notes in Computer Science, Springer-Verlag, pp. 461–476.

[51] Paun, G. Computing with membranes. Tech. Rep. 208, Institute of Mathematics of the Romanian Academy, November 1998.

[52] Pérez-Jiménez, M. J., and Romero-Campero, F. J. P systems, a new computational modelling tool for systems biology. *Transactions on Computational Systems Biology VI* (2006), 176–197.

[53] Petri, C. A. *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, 1962.

[54] Pierce, B. C., and Sangiorgi, D. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science 6*, 5 (1996), 409–454.

[55] Regev, A., Panina, E. M., Silverman, W., Cardelli, L., and Shapiro, E. Bioambients: An abstraction for biological compartments. *Theoretical Computer Science 325*, 1 (September 2004), 141–167.

[56] Regev, A., Silverman, W., and Shapiro, E. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Proceedings of the Pacific Symposium on Biocomputing* (2001), R. B. Altman, A. K. Dunker, and T. E. Klein, Eds., vol. 6, pp. 459–470.

[57] Riely, J., and Hennessy, M. A typed language for distributed mobile processes. In *Proceedings of the 25th. Annual Symposium on Principles of Programming Languages (POPL '98)* (January 1998), ACM Press, pp. 378–390.

[58] Sangiorgi, D. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, The University of Edinburgh, 1993.

[59] Sangiorgi, D. The name discipline of uniform receptiveness. *Theoretical Computer Science 221*, 2 (June 1999), 457–493.

[60] Sangiorgi, D. Asynchronous process calculi: the first- and higher-order paradigms. *Theoretical Computer Science 253*, 2 (February 2001), 311–350.

[61] SANGIORGI, D. Types, or: Where's the difference between CCS and π? In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)* (2002), no. 2421 in Lecture Notes in Computer Science, Springer-Verlag, pp. 76–97.

[62] SATOH, I. *Time and Asynchrony in Distributed Computing*. PhD thesis, Keio University, January 1996.

[63] SATOH, I., AND TOKORO, M. A timed calculus for distributed objects with clocks. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)* (July 1993), no. 707 in Lecture Notes in Computer Science, Springer-Verlag, pp. 326–345.

[64] STEFANI, J.-B. A calculus of kells. In *Proceedings of the 2nd European Association for Theoretical Computer Science International Workshop on Foundations of Global Computing (FGC '03)* (2003), vol. 85 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 40–60.

[65] TELLER, D., ZIMMER, P., AND HIRSCHKOFF, D. Using ambients to control resources. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)* (2002), L. Brim, P. Janar, M. Ketinsky, and A. Kuera, Eds., no. 2421 in Lecture Notes in Computer Science, Springer-Verlag, pp. 288–303.

[66] THOMSEN, B. A calculus of higher order communicating systems. In *Proceedings of the 16th. Annual Symposium on Principles of Programming Languages (POPL '89)* (January 1989), ACM Press, pp. 143–154.

[67] TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society* (1936), vol. 2, pp. 230–265.

[68] VITEK, J., AND CASTAGNA, G. Seal: A framework for secure mobile computations. In *Proceedings of the ICCL '98 Workshop on Internet Programming Languages* (1999), vol. 1686 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 47–77.

[69] WOJCIECHOWSKI, P. T. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, The University of Cambridge, March 2000.