Notes on Nominal Calculi for Security and Mobility

Andrew D. Gordon

Microsoft Research

Lecture notes for the FOSAD Summer School 2000, from works co-authored with Martín Abadi, Luca Cardelli, and Giorgio Ghelli

Abstract. There is great interest in applying nominal calculi—computational formalisms that include dynamic name generation—to the problems of programming, specifying, and verifying secure and mobile computations. These notes introduce three nominal calculi—the pi calculus, the spi calculus, and the ambient calculus. We describe some typical techniques, and survey related work.

1 Introduction

Programming a concurrent application is difficult. Deadlocks and race conditions are well known problems in multi-threaded applications on a single machine.

Programming a concurrent application running on a distributed network is more difficult, as we must deal with additional problems such as partial failure of one or more of the host machines. Moreover, if there are untrustworthy hosts on the network, as on the internet, we may need to resort to cryptographic protocols to achieve security, and such protocols are notoriously hard to get right.

Programming a concurrent application running on a network that includes mobile hosts or mobile software is still more difficult. We need to solve the communication problem of supporting reliable interaction between mobile devices or between mobile software agents. We need to solve the security problems induced by untrusted code and untrusted hosts.

These notes introduce an approach to these problems of security and mobility based on three related calculi for concurrency, all of which stress the importance of names. We call these *nominal calculi*. The three calculi are tiny but extremely expressive languages for programming concurrent computations. These calculi have well defined formal semantics upon which sophisticated semantic theories have been constructed. The point of defining the calculi and exploring their theories is to help shed light on the difficulties of programming concurrent, distributed, and mobile computations. The ways in which these calculi and their theories can help include the following. We can program intricate computations—such as protocols for communications or security—within these calculi, and apply their theories directly to try to prove properties or to expose flaws. We can use these calculi as simple settings in which to prototype programming models—such as communication or mobility primitives—that subsequently

R. Focardi and R. Gorrieri (Eds.): FOSAD 2000, LNCS 2171, pp. 262–330, 2001.

can be implemented as new libraries or language extensions in full programming langauges. Similarly, we can develop static checks such as type systems or flow analyses for these simple calculi and subsequently apply them to full languages.

In these notes on nominal calculi, we emphasise the application of equational reasoning and type systems to reasoning about security and mobility. Moreover, we survey other work on implementations and on other formal techniques such as logics and flow analyses.

Pure Names and Nominal Calculi

In his 1989 lecture notes on naming and security in distributed systems, Needham [Nee89] stresses the usefulness of pure names for referring to distributed objects. Needham defines a pure name to be "nothing but a bit pattern that is an identifier, and is only useful for comparing for identity with other bit patterns—which includes looking up in tables in order to find other information". An example of a pure name is the 128-bit GUID (Globally Unique Identifier) that uniquely identifies an interface or an implementation in the COM component model [Box98]. A pure name is atomic. In contrast, an impure name is one with some kind of recognisable structure, such as a file path or a URL containing a path. An impure name does more than simply name a single object. For example, the file name *rmn/animals/pig* may imply the presence of a directory *rmn* and a subdirectory *animals*.

The idea of a pure name is a useful abstraction for referring to many kinds of computational structures, not just distributed objects. All three formalisms described in these notes include an abstract set of pure names and an operator for local generation of fresh, unguessable names. This is what we mean when we say that a formalism is a nominal calculus.

The Pi Calculus—Programming with Names

The pi calculus [MPW92, Mil99, SW01] is a small but extremely expressive programming language. It is the original example of a nominal calculus, and is the archetype for many others. It was originally designed to be a foundation for concurrent computation, in the same way as the λ-calculus is a foundation for sequential computation. First published in the same year as Needham's lecture notes, it places a still greater emphasis on pure names. The pi calculus embodies the view that in principle most, if not all, distributed computation may usefully be explained in terms of exchanges of names on named communication channels.

Programs in the pi calculus are systems of independent, parallel processes that synchronise via message-passing handshakes on named channels. The channels a process knows about determine the communication possibilities of the process. Channels may be *restricted*, so that only certain processes may communicate on them. In this respect the pi calculus is similar to earlier process calculi such as CSP [Hoa85] and CCS [Mil89].

What sets the pi calculus apart from earlier calculi is that the scope of a restriction—the program text in which a channel may be used—may change

during computation. When a process sends a restricted name as a message to a process outside the scope of the restriction, the scope is said to *extrude*, that is, it enlarges to embrace the process receiving the channel. The communication possibilities of a process may change over time; a process may learn the names of new channels via scope extrusion. Thus, a channel is a transferable capability for communication.

A central technical idea of these notes is to use the restriction operator and scope extrusion from the pi calculus as a formal model of the possession and communication of secrets, such as cryptographic keys. These features of the pi calculus and other nominal calculi are essential in our descriptions of security protocols. At the formal level, we can guarantee freshness absolutely by treating a fresh name as a bound variable, distinct from all others. At the implementation level, there are several strategies to guarantee freshness; a common one in a distributed setting is to do so probabilistically by treating a fresh name as a random bitstring of sufficiently many bits to make collisions implausible.

The pi calculus enjoys a broad mathematical theory—including observational equivalences, program logics, and type systems—that addresses the difficulty of programming concurrent applications. Remarkably, a wide variety of data structures—from bits, tuples, and lists, through to objects—and procedural abstractions—such as functions and methods—can all be reduced to interactions on named channels. Hence, the pi calculus is a basis for semantic accounts of functional, imperative, and object-oriented programming, and for the design of several concurrent languages [FG96, PT00, Ode00], as well as other applications. In Part I of these notes, we introduce the pi calculus informally, as a simple programming notation for describing abstract versions of security protocols.

The Spi Calculus—Programming with Cryptography

Security protocols accomplish goals such as establishing the authenticity of one principal to another or preserving the secrecy of information during an interaction. Cryptographic protocols are security protocols implemented over a public network using cryptographic primitives such as encryption, digital signatures, and hashing. Widely-deployed examples include Kerberos and SSL. Designing cryptographic protocols is difficult, in part because they must work correctly even in the presence of an active adversary on the network, who may replay or modify messages. Even if we rule out cryptanalysis, that is, assume perfectly secure cryptographic primitives, cryptographic protocols are notorious for containing flaws, or being brittle in the sense that apparently innocuous changes in operating assumptions may cause failure. For example, Denning and Sacco [DS81] and Lowe [Low96] point out such brittleness in protocols proposed by Needham and Schroeder [NS78].

The spi calculus [AG99] is a version of the pi calculus equipped with abstract cryptographic primitives, in particular, with primitives for perfect encryption and decryption. In this nominal calculus, names represent encryption keys as well as communication channels. The idea is that to analyse a protocol, we begin by modelling it as a spi calculus program. We can then apply techniques

from the theory of the pi calculus such as equational reasoning or type systems to either show the protocol correct or identify a defect. In Part II of these notes, we introduce the spi calculus and explain how to apply equational reasoning to a series of example protocols.

The Ambient Calculus—Programming with Mobility

It is becoming more common for networks to include mobile devices or mobile software. When programming such networks, one area of difficulty is mobility: not so much how to move objects themselves, but how to specify which objects to move. This is a lesson reported by pioneers of mobile computation such as the designers of Telescript [Whi96] or Obliq [Car95]: in those systems, it is easy to move a single object or the whole running computation, but harder to specify a cluster of logically related objects that is to be moved. Another area of difficulty is security: this arises not so much from mobility itself, but from the careless or malicious crossing of administrative domains.

An *ambient* is an abstract collection or group of running processes and objects that functions both as a unit of mobility—of either software and hardware—and as a unit of security—an administrative domain or a security perimeter. An ambient is a bounded place where computation happens, with an inside and an outside. An ambient may contain other ambients, to model related clusters of object or to model hierarchical administrative domains. An ambient has an unforgeable name. An ambient's security rests on the controlled distribution of suitable credentials, or *capabilities*, derived from its name. A capability embodies the right to move a whole running ambient inside another, or the right to move one outside another, or the right to dissolve an ambient boundary.

The ambient calculus [CG00b, Car99] formalizes ambients by adopting the extreme position that everything is an ambient. Its purpose is to provide a formal model for describing mobility, and to be a prototypical programming language for mobile applications. Processes have a spatial structure induced by ambient nesting. Computations are series of re-arrangements of this spatial structures, representing ambient mobility. In this nominal calculus, names are the names of ambients rather than communication channels as in the pi calculus. Ambients are explicit boundaries: in the pi calculus, interaction depends on shared names—parties need to know the same communication channel to interact; in the ambient calculus, interaction depends on shared position—parties need to be inside the same ambient to interact. In this way, the ambient hierarchy regulates who may communicate with who.

In Part III of these notes, we introduce the ambient calculus, show how it can model a programming language for mobile computation with features akin to Telescript, describe a series of type systems for ambients, and show how we can type aspects of mobile computation.

Scope of These Notes

The goal of these notes is to introduce nominal calculi and their applications to security and mobility. The bulk of the notes consists of abridged versions of earlier articles [AG99, CG00b, CG99, CGG00a] concerning aspects of the three calculi we have discussed. Proof techniques and proofs omitted from these notes may be found in the full versions of the original articles.

Many nominal calculi that have been applied to security or mobility are of course not covered. Two prominent examples are the join calculus and the seal calculus. The join calculus [FG96] is a variant of the pi calculus based on asynchronous communications; a distributed implementation [FGL+96] has been used to implement the ambient calculus [FLS00], amongst other things. The seal calculus [VC99] is a calculus of mobile agents, akin to the ambient calculus, but with a richer set of primitive operations; it forms the basis of the JavaSeal platform for mobile agents [BV01].

Part I: The Pi Calculus

This part of the notes introduces the pi calculus as a programming notation for studying security protocols. Section 2 introduces the syntax and informal semantics of the pi calculus. In Section 3 we explain an application of the pi calculus to the study of abstract security protocols. Section 4 ends this part with pointers to some of the main works on the pi calculus.

Since the spi calculus of Part II is in fact an extension of the pi calculus of this part, we postpone formal definitions of operational semantics and equivalence until Part II.

2 Outline of the Pi Calculus

There are in fact several versions of the pi calculus. Here we review the syntax and semantics of a particular version. The differences with other versions are mostly orthogonal to our concerns.

We assume an infinite set of *names*, to be used for communication channels, and an infinite set of *variables*. We let m, n, p, q, and r range over names, and let x, y, and z range over variables. The set of *terms* is defined by the grammar:

Syntax of Terms:

$L, M, N ::=$	terms
n	name
(M, N)	pair
0	zero
$suc(M)$	successor
x	variable

In the standard pi calculus, names are the only terms. For convenience we have added constructs for pairing and numbers, (M, N), 0, and $suc(M)$. We have also distinguished variables from names.

The set of *processes* is defined by the grammar:

Syntax of Processes:

$P, Q, R ::=$	processes
$\overline{M}\langle N\rangle.P$	output
$M(x).P$	input
$P \mid Q$	composition
$(\nu n)P$	restriction
$!P$	replication
$[M \; is \; N] \; P$	match
$\mathbf{0}$	nil
$let \; (x, y) = M \; in \; P$	pair splitting
$case \; M \; of \; 0 : P \; suc(x) : Q$	integer case

In $(\nu n)P$, the name n is bound in P. In $M(x).P$, the variable x is bound in P. In $let \; (x, y) = M \; in \; P$, the variables x and y are bound in P. In $case \; M \; of \; 0 : P \; suc(x) : Q$, the variable x is bound in the second branch, Q. We write $P\{M\}x$ for the outcome of replacing each free occurrence of x in process P with the term M, and identify processes up to renaming of bound variables and names. We adopt the abbreviation $\overline{M}\langle N\rangle$ for $\overline{M}\langle N\rangle.\mathbf{0}$.

Intuitively, the constructs of the pi calculus have the following meanings:

- The basic computation and synchronisation mechanism in the pi calculus is *interaction*, in which a term N is communicated from an output process to an input process via a named channel, m.
  - An *output process* $\overline{m}\langle N\rangle.P$ is ready to output on channel m. If an interaction occurs, term N is communicated on m and then process P runs.
  - An *input process* $m(x).P$ is ready to input from channel m. If an interaction occurs in which N is communicated on m, then process $P\{N\}x$ runs.

  (The general forms $\overline{M}\langle N\rangle.P$ and $M(x).P$ of output and input allow for the channel to be an arbitrary term M. The only useful cases are for M to be a name, or a variable that gets instantiated to a name.)
- A *composition* $P \mid Q$ behaves as processes P and Q running in parallel. Each may interact with the other on channels known to both, or with the outside world, independently of the other.
- A *restriction* $(\nu n)P$ is a process that makes a new, private name n, which may occur in P, and then behaves as P.
- A *replication* $!P$ behaves as an infinite number of copies of P running in parallel.
- A *match* $[M \; is \; N] \; P$ behaves as P provided that terms M and N are the same; otherwise it is stuck, that is, it does nothing.

– The *nil* process **0** does nothing.

Since we added pairs and integers, we have two new process forms:

– A *pair splitting* process $let\ (x, y) = M\ in\ P$ behaves as $P\{N\}x\{L\}y$ if term M is the pair (N, L), and otherwise it is stuck.
– An *integer case* process $case\ M\ of\ 0 : P\ suc(x) : Q$ behaves as P if term M is 0, as $Q\{N\}x$ if M is $suc(N)$, and otherwise is stuck.

We write $P \simeq Q$ to mean that the behaviours of the processes P and Q are indistinguishable. In other words, a third process R cannot distinguish running in parallel with P from running in parallel with Q; as far as R can tell, P and Q have the same properties (more precisely, the same safety properties). We define the relation $\simeq$ in Part II as a form of testing equivalence. For now, it suffices to understand $\simeq$ informally.

3 Security Examples Using Restricted Channels

Next we show how to express some abstract security protocols in the pi calculus. In security protocols, it is common to find channels on which only a given set of principals is allowed to send data or listen. The set of principals may expand in the course of a protocol run, for example as the result of channel establishment. Remarkably, it is easy to model this property of channels in the pi calculus, via the restriction operation; the expansion of the set of principals that can access a channel corresponds to scope extrusion.

3.1 A First Example

Our first example is extremely basic. In this example, there are two principals A and B that share a channel, c_{AB}; only A and B can send data or listen on this channel. The protocol is simply that A uses c_{AB} for sending a single message M to B. In informal notation, we may write this protocol as follows:

$$\text{Message 1 } A \rightarrow B : M \text{ on } c_{AB}$$

A first pi calculus description of this protocol is:

$$\begin{aligned} A(M) &\triangleq \overline{c_{AB}}\langle M\rangle \\ B &\triangleq c_{AB}(x).\mathbf{0} \\ Inst(M) &\triangleq (\nu c_{AB})(A(M) \mid B) \end{aligned}$$

The processes $A(M)$ and B describe the two principals, and $Inst(M)$ describes (one instance of) the whole protocol. The channel c_{AB} is restricted; intuitively, this achieves the effect that only A and B have access to c_{AB}.

In these definitions, $A(M)$ and $Inst(M)$ are processes parameterised by M. More formally, we view A and $Inst$ as functions that map terms to processes,

called abstractions, and treat the M's on the left of $\triangleq$ as bound parameters. Abstractions can of course be instantiated (applied); for example, the instantiation $A(0)$ yields $\overline{c_{AB}}\langle 0\rangle$. The standard rules of substitution govern application, forbidding parameter captures; for example, expanding $\mathit{Inst}(c_{AB})$ would require a renaming of the bound occurrence of c_{AB} in the definition of Inst.

The first pi calculus description of the protocol may seem a little futile because, according to it, B does nothing with its input. A more useful and general description says that B runs a process F with its input. We revise our definitions as follows:

$$\begin{aligned} A(M) &\triangleq \overline{c_{AB}}\langle M\rangle \\ B &\triangleq c_{AB}(x).F(x) \\ \mathit{Inst}(M) &\triangleq (\nu c_{AB})(A(M) \mid B) \end{aligned}$$

Informally, $F(x)$ is simply the result of applying F to x. More formally, F is an abstraction, and $F(x)$ is an instantiation of the abstraction. We adopt the convention that the bound parameters of the protocol (in this case, M, c_{AB}, and x) cannot occur free in F.

This protocol has two important properties:

- Authenticity (or integrity): B always applies F to the message M that A sends; an attacker cannot cause B to apply F to some other message.
- Secrecy: The message M cannot be read in transit from A to B: if F does not reveal M, then the whole protocol does not reveal M.

The secrecy property can be stated in terms of equivalences: if $F(M) \simeq F(M')$, for any M, M', then $\mathit{Inst}(M) \simeq \mathit{Inst}(M')$. This means that if $F(M)$ is indistinguishable from $F(M')$, then the protocol with message M is indistinguishable from the protocol with message M'.

There are many sensible ways of formalising the authenticity property. In particular, it may be possible to use notions of refinement or a suitable program logic. However, we choose to write authenticity as an equivalence, for economy. This equivalence compares the protocol with another protocol. Our intent is that the latter protocol serves as a specification. In this case, the specification is:

$$\begin{aligned} A(M) &\triangleq \overline{c_{AB}}\langle M\rangle \\ B_{spec}(M) &\triangleq c_{AB}(x).F(M) \\ \mathit{Inst}_{spec}(M) &\triangleq (\nu c_{AB})(A(M) \mid B_{spec}(M)) \end{aligned}$$

The principal A is as usual, but the principal B is replaced with a variant $B_{spec}(M)$; this variant receives an input from A and then acts like B when B receives M. We may say that $B_{spec}(M)$ is a "magical" version of B that knows the message M sent by A, and similarly Inst_{spec} is a "magical" version of Inst.

Although the specification and the protocol are similar in structure, the specification is more evidently "correct" than the protocol. Therefore, we take the

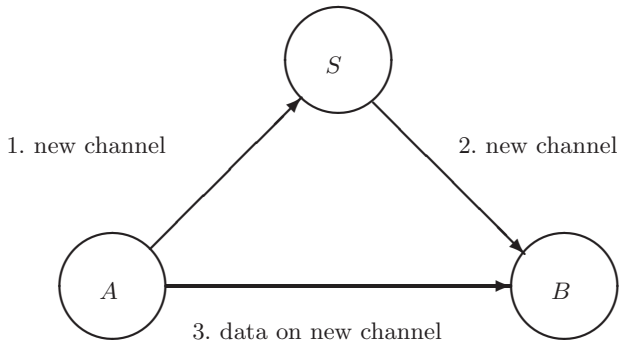

Fig. 1. Structure of the Wide Mouthed Frog protocol

following equivalence as our authenticity property: $Inst(M) \simeq Inst_{spec}(M)$, for any M.

In summary, we have:

Authenticity: $Inst(M) \simeq Inst_{spec}(M)$, for any M.

Secrecy: $Inst(M) \simeq Inst(M')$ if $F(M) \simeq F(M')$, for any M, M'.

Each of these equivalences means that two processes being equated are indistinguishable, even when an active attacker is their environment. Neither of these equivalences would hold without the restriction of channel c_{AB}.

3.2 An Example with Channel Establishment

A more interesting variant of our first example is obtained by adding a channel establishment phase. In this phase, before communication of data, the principals A and B obtain a new channel with the help of a server S.

There are many different ways of establishing a channel, even at the abstract level at which we work here. The one we describe is inspired by the Wide Mouthed Frog protocol [BAN89], which has the basic structure shown in Figure 1.

We consider an abstract and simplified version of the Wide Mouthed Frog protocol. Our version is abstract in that we deal with channels instead of keys; it is simplified in that channel establishment and data communication happen only once (so there is no need for timestamps). In the next section we show how to treat keys and how to allow many instances of the protocol, with an arbitrary number of messages.

Informally, our version is:

$$\begin{array}{l}\text{Message 1 } A \to S : c_{AB} \text{ on } c_{AS}\\ \text{Message 2 } S \to B : c_{AB} \text{ on } c_{SB}\\ \text{Message 3 } A \to B : M \quad \text{on } c_{AB}\end{array}$$

Here c_{AS} is a channel that A and S share initially, c_{SB} is a channel that S and B share initially, and c_{AB} is a channel that A creates for communication with B. After passing the channel c_{AB} to B through S, A sends a message M on c_{AB}. Note that S does not use the channel, but only transmits it.

In the pi calculus, we formulate this protocol as follows:

$$\begin{aligned} A(M) &\triangleq (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle\\ S &\triangleq c_{AS}(x).\overline{c_{SB}}\langle x\rangle\\ B &\triangleq c_{SB}(x).x(y).F(y)\\ \mathit{Inst}(M) &\triangleq (\nu c_{AS})(\nu c_{SB})(A(M) \mid S \mid B)\end{aligned}$$

Here we write $F(y)$ to represent what B does with the message y that it receives, as in the previous example. The restrictions on the channels c_{AS}, c_{SB}, and c_{AB} reflect the expected privacy guarantees for these channels. The most salient new feature of this specification is the use of scope extrusion: A generates a fresh channel c_{AB}, and then sends it out of scope to B via S. We could not have written this description in formalisms such as CCS or CSP; the use of the pi calculus is important.

For discussing authenticity, we introduce the following specification:

$$\begin{aligned} A(M) &\triangleq (\nu c_{AB})\overline{c_{AS}}\langle c_{AB}\rangle.\overline{c_{AB}}\langle M\rangle\\ S &\triangleq c_{AS}(x).\overline{c_{SB}}\langle x\rangle\\ B_{spec}(M) &\triangleq c_{SB}(x).x(y).F(M)\\ \mathit{Inst}_{spec}(M) &\triangleq (\nu c_{AS})(\nu c_{SB})(A(M) \mid S \mid B_{spec}(M))\end{aligned}$$

According to this specification, the message M is communicated "magically": the process F is applied to the message M that A sends independently of whatever happens during the rest of the protocol run.

We obtain the following authenticity and secrecy properties:

Authenticity: $\mathit{Inst}(M) \simeq \mathit{Inst}_{spec}(M)$, for any M.

Secrecy: $\mathit{Inst}(M) \simeq \mathit{Inst}(M')$ if $F(M) \simeq F(M')$, for any M, M'.

Again, these properties hold because of the scoping rules of the pi calculus.

4 Discussion: The Pi Calculus

In this part, we have briefly and informally introduced the pi calculus as a notation for describing and specifying security protocols. In the next part, we extend these ideas to apply to cryptographic protocols.

Starting with the original presentation [MPW92] there is by now an extensive literature on the pi calculus, covered, for instance, by introductory [Mil99] and advanced [SW01] textbooks. A good deal of the theory of the pi calculus concerns equational reasoning; two important works are on testing equivalence [BN95] and barbed bisimulation [MS92]. There are several works on logic [MPW93] and model checking [Dam96] for the pi calculus.

The study of type systems for the pi calculus is a booming research area. We cite just three out of many papers. The simplest type system for the pi calculus is a system of channel sorts proposed by Milner [Mil99]. Pierce and Sangiorgi [PS96] develop a more advanced system supporting subtyping. Igarashi and Kobayashi [IK01] propose a generic framework in which to understand a variety of previous systems.

Most versions of the pi calculus allow only passive data such as names to be transmitted on channels. Sangiorgi's higher-order pi calculus [SW01] is a variant in which processes may be transmitted on channels. Dam [Dam98] uses a second-order pi calculus to study security protocols.

Part II: The Spi Calculus

The spi calculus is an extension of the pi calculus with cryptographic primitives. It is designed for the description and analysis of security protocols, such as those for authentication and for electronic commerce. These protocols rely on cryptography and on communication channels with properties like authenticity and privacy. Accordingly, cryptographic operations and communication through channels are the main ingredients of the spi calculus.

In Part I of these notes, we used the pi calculus (without extension) for describing protocols at an abstract level. The pi calculus primitives for channels are simple but powerful. Channels can be created and passed, for example from authentication servers to clients. The scoping rules of the pi calculus guarantee that the environment of a protocol (the attacker) cannot access a channel that it is not explicitly given; scoping is thus the basis of security. In sum, the pi calculus appears as a fairly convenient calculus of protocols for secure communication.

However, the pi calculus does not express the cryptographic operations that are commonly used for implementing channels in distributed systems: it does not include any constructs for encryption and decryption, and these do not seem easy to represent. Since the use of cryptography is notoriously error-prone, we prefer not to abstract it away. We define the spi calculus in order to permit an explicit representation of the use of cryptography in protocols.

There are many other notations for describing security protocols. Some, which have long been used in the authentication literature, have a fairly clear connection to the intended implementations of those protocols (e.g., [NS78, Lie93]). Their main shortcoming is that they do not provide a precise and solid basis for reasoning about protocols. Other notations (e.g., [BAN89]) are more formal, but their relation to implementations may be more tenuous or subtle. The

spi calculus is a middle ground: it is directly executable and it has a precise semantics.

Because the semantics of the spi calculus is not only precise but intelligible, the spi calculus provides a setting for analysing protocols. Specifically, we can express security guarantees as equivalences between spi calculus processes. For example, we can say that a protocol keeps secret a piece of data X by stating that the protocol with X is equivalent to the protocol with X', for any X'. Here, equivalence means equivalence in the eyes of an arbitrary environment. The environment can interact with the protocol, perhaps attempting to create confusion between different messages or sessions. This definition of equivalence yields the desired properties for our security applications. Moreover, in our experience, equivalence is not too hard to prove.

Although the definition of equivalence makes reference to the environment, we do not need to give a model of the environment explicitly. This is one of the main advantages of our approach. Writing such a model can be tedious and can lead to new arbitrariness and error. In particular, it is always difficult to express that the environment can invent random numbers but is not lucky enough to guess the random secrets on which a protocol depends. We resolve this conflict by letting the environment be an arbitrary spi calculus process.

Our approach has some similarities with other recent approaches for reasoning about protocols. Like work based on temporal logics or process algebras (e.g., [FG94, GM95, Low96, Sch96a]), our method builds on a standard concurrency formalism; this has obvious advantages but it also implies that our method is less intuitive than some based on ad hoc formalisms (e.g., [BAN89]). As in some modal logics (e.g., [ABLP93, LABW92]), we emphasise reasoning about channels. As in state-transition models (e.g., [DY83, MCF87, Kem89, Mea92]), we are interested in characterising the knowledge of an environment. The unique features of our approach are its reliance on the powerful scoping constructs of the pi calculus; the radical definition of the environment as an arbitrary spi calculus process; and the representation of security properties, both integrity and secrecy, as equivalences.

Our model of protocols is simpler, but poorer, than some models developed for informal mathematical arguments because the spi calculus does not include any notion of probability or complexity (cf. [BR95]). It would be interesting to bridge the gap between the spi calculus and those models, perhaps by giving a probabilistic interpretation for our results. Recent work [LMMS98, AR00] makes progress in this direction.

Remainder of Part II

The remainder of this part is organised as follows. Section 5 extends the pi calculus with primitives for shared-key cryptography. Section 6 describes a series of protocol examples in the spi calculus. Sections 7 defines the formal semantics of the spi calculus. Section 8 discusses how to add primitives for hashing and public-key cryptography to the pi calculus. Finally, Section 9 offers some conclusions and discusses related work.

5 The Spi Calculus with Shared-Key Cryptography

Just as there are several versions of the pi calculus, there are several versions of the spi calculus. These differ in particular in what cryptographic constructs they include. In this section we introduce a relatively simple spi calculus, namely the pi calculus extended with primitives for shared-key cryptography. We then write several protocols that use shared-key cryptography in this calculus.

Throughout these notes, we often refer to the calculus presented in this section as "the" spi calculus; but we define other versions of the spi calculus in Section 8.

The syntax of the spi calculus is an extension of that of the pi calculus. In order to represent encrypted messages, we add a clause to the syntax of terms:

Syntax of Terms:

$L, M, N ::=$	terms
$\ldots$	as in Section 2
$\{M\}_N$	shared-key encryption

In order to represent decryption, we add a clause to the syntax of processes:

Syntax of Processes:

$P, Q ::=$	processes
$\ldots$	as in Section 2
case L *of* $\{x\}_N$ *in* P	shared-key decryption

The variable x is bound in P.

Intuitively, the meaning of the new constructs is as follows:

- The term $\{M\}_N$ represents the ciphertext obtained by encrypting the term M under the key N using a shared-key cryptosystem such as DES [DES77].
- The process *case* L *of* $\{x\}_N$ *in* P attempts to decrypt the term L with the key N. If L is a ciphertext of the form $\{M\}_N$, then the process behaves as $P\{M\}x$. Otherwise the process is stuck.

Implicit in this definition are some standard but significant assumptions about cryptography:

- The only way to decrypt an encrypted packet is to know the corresponding key.
- An encrypted packet does not reveal the key that was used to encrypt it.
- There is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected key.

It is not assumed that all messages contain information that allows each principal to recognise its own messages (cf. [BAN89]).

The semantics of the spi calculus can be formalised in much the same way as the semantics of the pi calculus. We carry out this formalisation in Section 7.

Again, we write $P \simeq Q$ to mean that the behaviours of the processes P and Q are indistinguishable. The notion of indistinguishability is complicated by the presence of cryptography. As an example of these complications, consider the following process:

$$P(M) \triangleq (\nu K)\overline{c}\langle\{M\}_K\rangle$$

This process simply sends M under a new key K on a public channel c; the key K is not transmitted. Intuitively, we would like to equate $P(M)$ and $P(M')$, for any M and M', because an observer cannot discover K and hence cannot tell whether M or M' is sent under K. On the other hand, $P(M)$ and $P(M')$ are clearly different, since they transmit different messages on c. Our equivalence $\simeq$ is coarse-grained enough to equate $P(M)$ and $P(M')$.

6 Security Examples Using Shared-Key Cryptography

The spi calculus enables more detailed descriptions of security protocols than the pi calculus. While the pi calculus enables the representation of channels, the spi calculus also enables the representation of the channel implementations in terms of cryptography. In this section we show a few example cryptographic protocols.

As in the pi calculus, scoping is the basis of security in the spi calculus. In particular, restriction can be used to model the creation of fresh, unguessable cryptographic keys. Restriction can also be used to model the creation of fresh nonces of the sort used in challenge-response exchanges.

Security properties can still be expressed as equivalences, although the notion of equivalence is more delicate, as we have discussed.

6.1 A First Cryptographic Example

Our first example is a cryptographic version of the example of Section 3.1. We consider two principals A and B that share a key K_{AB}; in addition, we assume there is a public channel c_{AB} that A and B can use for communication, but which is in no way secure. The protocol is simply that A sends a message M under K_{AB} to B, on c_{AB}.

Informally, we write this protocol as follows:

$$\text{Message 1 } A \rightarrow B : \{M\}_{K_{AB}} \text{ on } c_{AB}$$

In the spi calculus, we write:

$$\begin{aligned} A(M) &\triangleq \overline{c_{AB}}\langle\{M\}_{K_{AB}}\rangle \\ B &\triangleq c_{AB}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AB}}\ \mathit{in}\ F(y) \\ \mathit{Inst}(M) &\triangleq (\nu K_{AB})(A(M) \mid B) \end{aligned}$$

According to this definition, A sends $\{M\}_{K_{AB}}$ on c_{AB} while B listens for a message on c_{AB}. Given such a message, B attempts to decrypt it using K_{AB}; if this decryption succeeds, B applies F to the result. The assumption that A and B share K_{AB} gives rise to the restriction on K_{AB}, which is syntactically legal and meaningful although K_{AB} is not used as a channel. On the other hand, c_{AB} is not restricted, since it is a public channel. Other principals may send messages on c_{AB}, so B may attempt to decrypt a message not encrypted under K_{AB}; in that case, the protocol will get stuck. We are not concerned about this possibility, but it would be easy enough to avoid it by writing a slightly more elaborate program for B.

We use the following specification:

$$\begin{aligned} A(M) &\triangleq \overline{c_{AB}}\langle\{M\}_{K_{AB}}\rangle \\ B_{spec}(M) &\triangleq c_{AB}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AB}}\ \mathit{in}\ F(M) \\ \mathit{Inst}_{spec}(M) &\triangleq (\nu K_{AB})(A(M) \mid B_{spec}(M)) \end{aligned}$$

and we obtain the properties:

Authenticity: $\mathit{Inst}(M) \simeq \mathit{Inst}_{spec}(M)$, for any M.

Secrecy: $\mathit{Inst}(M) \simeq \mathit{Inst}(M')$ if $F(M) \simeq F(M')$, for any M, M'.

Intuitively, authenticity holds even if the key K_{AB} is somehow compromised after its use. Many factors can contribute to key compromise, for example incompetence on the part of protocol participants, and malice and brute force on the part of attackers. We cannot model all these factors, but we can model deliberate key publication, which is in a sense the most extreme of them. It suffices to make a small change in the definitions of B and B_{spec}, so that they send K_{AB} on a public channel after receiving $\{M\}_{K_{AB}}$. This change preserves the authenticity equation, but clearly not the secrecy equation.

6.2 An Example with Key Establishment

In cryptographic protocols, the establishment of new channels often means the exchange of new keys. There are many methods (most of them flawed) for key exchange. The following example is the cryptographic version of that of Section 3.2; it uses a simplified (one-shot) form of the Wide Mouthed Frog key exchange.

In the Wide Mouthed Frog protocol, the principals A and B share keys K_{AS} and K_{SB} respectively with a server S. When A and B want to communicate securely, A creates a new key K_{AB}, sends it to the server under K_{AS}, and the server forwards it to B under K_{SB}. All communication being protected by encryption, it can happen through public channels, which we write c_{AS}, c_{SB},

and c_{AB}. Informally, a simplified version of this protocol is:

$$\begin{array}{l}\text{Message 1 } A \to S : \{K_{AB}\}_{K_{AS}} \text{ on } c_{AS}\\ \text{Message 2 } S \to B : \{K_{AB}\}_{K_{SB}} \text{ on } c_{SB}\\ \text{Message 3 } A \to B : \{M\}_{K_{AB}} \quad \text{on } c_{AB}\end{array}$$

In the spi calculus, we can express this message sequence as follows:

$$\begin{aligned} A(M) &\triangleq (\nu K_{AB})(\overline{c_{AS}}\langle\{K_{AB}\}_{K_{AS}}\rangle.\overline{c_{AB}}\langle\{M\}_{K_{AB}}\rangle)\\ S &\triangleq c_{AS}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AS}}\ \mathit{in}\ \overline{c_{SB}}\langle\{y\}_{K_{SB}}\rangle\\ B &\triangleq c_{SB}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{SB}}\ \mathit{in}\\ &\qquad c_{AB}(z).\mathit{case}\ z\ \mathit{of}\ \{w\}_y\ \mathit{in}\ F(w)\\ \mathit{Inst}(M) &\triangleq (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B)\end{aligned}$$

where $F(w)$ is a process representing the rest of the behaviour of B upon receiving a message w. Notice the essential use of scope extrusion: A generates the key K_{AB} and sends it out of scope to B via S.

In the usual pattern, we introduce a specification for discussing authenticity:

$$\begin{aligned} A(M) &\triangleq (\nu K_{AB})(\overline{c_{AS}}\langle\{K_{AB}\}_{K_{AS}}\rangle.\overline{c_{AB}}\langle\{M\}_{K_{AB}}\rangle)\\ S &\triangleq c_{AS}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{AS}}\ \mathit{in}\ \overline{c_{SB}}\langle\{y\}_{K_{SB}}\rangle\\ B_{spec}(M) &\triangleq c_{SB}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{SB}}\ \mathit{in}\\ &\qquad c_{AB}(z).\mathit{case}\ z\ \mathit{of}\ \{w\}_y\ \mathit{in}\ F(M)\\ \mathit{Inst}_{spec}(M) &\triangleq (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B_{spec}(M))\end{aligned}$$

One may be concerned about the apparent complexity of this specification. On the other hand, despite its complexity, the specification is still more evidently "correct" than the protocol. In particular, it is still evident that $B_{spec}(M)$ applies F to the data M from A, rather than to some other message chosen as the result of error or attack.

We obtain the usual properties of authenticity and secrecy:

Authenticity: $\mathit{Inst}(M) \simeq \mathit{Inst}_{spec}(M)$, for any M.

Secrecy: $\mathit{Inst}(M) \simeq \mathit{Inst}(M')$ if $F(M) \simeq F(M')$, for any M, M'.

6.3 A Complete Authentication Example (with a Flaw)

In the examples discussed so far, channel establishment and data communication happen only once. As we demonstrate now, it is a simple matter of programming to remove this restriction and to represent more sophisticated examples with many sessions between many principals. However, as the intricacy of our examples increases, so does the opportunity for error. This should not be construed as a limitation of our approach, but rather as the sign of an intrinsic

difficulty: many of the mistakes in authentication protocols arise from confusion between sessions.

We consider a system with a server S and n other principals. We use the terms $suc(0)$, $suc(suc(0))$, ..., which we abbreviate to $\underline{1}$, $\underline{2}$, ..., as the names of these other principals. We assume that each principal has an input channel; these input channels are public and have the names c_1, c_2, ..., c_n and c_S. We also assume that the server shares a pair of keys with each other principal, one key for each direction: principal i uses key K_{iS} to send to S and key K_{Si} to receive from S, for $1 \leq i \leq n$.

We extend our standard example to this system of $n+1$ principals, with the following message sequence:

$$\begin{array}{ll} \text{Message 1 } A \rightarrow S : A, \{B, K_{AB}\}_{K_{AS}} & \text{on } c_S \\ \text{Message 2 } S \rightarrow B : \{A, K_{AB}\}_{K_{SB}} & \text{on } c_B \\ \text{Message 3 } A \rightarrow B : A, \{M\}_{K_{AB}} & \text{on } c_B \end{array}$$

Here A and B range over the n principals. The names A and B appear in messages in order to avoid ambiguity; when these names appear in clear, they function as hints that help the recipient choose the appropriate key for decryption of the rest of the message. The intent is that the protocol can be used by any pair of principals, arbitrarily often; concurrent runs are allowed. As it stands, the protocol has obvious flaws; we discuss it in order to explain our method for representing it in the spi calculus.

In our spi calculus representation, we use several convenient abbreviations. Firstly, we rely on pair splitting on input and on decryption:

$$\begin{array}{rl} c(x_1, x_2).P & \triangleq c(y).\mathit{let}\ (x_1, x_2) = y\ \mathit{in}\ P \\ \mathit{case}\ L\ \mathit{of}\ \{x_1, x_2\}_N\ \mathit{in}\ P & \triangleq \mathit{case}\ L\ \mathit{of}\ \{y\}_N\ \mathit{in} \\ & \quad \mathit{let}\ (x_1, x_2) = y\ \mathit{in}\ P \end{array}$$

where variable y is fresh. Secondly, we need the standard notation for the composition of a finite set of processes. Given a finite family of processes $P_1, \ldots, P_k$, we let $\prod_{i \in 1..k} P_i$ be their k-way composition $P_1 \mid \cdots \mid P_k$. Finally, we omit the inner parentheses from an encrypted pair of the form $\{(N, N')\}_{N''}$, and simply write $\{N, N'\}_{N''}$, as is common in informal descriptions.

Informally, an instance of the protocol is determined by a choice of parties (who is A and who is B) and by the message sent after key establishment. More formally, an instance I is a triple (i, j, M) such that i and j are principals and M is a message. We say that i is the source address and j the destination address of the instance. Moreover, we assume that there is an abstraction F representing the behaviour of any principal after receipt of Message 3 of the protocol. For an instance (i, j, M) that runs as intended, the argument to F is the triple $(\underline{i}, \underline{j}, M)$.

Given an instance (i, j, M), the following process corresponds to the role of A:

$$\mathit{Send}(i, j, M) \triangleq (\nu K)(\overline{c_S}\langle(\underline{i}, \{\underline{j}, K\}_{K_{iS}})\rangle \mid \overline{c_j}\langle(\underline{i}, \{M\}_K)\rangle)$$

The sending process creates a key K and sends it to the server, along with the names $\underline{i}$ and $\underline{j}$ of the principals of the instance. The sending process also

sends M under K, along with its name $\underline{i}$. We have put the two messages in parallel, somewhat arbitrarily; putting them in sequence would have much the same effect.

The following process corresponds to the role of B for principal j:

$$\begin{array}{l} \mathit{Recv}(j) \triangleq c_j(y_{cipher}).\mathit{case}\ y_{cipher}\ \mathit{of}\ \{x_A, x_{key}\}_{K_{Sj}}\ \mathit{in} \\ \qquad\quad c_j(z_A, z_{cipher}).[x_A\ \mathit{is}\ z_A] \\ \qquad\quad \mathit{case}\ z_{cipher}\ \mathit{of}\ \{z_{plain}\}_{x_{key}}\ \mathit{in}\ F(x_A, \underline{j}, z_{plain}) \end{array}$$

The receiving process waits for a message y_{cipher} from the server, extracts a key x_{key} from this message, then waits for a message z_{cipher} under this key, and finally applies F to the name x_A of the presumed sender, to its own name $\underline{j}$, and to the contents z_{plain} of the message. The variables x_A and z_A are both intended as the name of the sending process, so they are expected to match.

The server S is the same for all instances:

$$\begin{array}{l} S \triangleq c_S(x_A, x_{cipher}). \\ \qquad \prod_{i\in 1..n}[x_A\ \mathit{is}\ \underline{i}]\ \mathit{case}\ x_{cipher}\ \mathit{of}\ \{x_B, x_{key}\}_{K_{iS}}\ \mathit{in} \\ \qquad \prod_{j\in 1..n}[x_B\ \mathit{is}\ \underline{j}]\ \overline{c_j}\langle\{x_A, x_{key}\}_{K_{Sj}}\rangle \end{array}$$

The variable x_A is intended as the name of the sending process, x_B as the name of the receiving process, x_{key} as the new key, and x_{cipher} as the encrypted part of the first message of the protocol. In the code for the server, we program an n-way branch on the name x_A by using a parallel composition of processes indexed by $i \in 1..n$. We also program an n-way branch on the name x_B, similarly. (This casual use of multiple threads is characteristic of the pi calculus; in practice the branch could be implemented more efficiently, but here we are interested only in the behaviour of the server, not in its efficient implementation.)

Finally we define a whole system, parameterised on a list of instances:

$$\begin{array}{l} \mathit{Sys}(I_1, \ldots, I_m) \triangleq (\nu \boldsymbol{K_{iS}})(\nu \boldsymbol{K_{Sj}}) \\ \qquad\qquad\qquad (\mathit{Send}(I_1) \mid \cdots \mid \mathit{Send}(I_m) \mid \\ \qquad\qquad\qquad !S \mid \\ \qquad\qquad\qquad !\mathit{Recv}(1) \mid \cdots \mid !\mathit{Recv}(n)) \end{array}$$

where $(\nu \boldsymbol{K_{iS}})(\nu \boldsymbol{K_{Sj}})$ stands for:

$$(\nu K_{1S}) \ldots (\nu K_{nS})(\nu K_{S1}) \ldots (\nu K_{Sn})$$

The expression $\mathit{Sys}(I_1, \ldots, I_m)$ represents a system with m instances of the protocol. The server is replicated; in addition, the replication of the receiving processes means that each principal is willing to play the role of receiver in any number of runs of the protocol in parallel. Thus, any two runs of the protocol can be simultaneous, even if they involve the same principals.

As before, we write a specification by modifying the protocol. For this specification, we revise the sending and the receiving processes, but not the server:

$$
\begin{aligned}
Send_{spec}(i,j,M) &\triangleq (\nu p)(Send(i,j,p) \mid p(x).F(\underline{i},\underline{j},M)) \\
Recv_{spec}(j) &\triangleq c_j(y_{cipher}). \\
&\quad\ case\ y_{cipher}\ of\ \{x_A, x_{key}\}_{K_{Sj}}\ in \\
&\quad\ c_j(z_A, z_{cipher}).[x_A\ is\ z_A] \\
&\quad\ case\ z_{cipher}\ of\ \{z_{plain}\}_{x_{key}}\ in \\
&\quad\ \overline{z_{plain}}\langle * \rangle \\
Sys_{spec}(I_1,\ldots,I_m) &\triangleq (\nu \boldsymbol{K_{iS}})(\nu \boldsymbol{K_{Sj}}) \\
&\quad\ (Send_{spec}(I_1) \mid \cdots \mid Send_{spec}(I_m) \mid \\
&\quad\ !S \mid \\
&\quad\ !Recv_{spec}(1) \mid \cdots \mid !Recv_{spec}(n))
\end{aligned}
$$

In this specification, the sending process for instance (i, j, M) is as in the implementation, except that it sends a fresh channel name p instead of M, and runs $F(\underline{i}, \underline{j}, M)$ when it receives any message on p. The receiving process in the specification is identical to that in the implementation, except that $F(y_A, \underline{j}, z_{plain})$ is replaced with $\overline{z_{plain}}\langle * \rangle$, where the symbol $*$ represents a fixed but arbitrary message. The variable z_{plain} will be bound to the fresh name p for the corresponding instance of the protocol. Thus, the receiving process will signal on p, triggering the execution of the appropriate process $F(\underline{i}, \underline{j}, M)$.

A crucial property of this specification is that the only occurrences of F are bundled into the description of the sending process. There, F is applied to the desired parameters, $(\underline{i}, \underline{j}, M)$. Hence it is obvious that an instance (i, j, M) will cause the execution of $F(\underline{i'}, \underline{j'}, M')$ only if i' is i, j' is j, and M' is M. Therefore, despite its complexity, the specification is more obviously "correct" than the implementation.

Much as in previous examples, we would like the protocol to have the following authenticity property:

$$
\begin{aligned}
&Sys(I_1,\ldots,I_m) \simeq Sys_{spec}(I_1,\ldots,I_m), \\
&\qquad \text{for any instances } I_1, \ldots, I_m.
\end{aligned}
$$

Unfortunately, the protocol is vulnerable to a replay attack that invalidates the authenticity equation. Consider the system $Sys(I, I')$ where $I = (i, j, M)$ and $I' = (i, j, M')$. An attacker can replay messages of one instance and get them mistaken for messages of the other instance, causing M to be passed twice to F. Thus, $Sys(I, I')$ can be made to execute two copies of $F(\underline{i}, \underline{j}, M)$. In contrast, no matter what an attacker does, $Sys_{spec}(I, I')$ will run each of $F(\underline{i}, \underline{j}, M)$ and $F(\underline{i}, \underline{j}, M')$ at most once. The authenticity equation therefore does not hold. (We can disprove it formally by defining an attacker that distinguishes $Sys(I, I')$ and $Sys_{spec}(I, I')$, within the spi calculus.)

6.4 A Complete Authentication Example (Repaired)

Now we improve the protocol of the previous section by adding nonce handshakes as protection against replay attacks. The Wide Mouthed Frog protocol

$$
\begin{aligned}
Send(i,j,M) \triangleq\ & \overline{c_S}\langle \underline{i} \rangle \mid \\
& c_i(x_{nonce}).(\nu K)(\overline{c_S}\langle (\underline{i}, \{\underline{i}, \underline{i}, \underline{j}, K, x_{nonce}\}_{K_{iS}})\rangle \mid \overline{c_j}\langle (\underline{i}, \{M\}_K)\rangle) \\
S \triangleq\ & c_S(x_A).\textstyle\prod_{i\in 1..n}[x_A \ is\ \underline{i}]\ (\nu N_S)(\overline{c_i}\langle N_S\rangle \mid \\
& c_S(x'_A, x_{cipher}).[x'_A \ is\ \underline{i}] \\
& case\ x_{cipher}\ of\ \{y_A, z_A, x_B, x_{key}, x_{nonce}\}_{K_{iS}}\ in \\
& \textstyle\prod_{j\in 1..n}[y_A \ is\ \underline{i}]\ [z_A \ is\ \underline{i}]\ [x_B \ is\ \underline{j}]\ [x_{nonce} \ is\ N_S] \\
& (\overline{c_j}\langle * \rangle \mid c_S(y_{nonce}).\overline{c_j}\langle \{S, \underline{i}, \underline{j}, x_{key}, y_{nonce}\}_{K_{Sj}}\rangle)) \\
Recv(j) \triangleq\ & c_j(w).(\nu N_B)(\overline{c_S}\langle N_B\rangle \mid \\
& c_j(y_{cipher}). \\
& case\ y_{cipher}\ of\ \{x_S, x_A, x_B, x_{key}, y_{nonce}\}_{K_{Sj}}\ in \\
& \textstyle\prod_{i\in 1..n}[x_S \ is\ S]\ [x_A \ is\ \underline{i}]\ [x_B \ is\ \underline{j}]\ [y_{nonce} \ is\ N_B] \\
& c_j(z_A, z_{cipher}).[z_A \ is\ x_A] \\
& case\ z_{cipher}\ of\ \{z_{plain}\}_{x_{key}}\ in\ F(\underline{i}, \underline{j}, z_{plain})) \\
Sys(I_1, \ldots, I_m) \triangleq\ & (\nu \boldsymbol{K_{iS}})(\nu \boldsymbol{K_{Sj}}) \\
& (Send(I_1) \mid \cdots \mid Send(I_m) \mid \\
& !S \mid \\
& !Recv(1) \mid \cdots \mid !Recv(n))
\end{aligned}
$$

Fig. 2. Formalisation of the Seven-Message protocol

uses timestamps instead of handshakes. The treatment of timestamps in the spi calculus is possible, but it requires additional elements, including at least a rudimentary account of clock synchronisation. Protocols that use handshakes are fundamentally more self-contained than protocols that use timestamps; therefore, handshakes make for clearer examples.

Informally, our new protocol is:

Message 1 $A \rightarrow S : A$	on c_S
Message 2 $S \rightarrow A : N_S$	on c_A
Message 3 $A \rightarrow S : A, \{A, A, B, K_{AB}, N_S\}_{K_{AS}}$	on c_S
Message 4 $S \rightarrow B : *$	on c_B
Message 5 $B \rightarrow S : N_B$	on c_S
Message 6 $S \rightarrow B : \{S, A, B, K_{AB}, N_B\}_{K_{SB}}$	on c_B
Message 7 $A \rightarrow B : A, \{M\}_{K_{AB}}$	on c_B

Messages 1 and 2 are the request for a challenge and the challenge, respectively. The challenge is N_S, a nonce created by S; the nonce must not have been used before for this purpose. Obviously the nonce is not secret, but it must be unpredictable (for otherwise an attacker could simulate a challenge and later replay the response [AN96]). In Message 3, A says that A and B can communicate under K_{AB}, sometime after receipt of N_S. All the components A, B, K_{AB}, N_S appear explicitly in the message, for safety [AN96], but A could perhaps be elided. The presence of N_S in Message 3 proves the freshness of the message. In Message 4,

$*$ represents a fixed but arbitrary message; S uses $*$ to signal that it is ready for a nonce challenge N_B from B. In Message 6, S says that A says that A and B can communicate under K_{AB}, sometime after receipt of N_B. The first field of the encrypted portions of Messages 3 and 6 (A or S) is included in order to distinguish these messages; it serves as a "direction bit". Finally, Message 7 is the transmission of data under K_{AB}.

The messages of this protocol have many components. For the spi calculus representation it is therefore convenient to generalise our syntax of pairs and pair splitting to arbitrary tuples. We use the following standard abbreviations:

$$(N_1, \ldots, N_{k+1}) \triangleq ((N_1, \ldots, N_k), N_{k+1})$$

$$\begin{aligned} \mathit{let}\ (x_1, \ldots, x_{k+1}) = N\ \mathit{in}\ P \triangleq\ & \mathit{let}\ (y, x_{k+1}) = N\ \mathit{in} \\ & \mathit{let}\ (x_1, \ldots, x_k) = y\ \mathit{in}\ P \end{aligned}$$

where variable y is fresh.

In the spi calculus, we represent the nonces of this protocol as newly created names. We obtain the spi calculus expressions given in Figure 2. In those expressions, the names N_S and N_B represent the nonces. The variable subscripts are hints that indicate what the corresponding variables should represent; for example, x_A, x'_A, y_A, and z_A are all expected to be the name of the sending process, and x_{nonce} and y_{nonce} are expected to be the nonces generated by S and B, respectively.

The definition of Sys_{spec} is exactly analogous to that of the previous section, so we omit it. We obtain the authenticity property:

$$\begin{aligned} &\mathit{Sys}(I_1, \ldots, I_m) \simeq \mathit{Sys}_{spec}(I_1, \ldots, I_m), \\ &\qquad \text{for any instances } I_1, \ldots, I_m. \end{aligned}$$

This property holds because of the use of nonces. In particular, the replay attack of Section 6.3 can no longer distinguish $\mathit{Sys}(I_1, \ldots, I_m)$ and $\mathit{Sys}_{spec}(I_1, \ldots, I_m)$.

As a secrecy property, we would like to express that there is no way for an external observer to tell apart two executions of the system with identical participants but different messages. The secrecy property should therefore assert that the protocol does not reveal any information about the contents of exchanged messages if none is revealed after the key exchange.

In order to express that no information is revealed after the key exchange, we introduce the following definition. We say that a pair of instances (i, j, M) and (i', j', M') is *indistinguishable* if the two instances have the same source and destination addresses ($i = i'$ and $j = j'$) and if $F(\underline{i}, \underline{j}, M) \simeq F(\underline{i}, \underline{j}, M')$.

Our definition of secrecy is that, if each pair $(I_1, J_1), \ldots, (I_m, J_m)$ is indistinguishable, then $\mathit{Sys}(I_1, \ldots, I_m) \simeq \mathit{Sys}(J_1, \ldots, J_m)$. This means that an observer cannot distinguish two systems parameterised by two sets of indistinguishable instances. This property holds for our protocol.

In summary, we have:

Authenticity: $Sys(I_1, \ldots, I_m) \simeq Sys_{spec}(I_1, \ldots, I_m)$,
for any instances $I_1, \ldots, I_m$.

Secrecy: $Sys(I_1, \ldots, I_m) \simeq Sys(J_1, \ldots, J_m)$,
if each pair $(I_1, J_1), \ldots, (I_m, J_m)$
is indistinguishable.

We could ask for a further property of anonymity, namely that the source and the destination addresses of instances be protected from eavesdroppers. However, anonymity holds neither for our protocol nor for most current, practical protocols. It would be easy enough to specify anonymity, should it be relevant.

6.5 Discussion of the Examples

As these examples show, writing a protocol in the spi calculus is essentially analogous to writing it in any programming language with suitable communication and encryption libraries. The main advantage of the spi calculus is its formal precision.

Writing a protocol in the spi calculus may be a little harder than writing it in some of the notations common in the literature. On the other hand, the spi calculus versions are more detailed. They make clear not only what messages are sent but how the messages are generated and how they are checked. These aspects of the spi calculus descriptions add complexity, but they enable finer analysis.

7 Formal Semantics of the Spi Calculus

In this section we give a brief formal treatment of the spi calculus. In Section 7.1 we introduce the reaction relation; $P \rightarrow Q$ means there is a reaction amongst the subprocesses of P such that the whole can take a step to process Q. Reaction is the basic notion of computation in both the pi calculus and the spi calculus. In Section 7.2 we give a precise definition of the equivalence relation $\simeq$, which we have used for expressing security properties.

Syntactic Conventions

We write $fn(M)$ and $fn(P)$ for the sets of names free in term M and process P respectively. Similarly, we write $fv(M)$ and $fv(P)$ for the sets of variables free in M and P respectively. We say that a term or process is *closed* to mean that it has no free variables. (To be able to communicate externally, a process must have free names.) The set $Proc = \{P \mid fv(P) = \varnothing\}$ is the set of closed processes.

7.1 The Reaction Relation

The reaction relation is a concise account of computation in the pi calculus introduced by Milner [Mil92], inspired by the Chemical Abstract Machine of

Berry and Boudol [BB90]. One thinks of a process as consisting of a chemical solution of molecules waiting to react. A reaction step arises from the interaction of the adjacent molecules $\overline{m}\langle N\rangle.P$ and $m(x).Q$, as follows:

$$\text{(React Inter)} \quad \overline{m}\langle N\rangle.P \mid m(x).Q \;\rightarrow\; P \mid Q\{N\}x$$

Just as one might stir a chemical solution to allow non-adjacent molecules to react, we define a relation, *structural equivalence*, that allows processes to be rearranged so that the rule above is applicable. We first define the *reduction relation* $>$ on closed processes:

Reduction:

(Red Repl)	$!P > P \mid !P$
(Red Match)	$[M \; is \; M]\, P > P$
(Red Let)	$let\ (x,y) = (M,N)\ in\ P > P\{M\}x\{N\}y$
(Red Zero)	$case\ 0\ of\ 0 : P\ suc(x) : Q > P$
(Red Suc)	$case\ suc(M)\ of\ 0 : P\ suc(x) : Q > Q\{M\}x$
(Red Decrypt)	$case\ \{M\}_N\ of\ \{x\}_N\ in\ P > P\{M\}x$

We let structural equivalence, $\equiv$, be the least relation on closed processes that satisfies the following equations and rules:

Structural Congruence:

(Struct Nil)	$P \mid \mathbf{0} \equiv P$
(Struct Comm)	$P \mid Q \equiv Q \mid P$
(Struct Assoc)	$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
(Struct Switch)	$(\nu m)(\nu n)P \equiv (\nu n)(\nu m)P$
(Struct Drop)	$(\nu n)\mathbf{0} \equiv \mathbf{0}$
(Struct Extrusion)	$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \quad \text{if } n \notin fn(P)$

$$\text{(Struct Red)}\ \frac{P > Q}{P \equiv Q} \qquad \text{(Struct Refl)}\ \frac{}{P \equiv P} \qquad \text{(Struct Symm)}\ \frac{P \equiv Q}{Q \equiv P}$$

$$\text{(Struct Trans)}\ \frac{P \equiv Q \qquad Q \equiv R}{P \equiv R} \qquad \text{(Struct Par)}\ \frac{P \equiv P'}{P \mid Q \equiv P' \mid Q} \qquad \text{(Struct Res)}\ \frac{P \equiv P'}{(\nu m)P \equiv (\nu m)P'}$$

Now we can complete the formal description of the reaction relation. We let the *reaction relation*, $\rightarrow$, be the least relation on closed processes that satisfies the rule (React Inter) displayed above and the following rules:

Reaction:

$$\text{(React Struct)}\ \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q} \qquad \text{(React Par)}\ \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(React Res)}\ \frac{P \rightarrow P'}{(\nu n)P \rightarrow (\nu n)P'}$$

This definition of the reaction relation corresponds to the informal description of process behaviour given in Sections 2 and 5.

As an example, we can use the definition of the reaction relation to show the behaviour of the protocol of Section 6.2:

$$\begin{aligned} \mathit{Inst}(M) &\equiv (\nu K_{AS})(\nu K_{SB})(A(M) \mid S \mid B) \\ &\to (\nu K_{AS})(\nu K_{SB})(\nu K_{AB}) \\ &\qquad (\overline{c_{AB}}\langle \{M\}_{K_{AB}} \rangle \mid \overline{c_{SB}}\langle \{K_{AB}\}_{K_{SB}} \rangle \mid B) \\ &\to (\nu K_{AS})(\nu K_{SB})(\nu K_{AB}) \\ &\qquad (\overline{c_{AB}}\langle \{M\}_{K_{AB}} \rangle \mid \\ &\qquad c_{AB}(z).\mathit{case}\ z\ \mathit{of}\ \{w\}_{K_{AB}}\ \mathit{in}\ F(w)) \\ &\to (\nu K_{AS})(\nu K_{SB})(\nu K_{AB})F(M) \\ &\equiv F(M) \end{aligned}$$

The last step in this calculation is justified by our general convention that none of the bound parameters of the protocol (including, in this case, K_{AS}, K_{SB}, and K_{AB}) occurs free in F.

7.2 Testing Equivalence

In order to define equivalence, we first define a predicate that describes the channels on which a process can communicate. We let a *barb*, β, be an input or output channel, that is, either a name m (representing input) or a *co-name* $\overline{m}$ (representing output). For a closed process P, we define the predicate P *exhibits barb* β, written $P \downarrow \beta$, by the following rules:

Exhibition of a Barb:

(Barb In) $\dfrac{}{m(x).P \downarrow m}$ (Barb Out) $\dfrac{}{\overline{m}\langle M \rangle.P \downarrow \overline{m}}$ (Barb Par) $\dfrac{P \downarrow \beta}{P \mid Q \downarrow \beta}$

(Barb Res) $\dfrac{P \downarrow \beta \quad \beta \notin \{m, \overline{m}\}}{(\nu m)P \downarrow \beta}$ (Barb Struct) $\dfrac{P \equiv Q \quad Q \downarrow \beta}{P \downarrow \beta}$

Intuitively, $P \downarrow \beta$ holds just if P is a closed process that may input or output immediately on barb β. The *convergence* predicate $P \Downarrow \beta$ holds if P is a closed process that exhibits β after some reactions:

Convergence to a Barb:

(Conv Barb) $\dfrac{P \downarrow \beta}{P \Downarrow \beta}$ (Conv React) $\dfrac{P \to Q \qquad Q \Downarrow \beta}{P \Downarrow \beta}$

We let a *test* consist of any closed process R and any barb β. A closed process P *passes* the test if and only if $(P \mid R)\Downarrow\beta$. The notion of testing gives rise to a testing equivalence on the set *Proc* of closed processes:

$$P \simeq Q \triangleq \text{for any test } (R, \beta),\\ (P \mid R)\Downarrow\beta \text{ if and only if } (Q \mid R)\Downarrow\beta$$

The idea of testing equivalence comes from the work of De Nicola and Hennessy [DH84]. Despite superficial differences, we can show that our relation $\simeq$ is a version of De Nicola and Hennessy's may-testing equivalence. As De Nicola and Hennessy have explained, may-testing corresponds to partial correctness (or safety), while must-testing corresponds to total correctness. Like much of the security literature, our work focuses on safety properties, hence our definitions.

A test neatly formalises the idea of a generic experiment or observation another process (such as an attacker) might perform on a process, so testing equivalence captures the concept of equivalence in an arbitrary environment. One possible drawback of testing equivalence is that it is sensitive to the choice of language [BN95]. However, our results appear fairly robust in that they carry over smoothly to some extensions of our calculus.

8 Further Cryptographic Primitives

Although so far we have discussed only shared-key cryptography, other kinds of cryptography are also easy to treat within the spi calculus. In this section we show how to handle cryptographic hashing, public-key encryption, and digital signatures. We add syntax for these operations to the spi calculus and give their semantics. We thus provide evidence that our ideas are applicable to a wide range of security protocols, beyond those that rely on shared-key encryption. We believe that we may be able to deal similarly with Diffie-Hellman techniques and with secret sharing. However, protocols for oblivious transfer and for zero-knowledge proofs, for example, are probably beyond the scope of our approach.

8.1 Hashing

A cryptographic hash function has the properties that it is very expensive to recover an input from its image or to find two inputs with the same image. Functions such as SHA and RIPE-MD are generally believed to have these properties [Sch96b].

When we represent hash functions in the spi calculus, we pretend that operations that are very expensive are altogether impossible. We simply add a construct to the syntax of terms of the spi calculus:

Extension for Hashing:

$L, M, N ::=$	terms
$\ldots$	as in Section 5
$H(M)$	hashing

The syntax of processes is unchanged. Intuitively, $H(M)$ represents the hash of M. The absence of a construct for recovering M from $H(M)$ corresponds to the assumption that H cannot be inverted. The lack of any equations $H(M) = H(M')$ corresponds to the assumption that H is free of collisions.

8.2 Public-Key Encryption and Digital Signatures

Traditional public-key encryption systems are based on key pairs. Normally, one of the keys in each pair is private to one principal, while the other key is public. Any principal can encrypt a message using the public key; only a principal that has the private key can then decrypt the message [DH76, RSA78].

We assume that neither key can be recovered from the other. We could just as easily deal with the case where the public key can be derived from the private one. Much as in Section 5, we also assume that the only way to decrypt an encrypted packet is to know the corresponding private key; that an encrypted packet does not reveal the public key that was used to encrypt it; and that there is sufficient redundancy in messages so that the decryption algorithm can detect whether a ciphertext was encrypted with the expected public key.

We arrive at the following syntax for the spi calculus with public-key encryption. (This syntax is concise, rather than memorable.)

Extensions for Public-Key Cryptography:

$L, M, N ::=$	terms
$\ldots$	as in Section 5
M^+	public part
M^-	private part
$\{\!\lceil M \rceil\!\}_N$	public-key encryption
$P, Q ::=$	processes
$\ldots$	as in Section 5
case L *of* $\{\!\lceil x \rceil\!\}_N$ *in* P	decryption

If M represents a key pair, then M^+ represents its public half and M^- represents its private half. Given a public key N, the term $\{\!\lceil M \rceil\!\}_N$ represents the result of the public-key encryption of M with N. In *case* L *of* $\{\!\lceil x \rceil\!\}_N$ *in* P, the variable x is bound in P. This construct is useful when N is a private key K^-; then it binds x to the M such that $\{\!\lceil M \rceil\!\}_{K^+}$ is L, if such an M exists.

It is also common to use key pairs for digital signatures. Private keys are used for signing, while public keys are used for checking signatures. We can represent digital signatures through the following extended syntax:

Extensions for Digital Signatures:

$L, M, N ::=$	terms
$\ldots$	as above
$[\!\{ M \}\!]_N$	private-key signature

$P, Q ::=$	processes
$\ldots$	as above
case N *of* $[\{x\}]_M$ *in* P	signature check

Given a private key N, the term $[\{M\}]_N$ represents the result of the signature of M with N. Again, variable x is bound in P in the syntax *case* N *of* $[\{x\}]_M$ *in*P. This construct is dual to *case* L *of* $\{[x]\}_N$ *in* P. The new construct is useful when N is a public key K^+; then it binds x to the M such that $[\{M\}]_{K^-}$ is L, if such an M exists. (Thus, we are assuming that M can be recovered from the result of signing it; but there is no difficulty in dropping this assumption.)

Formally, the semantics of the new constructs is captured with two new rules for the reduction relation:

(Red Public Decrypt) *case* $\{[M]\}_{N^+}$ *of* $\{[x]\}_{N^-}$ *in* $P > P\{M\}x$
(Red Signature Check) *case* $[\{M\}]_{N^-}$ *of* $[\{x\}]_{N^+}$ *in* $P > P\{M\}x$

As a small example, we can write the following public-key analogue for the protocol of Section 6.1:

$$
\begin{aligned}
A(M) &\triangleq \overline{c_{AB}}\langle\{[M, [\{H(M)\}]_{K_A^-}]\}_{K_B^+}\rangle \\
B &\triangleq c_{AB}(x).\mathit{case}\ x\ \mathit{of}\ \{[y]\}_{K_B^-}\ \mathit{in} \\
&\quad \mathit{let}\ (y_1, y_2) = y\ \mathit{in} \\
&\quad \mathit{case}\ y_2\ \mathit{of}\ [\{z\}]_{K_A^+}\ \mathit{in} \\
&\quad [H(y_1)\ \mathit{is}\ z]\, F(y_1) \\
\mathit{Inst}(M) &\triangleq (\nu K_A)(\nu K_B)(A(M) \mid B)
\end{aligned}
$$

In this protocol, A sends M on the channel c_{AB}, signed with A's private key and encrypted under B's public key; the signature is applied to a hash of M rather than to M itself. On receipt of a message on c_{AB}, B decrypts using its private key, checks A's signature using A's public key, checks the hash, and applies F to the body of the message (to M). The key pairs K_A and K_B are restricted; but there would be no harm in sending their public parts K_A^+ and K_B^+ on a public channel.

Other formalisations of public-key cryptography are possible, perhaps even desirable. In particular, we have represented cryptographic operations at an abstract level, and do not attempt to model closely the properties of any one algorithm. We are concerned with public-key encryption and digital signatures in general rather than with their RSA implementations, say. The RSA system satisfies equations that our formalisation does not capture. For example, in the RSA system, $[\{\{[M]\}_{K^+}\}]_{K^-}$ equals M. Abadi and Fournet [AF01] investigate a pi calculus in which such equations may be imposed on terms.

9 Discussion: The Spi Calculus

In this part, we have applied the spi calculus to the description and analysis of security protocols. We showed how to represent protocols and how to ex-

press their security properties. Our model of protocols takes into account the possibility of attacks, but does not require writing explicit specifications for an attacker. In particular, we express secrecy properties as simple equations that mean indistinguishability from the point of view of an arbitrary attacker. To our knowledge, this sharp treatment of attacks has not been previously possible.

As examples, we chose protocols of the sort commonly found in the authentication literature. Although our examples are small, we have found them instructive and encouraging. In particular, there seems to be no fundamental difficulty in writing other kinds of examples, such as protocols for electronic commerce. Unfortunately, the specifications for those protocols do not yet seem to be fully understood, even in informal terms [Mao96].

Several proof techniques for the spi calculus have been developed since the work reported in this part of the notes was completed. Several researchers have devised proof techniques for equational reasoning in spi and its generalisations [AG98, BNP99, AF01]. A type system due to Abadi [Aba99] can prove equationally-specified secrecy properties including the one stated in Section 6.4. There is no comparable type system for proving equationally-specified authenticity properties. Still, recent work on type systems for the spi calculus [GJ01] allow the proof by type-checking of authenticity properties specified using the correspondence assertions of Woo and Lam [WL93].

Apart from the spi calculus, other nominal calculi to have been applied to cryptographic protocols include the sjoin calculus [AFG98], a version of the join calculus equipped with abstract cryptographic primitives. It is surprisingly difficult to encode encryption within the pi calculus; Amadio and Prasad [AP99] investigate one such encoding.

Part III: The Ambient Calculus

The ambient calculus is a nominal calculus whose basic abstraction, the *ambient*, represents mobile, nested, computational environments, with local communications. Ambients can represent the standard components of distributed systems, such as nodes, channels, messages, and mobile code. They can also represent situations where entire active computational environments are moved, as happens with mobile computing devices, and with multi-threaded mobile agents.

This part of the notes introduces the ambient calculus, and explains how we can regulate aspects of mobility by typing. It is organised as follows. In Section 10, we informally motivate the ambient abstraction, and then in Section 11 we present the basic untyped ambient calculus. Next, in Section 12, we motivate the development of type systems for the ambient calculus. In Section 13 we informally introduce a type system that only tracks communications. In Section 14 we give a precise definition of the same system, and a subject reduction result. Section 15 and 16 enrich this system to regulate the mobility of ambients. In Section 17, to illustrate the expressiveness of the ambient calculus and its type system, we present a typed encoding of a distributed programming language. Section 18 concludes.

10 Motivation for Ambients

There are two distinct areas of work in mobility: mobile computing, concerning computation that is carried out in mobile devices (laptops, personal digital assistants, etc.), and mobile computation, concerning mobile code that moves between devices (applets, agents, etc.). We aim to describe all these aspects of mobility within a single framework that encompasses mobile agents, the ambients where agents interact and the mobility of the ambients themselves.

The inspiration for this work comes from the potential for mobile computation over the World-Wide Web. The geographic distribution of the Web naturally calls for mobility of computation, as a way of flexibly managing latency and bandwidth. Because of recent advances in networking and language technology, the basic tenets of mobile computation are now technologically realizable. The high-level software architecture potential, however, is still largely unexplored, although it is being actively investigated in the coordination and agents communities.

The main difficulty with mobile computation on the Web is not in mobility per se, but in the handling of administrative domains. In the early days of the internet one could rely on a flat name space given by IP addresses; knowing the IP address of a computer would very likely allow one to talk to that computer in some way. This is no longer the case: firewalls partition the internet into administrative domains that are isolated from each other except for rigidly controlled pathways. System administrators enforce policies about what can move through firewalls and how.

Mobility requires more than the traditional notion of authorization to run or to access information in certain domains: it involves the authorization to enter or exit certain domains. In particular, as far as mobile computation is concerned, it is not realistic to imagine that an agent can migrate from any point A to any point B on the internet. Rather, an agent must first exit its administrative domain (obtaining permission to do so), enter someone else's administrative domain (again, obtaining permission to do so) and then enter a protected area of some machine where it is allowed to run (after obtaining permission to do so). Access to information is controlled at many levels, thus multiple levels of authorization may be involved. Among these levels we have: local computer, local area network, regional area network, wide-area intranet and internet. Mobile programs must be equipped to navigate this hierarchy of administrative domains, at every step obtaining authorization to move further. Similarly, laptops must be equipped to access resources depending on their location in the administrative hierarchy. Therefore, at the most fundamental level we need to capture notions of locations, of mobility and of authorization to move.

Today, it is very difficult to transport a working environment between two computers, for example, between a laptop and a desktop, or between home and work computers. The working environment might consist of data that has to be copied, and of running programs in various stages of active or suspended communication with the network that have to be shut down and restarted. Why can't we just say "move this (part of the) environment to that computer" and

carry on? When on a trip, why couldn't we transfer a piece of the desktop environment (for example, a forgotten open document along with its editor) to the laptop over a phone line? We would like to discover techniques to achieve all this easily and reliably.

With these motivations, we adopt a paradigm of mobility where computational ambients are hierarchically structured, where agents are confined to ambients and where ambients move under the control of agents. A novelty of this approach is in allowing the movement of self-contained nested environments that include data and live computation, as opposed to the more common techniques that move single agents or individual objects. Our goal is to make mobile computation scale-up to widely distributed, intermittently connected and well administered computational environments.

10.1 Ambients

An ambient, in the sense in which we are going to use this word, has the following main characteristics:

- An ambient is a bounded place where computation happens. The interesting property here is the existence of a boundary around an ambient. If we want to move computations easily we must be able to determine what should move; a boundary determines what is inside and what is outside an ambient. Examples of ambients, in this sense, are: a web page (bounded by a file), a virtual address space (bounded by an addressing range), a Unix file system (bounded within a physical volume), a single data object (bounded by "self") and a laptop (bounded by its case and data ports). Non-examples are: threads (where the boundary of what is "reachable" is difficult to determine) and logically related collections of objects. We can already see that a boundary implies some flexible addressing scheme that can denote entities across the boundary; examples are symbolic links, Uniform Resource Locators and Remote Procedure Call proxies. Flexible addressing is what enables, or at least facilitates, mobility. It is also, of course, a cause of problems when the addressing links are "broken".
- An ambient is something that can be nested within other ambients. As we discussed, administrative domains are (often) organized hierarchically. If we want to move a running application from work to home, the application must be removed from an enclosing (work) ambient and inserted in a different enclosing (home) ambient. A laptop may need a removal pass to leave a workplace, and a government pass to leave or enter a country.
- An ambient is something that can be moved as a whole. If we reconnect a laptop to a different network, all the address spaces and file systems within it move accordingly and automatically. If we move an agent from one computer to another, its local data should move accordingly and automatically.

More precisely, we investigate ambients that have the following structure:

- Each ambient has a name. The name of an ambient is used to control access (entry, exit, communication, etc.). In a realistic situation the true name of an ambient would be guarded very closely, and only specific capabilities would be handed out about how to use the name. In our examples we are usually more liberal in the handling of names, for the sake of simplicity.
- Each ambient has a collection of local agents (also known as threads, processes, etc.). These are the computations that run directly within the ambient and, in a sense, control the ambient. For example, they can instruct the ambient to move.
- Each ambient has a collection of subambients. Each subambient has its own name, agents, subambients, etc.

10.2 Technical Context: Systems

Many software systems have explored and are exploring notions of mobility. Among these are:

- Obliq [Car95]. The Obliq project attacked the problems of distribution and mobility for intranet computing. It was carried out largely before the Web became popular. Within its scope, Obliq works quite well, but is not really suitable for computation and mobility over the Web, just like most other distributed paradigms developed in pre-Web days.
- Telescript [Whi96]. Our ambient model is partially inspired by Telescript, but is almost dual to it. In Telescript, agents move whereas places stay put. Ambients, instead, move whereas agents are confined to ambients. A Telescript agent, however, is itself a little ambient, since it contains a "suitcase" of data. Some nesting of places is allowed in Telescript.
- Java [GJS96]. Java provides a working paradigm for mobile computation, as well as a huge amount of available and expected infrastructure on which to base more ambitious mobility efforts.
- Linda [CG89]. Linda is a "coordination language" where multiple processes interact in a common space (called a tuple space) by dropping and picking up tokens asynchronously. Distributed versions of Linda exist that use multiple tuple spaces and allow remote operations over those. A dialect of Linda [CGZ95] allows nested tuple spaces, but not mobility of the tuple spaces.

10.3 Technical Context: Formalisms

Many existing calculi have provided inspiration for our work. In particular:

- Enrichments of the pi calculus with locations have been studied, with the aim of capturing notions of distributed computation. In the simplest form, a flat space of locations is added, and operations can be indexed by the location where they are executed. Riely and Hennessy [RH98] and Sewell [Sew98]

propose versions of the pi calculus extended with primitives to allow computations to migrate between named locations. The emphasis in this work is on developing type systems for mobile computation based on existing type systems for the pi calculus. Riely and Hennessy's type system regulates the usage of channel names according to permissions represented by types. Sewell's type system differentiates between local and remote channels for the sake of efficient implementation of communication.

- The join calculus [FG96] is a reformulation of the pi calculus with a more explicit notion of places of interaction; this greatly helps in building distributed implementations of channel mechanisms. The distributed join calculus [FGL+96] adds a notion of named locations, with essentially the same aims as ours, and a notion of distributed failure. Locations in the distributed join calculus form a tree, and subtrees can migrate from one part of the tree to another. A significant difference from our ambients is that movement may happen directly from any active location to any other known location.
- LLinda [NFP97] is a formalization of Linda using process calculi techniques. As in distributed versions of Linda, LLinda has multiple distributed tuple spaces. Multiple tuple spaces are very similar in spirit to multiple ambients, but Linda's tuple spaces do not nest, and there are no restrictions about accessing a tuple space from any other tuple space.
- A growing body of literature is concentrating on the idea of adding discrete locations to a process calculus and considering failure of those locations [Ama97, FGL+96]. This approach aims to model traditional distributed environments, along with algorithms that tolerate node failures. However, on the internet, node failure is almost irrelevant compared with inability to reach nodes. Web servers do not often fail forever, but they frequently disappear from sight because of network or node overload, and then they come back. Sometimes they come back in a different place, for example, when a Web site changes its internet Service Provider. Moreover, inability to reach a Web site only implies that a certain path is unavailable; it implies neither failure of that site nor global unreachability. In this sense, an observed node failure cannot simply be associated with the node itself, but instead is a property of the whole network, a property that changes over time. Our notion of locality is induced by a non-trivial and dynamic topology of locations. Failure is only represented, in a weak but realistic sense, as becoming forever unreachable.

10.4 Summary of Our Approach

With respect to previous work on process calculi, we can characterize the main differences in the ambient calculus approach as follows. In each of the following points, our emphasis is on boundaries and their effect on computation. The existence of separate locations is represented by a topology of boundaries. This topology induces an abstract notion of distance between locations. Locations are not uniformly accessible, and are not identified by globally unique names. Process mobility is represented as crossing of boundaries. In particular, process

mobility is not represented as communication of processes or process names over channels. Security is represented as the ability or inability to cross boundaries. In particular, security is not directly represented by cryptographic primitives or access control lists. Interaction between processes is by shared location within a common boundary. In particular, interaction cannot happen without proper consideration of boundaries and their topology.

11 A Polyadic Ambient Calculus

The ambient calculus of this section is a slight extension of the original untyped ambient calculus [CG00b]. In that calculus, communication is based on the exchange of single values. Here we extend the calculus with communication based on tuples of values (polyadic communication), since this simple extension greatly facilitates the task of providing an expressive type system. We also add objective moves and we annotate bound variables with type information.

The ambient calculus is a derivative of the pi calculus. Four of its process constructions (restriction, inactivity, composition, and replication) are exactly as in the pi calculus. To these we add ambients, capabilities, and a simple form of communication. We briefly discuss these constructions; see [CG00b] for a more detailed introduction.

The restriction operator, $(\nu n{:}W)P$, creates a new (unique) name n of type W within a scope P. The new name can be used to name ambients and to operate on ambients by name. The inactive process, $\mathbf{0}$, does nothing. Parallel composition is denoted by a binary operator, $P \mid Q$, that is commutative and associative. As in the pi calculus, replication is a technically convenient way of representing iteration and recursion: the process $!P$ denotes the unbounded replication of the process P and is equivalent to $P \mid !P$.

An ambient is written $M[P]$, where M is the name of the ambient, and P is the process running inside the ambient.

The process $M.P$ executes an action regulated by the capability M, and then continues as the process P. We consider three kinds of capabilities: one for entering an ambient, one for exiting an ambient, and one for opening up an ambient. (The latter requires special care in the type system.) Capabilities are obtained from names; given a name n, the capability $\mathit{in}\, n$ allows entry into n, the capability $\mathit{out}\, n$ allows exit out of n and the capability $\mathit{open}\, n$ allows the opening of n. Implicitly, the possession of one or all of these capabilities is insufficient to reconstruct the original name n from which they were extracted. Capabilities can also be composed into paths, $M.M'$, with ϵ for the empty path.

Communication is asynchronous and local to an ambient. It is similar to channel communication in the pi calculus, except that the channel has no name: the surrounding ambient provides the context where the communication happens. The process $\langle M_1, \ldots, M_k \rangle$ represents the output of a tuple of values, with no continuation. The process $(x_1{:}W_1, \ldots, x_k{:}W_k).P$ represents the input of a tuple of values, whose components are bound to $x_1, \ldots, x_k$, with continuation P.

Communication is used to exchange both names and capabilities, which share the same syntactic class M of messages. The first task of our type system is to distinguish the messages that are names from the messages that are capabilities, so that each is guaranteed to be used in an appropriate context. In general, the type system might distinguish other kinds of expressions, such as integer and boolean expressions, but we do not include those in our basic calculus.

The process $go\, N.M[P]$ moves the ambient $M[P]$ as specified by the N capability, and has $M[P]$ as its continuation. It is called an *objective move* since the ambient $M[P]$ is moved from the outside, while a movement caused by a process $N.P$ which runs inside an ambient is called a *subjective move*. In the untyped calculus, we can define an objective move $go\, N.M[P]$ to be short for the process $(\nu k)k[N.M[out\ k.P]]$ where k is not free in P. As we will show in Section 16.2, a primitive typing rule for objective moves allows more refined typings than are possible with only subjective moves.

Messages and Processes:

$M ::=$	message
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
$P, Q, R ::=$	process
$(\nu n{:}W)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	action
$(x_1{:}W_1, \ldots, x_k{:}W_k).P$	input action
$\langle M_1, \ldots, M_k \rangle$	output action
$go\ N.M[P]$	objective move

The following table displays the main reduction rules of the calculus (the full set is presented in Section 14). The notation $P\{x_1 \leftarrow M_1\} \cdots \{x_k \leftarrow M_k\}$ in rule (Red I/O) denotes the outcome of a capture-avoiding simultaneous substitution of message M_i for each free occurrence of the corresponding name x_i in the process P, for $i \in 1..k$.

Reduction:

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ \ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)

$open\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$\langle M_1, \ldots, M_k \rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow P\{x_1 \leftarrow M_1\} \cdots \{x_k \leftarrow M_k\}$	(Red I/O)
$go(in\ m.N).n[P] \mid m[Q] \rightarrow m[go\ N.n[P] \mid Q]$	(Red Go In)
$m[go(out\ m.N).n[P] \mid Q] \rightarrow go\ N.n[P] \mid m[Q]$	(Red Go Out)

We will use the following syntactic conventions:

- parentheses may be used for precedence
- $(\nu n{:}W)P \mid Q$ is read $((\nu n{:}W)P) \mid Q$
- $!P \mid Q$ is read $(!P) \mid Q$
- $M.P \mid Q$ is read $(M.P) \mid Q$
- $(n_1{:}W_1, \ldots, n_k{:}W_k).P \mid Q$ is read $((n_1{:}W_1, \ldots, n_k{:}W_k).P) \mid Q$
- $n[] \triangleq n[\mathbf{0}]$
- $M \triangleq M.\mathbf{0}$ (where appropriate)

As an example, consider the following process:

$$a[p[out\ a.in\ b.\langle c \rangle]] \mid b[open\ p.(x).x[]]$$

Intuitively, this example represents a packet named p being sent from a machine a to a machine b. The process $p[out\ a.in\ b.\langle c \rangle]$ represents the packet, as a subambient of ambient a. The name of the packet ambient is p, and its interior is the process $out\ a.in\ b.\langle c \rangle$. This process consists of three sequential actions: exercise the capability $out\ a$, exercise the capability $in\ b$, and then output the name c. The effect of the two capabilities on the enclosing ambient p is to move p out of a and into b (rules (Red Out), (Red In)), to reach the state:

$$a[] \mid b[p[\langle c \rangle] \mid open\ p.(x).x[]]$$

In this state, the interior of a is empty but the interior of b consists of two running processes, the subambient $p[\langle c \rangle]$ and the process $open\ p.(x).x[]$. This process is attempting to exercise the $open\ p$ capability. Previously it was blocked. Now that the p ambient is present, the capability's effect is to dissolve the ambient's boundary; hence, the interior of b becomes the process $\langle c \rangle \mid (x).x[]$ (Red Open). This is a composition of an output $\langle c \rangle$ with an input $(x).x[]$. The input consumes the output, leaving $c[]$ as the interior of b (Red I/O). Hence, the final state of the whole example is $a[] \mid b[c[]]$.

As an example of objective moves, consider the following variation of the previous process:

$$a[go(out\ a.in\ b).p[\langle c \rangle]] \mid b[open\ p.(x).x[]]$$

In this case, the ambient $p[\langle c \rangle]$ is moved from the outside, out of a and into b (rules (Red Go Out), (Red Go In)), to reach the same state that was reached in the previous version after the (Red Out), (Red In) subjective moves:

$$a[] \mid b[p[\langle c \rangle] \mid open\ p.(x).x[]]$$

See the original paper on the ambient calculus [CG00b] for many more examples, including locks, data structures such as booleans and numerals, Turing Machines, routable packets and active networks, and encodings of the lambda calculus and the pi calculus.

12 Types for the Ambient Calculus

Type systems are, today, a widely applied technique allowing programmers to describe the key properties of their code, and to have these properties mechanically and efficiently checked. Mobile code makes types, and machine-checkable properties in general, useful for security reasons too, as has been demonstrated by the checking performed on Java applets [LY97] and on other mobile code [GS01].

In standard languages, the key invariants that are maintained by type systems have mainly to do with the contents of variables and with the interfaces of functions, procedures, or methods. In the ambient calculus, the basic properties of a piece of code are those related to its mobility, to the possibility of opening an ambient and exposing its content, and to the type of data which may be exchanged inside an ambient. To understand how groups arise in this context, consider a typical static property we may want to express in a type system for the ambient calculus; informally:

> The ambient named n can enter the ambient named m.

This could be expressed as a typing $n : \mathit{CanEnter}(m)$ stating that n is a member of the collection $\mathit{CanEnter}(m)$ of names that can enter m. However, this would bring us straight into the domain of dependent types [CH88], since the type $\mathit{CanEnter}(m)$ depends on the name m. Instead, we introduce type-level groups of names, G, H, and restate our property as:

> The name m belongs to group G.
> The ambient named n can enter any ambient of group G.

This idea leads to typings of the form: $m : G$, $n : \mathit{CanEnter}(G)$ which are akin to standard typings such as $x : \mathit{Int}$, $y : \mathit{Channel}(\mathit{Int})$.

To appreciate the relevance of groups in the description of distributed systems, consider a programmer coding a typical distributed system composed of nodes and mobile threads moving from one node to another, and where threads communicate by sending input and output packets through typed channels. In these notes, we define a type system where a programmer can:

- define groups such as *Node*, *Thread*, *Channel*, and *Packet*, which match the system structure;
- declare properties such as: this ambient is a *Thread* and it may only cross ambients which are *Nodes*; this ambient is a *Packet* and can enter *Channels*; this ambient is a *Channel* of type T, and it cannot move or be opened, but it may open *Packets* containing data of type T; this ambient is a *Node* and it cannot move or be opened;

– have the system statically verify all these properties.

Our *groups* are similar to *sorts* used in typed versions of the pi calculus [Mil99], but we introduce an operation, $(\nu G)P$, for creating a new group G, which can be used within the process P.

The binders for new groups, (νG), can float outward as long as this adjustment, extrusion, does not introduce name clashes. Because of extrusion, group binders do not impede the mobility of ambients that are enclosed in the initial scope of fresh groups but later move away. On the other hand, even though extrusion enlarges scopes, simple scoping restrictions in the typing rules prevent names belonging to a fresh group from ever being received by a process which has been defined outside the initial scope of the group.

Therefore, we obtain a flexible way of protecting the propagation of names. This is to be contrasted with the situation in most untyped nominal calculi, where names can (intentionally, accidentally, or maliciously) be extruded arbitrarily far, by the automatic and unrestricted application of extrusion rules, and communicated to other parties.

13 Introduction to Exchange Types

An ambient is a place where processes can exchange messages and where other ambients can enter and exit. We introduce here a type system which regulates communication, while mobility will be tackled in the following sections. This system generalizes the one presented in [CG99] by allowing the partitioning of ambients into groups.

13.1 Topics of Conversation

Within an ambient, multiple processes can freely execute input and output actions. Since the messages are undirected, it is easily possible for a process to utter a message that is not appropriate for some receiver. The main idea of the exchange type system is to keep track of the topic of conversation that is permitted within a given ambient, so that talkers and listeners can be certain of exchanging appropriate messages.

The range of topics is described in the following table by message types, W, and exchange types, T. The message types are $G[T]$, the type of names of ambients which belong to the group G and that allow exchanges of type T, and $Cap[T]$, the type of capabilities that when used may cause the unleashing of T exchanges (as a consequence of opening ambients that exchange T). The exchange types are Shh, the absence of exchanges, and $W_1 \times \ldots \times W_k$, the exchange of a tuple of messages with elements of the respective message types. For $k = 0$, the empty tuple type is called $\mathbf{1}$; it allows the exchange of empty tuples, that is, it allows pure synchronization. The case $k = 1$ allows any message type to be an exchange type.

Types:

$W ::=$	message type
$G[T]$	name in group G for ambients allowing T exchanges
$Cap[T]$	capability unleashing T exchanges
$S, T ::=$	exchange type
Shh	no exchange
$W_1 \times \cdots \times W_k$	tuple exchange (**1** is the null product)

For example, in a scope where the *Agent* and *Place* groups have been defined, we can express the following types:

- An ambient of the *Agent* group where no exchange is allowed (a quiet *Agent*): $Agent[Shh]$
- A harmless capability: $Cap[Shh]$
- A *Place* where names of quiet *Agent*s may be exchanged:

$$Place[Agent[Shh]]$$

- A *Place* where harmless capabilities may be exchanged:

$$Place[Cap[Shh]]$$

- A capability that may unleash the exchange of names of quiet *Agent*s:

$$Cap[Agent[Shh]]$$

13.2 Intuitions

Before presenting the formal type rules (in Section 14), we discuss the intuitions that lead to them.

Typing of Processes If a message M has message type W, then $\langle M \rangle$ is a process that outputs (exchanges) W messages. Therefore, we will have a rule stating that:

$$M : W \text{ implies } \langle M \rangle : W$$

If P is a process that may exchange W messages, then $(x{:}W).P$ is also a process that may exchange W messages. Therefore:

$$P : W \text{ implies } (x{:}W).P : W$$

The process **0** exchanges nothing, so it naturally has exchange type *Shh*. However, we may also consider **0** as a process that may exchange any type. This is useful when we need to place **0** in a context that is already expected to exchange some type:

$$\mathbf{0} : T \quad \text{for any } T$$

Alternatively, we may add a subtype relation among types, give $\mathbf{0}$ a minimal type, and add a rule which allows processes with a type to appear where processes with a supertype are required [Zim00]. We reject this approach here only because we want to explore the ideas of group-based exchange and mobility types in the simplest possible setting.

If P and Q are processes that may exchange T, then $P \mid Q$ is also such a process. Similarly for $!P$:

$$\begin{array}{ll} P : T, Q : T & \text{implies } P \mid Q : T \\ P : T & \text{implies } !P : T \end{array}$$

Therefore, by keeping track of the exchange type of a process, T-inputs and T-outputs are tracked so that they match correctly when placed in parallel.

Typing of Ambients An ambient $n[P]$ is a process that exchanges nothing at the current level, so, like $\mathbf{0}$, it can be placed in parallel with any process, hence we allow it to have any exchange type:

$$n[P] : T \quad \text{for any } T$$

There needs to be, however, a connection between the type of n and the type of P. We give to each ambient name n a type $G[T]$, meaning that n belongs to the group G and that only T exchanges are allowed in any ambient of that name. Hence, a process P can be placed inside an ambient with that name n only if the type of P is T:

$$n : G[T], P : T \text{ implies } n[P] \text{ is well-formed (and can have any type)}$$

By tagging the name of an ambient with the type of exchanges, we know what kind of exchanges to expect in any ambient we enter. Moreover, we can tell what happens when we open an ambient of a given name.

Typing of Open Tracking the type of I/O exchanges is not enough by itself. We also need to worry about *open*, which might open an ambient and unleash its exchanges inside the surrounding ambient.

If ambients named n permit T exchanges, then the capability *open* n may unleash those T exchanges. We then say that *open*n has a capability type $Cap[T]$, meaning that it may unleash T exchanges when used:

$$n : G[T] \quad \text{implies} \quad \mathit{open}\ n : \mathit{Cap}[T]$$

As a consequence, any process that uses a $Cap[T]$ must be a process that is already willing to participate in exchanges of type T, because further T exchanges may be unleashed:

$$M : \mathit{Cap}[T], P : T \quad \text{implies} \quad M.P : T$$

Typing of In and Out The exercise of an *in* or *out* capability cannot cause any exchange, hence such capabilities can be prepended to any process. Following the same pattern we used with **0** and ambients, the silent nature of these capabilities is formalized by allowing them to acquire any capability type:

$$\begin{array}{l} in\ n : Cap[T] \quad \text{for any } T \\ out\ n : Cap[T] \quad \text{for any } T \end{array}$$

Groups Groups are used in the exchange system to specify which kinds of messages can be exchanged inside an ambient. We add a process construct to create a new group G with scope P:

$$(\nu G)P$$

The type rule of this construct specifies that the process P should have an exchange type T that does not contain G. Then, $(\nu G)P$ can be given type T as well. That is, G is never be allowed to "escape" into the type of $(\nu G)P$:

$$P : T, \quad G \text{ does not occur in } T \text{ implies } \quad (\nu G)P : T$$

14 Typed Ambient Calculus

We are now ready for a formal presentation of the typed calculus which has been informally introduced in the previous section. We first present its syntax, then its typing rules, and finally a subject reduction theorem, which states that types are preserved during computation.

14.1 Types and Processes

We first recall the definition of the types of the exchange system.

Types:

$W ::=$	message type
$G[T]$	name in group G for ambients allowing T exchanges
$Cap[T]$	capability unleashing T exchanges
$S, T ::=$	exchange type
Shh	no exchange
$W_1 \times \cdots \times W_k$	tuple exchange (**1** is the null product)

Messages and processes are the same as in the untyped calculus of Section 11.

Messages and Processes:

$M ::=$	message
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

$P, Q, R ::=$	process
$(\nu G)P$	group creation
$(\nu n{:}W)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	action
$(x_1{:}W_1, \ldots, x_k{:}W_k).P$	input action
$\langle M_1, \ldots, M_k \rangle$	output action
go $N.M[P]$	objective move

We identify processes up to consistent renaming of bound names and groups. In the processes $(\nu G)P$ and $(\nu n{:}W)P$, the group G and the name n, respectively, are bound, with scope P. In the process $(x_1{:}W_1, \ldots, x_k{:}W_k).P$, the names x_1, $\ldots$, x_k are bound, with scope P.

The following table defines the free names of processes and messages, and the free groups of processes and types.

Free Names and Free Groups:

$fn((\nu G)P) \triangleq fn(P)$
$fn((\nu n{:}W)P) \triangleq fn(P) - \{n\}$
$fn(\mathbf{0}) \triangleq \varnothing$
$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$
$fn(!P) \triangleq fn(P)$
$fn(M[P]) \triangleq fn(M) \cup fn(P)$
$fn(M.P) \triangleq fn(M) \cup fn(P)$
$fn((x_1{:}W_1, \ldots, x_k{:}W_k).P) \triangleq fn(P) - \{x_1, \ldots, x_k\}$
$fn(\langle M_1, \ldots, M_k \rangle) \triangleq fn(M_1) \cup \cdots \cup fn(M_k)$
$fn(go\ N.M[P]) \triangleq fn(N) \cup fn(M) \cup fn(P)$
$fn(n) \triangleq \{n\}$
$fn(in\ M) \triangleq fn(M)$
$fn(out\ M) \triangleq fn(M)$
$fn(open\ M) \triangleq fn(M)$
$fn(\epsilon) \triangleq \varnothing$
$fn(M.N) \triangleq fn(M) \cup fn(N)$

$fg((\nu G)P) \triangleq fg(P) - \{G\}$
$fg((\nu n{:}W)P) \triangleq fg(W) \cup fg(P)$
$fg(\mathbf{0}) \triangleq \varnothing$
$fg(P \mid Q) \triangleq fg(P) \cup fg(Q)$
$fg(!P) \triangleq fg(P)$
$fg(G[T]) \triangleq \{G\} \cup fg(T)$
$fg(Cap[T]) \triangleq fg(T)$
$fg(Shh) \triangleq \varnothing$
$fg(W_1 \times \cdots \times W_k) \triangleq fg(W_1) \cup \cdots \cup fg(W_k)$

$fg(M[P]) \triangleq fg(P)$
$fg(M.P) \triangleq fg(P)$
$fg((x_1{:}W_1, \ldots, x_k{:}W_k).P) \triangleq fg(W_1) \cup \cdots \cup fg(W_k) \cup fg(P)$
$fg(\langle M_1, \ldots, M_k \rangle) \triangleq \varnothing$
$fg(go\ N.M[P]) \triangleq fg(P)$

The following tables describe the operational semantics of the calculus. The type annotations present in the syntax do not play a role in reduction; they are simply carried along by the reductions.

Processes are identified up to an equivalence relation, $\equiv$, called structural congruence. As in the pi calculus, this relation provides a way of rearranging processes so that interacting parts can be brought together. Then, a reduction relation, $\rightarrow$, acts on the interacting parts to produce computation steps. The core of the calculus is given by the reduction rules (Red In), (Red Out), (Red Go In), (Red Go Out), and (Red Open), for mobility, and (Red I/O), for communication.

The rules of structural congruence are similar to the rules for the pi calculus. The rules (Struct GRes ...) describe the extrusion behaviour of the (νG) binders. Note that (νG) extrudes exactly as (νn) does, hence it does not pose any dynamic restriction on the movement of ambients or messages.

Reduction:

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ \ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	(Red Open)
$\langle M_1, \ldots, M_k \rangle \mid (x_1{:}W_1, \ldots, x_k{:}W_k).P \rightarrow P\{x_1 \leftarrow M_1\} \cdots \{x_k \leftarrow M_k\}$	(Red I/O)
$go(in\ m.N).n[P] \mid m[Q] \rightarrow m[go\ N.n[P] \mid Q]$	(Red Go In)
$m[go(out\ m.N).n[P] \mid Q] \rightarrow go\ N.n[P] \mid m[Q]$	(Red Go Out)
$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$	(Red Par)
$P \rightarrow Q \Rightarrow (\nu n{:}W)P \rightarrow (\nu n{:}W)Q$	(Red Res)
$P \rightarrow Q \Rightarrow (\nu G)P \rightarrow (\nu G)Q$	(Red GRes)
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$	(Red Amb)
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	(Red $\equiv$)

Structural Congruence:

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n{:}W)P \equiv (\nu n{:}W)Q$	(Struct Res)
$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$	(Struct GRes)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)

$P \equiv Q \Rightarrow\ !P \equiv\ !Q$	(Struct Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \equiv Q \Rightarrow$ $(x_1{:}W_1, \ldots, x_k{:}W_k).P \equiv (x_1{:}W_1, \ldots, x_k{:}W_k).Q$	(Struct Input)
$P \equiv Q \Rightarrow go\ N.M[P] \equiv go\ N.M[Q]$	(Struct Go)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$n_1 \neq n_2 \Rightarrow$ $(\nu n_1{:}W_1)(\nu n_2{:}W_2)P \equiv (\nu n_2{:}W_2)(\nu n_1{:}W_1)P$	(Struct Res Res)
$n \notin fn(P) \Rightarrow (\nu n{:}W)(P \mid Q) \equiv P \mid (\nu n{:}W)Q$	(Struct Res Par)
$n \neq m \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P]$	(Struct Res Amb)
$(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P$	(Struct GRes GRes)
$G \notin fg(W) \Rightarrow (\nu G)(\nu n{:}W)P \equiv (\nu n{:}W)(\nu G)P$	(Struct GRes Res)
$G \notin fg(P) \Rightarrow (\nu G)(P \mid Q) \equiv P \mid (\nu G)Q$	(Struct GRes Par)
$(\nu G)m[P] \equiv m[(\nu G)P]$	(Struct GRes Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n{:}W)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$(\nu G)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero GRes)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$\epsilon.P \equiv P$	(Struct ϵ)
$(M.M').P \equiv M.M'.P$	(Struct .)
$go\ \epsilon.M[P] \equiv M[P]$	(Struct Go ϵ)

14.2 Typing Rules

In the tables below, we introduce typing environments, E, the five basic judgments, and the typing rules.

Environments, E, and the Domain, $dom(E)$, of an Environment:

$E ::= \varnothing \mid E, G \mid E, n{:}W$	environment
$dom(\varnothing) \triangleq \varnothing$	
$dom(E, G) \triangleq dom(E) \cup \{G\}$	
$dom(E, n{:}W) \triangleq dom(E) \cup \{n\}$	

Judgments:

$E \vdash \diamond$	good environment
$E \vdash W$	good message type W
$E \vdash T$	good exchange type T
$E \vdash M : W$	good message M of message type W
$E \vdash P : T$	good process P with exchange type T

Good Environments:

$$\text{(Env } \varnothing\text{)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env } n\text{)}\ \frac{E \vdash W \quad n \notin dom(E)}{E, n{:}W \vdash \diamond} \qquad \text{(Env } G\text{)}\ \frac{E \vdash \diamond \quad G \notin dom(E)}{E, G \vdash \diamond}$$

Good Types:

$$\text{(Type Amb)}\ \frac{G \in dom(E) \quad E \vdash T}{E \vdash G[T]} \qquad \text{(Type Cap)}\ \frac{E \vdash T}{E \vdash Cap[T]} \qquad \text{(Type Shh)}\ \frac{E \vdash \diamond}{E \vdash Shh} \qquad \text{(Type Prod)}\ \frac{E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash W_1 \times \cdots \times W_k}$$

Good Messages:

$$\text{(Exp } n\text{)}\ \frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W} \qquad \text{(Exp .)}\ \frac{E \vdash M : Cap[T] \quad E \vdash M' : Cap[T]}{E \vdash M.M' : Cap[T]}$$

$$\text{(Exp } \epsilon\text{)}\ \frac{E \vdash Cap[T]}{E \vdash \epsilon : Cap[T]} \qquad \text{(Exp In)}\ \frac{E \vdash n : G[S] \quad E \vdash T}{E \vdash in\ n : Cap[T]}$$

$$\text{(Exp Out)}\ \frac{E \vdash n : G[S] \quad E \vdash T}{E \vdash out\ n : Cap[T]} \qquad \text{(Exp Open)}\ \frac{E \vdash n : G[T]}{E \vdash open\ n : Cap[T]}$$

Good Processes:

$$\text{(Proc Action)}\ \frac{E \vdash M : Cap[T] \quad E \vdash P : T}{E \vdash M.P : T} \qquad \text{(Proc Amb)}\ \frac{E \vdash M : G[S] \quad E \vdash P : S \quad E \vdash T}{E \vdash M[P] : T}$$

$$\text{(Proc Res)}\ \frac{E, n{:}G[S] \vdash P : T}{E \vdash (\nu n{:}G[S])P : T} \qquad \text{(Proc GRes)}\ \frac{E, G \vdash P : T \quad G \notin fg(T)}{E \vdash (\nu G)P : T}$$

$$\text{(Proc Zero)}\ \frac{E \vdash T}{E \vdash \mathbf{0} : T} \qquad \text{(Proc Par)}\ \frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P \mid Q : T} \qquad \text{(Proc Repl)}\ \frac{E \vdash P : T}{E \vdash !P : T}$$

(Proc Input)
$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : W_1 \times \cdots \times W_k}$$

(Proc Output)
$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k}{E \vdash \langle M_1, \ldots, M_k \rangle : W_1 \times \cdots \times W_k}$$

(Proc Go)
$$\frac{E \vdash N : \mathit{Cap}[S'] \quad E \vdash M : G[S] \quad E \vdash P : S \quad E \vdash T}{E \vdash \mathit{go}\ N.M[P] : T}$$

14.3 Subject Reduction

We obtain a standard subject reduction result. A subtle point, though, is the need to account for the appearance of new groups (G_1, ..., G_k, below) during reduction. This is because reduction is defined up to structural congruence, and structural congruence does not preserve the set of free groups of a process. The culprit is the rule $(\nu n{:}W)\mathbf{0} \equiv \mathbf{0}$, in which groups free in W are not free in $\mathbf{0}$.

Lemma 1 (Subject Congruence). *If $E \vdash P : T$ and $P \equiv Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : T$.*

Theorem 1 (Subject Reduction). *If $E \vdash P : T$ and $P \rightarrow Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : T$.*

Subject reduction specifies that, if P is well-typed, it will only reduce to well-typed terms. This fact has some practical consequences:

- P will never reduce to meaningless processes allowed by the syntax like $(\mathit{in}\ n)[P]$;
- no process deriving from P will contain an ambient where a process attempts an input or output operation which does not match the ambient type.

Subject reduction has also interesting and subtle connections with secrecy of names. Consider a well-typed process $((\nu G).P) \mid O$, where O is a type-checked "opponent", and a name n is declared inside P with a type $G[T]$. Although (νG) can be extruded arbitrarily far, according to the extrusion rules, no process which derives from the opponent O will ever be able to read n through an input $(x{:}W).Q$. Any process $\langle n \rangle \mid (x{:}W).Q$ which derives from $((\nu G).P) \mid O$ is well-typed, hence $W = G[T]$, but the opponent was not, by assumption, in the initial scope of G, and therefore cannot even mention the type $G[T]$. Therefore, we can guarantee that names of group G can never be communicated to processes outside of the initial scope of G, simply because those processes cannot name G to receive the message.

This situation is in sharp contrast with ordinary name restriction, where a name that is initially held secret (e.g., a key) may accidentally be given away

and misused (e.g., to decrypt current or old messages). This is because scoping of names can be extruded too far, inadvertently. Scoping of groups can be extruded as well, but still offers protection against accidental or even malicious leakage.

Of course, we would have even stronger protection if we did not allow (νG) binders to extrude at all. But this would be too rigid. Since (νG) binders can be extruded, they do not impede the mobility of ambients that carry secrets. They just prevent those ambients from giving the secrets away. Consider the following example of travelling agents sharing secrets.

$$\begin{array}{l} a[(\nu G)(\nu k' : G[Shh])(\nu k'' : G[Shh])(\\ \qquad\qquad k'[out\ a.in\ b.out\ b.in\ c] \mid \\ \qquad\qquad k''[out\ a.in\ c.in\ k']) \\ \quad] \mid b[] \mid c[] \end{array}$$

Within an ambient a, two agents share a secret group G and two names k' and k'' belonging to that group. The two agents adopt the names k' and k'' as their respective names, knowing that those names cannot be leaked even by themselves. This way, as they travel, nobody else can interfere with them. If somebody interferes with them, or demonstrates knowledge of the names k' or k'', the agents know that the other party must be (a descendant of) the other agent. In this example, the first agent travels to ambient b and then to c, and the second agent goes to ambient c directly. The scope extrusion rules for groups and names allow this to happen. Inside c, out of the initial scope of (νG), the second agent then interacts with the first by entering it. It can do so because it still holds the shared secret k'.

We omit the proof that structural congruence preserves typing, but we comment here on the crucial case: the preservation of typing by the extrusion rule (Struct GRes Amb). For a well-typed P, $(\nu G)P$ is well-typed if and only if P does not communicate a tuple which names G in its type (rule (Proc GRes)): (νG) must not "see" G-typed names communicated at its own level. This intuition suggests that, referring to the following table, P' should be typeable ((νG) cannot "see" the output $\langle n \rangle$) while P'' should be not ($\langle n \rangle$ is at the same level as (νG)). However, the two processes are equivalent, modulo extrusion of (νG) (rule (Struct GRes Amb)):

$$\begin{array}{l} P' = (\nu G) m[(\nu n{:}G[Shh])\langle n \rangle] \\ P'' = m[(\nu G)(\nu n{:}G[Shh])\langle n \rangle] \end{array}$$

We go through the example step by step, to solve the apparent paradox. First consider the term

$$(\nu G)(\nu n{:}G[Shh])\langle n \rangle$$

This term cannot be typed, because G attempts to escape the scope of $(\nu G)(\nu n{:}G[Shh])$ as the type of the message n. An attempted typing derivation fails at the last step below:

$$
\begin{array}{ll}
\ldots & \\
\Rightarrow & G, n{:}G[Shh] \vdash n : G[Shh] \\
\Rightarrow & G, n{:}G[Shh] \vdash \langle n \rangle : G[Shh] \\
\Rightarrow & G \vdash (\nu n{:}G[Shh])\langle n \rangle : G[Shh] \\
\not\Rightarrow & \vdash (\nu G)(\nu n{:}G[Shh])\langle n \rangle : G[Shh] \text{ (because } G \in fn(G[Shh]))
\end{array}
$$

Similarly, the term

$$(\nu m{:}W)m[(\nu G)(\nu n{:}G[Shh])\langle n \rangle]$$

cannot be typed, because it contains the previous untypeable term. But now consider the following term, which is equivalent to the one above up to structural congruence, by extrusion of (νG) across an ambient boundary:

$$(\nu m{:}W)(\nu G)m[(\nu n{:}G[Shh])\langle n \rangle]$$

This term might appear typeable (contradicting the subject congruence property) because the message $\langle n \rangle{:}G[Shh]$ is confined to the ambient m, and $m[\ldots]$ can be given an arbitrary type, e.g., Shh, which does not contain G. Therefore (νG) would not "see" any occurrence of G escaping from its scope. However, consider the type of m in this term. It must have the form $H[T]$, where H is some group, and T is the type of messages exchanged inside m. But that's $G[Shh]$. So we would have

$$(\nu m{:}H[G[Shh]])(\nu G)m[(\nu n{:}G[Shh])\langle n \rangle]$$

which is not typeable because the first occurrence of G is out of scope.

This example tells us why (νG) *intrusion* (floating inwards) into ambients is not going to break good typing: (νG) cannot enter the scope of the $(\nu m{:}W)$ restriction which creates the name m of an ambient where messages with a G-named type are exchanged. This prevents (νG) from entering such ambients.

Indeed, the following variation (not equivalent to the previous one) is typeable, but (νG) cannot intrude any more:

$$(\nu G)(\nu m{:}H[G[Shh]])m[(\nu n{:}G[Shh])\langle n \rangle]$$

15 Opening Control

Ambient opening is a prerequisite for any communication to happen between processes which did not originate in the same ambient. On the other hand, opening is one of the most delicate operations in the ambient calculus, since the contents of the guest spill inside the host, with two different classes of possible consequences:

– the content of the guest acquires the possibility of performing communications inside the hosts, and of moving the host around;

– the host is now able to examine the content of the guest, mainly in terms of receiving messages sent by the processes inside the guest, and of opening its subambients.

For these reasons, a type system for ambients should support a careful control of the usage of the *open* capability.

15.1 The System

In this section, we enrich the ambient types, $G[T]$, and the capability types, $Cap[T]$, of the previous type system to control usage of the *open* capability.

To control the opening of ambients, we formalize the constraint that the name of any ambient opened by a process is in one of the groups $G_1, \ldots, G_k$, but in no others. To do so, we add an attribute $^\circ\{G_1, \ldots, G_k\}$ to ambient types, which now take the form $G[^\circ\{G_1, \ldots, G_k\}, T]$. A name of this type is in group G, and names ambients within which processes may exchange messages of type T and may only open ambients in the groups $G_1, \ldots, G_k$. We need to add the same attribute to capability types, which now take the form $Cap[^\circ\{G_1, \ldots, G_k\}, T]$. Exercising a capability of this type may unleash exchanges of type T and openings of ambients in groups $G_1, \ldots, G_k$. The typing judgment for processes acquires the form $E \vdash P : {}^\circ\{G_1, \ldots, G_k\}, T$. The pair $^\circ\{G_1, \ldots, G_k\}, T$ constrains both the *opening effects* (what ambients the process opens) and the *exchange effects* (what messages the process exchanges). We call such a pair an *effect*, and introduce the metavariable F to range over effects. It is also convenient to introduce metavariables $\mathbf{G}$, $\mathbf{H}$ to range over finite sets of groups. The following tables summarize these metavariable conventions and our enhanced syntax for types:

Group Sets:

$\mathbf{G}, \mathbf{H} ::= \{G_1, \ldots, G_k\}$	finite set of groups

Types:

$W ::=$	message type
$\quad G[F]$	name in group G for ambients which contain processes with F effects
$\quad Cap[F]$	capability (unleashes F effects)
$F ::=$	effect
$\quad {}^\circ\mathbf{H}, T$	may open $\mathbf{H}$, may exchange T
$S, T ::=$	exchange type
$\quad Shh$	no exchange
$\quad W_1 \times \cdots \times W_k$	tuple exchange

The definition of free groups is the same as in Section 14 except that we redefine $fg(W)$ by the equations $fg(G[F]) = \{G\} \cup fg(F)$ and $fg(Cap[F]) = fg(F)$, and we define $fg(F) = \mathbf{H} \cup fg(T)$ where $F = {}^\circ\mathbf{H}, T$.

The following tables define the type system in detail. There are five basic judgments as before. They have the same format except that the judgment $E \vdash F$, meaning that the effect F is good given environment E, replaces the previous judgment $E \vdash T$. We omit the three rules for deriving good environments; they are exactly as in the previous section. There are two main differences between the other rules below and the rules of the previous section. First, effects, F, replace exchange types, T, throughout. Second, in the rule (Exp Open), the condition $G \in \mathbf{H}$ constrains the opening effect $\mathbf{H}$ of a capability *open* n to include the group G, the group of the name n.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash W$	good message type W
$E \vdash F$	good effect F
$E \vdash M : W$	good message M of message type W
$E \vdash P : F$	good process P with F effects

Good Types:

(Type Amb)
$$\frac{G \in dom(E) \quad E \vdash F}{E \vdash G[F]}$$

(Type Cap)
$$\frac{E \vdash F}{E \vdash Cap[F]}$$

(Effect Shh)
$$\frac{\mathbf{H} \subseteq dom(E) \quad E \vdash \diamond}{E \vdash {}^{\circ}\mathbf{H}, Shh}$$

(Effect Prod)
$$\frac{\mathbf{H} \subseteq dom(E) \quad E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

Good Messages:

(Exp n)
$$\frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

(Exp ϵ)
$$\frac{E \vdash Cap[F]}{E \vdash \epsilon : Cap[F]}$$

(Exp .)
$$\frac{E \vdash M : Cap[F] \quad E \vdash M' : Cap[F]}{E \vdash M.M' : Cap[F]}$$

(Exp In)
$$\frac{E \vdash n : G[F] \quad E \vdash {}^{\circ}\mathbf{H}, T}{E \vdash in\ n : Cap[{}^{\circ}\mathbf{H}, T]}$$

(Exp Out)
$$\frac{E \vdash n : G[F] \quad E \vdash {}^{\circ}\mathbf{H}, T}{E \vdash out\ n : Cap[{}^{\circ}\mathbf{H}, T]}$$

(Exp Open)
$$\frac{E \vdash n : G[{}^{\circ}\mathbf{H}, T] \quad G \in \mathbf{H}}{E \vdash open\ n : Cap[{}^{\circ}\mathbf{H}, T]}$$

Good Processes:

(Proc Action)
$$\frac{E \vdash M : Cap[F] \quad E \vdash P : F}{E \vdash M.P : F}$$

(Proc Amb)
$$\frac{E \vdash M : G[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash M[P] : F'}$$

$$\frac{E, n{:}G[F] \vdash P : F'}{E \vdash (\nu n{:}G[F])P : F'} \text{ (Proc Res)} \qquad \frac{E, G \vdash P : F \quad G \notin fg(F)}{E \vdash (\nu G)P : F} \text{ (Proc GRes)}$$

$$\frac{E \vdash F}{E \vdash \mathbf{0} : F} \text{ (Proc Zero)} \qquad \frac{E \vdash P : F \quad E \vdash Q : F}{E \vdash P \mid Q : F} \text{ (Proc Par)} \qquad \frac{E \vdash P : F}{E \vdash !P : F} \text{ (Proc Repl)}$$

$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k} \text{ (Proc Input)}$$

$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k \quad \mathbf{H} \subseteq dom(E)}{E \vdash \langle M_1, \ldots, M_k \rangle : {}^{\circ}\mathbf{H}, W_1 \times \cdots \times W_k} \text{ (Proc Output)}$$

$$\frac{E \vdash N : Cap[{}^{\circ}\mathbf{H}, T] \quad E \vdash M : G[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash go\ N.M[P] : F'} \text{ (Proc Go)}$$

15.2 Subject Reduction

We obtain a subject reduction result.

Theorem 2. *If $E \vdash P : F$ and $P \to Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : F$.*

Here is a simple example of a typing derivable in this system:

$$G, n{:}G[{}^{\circ}\{G\}, Shh] \vdash n[\mathbf{0}] \mid open\ n.\mathbf{0} : {}^{\circ}\{G\}, Shh$$

This asserts that the whole process $n[\mathbf{0}] \mid open\ n.\mathbf{0}$ is well-typed and opens only ambients in the group G.

On the other hand, one might expect the following variant to be derivable, but it is not:

$$G, n{:}G[{}^{\circ}\{\}, Shh] \vdash n[\mathbf{0}] \mid open\ n.\mathbf{0} : {}^{\circ}\{G\}, Shh$$

This is because the typing rule (Exp Open) requires the effect unleashed by the *open* n capability to be the same as the effect contained within the ambient n. But the opening effect ${}^{\circ}\{\}$ specified by the type $G[{}^{\circ}\{\}, Shh]$ of n cannot be the same as the effect unleashed by *open* n, because (Exp Open) also requires the latter to at least include the group G of n.

This feature of (Exp Open) has a positive side-effect: the type $G[{}^{\circ}\mathbf{G}, T]$ of an ambient name n not only tells which opening effects may happen inside the ambient, but also tells whether n may be opened from outside: it is openable only if $G \in \mathbf{G}$, since this is the only case when $open\, n.\mathbf{0} \mid n[P]$ may be well-typed. Hence, the presence of G in the set $\mathbf{G}$ may either mean that n is meant to be

an ambient within which other ambients in group G may be opened, or that it is meant to be an openable ambient.

More generally, because of the shape of the open rule, the opening effects in the ambient type of n not only record the openings that may take place inside the ambient, but also the opening effects of any ambient m which is going to open n, and, recursively, of any ambient which is going to open m as well. A similar phenomenon occurs with exchange types and with the subjective-crossing effects of the next section.

While this turns out to be unproblematic for the examples we consider in these notes, one may prefer to avoid this "inward propagation" of effects by replacing (Exp Open) with the following rule:

$$\frac{E \vdash n : G[^\circ\mathbf{H}, T]}{E \vdash \mathit{open}\ n : \mathit{Cap}[^\circ(\{G\} \cup \mathbf{H}), T]}$$

With this rule, we could derive that the example process above, $n[\mathbf{0}] \mid \mathit{open}\ n.\mathbf{0}$, has effect $^\circ\{G\}, \mathit{Shh}$, with no need to attribute this effect to processes running inside n itself, but unfortunately, subject reduction fails. To see this, consider the process $\mathit{open}\ n \mid n[\mathit{open}\ m]$, which can be assigned the effect $^\circ\{G, H\}, \mathit{Shh}$:

$$G, H, m{:}G[^\circ\{\}, \mathit{Shh}], n{:}H[^\circ\{G\}, \mathit{Shh}] \vdash \mathit{open}\ n \,|\, n[\mathit{open}\ m] : {}^\circ\{G, H\}, \mathit{Shh}$$

The process reduces in one step to $\mathit{open}\ m$, but we cannot derive the following:

$$G, H, m{:}G[^\circ\{\}, \mathit{Shh}], n{:}H[^\circ\{G\}, \mathit{Shh}] \vdash \mathit{open}\ m : {}^\circ\{G, H\}, \mathit{Shh}$$

To obtain a subject reduction property in the presence of the rule displayed above, we should introduce a notion of subtyping, such that if $\mathbf{G} \subseteq \mathbf{H}$ and a process has type $^\circ\mathbf{G}, T$, then the process has type $^\circ\mathbf{H}, T$ too. This would complicate the type system, as shown in [Zim00]. Moreover, we would lose the indirect way of declaring ambient openability, so we prefer to stick to the basic approach.

16 Crossing Control

This section presents the third and final type system of these notes. We obtain it by enriching the type system of the previous section with attributes to control the mobility of ambients.

16.1 The System

Movement operators enable an ambient n to cross the boundary of another ambient m either by entering it via an $\mathit{in}m$ capability or by exiting it via an $\mathit{out}m$ capability. In the type system of this section, the type of n lists those groups that may be crossed; the ambient n may only cross the boundary of another ambient m if the group of m is included in this list. In our typed calculus, there

are two kinds of movement, subjective moves and objective moves, for reasons explained in Section 16.2. Therefore, we separately list those groups that may be crossed by objective moves and those groups that may be crossed by subjective moves.

We add new attributes to the syntax of ambient types, effects, and capability types. An ambient type acquires the form $G\frown\mathbf{G}'[\frown\mathbf{G},\circ\mathbf{H},T]$. An ambient of this type is in group G, may cross ambients in groups $\mathbf{G}'$ by objective moves, may cross ambients in groups $\mathbf{G}$ by subjective moves, may open ambients in groups $\mathbf{H}$, and may contain exchanges of type T. An effect, F, of a process is now of the form $\frown\mathbf{G},\circ\mathbf{H},T$. It asserts that the process may exercise *in* and *out* capabilities to accomplish subjective moves across ambients in groups $\mathbf{G}$, that the process may open ambients in groups $\mathbf{H}$, and that the process may exchange messages of type T. Finally, a capability type retains the form $Cap[F]$, but with the new interpretation of F. Exercising a capability of this type may unleash F effects.

Types:

$W ::=$	message type
$\quad G\frown\mathbf{G}[F]$	name in group G for ambients which cross $\mathbf{G}$ objectively and contain processes with F effects
$\quad Cap[F]$	capability (unleashes F effects)
$F ::=$	effect
$\quad \frown\mathbf{G},\circ\mathbf{H},T$	crosses $\mathbf{G}$, opens $\mathbf{H}$, exchanges T
$S,T ::=$	exchange type
$\quad Shh$	no exchange
$\quad W_1\times\cdots\times W_k$	tuple exchange

The definition of free groups is the same as in Section 14 except that we redefine $fg(W)$ by the equations $fg(G\frown\mathbf{G}[F])=\{G\}\cup\mathbf{G}\cup fg(F)$ and $fg(Cap[F])=fg(F)$, and we define $fg(F)=\mathbf{G}\cup\mathbf{H}\cup fg(T)$ where $F=\frown\mathbf{G},\circ\mathbf{H},T$.

The format of the five judgments making up the system is the same as in Section 15. We omit the three rules defining good environments; they are as in Section 14. There are two main changes to the previous system to control mobility. First, (Exp In) and (Exp Out) change to assign a type $Cap[\frown\mathbf{G},\circ\mathbf{H},T]$ to capabilities *in* n and *out* n only if $G\in\mathbf{G}$ where G is the group of n. Second, (Proc Go) changes to allow an objective move of an ambient of type $G\frown\mathbf{G}'[F]$ by a capability of type $Cap[\frown\mathbf{G},\circ\mathbf{H},T]$ only if $\mathbf{G}=\mathbf{G}'$.

Good Types:

(Type Amb)
$$\frac{G\in dom(E)\quad \mathbf{G}\subseteq dom(E)\quad E\vdash F}{E\vdash G\frown\mathbf{G}[F]}$$

(Type Cap)
$$\frac{E\vdash F}{E\vdash Cap[F]}$$

(Effect Shh)
$$\frac{\mathbf{G}\subseteq dom(E)\quad \mathbf{H}\subseteq dom(E)\quad E\vdash\diamond}{E\vdash\frown\mathbf{G},\circ\mathbf{H},Shh}$$

(Effect Prod)
$$\frac{\mathbf{G} \subseteq dom(E) \quad \mathbf{H} \subseteq dom(E) \quad E \vdash W_1 \quad \cdots \quad E \vdash W_k}{E \vdash \frown\mathbf{G},^\circ\mathbf{H}, W_1 \times \cdots \times W_k}$$

Good Messages:

(Exp n)
$$\frac{E', n{:}W, E'' \vdash \diamond}{E', n{:}W, E'' \vdash n : W}$$

(Exp ϵ)
$$\frac{E \vdash Cap[F]}{E \vdash \epsilon : Cap[F]}$$

(Exp .)
$$\frac{E \vdash M : Cap[F] \quad E \vdash M' : Cap[F]}{E \vdash M.M' : Cap[F]}$$

(Exp In)
$$\frac{E \vdash n : G \frown \mathbf{G}'[F] \quad E \vdash \frown\mathbf{G},^\circ\mathbf{H}, T \quad G \in \mathbf{G}}{E \vdash in\ n : Cap[\frown\mathbf{G},^\circ\mathbf{H}, T]}$$

(Exp Out)
$$\frac{E \vdash n : G \frown \mathbf{G}'[F] \quad E \vdash \frown\mathbf{G},^\circ\mathbf{H}, T \quad G \in \mathbf{G}}{E \vdash out\ n : Cap[\frown\mathbf{G},^\circ\mathbf{H}, T]}$$

(Exp Open)
$$\frac{E \vdash n : G \frown \mathbf{G}'[\frown\mathbf{G},^\circ\mathbf{H}, T] \quad G \in \mathbf{H}}{E \vdash open\ n : Cap[\frown\mathbf{G},^\circ\mathbf{H}, T]}$$

Good Processes:

(Proc Action)
$$\frac{E \vdash M : Cap[F] \quad E \vdash P : F}{E \vdash M.P : F}$$

(Proc Amb)
$$\frac{E \vdash M : G \frown \mathbf{G}[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash M[P] : F'}$$

(Proc Res)
$$\frac{E, n{:}G \frown \mathbf{G}[F] \vdash P : F'}{E \vdash (\nu n{:}G \frown \mathbf{G}[F])P : F'}$$

(Proc GRes)
$$\frac{E, G \vdash P : F \quad G \notin fg(F)}{E \vdash (\nu G)P : F}$$

(Proc Zero)
$$\frac{E \vdash F}{E \vdash \mathbf{0} : F}$$

(Proc Par)
$$\frac{E \vdash P : F \quad E \vdash Q : F}{E \vdash P \mid Q : F}$$

(Proc Repl)
$$\frac{E \vdash P : F}{E \vdash !P : F}$$

(Proc Input)
$$\frac{E, n_1{:}W_1, \ldots, n_k{:}W_k \vdash P : \frown\mathbf{G},^\circ\mathbf{H}, W_1 \times \cdots \times W_k}{E \vdash (n_1{:}W_1, \ldots, n_k{:}W_k).P : \frown\mathbf{G},^\circ\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Output)
$$\frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k \quad \mathbf{G} \subseteq dom(E) \quad \mathbf{H} \subseteq dom(E)}{E \vdash \langle M_1, \ldots, M_k \rangle : {\frown}\mathbf{G}, {\circ}\mathbf{H}, W_1 \times \cdots \times W_k}$$

(Proc Go)
$$\frac{E \vdash N : Cap[{\frown}\mathbf{G}, {\circ}\mathbf{H}, T] \quad E \vdash M : G{\frown}\mathbf{G}[F] \quad E \vdash P : F \quad E \vdash F'}{E \vdash go\ N.M[P] : F'}$$

Theorem 3. *If $E \vdash P : F$ and $P \to Q$ then there are $G_1, \ldots, G_k$ such that $G_1, \ldots, G_k, E \vdash Q : F$.*

16.2 The Need for Objective Moves

We can now show how primitive typing rules for objective moves allow us to assign better types in some crucial situations. Recall the untyped example from Section 11. Suppose we have two groups Ch and Pk (for channels and packets). Let W be any well-formed type (where Ch and Pk may appear), and set P to be the example process:

$$P = a[p[out\ a.in\ b.\langle c \rangle]] \mid b[open\ p.(x{:}W).x[]]$$

Let

$$\begin{aligned} E = {} & Ch, Pk, \\ & a{:}Ch\,{\frown}\{\}[{\frown}\{\}, {\circ}\{\}, Shh], \\ & b{:}Ch\,{\frown}\{\}[{\frown}\{Ch\}, {\circ}\{Pk\}, W], \\ & c{:}W, \\ & p{:}Pk\,{\frown}\{\}[{\frown}\{Ch\}, {\circ}\{Pk\}, W] \end{aligned}$$

and we can derive the typings:

$$\begin{aligned} & E \vdash out\ a.in\ b.\langle c \rangle : {\frown}\{Ch\}, {\circ}\{Pk\}, W \\ & E \vdash open\ p.(x{:}W).x[] : {\frown}\{Ch\}, {\circ}\{Pk\}, W \\ & E \vdash P : {\frown}\{\}, {\circ}\{\}, Shh \end{aligned}$$

From the typing $a : Ch\,{\frown}\{\}[{\frown}\{\}, {\circ}\{\}, Shh]$, we can tell that a is an immobile ambient in which nothing is exchanged and that cannot be opened. From the typings $p{:}Pk\,{\frown}\{\}[{\frown}\{Ch\}, {\circ}\{Pk\}, W], b{:}Ch\,{\frown}\{\}[{\frown}\{Ch\}, {\circ}\{Pk\}, W]$, we can tell that the ambients b and p cross only Ch ambients, open only Pk ambients, and contain W exchanges; the typing of p also tells us it can be opened. This is not fully satisfactory, since, if b were meant to be immobile, we would like to express this immobility invariant in its type. However, since b opens a subjectively mobile ambient, then b must be typed as if it were subjectively mobile itself. The problem is quite general, as it applies to any immobile ambient wishing to open a subjectively mobile one.

This problem can be solved by replacing the subjective moves by objective moves, since objective moves are less expressive than subjective moves, but they

cannot be inherited by opening another ambient. Let Q be the example process with objective instead of subjective moves:

$$Q = a[go(out\ a.in\ b).p[\langle c\rangle]] \mid b[open\ p.(x{:}W).x[]]$$

Let

$$\begin{aligned} E = {} & Ch, Pk, \\ & a{:}Ch \curvearrowright \{\}[\curvearrowright\{\}, {}^\circ\{\}, Shh], \\ & b{:}Ch \curvearrowright \{\}[\curvearrowright\{\}, {}^\circ\{Pk\}, W], \\ & c{:}W, \\ & p{:}Pk \curvearrowright \{Ch\}[\curvearrowright\{\}, {}^\circ\{Pk\}, W] \end{aligned}$$

and we can derive:

$$\begin{aligned} & E \vdash out\ a.in\ b : Cap[\curvearrowright\{Ch\}, {}^\circ\{\}, Shh] \\ & E \vdash go(out\ a.in\ b).p[\langle c\rangle] : \curvearrowright\{\}, {}^\circ\{\}, Shh \\ & E \vdash open\ p.(x{:}W).x[] : \curvearrowright\{\}, {}^\circ\{Pk\}, W \\ & E \vdash Q : \curvearrowright\{\}, {}^\circ\{\}, Shh \end{aligned}$$

The typings of a and c are unchanged, but the new typings of p and b are more informative. We can tell from the typing $p{:}Pk \curvearrowright \{Ch\}[\curvearrowright\{\}, {}^\circ\{Pk\}, W]$ that movement of p is due to objective rather than subjective moves. Moreover, as desired, we can tell from the typing $b{:}Ch \curvearrowright \{\}[\curvearrowright\{\}, {}^\circ\{Pk\}, W]$ that the ambient b is immobile.

This example suggests that in some situations objective moves lead to more informative typings than subjective moves. Still, subjective moves are essential for moving ambients containing running processes. An extended example in the next section illustrates the type system of this section; the treatment of thread mobility makes essential use of subjective moves.

17 Encoding a Distributed Language

In this section, we consider a fragment of a typed, distributed language in which mobile threads can migrate between immobile network nodes. We obtain a semantics for this form of thread mobility via a translation into the ambient calculus. In the translation, ambients model both threads and nodes. The encoding can be typed in all three of the systems presented in these notes; for the sake of brevity we describe the encoding only for the full system of Section 16. The encoding illustrates how groups can be used to partition the set of ambient names according to their intended usage, and how opening and crossing control allows the programmer to state interesting invariants. In particular, the typing of the translation guarantees that an ambient modelling a node moves neither subjectively nor objectively. On the other hand, an ambient modelling a thread is free to move subjectively, but is guaranteed not to move objectively.

17.1 The Distributed Language

The computational model is that there is an unstructured collection of named network nodes, each of which hosts a collection of named communication channels and anonymous threads. This is similar to the computational models underlying various distributed variants of the pi calculus, such as those proposed by Amadio and Prasad [AP94], Riely and Hennessy [RH98], and Sewell [Sew98]. In another paper [CG99], we show how to mimic Telescript's computational model by translation into the ambient calculus. In the language fragment we describe here, communication is based on named communication channels (as in the pi calculus) rather than by direct agent-to-agent communication (as in our stripped down version of Telescript). As in our previous paper, we focus on language constructs for mobility, synchronization, and communication. We omit standard constructs for data processing and control flow. They could easily be added.

To introduce the syntax of our language fragment, here is a simple example:

$$\begin{array}{l} \mathit{node}\ a\ [\mathit{channel}\ a_c \mid \mathit{thread}[\overline{a_c}\langle b, b_c\rangle]] \mid \mathit{node}\ b\ [\mathit{channel}\ b_c] \mid \\ \mathit{node}\ c\ [\mathit{thread}[\mathit{go}\ a.a_c(x{:}\mathit{Node}, y{:}\mathit{Ch}[\mathit{Node}]).\mathit{go}\ x.\overline{y}\langle a\rangle] \end{array}$$

This program describes a network consisting of three network nodes, named a, b, and c. Node a hosts a channel a_c and a thread running the code $\overline{a_c}\langle b, b_c\rangle$, which simply sends the pair $\langle b, b_c\rangle$ on the channel a_c. Node b hosts a channel b_c. Finally, node c hosts a single thread, running the code:

$$\mathit{go}\ a.a_c(x{:}\mathit{Node}, y{:}\mathit{Ch}[\mathit{Node}]).\mathit{go}\ x.\overline{y}\langle a\rangle$$

The effect of this is to move the thread from node c to node a. There it awaits a message sent on the communication channel a_c. We may assume that it receives the message $\langle b, b_c\rangle$ being sent by the thread already at a. (If there were another thread at node a sending another message, the receiver thread would end up receiving one or other of the messages.) The thread then migrates to node b, where it transmits a message a on the channel b_c.

Messages on communication channels are assigned types, ranged over by Ty. The type Node is the type of names of network nodes. The type $\mathit{Ch}[\mathit{Ty}_1, \ldots, \mathit{Ty}_k]$ is the type of a polyadic communication channel. The messages communicated on such a channel are k-tuples whose components have types $\mathit{Ty}_1, \ldots, \mathit{Ty}_k$. In the setting of the example above, channel a_c has type $\mathit{Ch}[\mathit{Node}, \mathit{Ch}[\mathit{Node}]]$, and channel b_c has type $\mathit{Ch}[\mathit{Node}]$.

Next, we describe the formal grammar of our language fragment. A *network*, Net, is a collection of nodes, built up using composition $\mathit{Net} \mid \mathit{Net}$ and restrictions $(\nu n{:}\mathit{Ty})\mathit{Net}$. A *crowd*, Cro, is the group of threads and channels hosted by a node. Like networks, crowds are built up using composition $\mathit{Cro} \mid \mathit{Cro}$ and restriction $(\nu n{:}\mathit{Ty})\mathit{Cro}$. A *thread*, Th, is a mobile thread of control. As well as the constructs illustrated above, a thread may include the constructs $\mathit{fork}(\mathit{Cro}).\mathit{Th}$ and $\mathit{spawn}\ n\ [\mathit{Cro}].\mathit{Th}$. The first forks a new crowd Cro inside the current node, and continues with Th. The second spawns a new node $\mathit{node}\ n\ [\mathit{Cro}]$ outside the current node, at the network level, and continues with Th.

A Fragment of a Typed, Distributed Programming Language:

$Ty ::=$	type
$\quad Node$	name of a node
$\quad Ch[Ty_1, \ldots, Ty_k]$	name of a channel
$Net ::=$	network
$\quad (\nu n{:}Ty)Net$	restriction
$\quad Net \mid Net$	network composition
$\quad node\ n\ [Cro]$	node
$Cro ::=$	crowd of channels and threads
$\quad (\nu n{:}Ty)Cro$	restriction
$\quad Cro \mid Cro$	crowd composition
$\quad channel\ c$	channel
$\quad thread[Th]$	thread
$Th ::=$	thread
$\quad go\ n.Th$	migration
$\quad \overline{c}\langle n_1, \ldots, n_k\rangle$	output to a channel
$\quad c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$	input from a channel
$\quad fork(Cro).Th$	fork a crowd
$\quad spawn\ n\ [Cro].Th$	spawn a new node

In the phrases $(\nu n{:}Ty)Net$ and $(\nu n{:}Ty)Cro$, the name n is bound; its scope is Net and Cro, respectively. In the phrase $c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$, the names x_1, $\ldots$, x_k are bound; their scope is the phrase Th.

The type system of our language controls the typing of messages on communication channels, much as in previous schemes for the pi calculus [Mil99]. We formalize the type system using five judgments, defined by the following rules.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash n : Ty$	name n has type Ty
$E \vdash Net$	good network
$E \vdash Cro$	good crowd
$E \vdash Th$	good thread

Typing Rules:

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash \diamond \quad n \notin dom(E)}{E, n{:}Ty \vdash \diamond} \qquad \frac{E, n{:}Ty, E' \vdash \diamond}{E, n{:}Ty, E' \vdash n : Ty} \qquad \frac{E, n{:}Ty \vdash Net}{E \vdash (\nu n{:}Ty)Net}$$

$$\frac{E \vdash Net \quad E \vdash Net'}{E \vdash Net \mid Net'} \qquad \frac{E \vdash n : Node \quad E \vdash Cro}{E \vdash node\ n\ [Cro]} \qquad \frac{E, n{:}Ty \vdash Cro}{E \vdash (\nu n{:}Ty)Cro}$$

$$\frac{E \vdash Cro \quad E \vdash Cro'}{E \vdash Cro \mid Cro'} \qquad \frac{E \vdash c : Ch[Ty_1, \ldots, Ty_k]}{E \vdash channel\ c} \qquad \frac{E \vdash Th}{E \vdash thread[Th]}$$

$$\frac{E \vdash n : Node \quad E \vdash Th}{E \vdash go\, n.Th} \qquad \frac{E \vdash c : Ch[Ty_1, \ldots, Ty_k] \quad E \vdash n_i : Ty_i \quad \forall i \in 1..k}{E \vdash \overline{c}\langle n_1, \ldots, n_k \rangle}$$

$$\frac{E \vdash c : Ch[Ty_1, \ldots, Ty_k] \quad E, x_1{:}Ty_1, \ldots, x_k{:}Ty_k \vdash Th}{E \vdash c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th}$$

$$\frac{E \vdash Cro \quad E \vdash Th}{E \vdash fork(Cro).Th} \qquad \frac{E \vdash n : Node \quad E \vdash Cro \quad E \vdash Th}{E \vdash spawn\, n\, [Cro].Th}$$

17.2 Typed Translation to the Ambient Calculus

In this section, we translate our distributed language to the typed ambient calculus of Section 16.

The basic idea of the translation is that ambients model nodes, channels, and threads. For each channel, there is a name for a buffer ambient, of group Ch^b, and there is a second name, of group Ch^p, for packets exchanged within the channel buffer. Similarly, for each node, there is a name, of group $Node^b$, for the node itself, and a second name, of group $Node^p$, for short-lived ambients that help fork crowds within the node, or to spawn other nodes. Finally, there is a group Thr to classify the names of ambients that model threads. The following table summarizes these five groups:

Global Groups Used in the Translation:

$Node^b$	ambients that model nodes
$Node^p$	ambients to help fork crowds or spawn nodes
Ch^b	ambients that model channel buffers
Ch^p	ambients that model packets on a channel
Thr	ambients that model threads

We begin the translation by giving types in the ambient calculus corresponding to types in the distributed language. Each type Ty gets translated to a pair $[\![Ty]\!]^b$, $[\![Ty]\!]^p$ of ambient calculus types. Throughout this section, we omit the curly braces when writing singleton group sets; for example, we write $\curvearrowright Node^b$ as a shorthand for $\curvearrowright\{Node^b\}$.

First, if Ty is a node type, $[\![Ty]\!]^b$ is the type of an ambient (of group $Node^b$) modelling a node, and $[\![Ty]\!]^p$ is the type of helper ambients (of group $Node^p$). Second, if Ty is a channel type, $[\![Ty]\!]^b$ is the type of an ambient (of group Ch^b) modelling a channel buffer, and $[\![Ty]\!]^p$ is the type of a packet ambient (of group Ch^p).

Translations $[\![Ty]\!]^b$, $[\![Ty]\!]^p$ of a Type Ty:

$$[\![Node]\!]^b \triangleq Node^b \curvearrowright Node^b[\curvearrowright\{\}, \circ Node^p, Shh]$$

$$
\begin{array}{l}
[\![Node]\!]^p \triangleq \\
\quad Node^p \curvearrowright Thr[\curvearrowright\{\},{}^\circ Node^p, Shh] \\
[\![Ch[Ty_1, \ldots, Ty_k]]\!]^b \triangleq \\
\quad Ch^b \curvearrowright\{\}[\curvearrowright\{\},{}^\circ Ch^p, [\![Ty_1]\!]^b \times [\![Ty_1]\!]^p \times \cdots \times [\![Ty_k]\!]^b \times [\![Ty_k]\!]^p] \\
[\![Ch[Ty_1, \ldots, Ty_k]]\!]^p \triangleq \\
\quad Ch^p \curvearrowright\{Thr, Ch\}[\curvearrowright\{\},{}^\circ Ch^p, [\![Ty_1]\!]^b \times [\![Ty_1]\!]^p \times \cdots \times [\![Ty_k]\!]^b \times [\![Ty_k]\!]^p]
\end{array}
$$

These typings say a lot about the rest of the translation, because of the presence of five different groups. Nodes and helpers are silent ambients, whereas tuples of ambient names are exchanged within both channel buffers and packets. None of these ambients is subjectively mobile—in this translation only threads are subjectively mobile. On the other hand, nodes and helpers may both objectively cross nodes, while buffers are objectively immobile, and packets objectively cross both threads and buffers. Finally, both nodes and helpers may open only helpers, and both buffers and packets may open only packets (actually, the ${}^\circ Ch^p$ annotation inside the type of a packet c^p of group Ch^p means that c^p can be opened, and similarly for helpers).

Next, we translate networks to typed processes. A restriction of a single name is sent to restrictions of a couple of names: either names for a node and helpers, if the name is a node, or names for a buffer and packets, if the name is a channel. A composition is simply translated to a composition. A network node n is translated to an ambient named n^b representing the node, containing a replicated $open\, n^p$, where n^p is the name of helper ambients for that node.

Translation $[\![Net]\!]$ of a Network Net:

$$
\begin{array}{l}
[\![(\nu n{:}Ty)Net]\!] \triangleq (\nu n^b{:}[\![Ty]\!]^b)(\nu n^p{:}[\![Ty]\!]^p)[\![Net]\!] \\
[\![Net \mid Net]\!] \triangleq [\![Net]\!] \mid [\![Net]\!] \\
[\![node\ n\ [Cro]]\!] \triangleq n^b[!open\ n^p \mid [\![Cro]\!]_n]
\end{array}
$$

The translation $[\![Cro]\!]_n$ of a crowd is indexed by the name n of the node in which the crowd is located. Restrictions and compositions in crowds are translated like their counterparts at the network level. A channel c is represented by a buffer ambient c^b of group Ch^b. It is initially empty but for a replicated $open\, c^p$, where c^p is the name, of group Ch^p, of packets on the channel. The replication allows inputs and outputs on the channel to meet and exchange messages.

An ambient of the following type models each thread:

$$Thr \curvearrowright\{\}[\curvearrowright Node^b, {}^\circ Sync, Shh]$$

From the type, we know that a thread ambient is silent, that it crosses node boundaries by subjective moves but crosses nothing by objective moves, and that it may only open ambients in the $Sync$ group. Such ambients help synchronize parallel processes in thread constructs such as receiving on a channel. A fresh group named $Sync$ is created by a $(\nu Sync)$ in the translation of each

thread. The existence of a separate lexical scope for *Sync* in each thread implies there can be no accidental transmission between threads of the names of private synchronization ambients.

Translation $[\![Cro]\!]_n$ of a Crowd Cro Located at Node n:

$$[\![(\nu m{:}Ty)\,Cro]\!]_n \triangleq (\nu m^b{:}[\![Ty]\!]^b)(\nu m^p{:}[\![Ty]\!]^p)[\![Cro]\!]_n$$
$$[\![Cro \mid Cro]\!]_n \triangleq [\![Cro]\!]_n \mid [\![Cro]\!]_n$$
$$[\![channel\ c]\!]_n \triangleq c^b[!open\ c^p]$$
$$[\![thread\ Th]\!]_n \triangleq$$
$$(\nu Sync)(\nu t{:}Thr \curvearrowright \{\}[\curvearrowright Node^b, \circ Sync, Shh])t[[\![Th]\!]^t_n] \quad \text{for } t \notin fn([\![Th]\!]^t_n)$$

The translation $[\![Th]\!]^t_n$ of a thread is indexed by the name t of the thread and by the name n of the node in which the thread is enclosed.

A migration $go\ m.Th$ is translated to subjective moves taking the thread t out of the current node n and into the target node m.

An output $\overline{c}\langle n_1, \ldots, n_k\rangle$ is translated to a packet ambient c^p that travels to the channel buffer c^b, where it is opened, and outputs a tuple of names.

An input $c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th$ is translated to a packet ambient c^p that travels to the channel buffer c^b, where it is opened, and inputs a tuple of names; the tuple is returned to the host thread t by way of a synchronization ambient s, that exits the buffer and then returns to the thread.

A fork $fork(Cro).Th$ is translated to a helper ambient n^p that exits the thread t and gets opened within the enclosing node n. This unleashes the crowd Cro and allows a synchronization ambient s to return to the thread t, where it triggers the continuation Th.

A spawn $spawn\ m\ [Cro].Th$ is translated to a helper ambient n^p that exits the thread t and gets opened within the enclosing node n^b. This unleashes an objective move $go(out\ n^b).m^b[!open\ m^p \mid [\![Cro]\!]_m]]$ that travels out of the node to the top, network level, where it starts the fresh node $m^b[!open\ m^p \mid [\![Cro]\!]_m]]$. Concurrently, a synchronization ambient s returns to the thread t, where it triggers the continuation Th.

Translation $[\![Th]\!]^t_n$ of a Thread Th Named t Located at Node n:

$$[\![go\ m.Th]\!]^t_n \triangleq out\ n.in\ m.[\![Th]\!]^t_m$$
$$[\![\overline{c}\langle n_1, \ldots, n_k\rangle]\!]^t_n \triangleq go(out\ t.in\ c^b).c^p[\langle n_1, n_1^p, \ldots, n_k, n_k^p\rangle]$$
$$[\![c(x_1{:}Ty_1, \ldots, x_k{:}Ty_k).Th]\!]^t_n \triangleq$$
$$(\nu s{:}Sync \curvearrowright \{Thr, Ch\}[\curvearrowright Node^b, \circ Sync, Shh])$$
$$(go(out\ t.in\ c^b).$$
$$c^p[(x_1^b{:}[\![Ty_1]\!]^b, x_1^p{:}[\![Ty_1]\!]^p, \ldots, x_k^b{:}[\![Ty_k]\!]^b, x_k^p{:}[\![Ty_k]\!]^p).$$
$$go(out\ c^b.in\ t).s[open\ s.[\![Th]\!]^t_n]] \mid$$
$$open\ s.s[])$$
$$\text{for } s \notin \{t, c^b, c^p\} \cup fn([\![Th]\!]^t_n)$$

$$
\begin{array}{l}
[\![fork(Cro).Th]\!]^t_n \triangleq \\
\quad (\nu s{:}Sync\curvearrowright Thr[\curvearrowright Node^b, \circ Sync, Shh]) \\
\quad\quad (go\ out\ t.n^p[go\ in\ t.s[] \mid [\![Cro]\!]_n] \mid open\ s.[\![Th]\!]^t_n) \\
\quad \text{for } s \notin \{t, n^p\} \cup [\![Cro]\!]_n \cup [\![Th]\!]^t_n \\
[\![spawn\ m\ [Cro].Th]\!]^t_n \triangleq \\
\quad (\nu s{:}Sync\curvearrowright Thr[\curvearrowright Node^b, \circ Sync, Shh]) \\
\quad\quad (go\ out\ t.n^p[go\ in\ t.s[] \mid go\ out\ n^b.m^b[!open\ m^p \mid [\![Cro]\!]_m]] \mid \\
\quad\quad\ open\ s.[\![Th]\!]^t_n) \\
\quad \text{for } s \notin \{t, n^b, n^p, m^b, m^p\} \cup fn([\![Cro]\!]_m) \cup fn([\![Th]\!]^t_n)
\end{array}
$$

Finally, we translate typing environments as follows.

Translation $[\![E]\!]$ of an Environment E:

$$
\begin{array}{rl}
[\![\varnothing]\!] & \triangleq Node^b, Node^p, Ch^b, Ch^p, Thr \\
[\![E, c{:}Ty]\!] & \triangleq [\![E]\!], c^b{:}[\![Ty]\!]^b, c^p{:}[\![Ty]\!]^p
\end{array}
$$

Our translation preserves typing judgments:

Proposition 1.

(1) *If* $E \vdash Net$ *then* $[\![E]\!] \vdash [\![Net]\!] : \curvearrowright\{\}, \circ\{\}, Shh$.
(2) *If* $E \vdash Cro$ *and* $E \vdash n : Node$ *then* $[\![E]\!] \vdash [\![Cro]\!]_n : \curvearrowright\{\}, \circ\{\}, Shh$.
(3) *If* $E \vdash Th$, $E \vdash n : Node$, $t \notin dom(E)$ *then*
$[\![E]\!], Sync, t{:}Thr\curvearrowright\{\}[\curvearrowright Node^b, \circ Sync, Shh] \vdash [\![Th]\!]^t_n : \curvearrowright Node^b, \circ Sync, Shh$.

Apart from having more refined types, this translation is the same as a translation to the type system with binary annotations of [CGG99]. The translation shows that ambients can model a variety of concepts arising in mobile computation: nodes, threads, communication packets and buffers. Groups admit more precise typings for this translation than were possible in the system with binary annotations. For example, here we can tell that a thread ambient subjectively crosses only node ambients, but never crosses helpers, buffers, or packets, and that it is objectively immobile; in the binary system, all we could say was that a thread ambient was subjectively mobile and objectively immobile.

18 Discussion: The Ambient Calculus

In this part, we introduced the ambient calculus as an abstract model for the mobility of hardware and software. We explained some of the type systems that have been proposed for ambients. We gave an application of the calculus as a semantic metalanguage for describing distributed computation. Our full type system tracks the communication, mobility, and opening behaviour of ambients, which are classified by *groups*. A group represents a collection of ambient names; ambient names belong to groups in the same sense that values belong to types. We studied the properties of a new process operator $(\nu G)P$ that lexically scopes

groups. Using groups, our type system can impose behavioural constraints like "this ambient crosses only ambients in one set of groups, and only dissolves ambients in another set of groups".

18.1 Related Work on Types

The ambient calculus is related to earlier distributed variants of the pi calculus, some of which have been equipped with type systems. The type system of Amadio [Ama97] prevents a channel from being defined at more than one location. Sewell's system [Sew98] tracks whether communications are local or non-local, so as to allow efficient implementation of local communication. In Riely and Hennessy's calculus [RH98], processes need appropriate permissions to perform actions such as migration; a well-typed process is guaranteed to possess the appropriate permission for any action it attempts. Other work on typing for mobile agents includes a type system by De Nicola, Ferrari, and Pugliese [DFP99] that tracks the access rights an agent enjoys at different localities; type-checking ensures that an agent complies with its access rights.

Our groups are similar to the *sorts* used as static classifications of names in the pi calculus [Mil99]. Our basic system of Section 14 is comparable to Milner's sort system, except that sorts in the pi calculus are mutually recursive; we would have to add a recursion operator to achieve a similar effect. Another difference is that an operator for sort creation does not seem to have been considered in the pi calculus literature. Our operator for group creation can guarantee secrecy properties, as we show in the setting of a typed pi calculus equipped with groups [CGG00b]. Our systems of Sections 15 and 16 depend on groups to constrain the opening and crossing behaviour of processes. We are not aware of any uses of Milner's sorts to control process behaviour beyond controlling the sorts of communicated names.

Apart from Milner's sorts, other static classifications of names occur in derivatives of the pi calculus. We mention two examples. In the type system of Abadi [Aba99] for the spi calculus, names are classified by three static *security levels*—*Public*, *Secret*, and *Any*—to prevent insecure information flows. In the flow analysis of Bodei, Degano, Nielson, and Nielson [BDNN98] for the pi calculus, names are classified by static *channels* and *binders*, again with the purpose of establishing security properties. (Similar flow analyses now exist for the ambient calculus [NNHJ99, HJNN99].) Although there is a similarity between these notions and groups, and indeed to sorts, nothing akin to our (νG) operator appears to have been studied.

There is a connection between groups and the region variables in the work of Tofte and Talpin [TT97] on region-based implementation of the λ-calculus. The store is split into a set of stack-allocated regions, and the type of each stored value is labelled with the region in which the value is stored. The scoping construct $letregion\,\rho\,in\,e$ allocates a fresh region, binds it to the region variable ρ, evaluates e, and on completion, deallocates the region bound to ρ. The constructs $letregion\ \rho\ in\ e$ and $(\nu G)P$ are similar in that they confer static scopes on the region variable ρ and the group G, respectively. One difference is that in

our operational semantics $(\nu G)P$ is simply a scoping construct; it allocates no storage. Another is that scope extrusion laws do not seem to have been explicitly investigated for *letregion*. Still, we can interpret *letregion* in terms of (νG), as is reported elsewhere [DG00].

18.2 Related Work on Ambients

The introduction to this part, Section 10, is extracted from the original article on the ambient calculus [CG00b]; more motivations may be found in another paper [Car99], which develops a graphical metaphor for ambients, the folder calculus.

The rest of this part reports research into type systems for the ambient calculus, some parts of which have been described in conference papers. In [CG99] we have investigated *exchange types*, which subsume standard type systems for processes and functions, but do not impose restrictions on mobility; no groups were present in that system. In [CGG99] we have reported on *immobility* and *locking* annotations, which are basic predicates about mobility, still with no notion of groups; Zimmer [Zim00] proposes inference algorithms for a generalization of this type system. In [CGG00a] we introduce the notion of groups; much of this part of the notes is drawn from that paper.

As well as work on types, there has been work on a variety of other techniques for reasoning about the ambient calculus. In [GC99], we define a form of testing equivalence for the ambient calculus, akin to the testing equivalence we introduced in Part II for the spi calculus; we develop some techniques for proving testing equivalence including a context lemma. Several papers investigate program logics for the ambient calculus; in [CG00a] we introduce a logic with both spatial modalities—for talking about the structure of ambient processes—and temporal modalities—for talking about their evolution over time. A recent paper extends the first with modal operators to express properties of restricted names [CG01]. Two other papers investigate the equivalence induced by the logic [San01] and the complexity of the model checking problem [CDG+01].

Levi and Sangiorgi [LS00] propose a variant of the calculus called Safe Ambients. As well as the original in, out, and $open$ capabilities, they introduce three dual capabilities, written $\overline{in}$, $\overline{out}$, and $\overline{open}$, respectively. To enter a sibling named n, an ambient needs to exercise the $out\ n$ capability, as before, but additionally, the sibling needs to exercise the $\overline{out}\ n$ capability. Similarly, to exit its parent named n, an ambient needs to exercise the $out\ n$ capability, as before, but additionally, the parent needs to exercise the $\overline{out}\ n$ capability. To dissolve an ambient named n, its environment needs to exercise the $open\ n$ capability, as before, but additionally, the ambient itself needs to exercise the $\overline{open}\ n$ capability. The resulting ambient calculus is a little more complicated than the one described here, but the advantages shown by Levi and Sangiorgi are that certain race conditions may be avoided, and in some ways more accurate typings are possible. Bugliese and Castagna [BC01] investigate an extension of Safe Ambients intended to describe security properties; their notion of ambient domain is akin to the notion of group discussed in these notes.

The first implementation of the ambient calculus was a Java applet [Car97]. More recent implementations support mobility between machines distributed on a network; they include an implementation of the original calculus using Jocaml [FLS00], and of Safe Ambients using Java RMI [SV01].

Acknowledgement C.A.R. Hoare and G. Castagna commented on a draft of these notes.

References

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999. 289, 323

[ABLP93] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993. 273

[AF01] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115, 2001. 288, 289

[AFG98] M. Abadi, C. Fournet, and G. Gonthier. Secure communications implementation of channel abstractions. In *13th IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 105–116, 1998. 289

[AG98] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5:267–303, 1998. 289

[AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999. An extended version with full proofs appears as Digital Equipment Corporation Systems Research Center report No. 149, January 1998. 264, 266

[Ama97] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*. Springer, 1997. 293, 323

[AN96] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996. 281

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS'94)*, volume 880 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 1994. 317

[AP99] R. Amadio and S. Prasad. The game of the name in cryptographic tables. In *Advances in Computing Science (ASIAN'99)*, volume 1742 of *Lectures Notes in Computer Science*, pages 5–26. Springer, 1999. 289

[AR00] M. Abadi and P. Rogaway. Reconciling two views of cryptography. In *Theoretical Computer Science (IFIP TCS 2000)*, volume 1872 of *Lectures Notes in Computer Science*, pages 3–33. Springer, 2000. 273

[BAN89] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. 270, 272, 273, 274

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 81–94, 1990. 284

[BC01] M. Bugliesi and G. Castagna. Secure safe ambients. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 222–235, 2001. 324

[BDNN98] C. Bodei, P. Degano, F. Nielson, and H. Nielson. Control flow analysis for the π-calculus. In *CONCUR'98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1998. 323

[BN95] M. Boreale and R. De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, August 1995. 272, 286

[BNP99] M. Boreale, R. De Nicola, and R. Pugliese. Proof techniques for cryptographic processes. In *14th IEEE Symposium on Logic in Computer Science*, pages 157–166, 1999. 289

[Box98] D. Box. *Essential COM*. Addison-Wesley, 1998. 263

[BR95] M. Bellare and P. Rogaway. Provably secure session key distribution: The three party case. In *27th ACM Symposium on Theory of Computing*, 1995. 273

[BV01] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. *Agent Systems Journal*, 2001. To appear. A preliminary version appears in the proceedings of the *3rd International Symposium on Mobile Agents (MA/ASA'99)*, 1999. 266

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. 265, 292

[Car97] L. Cardelli. Mobile ambient synchronization. SRC Technical Note 1997–013, Digital Equipment Corporation Systems Research Center, July 1997. 325

[Car99] L. Cardelli. Abstractions for mobile computation. In C. Jensen and J. Vitek, editors, *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999. 265, 324

[CDG+01] W. Charatonik, S. Dal Zilio, A. D. Gordon, S. Mukhopadhyay, and J.-M. Talbot. The complexity of model checking mobile ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'01)*, Lecture Notes in Computer Science. Springer, 2001. To appear. 324

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989. 292

[CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 79–92, January 1999. 266, 298, 317, 324

[CG00a] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 365–377, 2000. 324

[CG00b] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000. 265, 266, 294, 297, 324

[CG01] L. Cardelli and A. D. Gordon. Logical properties of name restriction. In *Typed Lambda Calculi and Applications (TLCA'01)*, Lecture Notes in Computer Science. Springer, 2001. To appear. 324

[CGG99] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In *26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, volume 1644 of *Lecture Notes in Computer Science*, pages 230–239. Springer, 1999. 322, 324

[CGG00a] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Theoretical Computer Science (IFIP TCS 2000)*, volume 1872 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2000. 266, 324

[CGG00b] L. Cardelli, G. Ghelli, and A. D. Gordon. Group creation and secrecy. In C. Palamidessi, editor, *CONCUR 2000—Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2000. 323

[CGZ95] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer, 1995. 292

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988. 297

[Dam96] M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996. 272

[Dam98] M. Dam. Proving trust in systems of second-order processes. In *31st Hawaii International Conference on System Sciences*, volume VII, pages 255–264, 1998. 272

[DES77] Data encryption standard. Fed. Inform. Processing Standards Pub. 46, National Bureau of Standards, Washington DC, January 1977. 274

[DFP99] R. De Nicola, G. Ferrari, and R. Pugliese. Types as specifications of access policies. In *Secure Internet Programming 1999*, volume 1603 of *Lecture Notes in Computer Science*, pages 117–146. Springer, 1999. 323

[DG00] S. Dal Zilio and A. D. Gordon. Region analysis and a π-calculus with groups. In *Mathematical Foundations of Computer Science 2000 (MFCS2000)*, volume 1893 of *Lectures Notes in Computer Science*, pages 1–21. Springer, 2000. 324

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976. 287

[DH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984. 286

[DS81] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981. 264

[DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983. 273

[FG94] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *Journal of Computer Security*, 3(1):5–33, 1994. 273

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the Join-calculus. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385, 1996. 264, 266, 293

[FGL+96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR'96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1996. 266, 293

[FLS00] C. Fournet, J.-J. Lévy, and A. Schmitt. An asynchronous, distributed implementation of mobile ambients. In *Theoretical Computer Science (IFIP TCS 2000)*, volume 1872 of *Lecture Notes in Computer Science*. Springer, 2000. 266, 325

[GC99] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1999. An extended version appears as Microsoft Research Technical Report MSR–TR–99–11, April 1999. 324

[GJ01] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. Submitted for publication. Available at *http://research.microsoft.com/~adg/Publications*, 2001. 289

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley, 1996. 292

[GM95] J. Gray and J. McLean. Using temporal logic to specify and verify cryptographic protocols (progress report). In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 108–116, 1995. 273

[GS01] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 248–260, 2001. 297

[HJNN99] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1999. 323

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985. 263

[IK01] A. Igarashi and N. Kobayashi. A generic type system for the pi calculus. In *28th ACM Symposium on Principles of Programming Languages*, pages 128–141, 2001. 272

[Kem89] R. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, 1989. 273

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992. 273

[Lie93] A. Liebl. Authentication in distributed systems: A bibliography. *ACM Operating Systems Review*, 27(4):31–41, 1993. 272

[LMMS98] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Fifth ACM Conference on Computer and Communications Security*, pages 112–121, 1998. 273

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lectures Notes in Computer Science*, pages 147–166. Springer, 1996. 264, 273

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 352–364, 2000. 324

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. 297

[Mao96] W. Mao. On two proposals for on-line bankcard payments using open networks: Problems and solutions. In *IEEE Symposium on Security and Privacy*, pages 201–210, 1996. 289

[MCF87] J. K. Millen, S. C. Clark, and S. B. Freedman. The interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, SE–13(2):274–288, 1987. 273

[Mea92] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992. 273

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989. 263

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992. 283

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. 263, 272, 298, 318, 323

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, pages 1–40 and 41–77, 1992. 263, 272

[MPW93] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993. 272

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In *19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, volume 623 of *Lecture Notes in Computer Science*. Springer, 1992. 272

[Nee89] R. M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. Addison-Wesley, 1989. 263

[NFP97] R. De Nicola, G.-L. Ferrari, and R. Pugliese. Locality based linda: programming with explicit localities. In *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 712–726. Springer, 1997. 293

[NNHJ99] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *CONCUR'99: Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 463–477. Springer, 1999. 323

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978. 264, 272

[Ode00] M. Odersky. Functional nets. In *European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000. 264

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. 272

[PT00] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. 264

[RH98] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 378–390, 1998. 292, 317, 323

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. 287

[San01] D. Sangiorgi. Extensionality and intensionality of the ambient logics. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 4–13, 2001. 324

[Sch96a] S. Schneider. Security properties and CSP. In *IEEE Symposium on Security and Privacy*, pages 174–187, 1996. 273

[Sch96b] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., second edition, 1996. 286

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 1998. 292, 317, 323

[SV01] D. Sangiorgi and A. Valente. A distributed abstract machine for Safe Ambients. Available from the authors, 2001. 325

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001. 263, 272

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 323

[VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, pages 47–77. Springer, 1999. 266

[Whi96] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/The MIT Press, 1996. 265, 292

[WL93] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993. 289

[Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'00)*, volume 1784 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2000. 300, 312, 324