A CALCULUS FOR DISTRIBUTED MOBILE COMPUTING WITH STATIC RESOURCE ACCESS AND USAGE CONTROL

By

Mayuresh Kulkarni

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Approved:

David Musser, Thesis Adviser

Carlos Varela, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Overview

We propose a process algebraic model of a distributed system with resource access and usage control. Our system has notions of explicitly located mobile agents, migration of agents between locations, location-transparent asynchronous directed communication (local and remote), local resources within locations and local resource access using synchronous communication. Thus, agents can communicate with local and remote agents but to access a resource, they have to migrate to the appropriate location and then establish a dedicated channel to communicate with the resource. We statically guarantee bounded usage of resources. The bound on the usage of a resource may be linearly distributed amongst other agents (including newly spawned agents). Our system borrows certain concepts from the π-calculus [34], the actor model [1] and mobile ambients [9]. The type system to enforce resource access and bounded resource usage is inspired by ideas for resource access control found in the $D\pi$ calculus [41] and linear type systems (c.f. [25]).

1.2 Motivation

As will be seen in chapter 2, there are already many foundational calculi to model distributed systems. A natural question that arises is "why one more?". In reply, we first note that we do not view our system as a foundational calculus. It has instances of mutually redundant concepts. The usual thrust of foundational calculi is to pick a minimalist set of concepts that are sufficiently powerful to model mobile computation and then study the semantic theory induced by these calculi. A common mechanism to prove the *foundational* nature of these calculi is to show that encoding other foundational calculi is possible. In these calculi, modeling interesting systems is possible in principle, but extremely inconvenient in practice. On the other hand we believe that we have adapted many of the *good* concepts found in several such calculi and created a calculus where non-trivial distributed systems can be

conveniently modeled. In spite of this being the case, we do not view our work as research into language design or distributed systems implementation issues. We note that languages like Pict [39], Nomadic Pict [49], SALSA [48] and LLinda [35] focus on such issues.

Though there are many calculi to model distributed systems, there is none that focuses directly on enforcing a bound on resource usage. The notion of bounded resource usage is useful to model an increasingly common scenario where a small business out-sources its IT infrastructure needs to a host company and the contract specifies not just which resources can be accessed at the host site, but also how many times they can be accessed per month. This can be (and most of the times is) done by implementing suitable run time checks. However, it is desirable to do away with these checks by statically guaranteeing certain constraints on resource access and usage. The concept of linear types can be used for this purpose. Linear types are well known in the π-calculus community. However, their main use in the π-calculus seems to be to obtain more fine grained equivalences by refining the contexts in which a process can be placed (c.f. [25]). Though statically bounded resource usage is a key feature of our calculus, we still do not view this as the main thrust of this work.

We mostly view this work as a starting point to investigate the applicability of formal tools like high level specification languages, rewriting logic systems, proof assistants, theorem provers and model checkers to help in designing interesting distributed systems and to deduce and prove interesting semantic properties of such systems.

1.3 Outline

In chapter 2, we survey the most influential foundational calculi for mobile computing. In chapter 3 we propose A^{π} — a calculus for distributed mobile computing with static resource access and usage control. In chapter 4, we conclude by summarizing and highlighting future work.

CHAPTER 2
FOUNDATIONAL CALCULI FOR MOBILE PROCESSES

2.1 Introduction

In this chapter we provide a survey of foundational calculi for concurrency and mobility, and the role of typing therein.

The λ-calculus is a well known model of a sequential programming language. It occupies an enviable position because it captures all the essential features of functional computation. Moreover, other models such as Turing machines are shown to be equivalent to it. Unfortunately, the world of concurrent computation is not so systematic. As noted by Pierce [36], the deep difference between the two cases stems from *what may be observed*. The only way to observe a functional computation is to watch the output values it yields when presented with different inputs. However, in a concurrent world, we may wish to observe a system of concurrently interacting processes at different levels of granularity — like observing or ignoring internal transitions, inherent parallelism, distribution of processes amongst physical processors, fault tolerance, locations, etc. Thus, a large body of work has arisen to propose different process calculi in the concurrent world, each one focusing on a particular subset of the observable features. Just as in the λ-calculus where everything is a function and function application is the heart of the reduction mechanism, in the concurrent world, everything is a *process* and *process interaction* is at the heart of the reduction mechanism. A process is a free standing computation that can interact with other processes according to interaction rules specified by that particular process calculus.

Here, we focus on three major models for concurrency and mobility — the π-calculus, mobile ambients and the actor model. We shall also look at some typical properties that are guaranteed by static type systems in these calculi. The calculus A^{π} that we propose in the next chapter borrows certain concepts from all of these models.

2.2 The π-calculus

The π-calculus of Milner et al. [34] grew out of an earlier formalism known as the Calculus of Communicating Systems (CCS) by Milner [30], which in turn was developed from the still earlier Communicating Sequential Processes (CSP) of Hoare [23]. The π-calculus is often considered to be the canonical model for concurrent computation. It is very simple, and yet, it can encode many of the common notions found in a concurrent setting.

The π-calculus is a name passing calculus in which *processes* communicate over shared *channels*. Channel names can be communicated between processes. In fact, names are the only entities that can be communicated. Value passing can be encoded in this name passing calculus (roughly in the same way that values are encoded in the λ-calculus). The scope of names can be restricted (and this is the key difference from CCS). Restriction is needed to write almost any interesting program, and, amongst other things, is indispensable to encode value passing. The λ-calculus has one combinator, function application, often denoted by $(M\ N)$. This combinator is neither commutative nor associative, as constrained by the sequential nature of the computation it models. The π-calculus also has one combinator — process composition (denoted by $P \mid Q$). However, in stark contrast with the function application, process composition is both commutative and associative. This is the deep difference between the two cases and allows us to model many of the common notions found in the concurrent world.[1] Process mobility is modeled indirectly through name restriction and name communication. To see how, we first note that there are two ways of thinking about mobility. The more intuitive way is to think of processes actually moving in physical space, and the less intuitive way is to think about the acquaintance list of a process changing dynamically. To cite an example (taken from [31]), consider a person talking on a cell phone. The cell phone is communicating with the tower in that particular cell. Now suppose the person walks out of her current cell and into the neighboring cell, so that the cell phone is communicating with the tower in the new cell. Now, if we want to think along the

[1] As noted by Milner in his Turing award lecture [32], *composition* can be thought of as a unifying concept for both the concurrent world and the sequential world. In the sequential world, the only way that processes (instructions) are allowed to be composed is in a temporal sequence.

first, more intuitive line, then we need to model the person, the tower, the notion of a cell and the notion of the person being in a particular cell. However, if we think along the second, less intuitive line, then we need to model the person, the tower and the notion of the current acquaintances of the person (which changes as the computation evolves). The π-calculus takes the second approach (and as discussed later, mobile ambients take the first approach). The reason for picking the second approach is in keeping in line with the design goals of the π-calculus: to avoid selecting anything more than a minimally necessary set of constructs to model the key notions in a concurrent world.

Note that scope restriction and extrusion are indispensable to model mobility. Scope extrusion happens when a restricted name is communicated to another process. The *scope* of this name is also transferred along with the name. This makes it convenient to model exchange of private resources. A restricted name which was initially known to only one process is now known to (only) those processes to whom it has been communicated and is thus a shared resource amongst these processes.

2.2.1 Syntax and Semantics of the π-calculus

The syntax and term reduction of the π-calculus is presented in Figure 2.1 (taken from [36]). As can be seen, the calculus provides constructs for synchronous input and output. Synchronicity is implied from the first rule in the operational semantics — a process reading from a channel (or writing to a channel) blocks until another process writes to (or reads from) the same channel.[2] Besides these, there are constructs for process composition and scope restriction. The replication construct, though standard, is not strictly needed and is usually included only for convenience.

We now provide a detailed description of the syntax and semantics of the π-calculus.

0 is the nil process. It does nothing. $x(y).P$ is a process that is waiting for input on channel x. After receiving an input along x, it continues as P, after the appropriate lexical substitution of y with the received value. $\overline{x}y.P$ is a process that outputs y on channel x and then continues as P. $P|Q$ denote a parallel composition

[2]For asynchronous versions of the π-calculus, we refer the reader to Honda et al. [24] and Boudol et al. [8]. The basic idea is to allow only the nil **0** process as the continuation of an output prefix.

Syntax:

$$
\begin{array}{rcll}
P, Q, R & ::= & \mathbf{0} & \text{inert process} \\
 & & x(y).P & \text{input prefix} \\
 & & \overline{x}y.P & \text{output prefix} \\
 & & P|Q & \text{parallel composition} \\
 & & (\nu x)P & \text{restriction} \\
 & & !P & \text{replication}
\end{array}
$$

Renaming of bound variables:

$$
\begin{array}{rcll}
x(y).P & = & x(z).([z/y]P) & \text{if } z \notin FV(P) \\
(vy)P & = & (vz)([z/y]P) & \text{if } z \notin FV(P)
\end{array}
$$

Structural Congruence:

$$
\begin{array}{rcll}
P|Q & \equiv & Q|P & \text{commutativity of parallel composition} \\
(P|Q)|R & \equiv & P|(Q|R) & \text{associativity of parallel composition} \\
((vx)P)|Q & \equiv & (vx)(P|Q) \quad \text{if } x \notin FV(Q) & \text{scope extrusion} \\
!P & \equiv & P|!P & \text{replication}
\end{array}
$$

Operational Semantics:

$$
\begin{array}{rcll}
\overline{x}y.P|x(z).Q & \rightarrow & P|[y/z]Q & \text{communication} \\
P|R & \rightarrow & Q|R \quad \text{if } P \rightarrow Q & \text{reduction under } | \\
(\nu x)P & \rightarrow & (\nu x)Q \quad \text{if } P \rightarrow Q & \text{reduction under } \nu
\end{array}
$$

$$
\frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \quad \text{structural congruence}
$$

Figure 2.1: Syntax and operational semantics of the π-calculus

of two processes. Two processes, when composed together, may interact with each other as governed by the reduction rules described below. $(\nu x)P$ says that x is universally fresh in P. Finally, $!P$ denotes an infinite number of P's composed in parallel. This is a common notational convenience used to program a reactive process, i.e., a process that receives a value, does something and then assumes its initial behavior again.

The structural congruence relation identifies processes that are *lexically* the same, i.e., processes that we can syntactically equate without even looking at their semantics.

The operational semantics is presented as a reduction relation. The rule for

communication represents synchronous communication between two processes. The next two rules say that if P can reduce to Q then this reduction is not affected if P is grouped with another process or if one or more names in P are scoped by a restriction operator. The final rule allows reduction of processes modulo the structural congruence relation. Note that it is this final rule which allows us to reduce expressions like $x(y).P \mid z(y).Q \mid \overline{x}a.R$ which otherwise are not in any of the "standard" forms in the semantics rules.

The π-calculus, more than any other model, has a large body of theoretical work studying both the properties that can be expressed via the calculus and meta level properties of the calculus itself. Equivalence of processes is usually based upon *bisimulation* relations and not upon *testing* theories which are more common in the actor model. Though we shall not be concerned with equivalence in this thesis, it is an interesting topic for future work. Research has also been done in designing and implementing programming languages based on the π-calculus. The Pict language [39] and Nomadic Pict [49] are two such languages. Pict is a language for concurrent programming while Nomadic Pict is a language for distributed programming. In particular, we highlight Nomadic Pict here because in the A^{π} calculus we propose in this thesis, we shall use location-transparent messaging — one of the concepts found in Nomadic Pict.

2.2.2 Typing in the π-calculus

Milner's notion of *sorting* [33] was historically the first attempt to impose a type discipline in the π-calculus. A sorting enforces constraints on the name tuples that can be communicated over a channel. Names are partitioned into a collection of sorts and a sorting function is defined which maps sorts onto sequences of sorts. If a sort s is mapped to the sequence (t_1, t_2), then the channel s can carry only pairs of names, where the first name is in t_1 and the second is in t_2. If x is in sort s, u in t_1 and v in t_2 then prefixes $x(u, v)$ and $\overline{x}\langle u, v\rangle$ respect the sorting. A process is said to be well sorted if all prefixes in it respect the given sorting. The main property guaranteed by this mechanism is that arity mismatches like $x(u, v).P \mid \overline{x}\langle y\rangle.Q$ are guaranteed not to occur during runtime. The main limitation

of sorting is that sort information is completely static: it cannot describe sequencing of values communicated over a channel. For example, it cannot describe a channel which is used to carry an alternating sequence of two and three tuples, or tuples of different sort sequences which have the same length. However, in the actor model, this dynamically changing communication pattern along with asynchronous one-to-one communication, state encapsulation and fairness, is precisely what makes actors useful. An actor name identifies a persistent entity, whose behavior evolves and changes with time. The ability for a name to be used in different ways in different contexts is desirable, and the notion of polymorphic types (c.f. [28] and [38]) provides partial support for this ability. In [36] Pierce et al. introduce I/O types — a refinement of Milner's sorting by imposing further constraints on the usage of names. The ability to read from a channel, the ability to write to a channel and the ability to both read and write are all distinguished from one another. Besides the sorting function, a function I is defined from sorts to a set of tags which specifies the operations allowed on names in a given sort. The tags are: r for reading, w for writing and b for both. The fact that a name that belongs to a sort which is tagged with b can be used in a context which expects a name of a sort that is tagged with only r or w imposes a natural sub-type relationship between sorts. In [25], Kobayashi et al. introduce the notion of linear types. This is a further refinement on I/O types where each sort is tagged with not just the ability to read or write or both, but also to specify a *multiplicity* — which is the number of times that each name can be used. In passing, we note that the basic sorting discipline as well as its various refinements are useful in at least three key areas. The first is to prevent careless programming errors. Second and more importantly, they reduce the number of contexts in which a given process can be correctly placed, thus making it easier to prove that two processes are equivalent. This is desirable when studying behavioral equivalences of processes. The third and probably the most important is that these type disciplines are *necessary* to prove certain encodings of other calculi into the basic π-calculus. Since the π-calculus aims to be the most foundational system to model mobile computation, it is necessary to be able to encode other paradigms into the π-calculus to prove its foundational nature. And to do so, it is

necessary to have, at least, a sorted π-calculus. The canonical example is [43] where a fully abstract translation is presented between the π-calculus and a higher order π-calculus (where processes themselves can be transmitted over channels). We refer the reader to Part III of the *The π-calculus—A theory of mobile processes* [42] for a more detailed treatment of typing in the π-calculus.

2.3 Mobile Ambients

Mobile ambients were first introduced by Cardelli and Gordon in [9]. Our presentation is drawn from the lecture notes in [10] which in turn summarize material from different ambient papers.

The mobile ambients formalism takes a more direct view of mobility than the one in π-calculus. As will be seen in Section 2.2, in the π-calculus, mobility is achieved by communicating and restricting the scope of channel names. It is like a graph in which the edges keep changing as the acquaintances of processes change with the evolution of the computation. Mobile ambients take a world view in which *processes* move in physical space between *locations*. This view is thus more intuitive. Further, mobile ambients model not just mobility of a process from one location to another, they also model mobility of a location from one domain to another—e.g., think of computations that are running in a laptop as the laptop is taken from one building to another. The design goals of mobile ambients are markedly different from those of the π-calculus. When mobile ambients were formulated, the π-calculus had already been more or less accepted as the analogue of the λ-calculus in the concurrent world. However, even for a foundational calculi, the π-calculus did not fairly represent distributed computing over the Internet (worldwide computing) which has become ubiquitous now. This is because locations, mobility and security are fundamental concepts in worldwide computing and these concepts enjoy at best a second class status in the π-calculus, since they exist only as encodings of the name passing mechanism. However, in mobile ambients, these concepts enjoy a first class status since they are present more directly in the calculus.

An *ambient* is a place that is delimited by a boundary and where multiple processes execute. Each ambient has a *name*, a collection of local *processes*, and

a collection of *sub-ambients*. Ambients can move in and out of other ambients, subject to *capabilities* that are associated with ambient names. Ambient names are unforgeable. This is the most basic security feature. Thus, a process may obtain a capability to enter an ambient but from this capability (or by any other means for that matter) it may not deduce the name of an ambient.

2.3.1 Mobile ambients: an informal introduction

In this section we present a pictorial introduction to mobile ambients. The formal syntax and semantics described in subsequent sections are easier to understand after first reading this section.

In general, an ambient is a collection of processes and sub-ambients. The structure of an ambient with name n may be represented as shown in Figure 2.2.

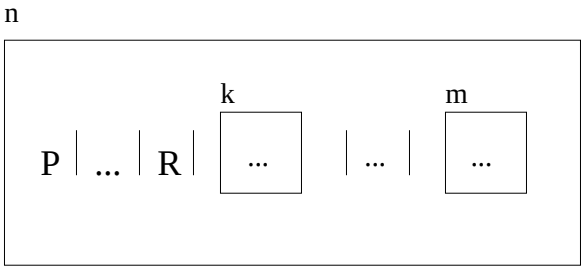

Figure 2.2: Basic ambient structure

Note that nothing prevents the existence of two or more ambients with the same name, either nested or at the same level. Once a name is created , it can be used to name multiple ambients. Moreover, $!n[P]$ generates multiple ambients with the same name. This feature is used to model replication of services.

2.3.1.1 Actions and Capabilities

Operations that change the hierarchical structure of ambients can be interpreted as crossing of firewalls or decoding encrypted data. Such operations are thus sensitive and are restricted by *capabilities*. As we noted earlier, an ambient can allow other ambients to perform certain operations (like entering it or communicating with it) without revealing its true name. Capabilities can be transmitted as messages.

The process $M.P$ executes an action regulated by capability M, and then continues as process P. Thus M is a guard for P. Capabilities are syntactically derived from names. Given an ambient name m, the capability $in\ m$ allows entry into m, the capability $out\ m$ allows exit out of m and the capability $openm$ allows the opening of m. Implicitly, the possession of one or all of these capabilities is insufficient to reconstruct the original name m from which they were extracted.

The following figures explain the three basic capabilities:

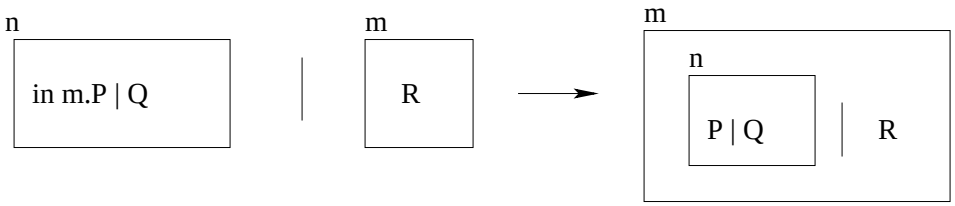

Figure 2.3: Entry Capability

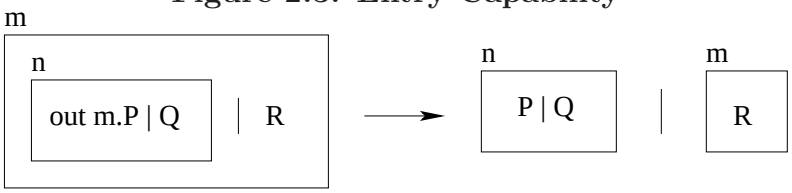

Figure 2.4: Exit Capability

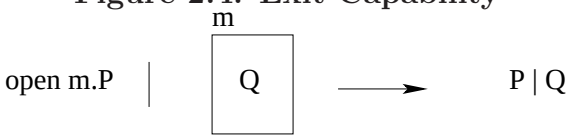

Figure 2.5: Open Capability

We note that while exercising the entry capability — $in\ m.P$ — if there is no ambient named m, then the executing process blocks until such an ambient becomes available. And if there is more than one ambient named m, then one of them is chosen non-deterministically.

The open capability may seem like a bad idea because the process that has a capability to open an ambient has no idea of what it might be unleashing by opening that ambient. However, note that since names cannot be guessed, the only way that a process can obtain a capability to open an ambient is by requesting precisely such a capability. Thus, in some ways, the result is not entirely expected. At the same time, it is desirable for the target ambient to be able to control attempts by other ambients to move into or open the target ambient. *Safe ambients* [26], an

extension to mobile ambients provide such facilities. The key idea in safe ambients is that capabilities always come in pairs: for instance the usual capability $in\ a$ and its co-capability $\overline{in}\ a$. Both the capabilities must be present for that action to be performed. Consider the following reduction:

$$a[\overline{in}\ a.open\ b.in\ c] \mid b[in\ a.\overline{open}\ b.in\ d]$$

The move of b into a is legal because of the presence of the $in\ a$ capability in b and the corresponding $\overline{in}\ a$ capability in a. Similarly the *open* action is legal because of the presence of the *open* capability and the $\overline{open}$ co=capability. This extension makes this calculus more secure than basic mobile ambients.

2.3.1.2 Communication primitives

We first note that the *pure* mobile ambient calculus — which only has the three capabilities outlined above — is actually Turing-complete. However, for convenience, we also include the ability to communicate.

The usual communication mechanism chosen in work involving ambient calculus is *local, undirected* communication. Local means that communication can happen only within an ambient. Undirected means that messages are not directed to a particular process or agent. Thus a process just emits a message into the ether, so to speak, and *any* neighboring process which is waiting for a message can snap up this message. Later on, we describe a type mechanism to impose some structure into this otherwise chaotic communication mechanism.

2.3.1.3 Orthogonality of mobility primitives and communication primitives

With the communication mechanism we just described, if a process wants to communicate with another process in a different ambient, it first has to physically go to the other ambient. To do so it requires the capability to exit its current ambient and another capability to enter the new ambient. The alert reader will note that only these two are not sufficient. In addition, some process in the destination ambient must have the capability to open the source ambient so that the two processes are

finally adjacent to each other and communication may commence.

This highlights an interesting design property of the calculus. The choice of a communication mechanism is orthogonal to the choice of the mobility primitives. Indeed, we can choose one set of mobility primitives and another set of communication primitives at will, and our choices will affect the expressiveness and computational power of the resultant calculus.

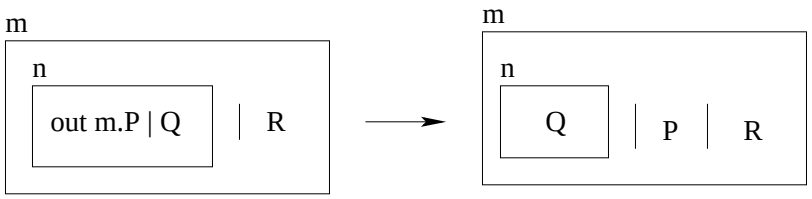

Figure 2.6: A different exit capability

In particular, as seen in Figure 2.6, we can choose to include an additional capability which allows a process to escape from an ambient. Recall that the *exit* capability detailed in Figure 2.4 causes the *surrounding ambient* of the exercising process to exit *its surrounding ambient*.[3] With this additional capability, we may then choose to leave out the *open* capability and still be able to encode the above mentioned scenario.

From this angle A^{π}, the main calculus described in this thesis, can be roughly viewed as a mobile ambient calculus with a more restrictive set of of mobility primitives and a more expressive set of communication primitives (*directed, local and non-local* communication).

2.3.2 Mobile Ambients: Syntax and Semantics

The formal syntax of mobile ambients calculus is presented in Figure 2.7. We now briefly explain the constructs.

- *Restriction, Inaction, Composition and Replication*

  These have the same meaning as in the π-calculus. $(\nu n)P$ creates a new unique

[3]In the literature the initial exit capability we had is often referred to as a *subjective* capability because, from the viewpoint of the ambient it can be read as "I move". The second exit capability is often referred to as an *objective* capability because from the view point of the ambient, it may be read as "I make this process move".

Syntax:

n			names
P, Q	::=		processes
		$(\nu n)P$	restriction
		$\mathbf{0}$	inactivity
		$P \mid Q$	composition
		$!P$	replication
		$M[P]$	ambient
		$M.P$	capability action
		$(x).P$	input action
		$\langle M \rangle$	async output action
M	::=		messages
		x	variable
		n	name
		in M	can enter into M
		out M	can exit out of M
		open M	can open M
		ε	null
		$M.M'$	path

Figure 2.7: Syntax of Mobile Ambients with communication primitives

name n with scope P. The new name can be used to name ambients and to operate on ambients by name. Like the π-calculus, the (νn) binder can float outwards (when possible) to extend the scope of a name and can float inwards (when possible) to restrict the scope of a name.

- *Ambients*

  $M[P]$: If M is the name of an ambient, say n, then we get a meaningful construct: $n[P]$ creates an ambient n executing the process P. However, if M is a capability or a path, then we get a meaningless construct. Such meaningless constructs are permitted in the syntax, but they have no computational reduction rule and so cannot be reduced. A question that arises is why permit them in the first place? This is done to achieve a uniform messaging system where a variable can be used to denote both an ambient name and also a capability. We shall talk about this later in Section 2.3.3.

- *Capability action, input and output*

  $M.P$ first executes the action governed by capability M and then executes

P. The process blocks until the action governed by M can be performed. $(x).P$ inputs a message x and then continues as P (with the appropriate substitutions). Recall that communication is undirected, so a message is not targeted to a particular process. Also, a message may be received by only one process. $\langle M \rangle$ represents an asynchronous output action. Note that here a message is being represented as a process which has no continuation (same as in the asynchronous π-calculus).

The behavior of processes is given by the reduction relation in Figure 2.8. The first three rules are the reduction rules for actions associated with the *in*, *out* and *open* capabilities. The next three rules propagate reductions across scopes, ambient nesting and parallel composition. The next rules allows structural equivalence during reduction and the final rule allows communication to occur.

Structural Congruence:

$$
\begin{array}{l}
P \equiv P \\
P \equiv Q \Rightarrow Q \equiv P \\
P \equiv Q, Q \equiv R \Rightarrow P \equiv R \\
P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q \\
P \equiv Q \Rightarrow P|R \equiv Q|R \\
P \equiv Q \Rightarrow !P \equiv !Q \\
P \equiv Q \Rightarrow n[P] \equiv n[Q] \\
P \equiv Q \Rightarrow M.P \equiv M.Q \\
P|Q \equiv Q|P \\
(P|Q)|R \equiv P|(Q|R) \\
!P \equiv P|!P \\
(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \\
(\nu n)(P|Q) \equiv P|(\nu n)Q \text{ if } n \notin fn(P) \\
(\nu n)(m[P]) \equiv m[(\nu n)P] \text{ if } n \neq m \\
P|\mathbf{0} \equiv P \\
(\nu n)\mathbf{0} \equiv \mathbf{0} \\
!\mathbf{0} \equiv \mathbf{0} \\
P \equiv Q \Rightarrow M[P] \equiv M[Q] \\
P \equiv Q \Rightarrow (x).P \equiv (x).Q \\
\varepsilon.P \equiv P \\
(M.M').P \equiv M.M'.P
\end{array}
$$

Reduction:

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P\|Q] \mid R]$	**Red In**
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P\|Q] \mid m[R]$	**Red Out**
$open\ n.P \mid n[Q] \rightarrow P\|Q$	**Red Open**
$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$	**Red Res**
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	**Red Amb**
$P \rightarrow Q \Rightarrow P\|R \rightarrow Q\|R$	**Red Par**
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	**Red** $\equiv$
$(x).P\|\langle M \rangle \rightarrow P_{M/x}$	**Red Comm**

Figure 2.8: Semantics of Mobile Ambients with communication primitives

2.3.3 Typing in mobile ambients

As seen above, there are two different concepts involved in mobile ambients — mobility and communication. Mobility is governed by run-time capabilities. However, communication can be further refined by imposing a type discipline. Consider the following system: $n[(x : Int).P \mid \mathit{open}\ m] \mid m[\mathit{in}\ n.\langle 3\rangle]$. In essence, the system consists of two processes, one of which is trying to communicate the integer value 3 to the other which is expecting an integer value. The type system must be able to ensure that the system is well typed, in spite of the reading and writing actions being initially located in different ambients. This is done by keeping track of the *topic of conversation* within an ambient. In general, an ambient may have many processes, all of them performing input and output actions. Since output is undirected, it is necessary that all the processes are dealing with the same type of messages. This type is the topic of conversation allowed in that ambient. To ensure that all processes and ambients are well typed with respect to the topic of conversation, the following three entities are controlled:

- The processes must be tagged to indicate what type of messages they are going to exchange. This is reminiscent of the *sorting* discipline in the π-calculus, where a name can be used to communicate only one type of value.

- The ambients must be tagged to indicate the topic of conversation permitted in the ambient.

- The *open* capability must be tagged to indicate what type of ambients it can open. It is not necessary to tag the *in* or *out* capabilities (since unless an ambient is opened, no communication can occur).

Once the above additional tagged information is available, it is possible to devise an inference mechanism that looks at the syntax of the system and determines if it is well typed.

Further note that as shown in Figure 2.7, a variable is used to denote an ambient name and also a capability. This leads to syntactic anomalies as is seen from the expression: $\big((x).x.P\big) \mid \langle n\rangle$ Here, the process wishes to input a capability and then exercise that capability. Instead, it ends up receiving an ambient name and

reduces to the meaningless term: $n.P$. A similar anomaly may arise if a process that wishes to input an ambient name ends up receiving a capability. Such anomalies are also handled in the type system.

We refer the reader to [13], [12] and [11] for further details.

2.4 The Actor model

The actor model was first proposed by Hewitt [21] who developed it further in [22]. It has evolved over the years with a trend towards treating a collection of actors as the basic unit of discourse instead of a single actor. Our presentation is based on ideas found in Agha et al. [2] and Thati [46] and these are themselves based on the system described in Agha [1]. Some alternative formulations can be found in Clinger [15], Gaspari [29] and Talcott [45].

The actor model is based on asynchronous communications between *actors*. Actors are very much like processes — free standing computations, however, actors are usually persistent, reactive entities. A computational system in this model, also known as a *configuration* , consists of a collection of concurrently executing actors and a collection of messages in transit, and has an interface to its external environment. Actors communicate via asynchronous messages. These messages may consist of either values or actor names. A message contains values targeted to a single actor called the target actor. A message is consumed once it is received by the target. The programming style usually adopted in this model is an event based style. Actors are reactive in nature. On receiving a message, an actor can *concurrently* perform some or all of the following actions:

1. Send a finite number of messages to other actors.
2. Spawn a finite number of new actors with universally fresh names and instantiate these actors with behaviors.
3. Take a finite number of internal computation steps.
4. Assume a new behavior with the same name.

The model has certain assumptions as follows:

1. Fairness: It is assumed that the underlying message delivery system provides certain fairness guarantees — roughly that no message delivery will be infinitely postponed. Fairness plays a crucial role in proving certain observational equivalences in the model and is also indispensable while composing configurations. We refer the reader to Agha et al. [2] for details.

2. An actor knows the names of only a finite number of other actors (its *acquaintances*). Further, in one computational step, it can send only a finite number of messages to a finite number of actors.

3. A message can carry a finite number of names.

4. Like names in Mobile Ambients, actor names cannot be guessed. This is the most basic form of security. Thus, an actor may only learn about new actors via messages that it receives. Further, as the computation proceeds, it may forget some names. In the system that we are going to describe, this is syntactically represented via (static) name scoping. A name is known to the actor only within its lexical scope.

As compared to other foundational calculi, the actor model has much better support for *encapsulation*. Each configuration has an interface to its environment. The interface determines how this configuration can interact and compose with other configurations. The interface changes dynamically as the computation evolves and as actors gain new acquaintances and forget old ones. The interface consists of a set of actors known as *receptionists* and another set known as *external actors*. Some actors within a configuration are designated as *receptionists* while the known actors belonging to other configurations are designated as *external actors*. Only the receptionists can receive messages from outside the configuration and actors within the configuration can send messages either to other actors within the configuration or to the external actors. As we noted above, current research in actor semantics abstracts away from the internal events of a configuration and concentrates only on how configurations evolve and interact with each other. We refer the reader to [45] and [46] for details. An attempt to model locations and security in the actor model is found in Toll et al. [47] where a global resource map in conjunction with location

specific resource maps are used to determine the resource that an actor is trying to access. The local resource map at a location is first consulted to resolve the target resource. If there is no entry in the local resource map, then the global resource map is consulted. Thus, each location may override the default global behaviour. This is useful, for instance, to map *stdout* to different resources in different locations.

The actor model predates the π-calculus and has significantly influenced research in programming abstractions for open distributed systems, concurrent object oriented languages and projects on high performance computing. In many ways, the model was ahead of its time. The increasing availability of ubiquitous distributed computing power provides the right infrastructure for the actor model.

Syntax:

$$
\begin{array}{rcl}
C & := & \mathbf{0} \\
 & | & x(\tilde{y}).C \\
 & | & \overline{x}\langle\tilde{y}\rangle \\
 & | & (\nu x)(C) \\
 & | & [x = y](C_1, C_2) \\
 & | & C_1 | C_2 \\
 & | & B\langle x, \tilde{y}\rangle
\end{array}
$$

Figure 2.9: Syntax of System A

2.4.1 System A: An actor algebra

As we noted above, the system described by Agha et al. [2] is the one often used while describing the actor model. They take a simple functional language based on the call by value lambda calculus and extend it with additional constructs to capture the actor model. However, here, we shall describe *System A*, an actor algebra proposed by Thati [46], since we believe that the style of this algebra is very similar to the other calculi described in this chapter.

The syntax of System A is presented in Figure 2.9. For those familiar with the more common presentation of the Actor system [2], we note that in System A, the concept of configurations is carried to its extreme. Each actor is viewed as a configuration (or equivalently as a configuration that has a single actor, which is also a receptionist).

Let $\mathcal{N}$ be an infinite set of names; $\mathcal{B}$ an infinite set of behavior identifiers; u, v, w, x, y, z range over $\mathcal{N}$; A, B range over $\mathcal{B}$; and $\tilde{x}$ and $\tilde{y}$ denote tuples of names. The set of *preterms* $\mathcal{C}$ is defined by the context free grammar in Figure 2.9. C ranges over $\mathcal{C}$.

Here is a description of each preterm:

1. *Nil Process*, 0: This represents a configuration with no actors or messages.

2. *Output*, $\overline{x}\langle\tilde{y}\rangle$: This represents a configuration containing a single message. The message is targeted to an external actor x and contains the tuple of names $\tilde{y}$.

3. *Input*, $x(\tilde{y}).C$: This represents a configuration containing a single actor named x. The actor is also a receptionist and may receive messages from the environment. All names in $\tilde{y}$ are distinct and are bound by the input $x(\tilde{y})$. The actor receives a message containing a tuple $\tilde{z}$ of the same arity as $\tilde{y}$, and replaces itself with the configuration $C_{\tilde{z}/\tilde{y}}$. The configuration $C_{\tilde{z}/\tilde{y}}$ should in turn contain the actor x with a new behavior (this is guaranteed by System A's type system), and possibly some freshly created actors and messages.

4. *Composition*, $C_1 \mid C_2$: This represents a composition of two actor configurations. For those familiar with Agha et al. [2], we note that unlike that work, in System A it is assumed that in a configuration, all actors except the receptionists have universally fresh names. Normally, two actor configurations can be composed if there is no actor with the same name in both the configurations. In System A, this condition is equivalent to saying that there is no receptionist with the same name in both configurations.

5. *If-then-else*, $[x = y](C_1, C_2)$: This preterm is C_1 if x and y are the same names. Otherwise it is C_2.

6. *Restriction*, $(\nu x)C$: This represents the configuration C, but with actor x no longer a receptionist. The name x is private to the configuration. It is bound by restriction (νx).

7. *Behavior Instantiation*, $B\langle x, \tilde{y}\rangle$. The behavior identifier B represents a template for an actor behavior and this template is instantiated with the appropriate values.

NIL:
$$\frac{}{0 : [\phi, \phi]}$$

MSG:
$$\frac{}{\overline{x}\langle\tilde{y}\rangle : [\phi, \{x, \tilde{y}\}]}$$

ACT:
$$\frac{C : [\{x\}, \chi]}{x(\tilde{y}).C : [\{x\}, fn(C) - \{x, \tilde{y}\}]} \quad x \notin \{\tilde{y}\}$$

RES:
$$\frac{C : [\rho \cup \{x\}, \chi]}{(\nu x)C : [\rho, \chi]} \quad x \notin \rho$$

COMP:
$$\frac{C_1 : [\rho_1, \chi_1] \;\; C_2 : [\rho_2, \chi_2]}{C_1 | C_2 : [\rho_1 \cup \rho_2, (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)]} \quad \rho_1 \cap \rho_2 = \phi$$

COND:
$$\frac{C_1 : [\rho, \chi_1] \;\; C_2 : [\rho, \chi_2]}{[x = y](C_1, C_2) : [\rho, \chi]}$$

BINST:
$$\frac{}{B\langle x, \tilde{y}\rangle : [\{x\}, \{\tilde{y}\} - \{x\}]}$$

Figure 2.10: Type rules for System A

2.4.2 Type system

We note that not all preterms are valid actor configurations. To see why, consider the following illustrative examples:

1. The preterm $x(y).C_1 \mid x(y).C_2$ contains two actors with name x and hence violates uniqueness of actor names.

2. The preterm $x(y).0$ violates the persistence of actors since the actor x disappears after receiving a message.

3. The preterm $x(y).(x(z).C_1 \mid y(z).C_2)$ violates anonymity of newly created actors since an actor is created with a name received as an input message. New actors must be created with fresh names.

4. The preterm $x(y).(\nu z)(x(y).C \mid \overline{u}\langle z\rangle)$ where $z \notin fn(C)$ is not a valid configuration because a new name z is generated without associating it with a behaviour. Recall that in System A, all names are names of actors and must hence be associated with a behavior.

In System A, the type system plays a central role since it is used filter out all the preterms that are not valid actor configurations. The preterms that are well typed are known as *terms*.

The basic type judgment is of the form $C : [\rho, \chi]$ which means that the configuration C is a valid actor configuration with receptionists ρ and external actors χ. The type rules are presented in Figure 2.10. The type system is implicit. The rules are self explanatory for the most part. Of particular interest are the rules ACT and $COMP$ and we shall only explain these here.

The ACT rule concerns a single actor. It says that when an actor receives a message, it can replace itself with a configuration in which it is the only receptionist. This in turn implies that neither can an actor simply disappear, nor can it create any actors with well known names. It can only create anonymous actors. Further, all the free names that it learns about from the message it receives are added to its external actor set.

The $COMP$ rule concerns the composability of two actor configurations. Consider the two actor configurations C_1 and C_2 with interfaces $[\rho_1, \chi_1]$ and $[\rho_2, \chi_2]$. According to the rules of the general actor model, C_1 and C_2 are composable if and only if:

1. No two actors in the configurations have the same name.

2. The configurations respect each other's interface:

$$\begin{aligned} \chi_1 \cap dom(C_2) &\subset \rho_2 \\ \chi_2 \cap dom(C_1) &\subset \rho_1 \end{aligned}$$

These rules reflect the condition that for any configuration, the names of actors that are not receptionists are unknown to the environment.

If the above two conditions are satisfied, then, the configuration C obtained by composing C_1 and C_2 contains all the actors and messages in the two configurations and its interface $[\rho, \chi]$ is given by:

$$\begin{aligned} \rho &= \rho_1 \cup \rho_2 \\ \chi &= (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2) \end{aligned}$$

Now, in system A, we note that any newly created name is assumed to be universally fresh. Further, as per the type rules, any name that is not within the scope of a ν operator is a receptionist for that configuration. Thus, in system A, the two preconditions for composability reduce to the single condition: $\rho_1 \cap \rho_2 = \phi$.

In passing we highlight an important consequence of treating the receptionists and external actors of a configuration as the type of that configuration. The type is computed based solely on the syntactic information present in a term. Thus, the ρ and χ sets reflect the *current* interface of the configuration. The past history is not reflected in the current interface. This is in contrast with the situation in Agha et al. [2] where the interface of an configuration increases monotonically; i.e., a receptionist will always remain a receptionist and a known external name will always be known.

2.4.3 Semantics of System A

The semantics of system A is given as a labeled transition system. A transition is of the form $C_1 \xrightarrow{\alpha} C_2$ and means that C_1 can perform the action α and evolve into C_2. The actions that can be performed are presented in Table 2.1.

Action(α)	Kind	$fn(\alpha)$	$bn(\alpha)$
τ	Silent	ϕ	ϕ
in($\overline{x}\langle\tilde{y}\rangle$)	Input	$x, \tilde{y}$	ϕ
out($(\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle$)	Output	$\{x, \tilde{y}\} - \{\tilde{y'}\}$	$\{\tilde{y'}\}$

Table 2.1: Action labels

The transition $C_1 \xrightarrow{\tau} C_2$ represents the delivery of a single message to its target in C_1. After the delivery, C_1 evolves into C_2. This action does not involve any interaction with the environment. All such message deliveries are uniformly abstracted

as a silent transition since we are not interested in the internal details of a configuration. The remaining two actions, input and output, represent interactions of a configuration with its environment. The input action is used to label a transition in which a configuration accepts a message from the environment targeted to a receptionist in the configuration. The output action is used to label a transition in which a configuration sends out a message to an external actor. The message may consist of names of receptionist as well as non-receptionist actors of the configuration.

Structural congruence:

$C_1 \equiv C_2$ **if** $C_1 \equiv_\alpha C_2$	**congruence up-to alpha equivalence**
$C_1 \mid C_2 \equiv C_2 \mid C_1$	**commutativity of composition**
$(C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)$	**associativity of composition**
$(\nu x)(\nu y)C \equiv (\nu y)(\nu x)C$	
$(\nu x)C_1 \mid C_2 \equiv (\nu x)(C_1 \mid C_2)$	**if** $x \notin fn(C_2)$
$C \mid 0 \equiv C$	
$B\langle x, \tilde{y}\rangle \equiv C(x, \tilde{y})/(u, \tilde{v})$	**if** $B ::= (u, \tilde{v})u(\tilde{w}).C$
$[x = y](C_1, C_2) \equiv C_1$	**if** $x = y$
$[x = y](C_1, C_2) \equiv C_2$	**if** $x \neq y$
$C_1 \equiv C_2 \Rightarrow (\nu x)C_1 \equiv (\nu x)C_2$	
$C_1 \equiv C_2 \Rightarrow C \mid C_1 \equiv C \mid C_2$	

Labeled transition system:

EQUIV:
$$\frac{C_1 \equiv C_1' \xrightarrow{\alpha} C_2' \equiv C_2}{C_1 \xrightarrow{\alpha} C_2}$$

RECV:
$$\frac{}{x(\tilde{y}).C \mid \overline{x}\langle \tilde{z}\rangle \xrightarrow{\tau} C\{\tilde{z}/\tilde{y}\}} \quad len(\tilde{y}) = len(\tilde{z})$$

PAR:
$$\frac{C_1 \xrightarrow{\tau} C_1'}{C_1 \mid C_2 \xrightarrow{\tau} C_1' \mid C_2}$$

HIDE:
$$\frac{C \xrightarrow{\tau} C'}{(\nu x)C \xrightarrow{\tau} (\nu x)C'}$$

IN:
$$\frac{}{C \xrightarrow{\text{in}(\overline{x}\langle \tilde{y}\rangle)} C \mid \overline{x}\langle \tilde{y}\rangle} \quad C : [\rho, \chi], x \in \rho$$

OUT:
$$\frac{}{(\nu \tilde{y'})(C \mid \overline{x}\langle \tilde{y}\rangle) \xrightarrow{\text{out}((\nu \tilde{y'})\overline{x}\langle \tilde{y}\rangle)} C} \quad C : [\rho, \chi], x \notin \rho, \{\tilde{y'}\} \subset \{\tilde{y}\}$$

Figure 2.11: Semantics of System A

The transitions are presented in Figure 2.11. Of particular interest are the rules *IN* and *OUT*. After inputting a message from the environment via the *IN* transition, the free names in the message are added to the external actor set of the configuration (this follows from the type rules to compute the type of the resultant configuration). Similarly, after outputting a message via the *OUT* rule, the private names that are exported in the message are no longer private and become receptionists.

2.5 Other models

We briefly describe a few other models that are relevant the system we propose in the next chapter.

- $D\pi$ is a distributed π-calculus proposed by Hennessy et al. [41]. Computation is distributed over different *locations* in which processes may *migrate* from one site to another and in which sites may *fail*. Distribution is achieved by requiring that processes be located at locations. As in mobile ambients, locations may be nested, giving rise to hierarchies of locations. Communication channels are explicitly located: the use of a channel requires knowledge of both the channel and its location. There are primitives to create new channels and new locations as well as to *halt* locations. Like mobile ambients, names are endowed with *permissions*. The holder of a name may use that name in the manner allowed by these permissions. The following permissions are used (some permissions apply to channel names, others to location names):

  - sending data along a channel.
  - receiving data along a channel.
  - running a thread at a location.
  - placing sub-locations inside a location.
  - creating new channels at a location.
  - move a location inside another location.
  - kill a location

$D\pi$ has a type system, whose main function is to control the capabilities associated with each *instance* of a name. For example, when exporting a new location name, it is natural to constrain the usage of that location by the recipient. These constraints can be specified as a subset of the permissions mentioned above. For instance, the recipient of a location name can be given permission to read from only certain channels within that location, write to only certain channels, and be able to create new channels at that location. Thus, for example, the recipient cannot kill that location or move that location inside some other location. Thus the type system is primarily a means for resource access control. Hennessy et al. present a version of $D\pi$ in [20] where they focus exclusively on resource access control in a distributed setting. The type system there is slightly different in that they tag agents with accumulated capabilities as agents roam around instead of typing each instance of a resource name as is done in the basic $D\pi$ typing system. To prove type safety, they use the notion of a *tagged operational semantics* which was first proposed by Pierce and Sangiorgi in [37].

- The join-calculus [18] can be interpreted as a reformulation of the π-calculus with a more explicit notion of places of interaction. It is one of the first attempts to model locality. The join-calculus is a specific instance of the reflexive CHAM — a general semantic framework based on the CHAM framework [7]. In CHAM[4] (Chemical Abstract Machine), molecules float around in a soup. The soup can be heated or cooled to break terms into molecules or coalesce molecules into terms. This is thus a reversible operation. Two *complementary* molecules can react as shown:

$$[a.p, M, N, \overline{a}.q] \rightarrow [p, M, N, q]$$

This reaction is non-reversible. Further note that there is no notion of locality here. Any two complementary molecules in the soup may react.

[4]CHAM is essentially a term rewriting system modulo structural equivalence while reflexive CHAM is essentially a first order rewriting system modulo structural equivalence.

The reflexive CHAM extends CHAM in two important ways. First, it imposes locality. Second, reactions are allowed to produce not only new molecules but also reaction rules for those molecules. Locality is not imposed directly but is an indirect interpretation. A reaction rule is to be thought of as a location and the molecules that reduce according to this rule are thought to be processes that travel to this location before they can interact with each other. An interaction can produce new molecules, and more interestingly, can produce new reaction rules (locations). However, new reaction rules are created in a safe way. In particular, a process that receives input names may create a new reaction rule that *uses* these names but cannot create reaction rules *for* these names. As noted above, locality is present in a rather primitive form. The distributed join-calculus [17] adds a notion of named locations and a notion of distributed failures.

- The Kell calculus [44] is a very recent formulation. It is a higher order distributed process calculus.[5] Most of the calculi for mobile computation stick to a first order approach. They show that higher order can be encoded in the first order setting and then insist that this is the correct approach because a first order approach has a simpler semantic theory. The Kell calculus is different in this aspect. In [44], the authors argue that certain constructs like distributed software updates, introduction of new components in a system and dynamic reconfiguration in general are inherently higher order constructs. They further argue that the semantic theory induced by the higher order calculi is not as complicated as the first order camp believes it to be.

2.6 Conclusion

In this chapter we looked mainly at the π-calculus, mobile ambients and the actor model. These three foundational systems have been very influential in stimulating research in process algebraic modeling of mobile computation (amongst other areas).

[5] Strictly speaking, it is actually a family of calculi.

In the next chapter we propose the A^{π} calculus for resource access and usage control. It borrows certain concepts from the three systems we described here and its type system is inspired by the style found in the $D\pi$ calculus.

CHAPTER 3
THE A^π CALCULUS

3.1 Introduction

In this chapter we propose A^π — a calculus for distributed mobile computing with static resource access and usage control. We first present the core system. In a subsequent section, we show how to equip it with various useful extensions.

3.2 Basic A^π

As in the design of any system, there are several alternatives for each concept that we wish to model. Our main focus is on resource access control and bounded resource usage in a distributed setting. So, when presented with alternatives for concepts that are orthogonal to our main concern, we often pick the simplest possible alternative. We use the name *basic* A^π for the system we obtain from this exercise. We describe basic A^π in this section.

The main entity in A^π is an *agent*. An *agent* is a computation that has a name. An agent is located in a *location*. It may migrate to other locations. Agents can communicate with other local as well as remote agents via asynchronous message passing with blocking receives and non-blocking sends. Unlike mobile ambients, message passing is *directed*; i.e., each message is addressed to a particular agent. In basic A^π, message communication is location transparent; i.e., an agent just needs to know the name of another agent to communicate with it. This is similar to the high level location-independent messaging primitive found in the Nomadic Pict language [49]. Location transparent messaging is simpler to model because it requires only a global mailbox. Later, this is extended to location-aware communication.

Message communication is typed, and is similar to the system in the Erlang language [6]. Agents explicitly announce the type of the message they are sending. This type is tagged to the message as the message makes its way to the destination agent. The receiving agent can choose to receive messages of a particular type. We note that the information tagged to a message is essential to our system. In general,

when we have an asynchronous message passing system, it is not possible to type the processes (or actors or agents) solely and some typing information has to be passed along with the message itself. We shall talk about this more when we present the type system, but in passing we note that a limited form of tuple communication is possible in basic A^{π} (see Section 3.3.6).

Each location has certain *resources* that are static and cannot leave that location. Agents communicate with resources via *channels*. A channel is to be viewed as a dedicated communication link between an agent and a resource. We note that resources are not modeled directly but a channel implicitly represents a resource. Further, since channel names are scoped by the agent that creates them, a channel name is more accurately interpreted as an instance of a communication link between a resource and an agent. There may be several such links between a resource and the same or different agents, and these links do not interfere with each other. A channel is a *located* entity; i.e., to use a channel, the agent must first migrate to the location of the channel. Channel communication is synchronous with blocking sends and blocking receives. A new agent may be spawned to perform a potentially blocking computation to prevent the initial agent from being blocked. When requesting access to a resource, the agent specifies an usage bound. We impose a type system that guarantees the following: In response to a request, if an agent manages to obtain access to a resource (via a channel), then a well typed agent will respect this usage bound. We note that in basic A^{π}, channels may only communicate *values*. This keeps things simple, but it is straightforward to extend this to a system where channels can communicate *names* as well.

In basic A^{π}, any agent can create located channels as well as new locations. Remote channel creation is not allowed; i.e., an agent has to migrate to the remote location and then create a channel there. Unlike the $D\pi$-calculus or mobile ambients, we have a flat space for locations and there is no hierarchy between locations. All new locations are created in this same flat space. Further, locations are static and cannot be moved (movement does not make sense in a flat space). We later extend this to a system where only certain *privileged* agents can create new locations and/or new located channels.

In basic A^π, there is no control over migration into and out of locations, or the ability to communicate remotely. In the actor model, the interface of a configuration is controlled via the notions of receptionists and external actors. We later extend our system to provide similar control. However, we take a more fine-grained approach in which appropriate run-time capabilities are required to execute any of the following actions: sending a remote message, receiving a remote message, migrating out from a location, migrating into a location, creating new channels at a location, creating new locations.

We assume that names (of either agents, locations or channels) are unforgeable entities, and cannot be guessed. An agent begins execution with the knowledge of certain names. It can learn new names from fellow agents or it can create new names and communicate them to fellow agents.

3.2.1 Syntax of basic A^π

Since message typing is explicit in the syntax, we first explain the types in the language. (The type system itself is explained later in Section 3.2.2.) The types are outlined in Table 3.1.

Type	
$\mathcal{N}$	Agent names
$\mathcal{L}$	Location names
$\mathcal{V}$	Values
$\zeta ::= \mathcal{N} \mid \mathcal{L} \mid \mathcal{V}$	Convenience type
$\mathcal{C}_i \quad i \geq 0$	Channel names
$\mathcal{T}_i ::= \mathcal{L}[\mathcal{C}_i] \quad i \geq 0$	Located channels

Table 3.1: Types for the A^π calculus

$\mathcal{N}$ and $\mathcal{L}$ are ground unstructured types used for agent names and location names. $\mathcal{C}_i$ actually stands for a collection of types. $\mathcal{C}_n$ is the type of a channel that can be used at most n times. Channel names themselves are unstructured entities. $\mathcal{T}_i$ is used to type located channels. It is a structured type and can be used to type values conforming to this structure — e.g., $l[c]$ is a channel c located at location l. Channels can only communicate values of type $\mathcal{V}$. In basic A^π, $\mathcal{V}$ is assumed to be a ground unstructured type. However, this is not a requirement and in a subsequent

extension, we interpret $\mathcal{V}$ as a structured type. The types $\mathcal{N}$, $\mathcal{L}$ and $\mathcal{V}$ are grouped together into type ζ for convenience, since many of the typing and reduction rules involving these types are similar. Agents can communicate values of type $\mathcal{T}_i$ or ζ.

Threads:

p	::=	**nil**	
		$\overline{\beta}(d:X).p \;\; X = \zeta \mid \mathcal{T}_i$	agent communication: asynchronous output
		$?(d:X).p \;\; X = \zeta \mid \mathcal{T}_i$	agent communication: input
		$\nu_{agt}(\beta(q)).p$	New agent
		$\overline{\overline{a}}(v).p$	synchronous writing to a resource
		$a??(v:\mathcal{V}).p$	synchronous reading from a resource
		$\nu_{ch}(a:\mathcal{C}_i).p$	New Located Channel
		$\nu_{loc}(l:\mathcal{L}).p$	New location
		$l :: p$	moving to location l
		$d = e \; ? \; p : q$	match
		$*p$	replication

Agents:

A	::=	**nil**	
		$\alpha[p] \mid A$	Agent with name α composed with other agents

Channels:

C	::=	**nil**	
		$a \mid C$	Channel a

Location:

L	::=	**nil**	
		$l\left[A \mid C\right] \;\Big\vert\; L$	Locations with agents, and located channels

Directed Messages:

M	::=	**nil**	
		$\alpha\langle d:\zeta\rangle \mid M$	Message for agent α with data d composed with other messages

System:

S	::=	**nil**	
		$L \mid M$	A collection of locations with a global mailbox

Figure 3.1: Syntax of the A^π calculus

The syntax of the system is presented in Figure 3.1. Threads are basic sequences of computational instructions. An agent is a named thread. We use p, q, r to denote threads and α, β, γ to denote agent names. l, m, n are used for location names and a, b, c are used for channel names.

We now explain the syntactic constructs:

- $\alpha[\mathbf{nil}]$: Agent named α is executing the nil thread.

- $\alpha[\overline{\beta}(d : X).p]$: Agent named α outputs a message d with type X to agent named β and then proceeds to execute p. As indicated by the side condition, X is ζ or a located channel type $\mathcal{T}_i$. Strictly speaking, this annotation is required only if d is of type $\mathcal{T}_i$. If it is of type ζ, then we could infer its type implicitly instead of requiring it to be explicit in the syntax. For uniformity, we require it to be explicit. As will become clear later, the reason why we cannot infer the type $\mathcal{T}_i$ of a located channel is because it is possible to output a located channel with a subtype of its original type; i.e., if the original type was $\mathcal{T}_i$ then it is possible to output this located channel as belonging to the type $\mathcal{T}_j$, where $0 \leq j \leq i$. Thus, it becomes necessary to explicitly annotate the output construct with a type. The type system ensures that we cannot output a located channel of type $\mathcal{T}_i$ with type $\mathcal{T}_j$ where $j > i$.

- $\alpha[?(d : X).p]$: Agent named α is waiting for a message of type X and continues to execute p after appropriate parameter substitution.. As indicated by the side condition, X is of type ζ or $\mathcal{T}_i$.

- $\alpha[\nu_{agt}(\beta(q)).p]$: Agent named α creates a sub-agent named β and then continues to execute thread p. β *concurrently* starts executing q (this follows from the interleaved concurrency in the operational semantics). β is assumed to be a universally fresh agent name.

- $\alpha[\overline{\overline{a}}(v).p]$: Agent named α writes value v to resource represented by channel a and then continues to execute p. As will be seen in the operational semantics, the communication is synchronous. We note that since channel names and agent names belong to different types, it would be possible to use the same

output and input constructs for both channels and agents. However, we choose to use different syntactic constructs for better readability.

- $\alpha[a??(v : \mathcal{V}).p]$: Agent named α reads a value from the resource represented by channel a and then proceeds to execute p.

- $\alpha[\nu_{ch}(a : \mathcal{C}_i).p]$: Agent named α creates a new channel named a of type $\mathcal{C}_i$ and then continues to execute p. a is assumed to be fresh within the current location of α. The type system will automatically infer the corresponding located channel type of a.

- $\alpha[\nu_{loc}(l : \mathcal{L}).p]$: Agent named α creates a new location named l. l and then continues to execute p. is assumed to be an universally fresh location name.

- $\alpha[l :: p]$: Agent named α migrates to location l and then executes p in the new location.

- $\alpha[d = e \ ? \ p : q]$: Agent named α executes the standard match construct. Depending on the result of the match, it either executes p or q. The type system ensures that the values d and e are of the same type.

- $\alpha[*p]$: Agent named α repeatedly executes thread p. We note that this particular style of repetition (coupled with an appropriate type rule as shown in Figure 3.2) allows us to get away without needing recursive types. The other common style is to not have any construct for repetition, but to make repetition explicit in the program. For example $p ::= (A \mid B).p$ is used to define a process p that does action A or B, and then continues again as p. This style requires some notion of recursive types to type p.

As seen in Figure 3.1, the structure of the system consists of a collection of locations and a global mailbox. The mailbox is not modeled explicitly, it is implicit in the structure of S. Each location consists of a collection of agents and a collection of channels local to that location. We further note that an agent never disappears. It may reach a stage where it is executing the *nil* thread and thus will never process any outstanding messages targeted to it. This is somewhat similar to the actor

model where an actor may start performing an internal computation step that is divergent.

3.2.2 The Type System

We first note that not all well formed syntactic terms are valid systems. Broadly, a well formed term may fail to be a valid system because of two reasons. First, a name may be used improperly. Consider the term $\alpha[\overline{\beta}(d:\mathcal{V}).\beta??(v:\mathcal{V}).p]$. The agent named α is trying to use the name β as both an agent and a channel. This is clearly invalid. Second, a located channel created (or received) with type $\mathcal{T}_i$ may be passed along to other agents or new child agents in such a way that it eventually ends up being used more than i times. This too is a violation. We now present a type system to prevent violations in both of these categories. In passing we note that since we assume names to be unforgeable and unguessable, we do not have to worry about agents having unauthorized access to resources.

Our description will avoid those mathematical technicalities which are either not needed to understand the type system (e.g., the lattice structure of the subtyping relationship.) or which are obvious (e.g., tuple arity checks or arithmetic involving ∞.) Only the intended meaning of these constructs shall be described.

The types in our system have already been described in Table 3.1. Also, as may be seen from the syntax of the language, the typing is explicit. Whether explicit typing or implicit type inference is to be preferred is not so clear. Implicit type inference is desirable in open distributed systems where not all participants can be expected to adhere to the same explicit type discipline and moreover, a principal need not rely on any trust in the type annotations of incoming code. Intuitively, implicit type inference looks less powerful than explicit type annotations. However, as presented in [27], in a $D\pi$-like calculus with notions of locations and migration, there exists an algorithm for implicit type inference.

The basic type judgment has the form

$$\Gamma \vdash_m \alpha[p]$$

This means that with the type environment Γ, the agent with name α and thread

p is well typed to run at location m. A type environment is a collection of agent names, location names, values and located channel names, each with an associated type. The usage bound of a located channel is explicit in its type. Initial type environments are assumed to be *consistent*. This means that no entity with the same name is typed twice in the environment.[6] The type rules preserve the consistency of a consistent type environment.

We will sometimes omit the location subscript to indicate that the agent is well typed at all locations:

$$\Gamma \vdash \alpha[p]$$

Consider the type environment $\Gamma = \{l : \mathcal{L}, l[a] : \mathcal{T}_4, l[b] : \mathcal{T}_\infty, m : \mathcal{L}, \alpha : \mathcal{N}, \beta : \mathcal{N}\}$ In this environment, at location l, resource a may be used at most four times and resource b may be used an infinite number of times. Further, no resources are known at location m and α and β are the only known agent names. Thus for an agent to be well typed at location l in this environment, amongst other things, it has to be named either α or β and can communicate with only these two agents. However, as will be seen in the type rules, it can learn the names of new agents and begin communicating with them.

The basic type judgment uses subsidiary type judgments of the form:

$$\Gamma \vdash X : T$$

This says that in the type environment Γ, X is well typed with type T. Typically X is either an agent name or a location name or a located channel name. Thus, $\Gamma \vdash \alpha : \mathcal{N}$ means that in the type environment Γ, name α is well typed to be an agent name. Similarly, $\Gamma \vdash l[c] : \mathcal{T}_i$ means that in type environment Γ, c is a channel with usage bound i located at location l.

A type environment is thus a collection of declarations of the form $X : T$. We say that X is the *subject* of the declaration. We use the notation $\Gamma, X : T$ to denote a type environment which is obtained by extending Γ with the global declaration

[6] This is not strictly required. We can have name overloading, but we would like to keep things simple.

$X : T$. For this to be well defined, X should not be the subject of any declaration in Γ. We use the notation $\Gamma_{X:T}$ to denote a type environment which is the same as Γ but in which the type of X is now T. For this to be well defined Γ must have a declaration with subject X.

$$\text{(t-ground)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, m : \mathcal{L}}{\Gamma \vDash_m \alpha[nil]}$$

$$\text{(t-agt-out-1)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, \beta : \mathcal{N}, d : \zeta \quad \Gamma \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ \overline{\beta}(d : \zeta).p\ \big]}$$

$$\text{(t-agt-out-2)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, \beta : \mathcal{N}, l[c] : \mathcal{T}_i \quad \Gamma_{l[c]:\mathcal{T}_j} \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ \overline{\beta}(l[c] : \mathcal{T}_k).p\ \big]}\ i \geq j + k$$

$$\text{(t-agt-in-1)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma, d : \zeta \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ ?(d : \zeta).p\ \big]}$$

$$\text{(t-agt-in-2)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma, l : \mathcal{L}, l[a] : \mathcal{T}_i \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ ?(l[a] : \mathcal{T}_i).p\ \big]}$$

$$\text{(t-ch-out)}\ \frac{\Gamma \vdash m[a] : \mathcal{T}_i, v : \mathcal{V}, \alpha : \mathcal{N} \quad \Gamma_{m[a]:\mathcal{T}_{i-1}} \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ \overline{\overline{a}}(v).p\ \big]}\ i > 0$$

$$\text{(t-ch-in)}\ \frac{\Gamma \vdash m[a] : \mathcal{T}_i, \alpha : \mathcal{N} \quad \Gamma_{m[a]:\mathcal{T}_{i-1}}, v : \mathcal{V} \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\ a??(v : \mathcal{V}).p\ \big]}\ i > 0$$

$$\text{(t-new-agt)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma_1, \beta : \mathcal{N} \vDash_m \alpha[p] \quad \Gamma_2, \beta : \mathcal{N} \vDash_m \beta[q] \quad \Gamma = \Gamma_1 \uplus \Gamma_2}{\Gamma \vDash_m \alpha\big[\ \nu_{agt}(\beta(q)).p\ \big]}$$

$$\text{(t-new-ch)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma, m[a] : \mathcal{T}_i \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\nu_{ch}(a : \mathcal{C}_i).p\big]}\ i \geq 0$$

$$\text{(t-new-loc)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma, l : \mathcal{L} \vDash_m \alpha[p]}{\Gamma \vDash_m \alpha\big[\nu_{loc}(l : \mathcal{L}).p\big]}$$

$$\text{(t-move)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, n : \mathcal{L} \quad \Gamma \vDash_n \alpha[p]}{\Gamma \vdash \alpha[n :: p]}$$

$$\text{(t-rep)}\ \frac{\Gamma \vdash \alpha : \mathcal{N} \quad \Gamma \ \vDash^{\infty}_m \alpha[p]}{\Gamma \vDash_m \alpha[*p]}$$

$$\text{(t-match-1)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, d : \zeta, e : \zeta \quad \Gamma \vDash_m \alpha[p], \alpha[q]}{\Gamma \vDash_m \alpha\big[d = e\ ?\ p : q\big]}$$

$$\text{(t-match-2)}\ \frac{\Gamma \vdash \alpha : \mathcal{N}, d : \mathcal{T}_i, e : \mathcal{T}_j \quad \Gamma \vDash_m \alpha[p], \alpha[q]}{\Gamma \vDash_m \alpha\big[d = e\ ?\ p : q\big]}\ i, j \geq 0$$

Figure 3.2: Type rules for agents in the A^{π} calculus

The type rules are presented in Figure 3.2.

- *t-ground* : This is a ground rule that specifies that an agent executing the *nil* thread is well typed.

- *t-agt-out-1*: For $\alpha\big[\ \overline{\beta}(d : \zeta).p\ \big]$ to be well typed in Γ, α, β and d should be well typed in Γ and the continuation $\alpha[p]$ should also be well typed in Γ.

- *t-agt-out-2*: This is similar to the previous one, except that we are now dealing with a located channel and so we need to account for the usage bound. If a located channel with initial bound i is being output with bound k, then we need ensure that in the continuation, it is being used at most j times where $i \geq j + k$.

- *t-agt-in-1* and *t-agt-in-2*: For the input constructs to be well typed , we need to ensure that the continuation is well typed in Γ extended with d or $l[a]$. Recall that the definition of environment extension requires d or $l[a]$ to be fresh in Γ. If it is not, then the process can still be well typed after alpha renaming d or $l[a]$. Note that in *t-agt-in-2*, the type environment is also extended with the location name l. Thus, communicating a located channel is tantamount to communicating the location name as well as the channel name within that location.

- *t-ch-out* and *t-ch-in*: To perform I/O via a located channel $m[a]$, the agent must first be located at location m. In the type system this is captured by checking if $m[a]$ is globally typed and if so, then, locally typing the agent at location m. Further, we must ensure that the usage bound on $m[a]$ is respected by the continuation.

- *t-new-agt*: The essential idea here is to split the initial environment Γ into two environments Γ_1 and Γ_2 such that the new agent named β is typable in Γ_2 extended with β and the continuation is typable in Γ_1 extended with β. The splitting of Γ into Γ_1 and Γ_2 is captured by the $\uplus$ operator: $\Gamma = \Gamma_1 \uplus \Gamma_2$. The definition of $\uplus$ imposes different interpretations on the ν_{agt} operator.

Any definition of $\uplus$ must necessarily satisfy the following condition: After splitting the initial environment, the usage bounds on all located channels must be respected. The other issue to be decided is "what names does the new agent initially know of?"

Consider a type environment $\Gamma = \{l[a] : \mathcal{T}_5, m[a] : \mathcal{T}_6, \beta : \mathcal{N}, n : \mathcal{L}\}$.

One way of splitting it is the following:

$$\Gamma_1 = \{l[a] : \mathcal{T}_5, m[a] : \mathcal{T}_5, \beta : \mathcal{N}, n : \mathcal{L}\}$$
$$\Gamma_2 = \{m[a] : \mathcal{T}_1\}$$

In this split, the usage bounds on both located channels are respected, but the new agent knows only a subset of the names known to the initial agent. This resembles a scenario where a child agent is being created for a specific purpose.

Another way of splitting it is the following:

$$\Gamma_1 = \{l[a] : \mathcal{T}_2, m[a] : \mathcal{T}_5, \beta : \mathcal{N}, n : \mathcal{L}\}$$
$$\Gamma_2 = \{l[a] : \mathcal{T}_3, m[a] : \mathcal{T}_1, \beta : \mathcal{N}, n : \mathcal{L}\}$$

In this split, the usage bounds on both located channels are respected, and the new agent knows all the names of the initial agent. This resembles a UNIX like scenario where a process forks into two.

The exact definition of the $\uplus$ operator is an implementation issue and there is no need to decide it here.

- *t-new-ch* and *t-new-loc*: These are simple to understand. The only interesting issue is that the *t-new-ch* rule does *not* rule out remote channel creation. However, it ensures that remote channel creation is safe in the following way: An agent that is well typed in location m will also be well typed to create a new channel at location m, irrespective of its current location. Remote channel creation is ruled out in the operational semantics.

- *t-move*: This says that for $\alpha[n :: p]$ to be well typed at *any* location, all that

is required is that the continuation $\alpha[p]$ be well typed at location n.

- *t-rep*: Here, we use a new type judgment: $\models_m^\infty$. We say that $\Gamma \models_m^\infty \alpha[p]$ if :

  1. $\Gamma \vdash_m \alpha[p]$ and
  2. The only located channels that p uses (to read from or write to) have a bound ∞. Note that p is allowed to communicate non-∞ located channels.

- *t-match-1* and *t-match-2*: There are a couple of interesting things here. First note that we require d and e to be of the same type. Further, if they are located channels, then they are allowed to have different usage bounds. Second, we see that using a channel name in a *match* construct is not counted as a usage of that name as far as the usage bound limits are concerned. Third, we see that for the match construct to be well typed, both of the possible continuations p and q must be well typed in the same environment. This is a conservative rule but it is the best we can do in a simple type system. It is conservative because it throws away any information available by looking at the possible values of d and e and treats both continuations in an uniform way. In more powerful type systems with notions like dependent types (c.f. the Epigram language [3]) or proof-based approaches, it is possible to treat both continuations non-uniformly.

3.2.3 Operational Semantics

The operational semantics of A^π are given modulo the structural equivalence in Figure 3.3.

The first rule gives equivalence up-to alpha renaming. The next five rules deal with commutativity and associativity, at the location and system levels. These rules are based on the structure of locations and systems and are obvious. The only interesting rule is the one for replication. We note that this particular rule for *unwinding* replication enforces sequentiality. Another common alternative, found for instance in mobile ambients, is to choose the following rule: $\alpha[*p] \equiv \alpha[p] \mid \alpha[*p]$. This rule can be interpreted as parallel unwinding. It is useful to model replication

Structural Equivalence:

$$
\begin{array}{rcll}
S_1 \equiv_\alpha S_2 & \Rightarrow & S_1 \equiv S_2 & \text{alpha renaming} \\
\mathbf{nil} \mid x & \equiv & x & x \in \{A, M, C, L, S\} \\
x \mid y & \equiv & y \mid x & x, y \in \{L, M\} \\
(x \mid y) \mid z & \equiv & x \mid (y \mid z) & x, y, z \in \{L, M\} \\
l[x \mid y] & \equiv & l[y \mid x] & x, y \in \{A, C\} \\
l[(x \mid y) \mid z] & \equiv & l[x \mid (y \mid z)] & x, y, z \in \{A, C\} \\
\alpha[*p] & \equiv & \alpha[p. * p] & \text{replication}
\end{array}
$$

Figure 3.3: Structural Equivalence in the A^π calculus

of services. Obviously, to use this rule we would need a calculus where uniqueness of names is not assumed.

$$l\Big[\alpha[\,\overline{\beta}(d:\zeta).p]\Big] \longrightarrow l\Big[\alpha[p]\Big] \;\Big|\; \beta\langle d:\zeta\rangle \qquad \text{r-m-out-1}$$

$$l\Big[\alpha[\,\overline{\beta}(d:\mathcal{T}_i).p]\Big] \longrightarrow l\Big[\alpha[p]\Big] \;\Big|\; \beta\langle d:\mathcal{T}_i\rangle \qquad \text{r-m-out-2}$$

$$l\Big[\alpha[?(d:\zeta).p]\Big] \;\Big|\; \alpha\langle e:\zeta\rangle \longrightarrow l\Big[\alpha[p_{e/d}]\Big] \qquad \text{r-m-in-1}$$

$$l\Big[\alpha[?(d:\mathcal{T}_i).p]\Big] \;\Big|\; \alpha\langle e:\mathcal{T}_j\rangle \longrightarrow l\Big[\alpha[p_{e/d}]\Big] \text{ if } i \leq j \qquad \text{r-m-in-2}$$

$$l\Big[\alpha[\,\overline{\overline{a}}(u).p] \mid \beta[a??(v:\mathcal{V}).q]\Big] \longrightarrow l\Big[\alpha[p] \mid \beta[q_{u/v}]\Big] \qquad \text{r-ch-comm}$$

$$l\Big[\alpha[d = e\ ?\ p : q]\Big] \longrightarrow l\Big[\alpha[p]\Big] \text{ if } d = e \qquad \text{r-match-1}$$

$$l\Big[\alpha[d = e\ ?\ p : q]\Big] \longrightarrow l\Big[\alpha[q]\Big] \text{ if } d \neq e \qquad \text{r-match-2}$$

$$l\Big[\alpha[\nu_{agt}(\beta(q)).p]\Big] \longrightarrow l\Big[\alpha[p] \mid \beta[q]\Big] \text{ globally fresh } \beta \qquad \text{r-new-agt}$$

$$l\Big[\alpha[\nu_{ch}(a:\mathcal{C}_i).p] \mid C\Big] \longrightarrow l\Big[\alpha[p] \mid C \mid a\Big] \text{ if } a \notin C \qquad \text{r-new-ch}$$

$$l\Big[\alpha[\nu_{loc}(k:\mathcal{L}).p]\Big] \;\Big|\; L \longrightarrow l\Big[\alpha[p]\Big] \;\Big|\; L \;\Big|\; k\Big[\mathbf{nil}\Big] \text{ if } k \notin \{l \cup dom(L)\} \qquad \text{r-new-loc}$$

$$l\Big[\alpha[m :: p] \mid A_1 \mid C_1\Big] \;\Big|\; m\Big[A_2 \mid C_2\Big] \longrightarrow l\Big[A_1 \mid C_1\Big] \;\Big|\; m\Big[\alpha[p] \mid A_2 \mid C_2\Big] \text{ if } l \neq m \qquad \text{r-agt-migrate-1}$$

$$l\Big[\alpha[m :: p]\Big] \longrightarrow l\Big[\alpha[p]\Big] \text{ if } l = m \qquad \text{r-agt-migrate-2}$$

Figure 3.4: Operational Semantics of the A^π calculus—I

$$\frac{l\left[\alpha[p] \mid A_1 \mid C_1\right] \mid M_1 \longrightarrow m\left[\alpha[p'] \mid A_1' \mid C_1'\right] \mid M_1'}{l\left[\alpha[p] \mid A_1 \mid C_1 \mid A_2 \mid C_2\right] \mid M_1 \mid M_2 \longrightarrow m\left[\alpha[p'] \mid A_1' \mid C_1' \mid A_2 \mid C_2\right] \mid M_1' \mid M_2'} \text{r-struct-1}$$

$$\frac{S \longrightarrow S'}{S \mid S_1 \longrightarrow S' \mid S_1} \quad \text{r-struct-2}$$

$$\frac{S_1 \equiv S_2 \quad S_2 \longrightarrow S_3 \quad S_3 \equiv S_4}{S_1 \longrightarrow S_4} \quad \text{r-struct-3}$$

Figure 3.5: Operational Semantics of the A^π calculus—II

The operational semantics are presented in Figures 3.4 and 3.5. They are meant to apply to well typed terms only. Further, we require that initially the system is *well formed*. For a system to be *well formed* there should be no agents in any location with the same name, i.e., agent names are globally unique. Further, at any location, there should be no channels with the same name, i.e., channel names are locally unique. If a system is well formed, our transition system guarantees that any transition preserves the well-formed-ness of the system. (More strongly, the type system guarantees that any well typed program cannot break either the well-formed-ness of the system or exceed the usage bounds on any of the resources in the system. However, we shall not be getting into Subject reduction and Type safety here. See Chapter 4 for our future plans to use formal techniques to prove such results.)

The details of the underlying message passing system are mostly orthogonal to our concerns. As is normal in the actor model, the only constraint we impose on the message delivery system is fairness. We do not assume ordered delivery of messages.

We now describe the transition system:

- *r-m-out-1* and *r-m-out-2*: These just put the message into the global mailbox. The structural equivalence rules (Figure 3.3) imply that the mailbox is treated as a *multiset*.

- *r-m-in-1* and *r-m-in-2*: This delivers a message from the global mailbox to the target agent. There are a couple of interesting points here.

  First, the structure of this rule indicates that we are modeling location-transparent messaging. Only the name of an agent need be known to communicate with it.

  Second, as done in the Erlang language [6], the receiving agent specifies a type/pattern while attempting to receive a message. The pattern matching notion along with the structure of the transition rules serves to have the following effect: From all the messages for this agent that are currently in the global mailbox, the system randomly picks one that pattern matches and delivers it to the agent. After the message is delivered, the appropriate substitution of actual names for variables is done. Substitution is defined structurally in the expected way.

  Third, note the side condition on the *r-m-in-2* rule. If an agent is expecting to receive a located channel with an usage bound of i, then the system may deliver one with a usage bound of $j \geq i$.

- *r-ch-comm*: This is as expected. Channel communication is local and synchronous. The interesting issue here is that we do not need to explicitly ensure the presence of a in the channel pool C, i.e., the following is *not* required:

$$l\Big[\alpha[\,\overline{\overline{a}}(u).p] \mid \beta[a??(v:\mathcal{V}).q]\Big] \;\longrightarrow\; l\Big[\alpha[p] \mid \beta[q_{u/v}]\Big] \quad \text{r-ch-comm}$$

  This is because we already know that agents α and β are well typed at location l. So, the type system has ensured that both of them have already learnt about the channel name a before attempting to use it. In particular note that if a is a valid channel name at l and α has learnt the name a while β has not, then only α will be well typed to use it. This is an instance of resource access control provided via the type system.

- *r-match-1* and *r-match-2*: The type system has already ensured that both d and e belong to the same type. The only interesting detail is when they

are located channels. In this case, equality is checked structurally. Both the location name and the channel name should match.

- *r-new-agt*, *r-new-ch*, *r-new-loc*: These are as expected. Care is taken to ensure local and global uniqueness of appropriate names. Because of this, migration requires no side conditions.

  Note that we have chosen to give an explicit computational significance to the various ν operators. This is in sharp contrast to the presentation style in the π-calculus where the ν operators have an indirect interpretation as "creation of new channels". Usually, a combination of an appropriate structural equivalences and a conditional transition rule of the following form is present:

  $$\frac{P \rightarrow Q}{(\nu x).P \rightarrow (\nu x).Q}$$

  Because of our more explicit style, it is easier to visualize the reduction of a term. In the π-calculus approach, the reduction of processes involving scope restriction and scope extrusion involves heavy usage of the structural equivalences and requires some experience to be comfortable with. Another issue affected by these two different styles is the addition of failure primitives to the language. Suppose we desire to add primitives for channel deletion, location deletion, location halting, etc., which are constructs to model failure in calculi for distributed systems. The approach we need to take depends on which style we have chosen for the channel and location creation primitives. We shall elaborate upon this in Section 3.3.5.

- *r-agt-migrate-1* and *r-agt-migrate-2*: The second one may seem odd at first but it is needed because an agent that tries to migrate to its current location is indeed well typed and so must be accounted for in the operational semantics.

- *r-struct-1*, *r-struct-2* and *r-struct-3* : These are the usual context laws. They allow us to specify the transitions of a term irrespective of the context in which it is embedded (wherever the said transition is not dependent on the context).

3.3 Limitations and Extensions

In this section we point out the limitations of the basic system and indicate how it can be extended along various dimensions. We have already alluded to some of these issues in the previous sections but we shall repeat those again to collect everything in the same section.

3.3.1 Encapsulation via capabilities

3.3.1.1 The tagged language approach

In basic A^π, there is no encapsulation (as found in the actor model). We do have locations, but any agent is free to migrate to any location, is free to create new locations as well as new channels at any locations and is free to communicate with any agent whose name it knows. This is not desirable in a distributed system. In the actor model, each configuration[7] has a well defined interface (receptionist set and external actor set) which is used to encapsulate the visibility of that configuration in the environment. We take a more fine grained approach with *capabilities*. There exists a more or less standard way of using a *tagged* language to add capabilities to a system. This approach has been used by Pierce and Sangiorgi [37] in the context of refining channel usage by adding I/O capabilities. We extend the same approach here. (Keeping in line with our central theme of bounded resource usage, the extension deals with adding a constraint on the number of times that a capability may be exercised.)

Our basic idea is the following: suppose we want to add the capabilities of exiting a location and entering a location; i.e., to migrate to a new location, an agent must already possess the capabilities to exit from its current location and enter the new location. Let us represent this as the capability set $C ::= \{in, out\}$. Then, we first generate the power-set of C, in this case $2^C = \big\{\{in\}, \{out\}, \{in, out\}, \phi\big\}$. Then, we tag each *instance* of a location name with an element from 2^C. For instance, $l_{\{in\}}$ indicates a "black hole" instance of location l where only entering l is permitted. Different agents may obtain different instances of l and thus have differ-

[7] We would like to note that a configuration in the actor model is a more abstract notion meant to model composability, as compared to a location which is intended to model a real physical location or site.

ent capabilities. At the operational semantics level, we need to add an appropriate conservative rule, such as:

$$l\Big[\alpha[?(M_{\{in\}} : \mathcal{L}).p]\Big] \;\Big|\; \alpha\langle m_{\{in,out\}} : \mathcal{L}\rangle \longrightarrow l\Big[\alpha[p_{m_{\{in\}}/M_{\{in\}}}]\Big]$$

Note that the system may deliver a name which has additional capabilities beyond those requested, after stripping the name of its additional capabilities. Also note that some combinations of capabilities may not make sense. For example just the capability to exit a location is of no use to an agent that is not initially located in this location. It is straightforward to tackle this issue. We need to define a constraining function that filters the power-set to drop out all the non-sensical combinations.

It is relatively easy to modify the type system to accommodate this tagged syntax. To see this from an abstract standpoint, we may view the type system as a mechanism to specify a relation between different syntactic constructs. For example in basic A^{π}, the migration construct, ::, can be viewed as an unary relation that holds for all location names; i.e., l :: holds if l is a location name. So, extending the type system to accommodate the tagged syntax involves refining this relation. In the tagged syntax, the relation holds only for location names that have at least the *in* tag, i.e., $l_{\{in,\ldots\}}$ :: holds.

The tags are meant to be explicitly specified only in an input prefix when inputting a name with a desired capability set. After that, the name may be used without the tags. A tag inference algorithm can then reconstruct the tags to get an explicitly tagged program. This is done because in the approach used by Pierce and Sangiorgi [37], the operational semantics is defined over a tagged language. The tagged operational semantics is used to formalize the notion of a run-time error and thus prove type safety.

We note that in the tagged language approach, there is no direct way to modify the existing capabilities on a name. The only way to modify existing capabilities is to try to obtain a fresh instance of a name tagged with the desired capabilities.

3.3.1.2 Bounded capabilities

We shall now introduce a bound on the number of times that a capability might be exercised. Then, we shall make a modification to the types of basic A^{π} to obtain a more homogeneous treatment.

As we did with located channels, it is straightforward to introduce an integer into the structure of the tag to indicate how many times the capability may be exercised. Thus $l_{\{in_5\}}$ indicates a capability that endows the owner to enter location l at most five times. As was the case with located channels, a part of this usage bound may be handed out to other agents. The required extensions to the type system and the operational semantics will follow on the same lines as the rules that we currently have for located channels.

We note that in basic A^{π}, we type located channels as belonging to a family of types indexed with an integer: $\mathcal{T}_i$. But in essence, what we have is a bounded capability to access this located channel. So, we can now type all located channels as belonging to the same type $\mathcal{T}$ and then tag each name instance with a bounded capability to access this located channel.

Finally, the only issue that remains is "which capabilities should we add?" The tagged language approach is very general since it allows us to associate any number of capabilities with any syntactic construct in our language. Table 3.2 shows a representative set of capabilities.

Type	Tag	
$\mathcal{L}$	in	Entering a location
	out	Exiting a location
	chcreate	Creating a new channel
$\mathcal{T}$	i	reading from a channel
	o	writing to a channel

Table 3.2: Representative capability set

In closing we mention two issues. First, note that the notion of bounded capabilities has to be used carefully. For instance, suppose a location wants to control the total number of agents that may be present in the location at any given time. In such a scenario, giving one agent the capability to enter this location with

a usage bound of 5 may be o.k. because every time it executes this capability, the number of agents in the location increases by just one. However, it may not be o.k. if this agent distributes this capability to five other agents with a usage bound of 1 each, because, if all of them enter the location, then the number of agents in the location will increase by five. This situation may be remedied by tagging each capability with a set of agent names (in this case just one agent name) specifying the agents that may exercise this capability.

Second, note that certain capabilities cannot be (directly) associated with an instance of some name. For example the capability to create new locations or halt existing locations (assuming such constructs exist in our language). We wish to associate these capabilities with an agent, but it does not seem sensible to tag these capabilities to agent names (since the only action we can perform with an agent name is communicate with it, which is unrelated to the capabilities we are talking about). To accommodate such capabilities, we need to introduce a notion of a *privileged* type environment. An agent that tries to create or halt a location needs to be well typed in a privileged type environment.

3.3.2 Static resources

So far we have been thinking of channel communication as resource access. However in basic A^{π}, this is not strictly true. Basic A^{π} is more accurately described as a system of agents that have two communication primitives available to them: local and remote asynchronous message passing and local synchronous communication over shared channels. This is because in basic A^{π}, an agent that wishes to communicate with a resource actually communicates with another agent on a shared channel name. Thus this other agent is being thought of as a resource manager. However, there is nothing in the calculus itself that prevents the resource manager from migrating to a different location. The static nature of the resource manager is assumed at the application level. It is desirable to make this static nature explicit in the calculus itself; i.e., resource managers (and implicitly resources) do not migrate and thus resources are static and local to a location.

The simplest way to do this is by introducing a new category R of resource

managers (along with threads, agents, channels, locations, messages and systems). Resource managers are only allowed to read from and write to a channel, create new channels (representing a new port to communicate with the resource) and spawn new resource managers (to enable different agents to concurrently access the same resource, where this makes sense). The typing system can easily ensure that a resource manager that attempts to use any other syntactic construct is not well typed. If desired, the operational semantics rules may be modified so that channel communication is permitted only if one of the participating entities is an agent and the other is a resource manager (to rule out agent-agent communication via channels). The above extensions will make the notions of static resources and resource communication via channels more concrete in the calculus.

3.3.3 Extending the type system

Note that the current type system is too simple because it cannot deal with arithmetic expressions. So, it will not be able to type expressions involving located channels whose bound is an arithmetic variable (instead of an actual number). A simple example that cannot be typed is the following agent that receives a located channel with bound $2k$, then creates two new agents (that execute a thread q) and communicates this located channel name with new bound k to each.

$$\alpha\big[?(d : \mathcal{T}_{2k}) \,.\, \nu_{agt}\ \beta(q) \,.\, \nu_{agt}\ \gamma(q) \,.\, \overline{\beta}(d : \mathcal{T}_k) \,.\, \overline{\gamma}(d : \mathcal{T}_k)\big]$$

To handle such expressions, the usual approach is to have the type system generate verification conditions along with each typing judgment. In this case, the verification condition would be $k + k \leq 2k$. For the term to be well typed, the verification condition needs to be discharged. This can be done by proving the condition using a theorem prover that understands arithmetic. If the condition is such that it cannot be proven statically (typically in cases where it depends on run-time values) then an additional run-time check is generated. The operational semantics also need to be augmented so that any such run-time checks are discharged before the reduction rule is allowed to go through.

3.3.4 Location aware messaging

In basic A^{π}, agent-to-agent communication is location transparent. This seems to be a reasonable primitive when agents are mobile and can move around. However, it is desirable to be able to remotely communicate with resources. Currently, the only way to communicate with resources is by establishing a location-specific channel and communicating over that channel. We view a channel as a dedicated means of communication. It is thus *expensive* to implement and so it is not desirable to add global channels that spawn over different locations. However, what can be done is to add the capability to communicate with resources via asynchronous message passing. To do so, we now need to model resources explicitly. Recall that currently they are implicitly modeled via channels. By modeling resources explicitly as belonging to a different type, say R, it is also possible to use the above mentioned capabilities approach to limit remote resource access for resources that we do not wish to be remotely accessible.

3.3.5 Modeling failures

The ability to model failures in a distributed system seems to be useful for many scenarios. The $D\pi$ calculus has primitives to *halt* a location and predicates to test if a location[8] is alive. We have briefly touched upon this issue while talking about the operational semantics rules *r-new-ch* and *r-new-loc* in Section 3.2.3.

To recall, there are two ways in which we can handle creation primitives. The first is the approach taken in the π-calculus. Here channel (or name) creation is not modeled explicitly in the operational semantics but is modeled via a combination of structural equivalence rules for scope restriction and extrusion and a conditional transition rule of the form:

$$\frac{P \rightarrow Q}{(\nu x).P \rightarrow (\nu x).Q}$$

To see what is happening, consider a process that creates a new name and then sends it to another process: $(\nu u)\overline{a}u.P$. The reduction of a term involving such a process would go like this: (assuming $u \notin fn(Q)$)

[8]Or any of its ancestors since $D\pi$ has hierarchical locations.

$$(\nu u)\overline{a}u.P \mid a(x).Q \longrightarrow (\nu u)(\overline{a}u.P \mid a(x).Q) \longrightarrow (\nu u)(P \mid Q_{u/x})$$

The first reduction is obtained via a structural congruence rule that allows us to extend the scope of u since $u \notin fn(Q)$. The second reduction is obtained via a transition rule that allows two processes to communicate over a mutually known name.

Further, via alpha renaming and scope extension as described above, all ν operators can be pulled up to the topmost level. Thus, a probable snapshot of a system is

$$(\nu a)(\nu b)(\overline{a}c.P \mid a(x).\overline{b}x.Q \mid b(x).R)$$

Here, a and b were initially created somewhere within P, Q and R. Then, using appropriate structural equivalence rules, they were pulled up to the top so that they now scope all three processes. However, only the first two processes use a and the last two processes use b. c is a name that is assumed to be known to the first process.

Now, let us consider how we can add a primitive to delete a name. Let us represent the primitive with: $\downarrow (n)$. A simple way to model deletion of a name n would be to annotate the surrounding (νn) to represent that n is now dead:

$$(\nu n)(\downarrow (n).P \mid Q) \longrightarrow (\nu n^{'})(P \mid Q)$$

So, from now on, any process which was initially scoped by n will not be able to use n since there will be no appropriate reduction rule that may be applied. Note that the definition of free and bound names within processes needs to be modified since the $(\nu n')$ should still bind all occurrences of n to prevent the previously bound n from suddenly becoming free. There are other issues that need to be tackled like "how can globally known names be deleted" but the above approach can be elaborated as needed.

On the other hand, consider our channel creation primitive *r-new-ch* :

$$l\left[\alpha[\nu_{ch}(a:\mathcal{C}_i).p] \mid C\right] \longrightarrow l\left[\alpha[p] \mid C \mid a\right] \text{ if } a \notin C \quad \text{r-new-ch}$$

As mentioned, we model channel creation explicitly by creating a new channel in the pool of channels C. Thus, adding a primitive for channel deletion is straightforward. Let $\downarrow_{ch}\ (a:\mathcal{C}_i)$ represent channel deletion.

$$l\left[\alpha[\downarrow_{ch}\ (a:\mathcal{C}_i).p] \mid a \mid C\right] \longrightarrow l\left[\alpha[p] \mid C\right] \quad \text{r-del-ch}$$

Note that in either case, there are common design issues that need to be taken care of like "Can any name be deleted" or "who can delete names" and other issues usually dealt with in distributed garbage collection.

The *transactor* model introduced by Field and Varela in [16] models failures by extending the actor model with notions of state dependencies between actors, explicit modeling of secondary storage through commit and rollback primitives, and the notion of globally consistent states.

3.3.6 Miscellany

We noted earlier that a limited form of tuple communication is possible in basic A^{π} itself. We now elaborate on this. For simplicity, we shall consider only agent to agent communication. We recall that we use the convenience type ζ to stand for any of $\mathcal{N}$, $\mathcal{L}$ and $\mathcal{V}$. We can see that in the current system, there are no problems if we start communicating tuples with values of types ζ. We simply need to make the necessary minor changes to the syntax, the type system and the operational semantics to accommodate tuple communication. However, the interesting case is when we wish to have full fledged tuple communication, i.e., allow the tuples to have values of type $\mathcal{T}_i$ also. Even in this case, relatively minor changes to the type system and operational semantics suffice. If a message contains more than one occurrence of a name of type $\mathcal{T}_i$, then, in essence, we simply need to do what we currently do with all these names; i.e., we ensure that the usage bound on each located channel is respected and so on.

Another issue is whether our syntax is too verbose. We believe that our syntax is not too unwieldy and some of the redundancy and explicit typing makes it simpler to read and easier to understand. However, it is possible to compact the syntax. We do not need to explicitly type names of types ζ. A type inference algorithm can easily infer the type of such a name since it depends only on the syntactic operations that the name is taking part in. (Note that this is possible partly because we have different operators for agent-agent communication and agent-resource communication. Alternatively, we can retain the explicit typing and then use the same operator for both kinds of communication.)

3.4 Examples

We now provide some illustrative examples.

3.4.1 Preliminaries

First, suppose that the A^π rules have been slightly modified so that channel names of bound ∞ can be communicated over channels (indeed, only slight modifications in the type system and operational semantics are needed for this). In such a A^π system, consider a collection of agents that are all co-located in the same location. Suppose that they never use the asynchronous messaging system, never migrate out of that location, never create new locations and only communicate located channels of bound ∞. Then, it is clear that any action in a π-calculus system can be mapped to an action in such a A^π system. Thus the standard data encodings from the π-calculus can be mimicked by such a A^π system. So, from now on, we assume that data types like integers and strings are available. Since data types and basic arithmetic are available, arbitrary computable functions are also available. We use $\mathbb{F}$, $\mathbb{G}$ and $\mathbb{H}$ to represent computable functions with arbitrary arities.

We have already indicated how to obtain tuple communication (see Section 3.3). (For simplicity we shall leave out notation required to ensure correct arities) Tuples are represented by square brackets: $[a, b]$.

In the spirit of the π-calculus, state is obtained via cells. To briefly recall, a cell is represented as a process that is trying to write the cell value to a private

channel. The cell process will usually be blocked since channel communication is synchronous. A process that wishes to read the cell value is composed with the cell. It reads from the private channel and after reading, it writes the same value back. Similarly, a process that wishes to set the cell value is composed with the cell. It reads the current value from the private channel and after reading it, it writes the new value back. With suitable discipline, the above idea can be implemented as a fully functional cell.

For clarity, we shall omit explicitly mentioning the type of output values wherever the type is clear from the context. Also, in addition to the normal agent and resource names we shall also use more descriptive names like *bank*, *seller* and so on. These descriptive names will be used primarily for agents but may also be used for located channels. The context will always illuminate the intended category.

3.4.2 Two useful primitives

We now show how two useful primitives can be implemented in terms of existing constructs. We shall then assume that these primitives are available.

- *Remote channel creation:* Currently channels can only be created locally. However, it is simple to obtain global channel creation by spawning an agent that migrates to the destination location, creates a channel there and then becomes the *nil* agent. So, we shall assume that global channel creation is also available. We use the operator $\nu_{ch}^{rem}(l, a : \mathcal{C}_i)$ to denote creation of channel a with type $\mathcal{C}_i$ at location l. Then, the remote channel creation primitive can implemented as follows:

  $$\alpha\big[\nu_{ch}^{rem}(l, a : \mathcal{C}_n)\ .\ p\big] \equiv \alpha\big[\nu_{agt}\ \beta\big(l :: \nu_{ch}(a : \mathcal{C}_n)\big)\ .\ p\big]$$

- *Receive from:* Currently, only type/pattern matching is allowed when an agent wishes to receive a message. However, it is often convenient to be able to specify "I want to receive a message of type so and so from this particular agent." This construct can be represented by $\beta\big[?_\alpha(a : T)\ .\ p\big]$, which means that agent β is waiting for a message of type T from agent α and will then continue as p.

It is not possible to model this notion as a *general* construct in our system. It can however be modeled as a communication protocol between two participating agents. Suppose agents α and β are communicating via this protocol. α tags every message that it sends to β with its name as an additional tuple component. Now, if β wants to receive a message from α with type T, then it simply invokes $?(m : [T, \mathcal{N}])$. On receiving such a message, it compares the second tuple component with α. If the names do not match, then it simply sends this message out to itself.[9] It keeps doing this till it receives an appropriate message.

Thus, it is possible to implement this primitive as a protocol. For simplicity, we shall simply assume the *receive from* primitive without explicitly showing the protocol interactions.

3.4.3 Digital Cash

We note that until now we have been thinking of located channel names as an implicit representation of resources. However, there is nothing in the calculus itself that enforces this representation. It is merely an interpretation. Alternatively, we can think of a located channel name that is typed $\mathcal{T}_i$ as digital cash with value i. Just like real cash, this can be used to buy (digital) goods. All of it or part of it can be given off to another agent. Thus the communication of the name can be thought of as the flow of digital cash in the system. For this to work, only certain *privileged* agents can create new channels. Creation of new channels is interpreted as petty cash entering the system (after an equivalent amount has been debited from a corresponding electronic account). We have already talked about privileged type environments (see Section 3.3).

We shall now program this scenario. Consider an agent β that is selling some information. It waits for a purchase offer; if the price offered is acceptable, it sends the information out and then collects payment via digital cash. It can now use this cash as it wishes. In our example scenario, it deposits this cash into its bank account.

[9]The protocol thus assumes fairness (at the least).

The behavior of β can be programmed as follows:

$$\beta\Big[*\,?\big(\,[\mathit{Offer},\alpha,n]\,\big)\,.\;\mathbb{F}\big(\,[\mathit{Offer},\alpha,n]\,\big) = \mathit{acceptable}?\;\overline{\alpha}(v)\,.\,?_{\alpha}(M:\mathcal{T}_n)\,.\,\overline{\mathit{bank}}\big([\mathit{deposit},\beta,n,M:\mathcal{T}_n]\big)\;:\;\big(\overline{\alpha}(\mathit{rejected})\big)\Big]$$

Here, $\mathbb{F}$ represents an evaluation function that evaluates the offer. As seen above, β repeatedly does the following: It first receives an offer $[\mathit{offer},\alpha,n]$ where *offer* represents the details of the offer made by α who agrees to pay cash worth n units. If the offer is acceptable, it sends the information v to α and then waits for its payment. Note that it is expecting to receive the same amount as what was promised in the offer. After receiving the payment, it deposits it into the bank. We note that issues like ordered delivery and message loss are assumed to be resolved at a lower middle-ware layer. Likewise issues of trust, etc., are also orthogonal. (For example in the above scenario, what happens if the buyer refuses to pay after receiving the information?)

Consider the buyer γ. Assume that it is transacting with seller β.

$$\gamma\Big[\overline{\mathit{wallet}}(m)\,.\,?_{\mathit{wallet}}(\,\mathit{Cash}:\mathcal{T}_m\,)\,.\;\overline{\beta}\big(\,[\mathit{Offer},\gamma,m]\,\big)\,.\;?_{\beta}(\mathit{res})\,.\,\mathit{res} = \mathit{rejected}?\;\overline{\mathit{wallet}}(\mathit{Cash}:\mathcal{T}_m)\;:\;\overline{\beta}(\mathit{Cash}:\mathcal{T}_m)\Big]$$

It first withdraws some cash from its wallet. Note that cash intended to be of value m is typed as $\mathcal{T}_m$. It then proceeds to make an offer and if the offer is accepted, it pays the cash; otherwise, it puts the cash back in its wallet.

Because of the ν_{agt} construct, extending the above to a scenario with two sellers is straightforward. This time, assume that the buyer γ is transacting with

two sellers β_1 and β_2:

$$
\begin{array}{l}
\gamma\Big[\overline{wallet}(m)\ .\ ?_{wallet}(\ Cash : \mathcal{T}_m\ )\ . \\
\quad \nu_{agt}\ \gamma_1\Big(\overline{\beta_1}(\ [Offer, \gamma_1, m_1]\ )\ . \\
\qquad\qquad ?_{\beta_1}(res)\ .\ res = rejected? \\
\qquad\qquad\qquad \overline{wallet}(Cash : \mathcal{T}_{m_1}) \\
\qquad\qquad\qquad : \\
\qquad\qquad\qquad \overline{\beta}(Cash : \mathcal{T}_{m_1})\ .\ \overline{\gamma}(res)\ \Big)\ . \\
\quad \nu_{agt}\ \gamma_2\Big(\overline{\beta_2}(\ [Offer, \gamma_2, m_1]\ )\ . \\
\qquad\qquad ?_{\beta_2}(res)\ .\ res = rejected? \\
\qquad\qquad\qquad \overline{wallet}(Cash : \mathcal{T}_{m_2}) \\
\qquad\qquad\qquad : \\
\qquad\qquad\qquad \overline{\beta}(Cash : \mathcal{T}_{m_2})\ .\ \overline{\gamma}(res)\ \Big)\ . \\
?_{\gamma_1}(result_1)\ . \\
?_{\gamma_2}(result_2)\ \Big]
\end{array}
$$

This time, the buyer takes out cash worth m units from its wallet. It then spawns two agents, one of which makes an offer of m_1 to the first seller and the other makes an offer of m_2 to the second seller.

Note that the type system ensures that $m_1 + m_2 \leq m$ for the above program to type check. After pursuing their transactions, the two newly spawned agents communicate the result back to the original agent.

Finally, consider the bank account. So far, cash was just circulated in the system. But the bank is where cash is *created* (by debiting an appropriate electronic account that has sufficient funds) and is *destroyed* (by crediting an appropriate electronic account). Recall that cash worth k units is nothing but a located channel that may be used k times. The bank manages the creation of these located channels.

The bank agent can receive messages of two kinds — a deposit instruction and a withdraw instruction. In response to an instruction to deposit digital cash worth n units, it first credits the concerned electronic account with n. Then, the name that represents the petty cash is used n times. Thus now the name can never be

used again. (This property is guaranteed by the type system.) This "means" that cash has now been deposited into the bank account.

Similarly, when it receives a message with an instruction to withdraw digital cash worth n units, it first checks if the concerned electronic account has enough funds. If so, it then debits the account with n and creates new petty cash worth n units. (Recall that digital cash worth n units is nothing but a located channel with a usage bound of n.)

Here is how we can program this scenario. Note that as explained in Section 3.3.3, we need to extend the type system before the bank code can be typed. We assume *bank* is a privileged banking account and $*^k.p$ represents k iterations of p. All agents communicate with this account and pass their names along with each transaction. For an agent named α, we assume that α_{act} is a cell containing the electronic account of α.

$$
\begin{array}{l}
bank\Big[*\,?\big(\,[Transaction, cust, k, amt : \mathcal{T}_k]\,\big)\,. \\
\quad Transaction = deposit? \\
\qquad \overline{act}(\,[cust, increment, k]\,)\,.\,\nu_{agt}\; tempagt1\big(*^k\,\overline{\overline{amt}}(v)\big) \\
\qquad : \\
\qquad nil\,. \\
\quad Transaction = withdraw? \\
\qquad \overline{act}(\,[cust, get, Current]\,)\,. \\
\qquad Current < k? \\
\qquad\qquad \overline{\overline{cust}}(declined) \\
\qquad\qquad : \\
\qquad\qquad \nu_{loc}(l)\,.\,\nu_{ch}^{rem}(l, c : \mathcal{C}_{2k})\,. \\
\qquad\qquad \overline{\overline{cust}}(l[c] : \mathcal{T}_k)\,. \\
\qquad\qquad \nu_{agt}\; tempagt2\big(l :: *^k\; c??(v)\big) \\
\qquad : \\
\qquad nil
\end{array}
$$

The interesting case is the withdrawal. To withdraw amount k, the agent first

creates a new located channel with usage bound $2k$. Half of this capacity, i.e., k is passed back as digital cash. The other half is given to a private agent. This mechanism is necessary because channel communication is synchronous. So, when it is eventually time to destroy the petty cash, there should be two entities and one of them should "write" to the located channel k times and the other should "read" k times. Thus, this procedure encodes the creation and destruction of digital cash in the system.

3.4.4 Fair polling

Consider a web-site that wishes to conduct a poll. The poll is open to visitors who satisfy certain criteria. However, to prevent the poll from being skewed, the web-site wishes to ensure that each visitor vote at most once.

We can model this scenario easily as follows: The polling is done by two agents α and β working together. Consider α:

$$\alpha\Big[*?_{vd}(X:\mathcal{N})\,.\,\nu_{agt}\;tmpagt(q)\,.\,\overline{tmpagt}(l:\mathcal{L})\,.\,\overline{tmpagt}(X:\mathcal{N})\Big]$$

α repeatedly does the following: It first waits to learn about a new visitor to the location. It does this by waiting to hear from vd, which is a visitor daemon that keeps track of visitors. When α learns about the name of a new visitor, it spawns a new agent (*tmpagt*) to handle this visitor. It sends the current location and the name of the visitor to the *tmpagt* and then goes back to its initial behavior. Note that typing the messages makes it possible to correctly communicate the location and name inspite of no ordering in the message delivery system. The newly spawned *tmpagt* sends the survey form to the visitor. If it receives a response and if the response qualifies that visitor to vote, then it creates a new located channel and passes its name with bound 1 to the visitor. This models the requirement that the visitor can vote at most once. If the visitor does use the name to vote, then the result is sent on to the aggregator β. Note that a new channel is being created for each visitor/tmpagt pair, and the *tmpagt* always reads from this channel. If the visitor misuses the channel name by trying to read from it, then it (along with the *tmpagt*) will hang forever since both will be trying to read from the same channel.

The thread q executed by each *tmpagt* is written as:

$$
\begin{array}{l}
q \equiv \\
\quad ?(L : \mathcal{L}) \ . \\
\quad ?(Y : \mathcal{N}) \ . \\
\quad \overline{Y}(SurveyForm) \ . \\
\quad ?_Y(EvaluatedForm) \\
\quad \mathbb{G}(EvaluatedForm) = qualify? \\
\qquad \nu_{ch} \ (a : \mathcal{C}_2) \ . \\
\qquad \overline{Y}(L[a] : \mathcal{T}_1) \ . \\
\qquad a??(Res) \ . \\
\qquad \overline{\beta}(Res) \\
\qquad : \\
\qquad \overline{Y}(sorry)
\end{array}
$$

And finally, here is the vote aggregator β. After receiving the result, it stores it in the data bank for later processing.

$$\beta \Big[* \ ?(Result) \ . \ \overline{data}[add, Result] \Big]$$

3.5 Conclusion and Comparison

In this chapter we proposed a calculus to model distributed mobile computing. Resource access and the usage bounds on accessible resources are statically controlled. We indicated several possible extensions and then provided illustrative examples.

We view the lack of the following abilities as more serious limitations and believe that these are not optional but are necessary in the core A^π itself:

- The ability to control traffic into and out of a location.
- Explicitly enforcing the immobile nature of resources.
- A more powerful type system that has logic to deal with arithmetic expressions involving variables.

In Section 3.3 we indicated how the above concepts may be added to A^π. We shall now consider an A^π in which the above mentioned concepts have been added and compare it with other formalisms.

One major way in which A^π differs from other calculi is that it includes two different kinds of communication mechanisms which are meant to be used for different purposes. One is synchronous and the other is asynchronous and point-to-point. We believe that this is a direct ramification of the more fundamental issue that in an open distributed system, everything cannot be treated uniformly. Certain things *are* different and it is more natural to allow this difference to creep into the calculus instead of trying to encode it using some uniform mechanism (unless the aim is a strictly foundational treatment). Of these two communication mechanisms, synchronous communication is allowed only locally while asynchronous communication is allowed locally as well as globally. This mirrors the fact that implementing synchronous communication spanning multiple locations is very costly and should be avoided. Moreover synchronous communication via a channel between different locations incorrectly models remote communication as an instantaneous enterprise and so should be avoided. In the π-calculus, synchronous as well as asynchronous communication is possible, but unlike the A^π calculus, there is nothing that explicitly constrains the usage of synchronous communication to a location (and there are no locations to begin with). In the distributed π-calculus, $D\pi$, locations are modeled explicitly, but only local synchronous communication is permitted. The usual communication mechanism in mobile ambients and join-calculus is more severely limited since it is local, asynchronous and undirected. Finally in the actor model, only asynchronous point-to-point messaging is present which is good when modeling remote communication but does not capture dedicated high speed communication between two co-located processes.

In A^π, locations are modeled directly and mobility is modeled via agent movement between locations. In particular, as in the π-calculus, mobility is *not* modeled via communication of scoped names. Unlike mobile ambients, location movements are not modeled. This directly rules out an entire class of *mobile computing* scenarios where devices like laptops and PDAs move between different administrative domains.

Further, in difference to $D\pi$, locations are present in a flat space. Richer notions like location nesting, location hierarchies and location failures are not present.

As we saw above, the lack of location movement primitives and location nesting makes A^π unsuitable to model mobile computing amidst different administrative domains. On the other hand, the presence of explicit static resources, a separate communication mechanism between agents and resources and resource usage bounds makes A^π more suitable to model electronic commerce scenarios where mobile agents roam a network and perform certain tasks at different locations. These notions are not present in any of the other formalisms. Further, both resource access and the usage limits of resources are statically constrained via the type system. $D\pi$ has notions of static resource access control but no usage bounds. Linear types are well known in the π-calculus community to model constraints on resource usage.

We believe that we have not just thrown in several known concepts but have rather only included those which mix well with each other and, seen in terms of the communication primitives, have included appropriate constraints so that the concepts may be used in a meaningful way.

CHAPTER 4
CONCLUSION AND FUTURE WORK

In closing, we summarize and point towards future plans. The A^{π} calculus that we constructed in this thesis is a process algebraic model of a distributed system. It models explicitly located mobile agents, migration of agents between locations, location-transparent asynchronous directed communication (local and remote), local resources within locations and local resource access using synchronous communication. As shown by the examples, realistic scenarios can be conveniently modeled in small amounts of code. The type system provides static control over resource access and resource usage bounds.

As noted in the introduction, we view the main import of this work as a vehicle to investigate the application of formal techniques to design distributed systems and prove some of their properties. A common critique of formal techniques is that they are usually not applicable to any realistic scenario. We agree that in spite of steady progress in the field and impressive advances in the computing speedup (which in turn helps in building faster formal methods tools), the current state of the art in the area is not easily applicable to realistic industrial scenarios. However, as discussed in [19], formal methods are already being used extensively in the area of hardware circuit design. We believe that the area of formal methods shows great promise as a better means for mechanized assurance than currently existing techniques like testing and simulation.

Broadly speaking, we can talk of two kinds of properties. One is properties of the system itself. And the other is properties of programs written in this system. An example of the former kind in A^{π} would be type safety modulo usage bounds on resources; i.e., to prove a result that says "In a well typed program all resource usage bounds will be respected". An interesting task for formal tools would be to try to prove this property. An example of the second kind of property, with regards to the digital cash example we provided in the previous chapter, is "In the digital cash implementation, cash is never spuriously created from nothing, but it may sometimes

be lost from the system without getting credited back to some account." (This may happen because of the structure of the *r-m-in-2* rule in the operational semantics.) For properties of the first kind, the Maude system[10] seems to be a good fit. Maude is a very powerful executable specification language based on rewriting logic [14]. Rewriting logic is a natural logic to talk about operational semantics which are usually presented as transition systems themselves. Maude comes with its own model checker, with support for Linear Temporal Logic, and an inductive theorem prover. However, when it comes to properties of the second kind, a more expressive logic like full first order or even temporal logic seems to be a better choice. There are various external theorem provers and proof assistants to deal with these logics. One common problem with most theorem provers is that they can only be used to obtain binary answers. They are of limited use when trying to understand the reason behind that answer. The concept of *denotational proof languages* proposed in Arkoudas [4] and as implemented in Athena[11] provides a solution to this issue. Athena, when hooked up with external first order theorem provers like Vampire [40], may be thought of as an assistant for formal proof discovery and presentation in a naturalistic human readable format. The structure of the proofs in Athena closely resembles human reasoning and this greatly helps in proof comprehension. Besides theorem provers, Athena can also be hooked up to external model checkers. A seamless usage of formal tools in the system design phase to avoid buggy designs has been demonstrated in Arkoudas [5] by combining model checking and theorem proving. Thus, we believe Maude and Athena are appropriate tools to prove both kinds of properties about the A^{π} calculus.

[10] `http://maude.uiuc.cs.edu`

[11] `http://www.cag.csail.mit.edu/~kostas/dpls/athena/`

LITERATURE CITED

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[4] Konstantine Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000.

[5] Konstantine Arkoudas. Specification, abduction, and proof. In *ATVA : 2nd International Symposium on Automated Technology for Verification and Analysis*, National Taiwan University, October 2004.

[6] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[7] Gerard Berry and Gerard Boudol. The Chemical Abstract Machine. In *Seventeenth Annual ACM Symposium onn Principles of Programming Languages*, pages 81–94, San Francisco, California, January 17–19, 1990. ACM Press, New York.

[8] G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, Department of Computer Science, Inria University, May 1992.

[9] L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of System Specification and Computational Structures*, LNCS 1378, pages 140–155. Springer Verlag, 1998.

[10] Luca Cardelli. Mobility and security : Lecture notes for the marktoberdorf summer school. webpage, 1999. http://www.luca.demon.co.uk.

[11] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Types for the ambient calculus. *Inf. Comput.*, 177(2):160–194, 2002.

[12] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 79–92, New York, NY, USA, 1999. ACM Press.

[13] Luca Cardelli, Andrew D. Gordon, and Giorgio Ghelli. Mobility types for mobile ambients. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 230–239, London, UK, 1999. Springer-Verlag.

[14] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[15] W. D. Clinger. *Foundations Of Actor Semantics*. PhD thesis, AI Laboratory, MIT, 1981.

[16] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 195–208, New York, NY, USA, 2005. ACM Press.

[17] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *CONCUR'96: Proceedings of the 7th International Conference on Concurrency Theory*, pages 406–421, 1996.

[18] Cedric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[19] John Harrison. Formal verification at Intel. In *18th Annual IEEE Symposium on Logic in Computer Science (LICS'03)*, 2003.

[20] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Inf. Comput.*, 173(1):82–120, 2002.

[21] C. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. PhD thesis, MIT, 1971.

[22] C. Hewitt. Viewing control structures as patterns of message passing. *Journal of Artificial Intelligence*, 8(3):323–364, September 1977.

[23] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[24] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 133–147, London, UK, 1991. Springer-Verlag.

[25] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the π-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 358–371, New York, NY, USA, 1996. ACM Press.

[26] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 352–364, New York, NY, USA, 2000. ACM Press.

[27] Cedric Lhoussaine. Type inference for a distributed π-calculus. *Sci. Comput. Program.*, 50(1-3):225–251, 2004.

[28] X. Liu and David Walker. A polymorphic type system for the polyadic π-calculus. In *CONCUR'95 : Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 103–116, Berlin, 1995. Springer Verlag.

[29] M.Gaspari. An algebra of actors. Technical Report UBLCS-97-4, Department of Computer Science, University of Bologna(Italy), May 1997.

[30] R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, 1982.

[31] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[32] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1), 1993.

[33] Robin Milner. The polyadic π-calculus: a tutorial. In F.L.Hamer, W.Brauer, and H.Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer Verlag, 1993.

[34] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I-II. *Information and Computation*, 100(1):1–77, 1992.

[35] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Locality based Linda: Programming with explicit localities. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 712–726, London, UK, 1997. Springer-Verlag.

[36] B. Pierce. Foundational calculi for programming languages. In A. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 102, pages 2190–2199. CRC press and ACM, 1997.

[37] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. 6(5):409–454, 1996.

[38] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic π-calculus. *J. ACM*, 47(3):531–584, 2000.

[39] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the π-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[40] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2):91–110, 2002.

[41] James Riely and Matthew Hennessy. A typed language for distributed mobile processes (extended abstract). In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–390, New York, NY, USA, 1998. ACM Press.

[42] D. Sangiorgi and D. Walker. *The π-calculus - A theory of mobile processes*. Cambridge University Press, 2001.

[43] Davide Sangiorgi. From π-calculus to higher-order π-calculus—and back. In *TAPSOFT '93: Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 151–166, London, UK, 1993. Springer-Verlag.

[44] Alan Schmitt and Jean-Bernard Stefani. The kell calculus: A family of higher-order distributed process calculi. *LNCS*, 3267:146–178, January 2005.

[45] C. Talcott. Interaction semantics for components of distributed systems. In *First IFIP workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96)*, Paris, France, March 1996.

[46] P. Thati. Towards an algebraic formulation of actors. Master's thesis, University of Illinois at Urabana-Champaign, 2000.

[47] Robin Toll and Carlos Varela. Mobility and security in worldwide computing. In *Proceedings of the 9th ECOOP Workshop on Mobile Object Systems*, Darmstadt, Germany, July 2003.

[48] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[49] Pawel Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.