MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Type systems and static analysis for mobile and distributed computing: state of the art

MIKADO Deliverable 2.1.0

Title :	Type systems and static analysis for mobile and distributed computing: state of the art
Editor :	M. Hennessy
Authors :	E. Giovannetti,M. Hennessy, J. Rathke
Classification :	Deliverable
Reference :	RR/WP2/0
Version :	0.0
Date :	December 2002

Contents

1 Introduction

Static typing is now a well-developed and widely used technology for avoiding run-time errors in high-level programming languages. As a simple example one might want to avoid, during program execution, boolean variables being assigned integer or real values. The consequences of such assignments are in general disastrous as it usually entails memory being corrupted. Rather than implementing costly run-time checks to forbid such behaviour one can do a *compile time* static analysis on the program code which tries to predict if variables will ever be assigned inappropriate values. Typing is essentially one form of static analysis which establishes an *invariant* of the code; a well-typed program

- does not misuse variables
- remains well-typed during execution.

This simple idea has been successfully extended to many other scenarios:

- ensuring functions in functional languages such as ML, are only applied to appropriate arguments; and methods in object-oriented languages such as Java are only called for appropriate objects;
- memory management: determining where and when memory can be allocated and deallocated in functional languages; [TT97].

- compiling: typed assembly languages have been developed in [MCGW98] which support high-level programming abstractions and ensure that these are not violated by well-typed assembly programs.

In these applications the form of types used, and the type inference, is often quite sophisticated. Of particular interest to us is the use of *annotated* types, (also called *type and effect systems*), in which types are decorated with representations of intensional behaviour. Assigning a program fragment such a type means not only that it conforms to the requirements of that type but also some part of its intentional behaviour conforms to the annotation. For example in [TT97] cited above the types used are annotated with *memory regions* and the type inference indicates, statically, in which regions data items are allocated, and when deallocation of a region is safe. Similarly in [MCGW98] the types may refer to specific registers and code allocated such types must have appropriate values allocated to those registers.

In this deliverable we give a brief overview of the recent trends in these *annotated* typing systems which we believe are of most relevant to the aims of Mikado. We intend to develop typing systems for controlling the behaviour of distributed systems. This is a rather large remit but we intend to focus on certain areas, which on the one hand are of particular interest to the project, and on the other offer reasonable chances of success within the constraints, in terms of time and manpower, of the project.

Accordingly this review concentrates on three topics:

Resource access control: Resource control has long been of concern in the design of operating systems, particularly multiprocessor systems, and many techniques have evolved for managing access. In wide area distributed systems this concern is even more acute. We discuss briefly the problems, and some current work on addressing the issues using types.

Mobility control: In wide area distributed systems, it is expected that much of the computation and organisation of the network will be carried out by mobile agents or migrating code, which travel between network hosts, updating information, installing new processes, seeking resources, changing network connectivity, etc.. Controlling where such agents are allowed to migrate, and under what circumstances, is essential to the proper management of distributed networks, and to the integrity of individual hosts. One approach to *mobility control* is, of course, using annotated types; for example

- the particular type at which you learn a host name may determine your ability to migrate there, or even your capabilities once you arrive.
- the particular mobility type at which an agent is generated may determine to extent to which it can roam around the network.

Luckily there has already been considerable work on *mobility types* for one particular distributed process language, namely *ambients*, [CG98a]. So we give a fairly detailed review of this work.

Information Control: This is a more delicate issue. In multi-user systems it has long been important to control the flow of sensitive information; here not only must access to information be controlled but also its propagation, both explicit and implicit. There is already a well-established literature in control mechanisms, [BL75, Lam74], for regulating such information flows, although there does not yet seem to be an accepted definition of what exactly constitutes an *implicit information flow*. More recent work have investigated various type systems for enforcing these mechanisms, particularly in sequential programs, [SV98], but also to some extent also in shared-memory concurrent programs. We will give an outline of these type systems with a view to their application in the Mikado project.

2 Resource Access Control and Secure Code

Current trends in computing, and especially distributed computing, are such that the execution of untrusted code on one's own system is becoming commonplace. Clearly, we wish to protect our systems from malicious abuse of system resources by this untrusted code. Abuse may take different forms, such as simple unauthorised access of resources, or, in a more sophisticated system, over consumption of resources. The typical approach taken to protect systems is to install some kind of monitoring in the operating system. Resource guardians can be set in place to oversee use of any given resource by making run time checks whenever a resource is called for. This is of course a costly business and may degrade system performance to such an extent so as to render the solution unworkable. An alternative approach is to offer a static analysis at compile time as a means of checking resource access. This can be achieved through the use of types.

An immediate problem which arises if one adopts this route is the choice of the language level at which protection is provided. For instance, one may ask that all foreign, untrusted, code be provided as source code of a specific high-level language. We can then take this source code and, as part of the compilation process, use type-checking to guarantee against bad resource usage. Alternatively, we could ask that we receive machine code, assembly language instructions, or intermediate level code, such as Java byte-code, suitably annotated with type information to facilitate our static checks.

There are instances of each of these approaches in the literature. We now proceed to examine some of these and describe the nature of protection each offer along with the technology used to do so. The research we consider is not intended to provide an exhaustive summary of all such protection mechanisms, rather, a representative sample of the state of the art in this area.

2.1 Type annotated assembly language

In this section we consider the use of type annotations in low-level code. The two main vehicles for these ideas are Necula and Lee's Proof-Carrying Code (PCC) [NL96] and Morrisett, Walker, Crary and Glew's Typed Assembly Language (TAL) [MCGW98].

The motivation for PCC derives from the use of operating system kernel extensions supplied by the manufacturer, or more importantly, untrusted sources. Administrators would like assurance that these extensions will not cause havoc in their system. The approach taken is roughly described as follows: the kernel, or *code consumer*, provides a *safety policy* against which it will verify any untrusted native-code binaries supplied by an untrusted *code producer*. The native-code binaries must be created in a specific form called *proof-carrying code*. The binary will in fact contain a representation of an actual proof that the supplied code meets the stated safety policy. These proofs must be constructed in such a way as to facilitate simple checking and in this respect will resemble type derivation trees. The safety policy is specified as some kind of logical predicate, along with axioms and inference rules which may be used to derive this predicate. Figure 1, taken from [NL96], describes the proof-carrying code mechanism succinctly.

The flexibility of the PCC approach, in that it uses first-order predicate logic with arithmetic, allows for sophisticated safety policies to be expressed. Unfortunately, this flexibility seems to limit its applicability also. The key technical difficulty seems to be generating the proofs for large policies to embed in the code. Automated theorem provers can be of assistance here but the limitations are still considerable.

The research on Typed Assembly Language uses a more constrained approach. Rather than employing the full power of predicate logic, instead they use a language of types to express the safety properties and annotate assembly language instructions with type information. This allows straightforward type checking according to pre-defined type inference rules, which is tantamount to checking validity of a proof. In this sense, TAL may be viewed as a restricted form of PCC, with the benefit that the proofs are easily generable. In common with PCC, both the (high-level) source and (low-level) target languages for TAL compilers use

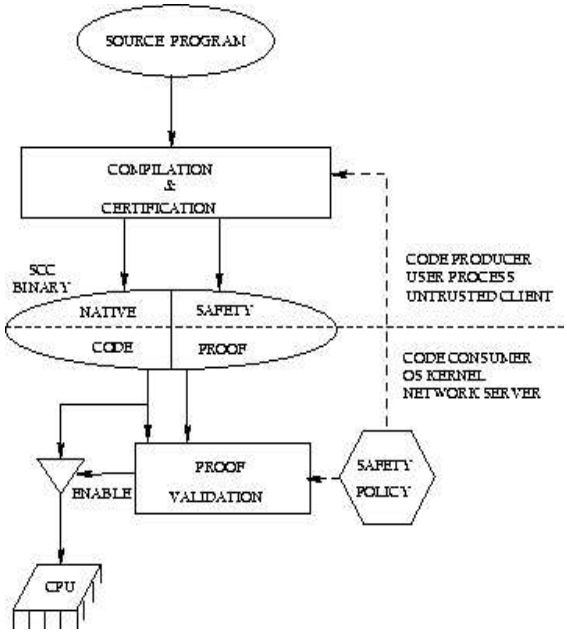

Figure 1: The process begins with the code consumer defining and publicising a safety policy. This policy defines formally what is meant by "safety" and also specifies the interface between the consumer and any binary provided by the producer. Taking the policy into account, the code producer compiles (or assembles) and proves the safety of a source program, through a process which we call certification. This results in a binary that can be delivered to the code consumer. Upon receipt, the consumer validates the safety proof enclosed in the binary. Finally, if the proof is found to be valid, the code consumer can safely execute the native-code part of the binary.

type annotations. Initial work on TAL concentrated on providing type safe compilations of polymorphic functional languages into TAL [MCGW98]. The role of the type system here was to enforce high-level abstractions such as closures, tuples, and objects. In subsequent work [MCG+99], other safety properties were addressed such as safe stack allocation for active records, separate compilation and linking, as well as several high-level abstractions including polymorphism and recursive types.

Also using the TAL approach, work in [CW00] focuses on safety properties in which resource consumption is monitored quantitatively and operation within certain consumption bounds is guaranteed. In this work they present a decidable type system capable of specifying and certifying bounds on resource consumption. In particular they concentrate on running-time bounds by augmenting the semantics with a virtual clock to monitor time usage. The onus is on the programmer to actually discover the usage bounds and annotate the source code with them. Crary and Weirich's system then compiles annotated code to certified TAL. This system appears to be flexible enough to cater for other kinds of resource usage, such as stack space, and its main feature appears to be the subtle use of a simple form of dependent types through sum types and inductive kinds.

2.2 Typed intermediate languages

The use of type annotations at the low level of assembly code does offer great flexibility for static analysis of code but certain safety properties do not seem best suited to reasoning at this low-level. An alternative approach is to compile a type annotated high-level language down to some typed mid-way representation between source and target code which still incorporates some high-level abstractions. The best known example of this approach is probably Java Byte-code [LY96] although the safety properties expressible there are limited.

A system proposed by Crary, Walker and Morrisett [CWM99] also makes use of a typed intermediate language called the Calculus of Capabilities. This supports region-based memory management so that type safety can be verified before a straightforward compilation to TAL. The typed intermediate language supports explicit operations for allocating, freeing and accessing memory regions along with type information expressed as the capability to perform these operations. The main technical challenge here seemed to be the control of pointer aliasing. For instance, if a particular region of memory pointed to by p has been allowably freed, it must be checked that no accesses are made to the region of memory q if q is in fact an alias of p. Various techniques, with limited use, for this sort of control exist in the literature, using linear types, or syntactic control of interference, but Crary, Walker and Morrisett employ the use of (non-)alias tracking to ensure safety. As in [CW00] no provision is made for type inference; instead region inference algorithms must be employed before type checking can occur.

An interesting use of intermediate languages can also be found in [Wal00]. Walker proposes the use of *certified code* as a general mechanism for enforcing security policies. First, a security policy is specified using a *security automata* [Sch00]. Then the programmer provides source code written in a high-level language which is believed to adhere to the policy. The compiler then instruments the source code with run-time checks to guarantee adherence to the policy — this of course may result in bad programs being terminated. These run-time checks are then statically analysed using the type system and removed where it is provably safe to do so. Essentially, the run-time check is replaced with a type annotation so that static verification can still take place. The benefits of this approach is that it blends a very expressive mechanism for specifying safety properties [Sch00], with the power of static analysis. The instrumentation phase guarantees that safe, although not necessarily statically verifiably safe, code will be produced.

2.3 Types for resource access control in mobile systems

The nearest thing we have to a canonical semantic model for mobile agents is the π-calculus [Mil99, SW01]. There have been various proposals for type systems for π-calculus, all of which make some use of the notion of channel type. The first incarnation of resource access control that we have seen in this setting appears in [PS96]. Here it is proposed that not only do the types record channel information about what sort of data is intended to be transmitted along it but also the allowed use of the channel. We can view a channel as some sort of object with both send and receive methods and type the channel as one might type such objects. Essentially the type system in [PS96] uses record types for channels where records contain both channel send types and channel receive types. The use of subtyping then allows for restricted access to the send and receive methods in the same way that methods are hidden using upwards casts in object-oriented languages. The i/o-subtyping systems for π-calculus have been refined to include receptivity types and linear types [SW01] and type inference algorithms for such systems proposed [IK00].

The π-calculus makes no explicit mention of distribution in computations. This has led to a surfeit of proposals for extending this core language with a notion of location and primitives for process mobility. Many of type systems proposed for such languages tend to be concerned with mobility control and we will survey these in Section 3. However, we make mention here of the language Dπ of Hennessy and Riely [HR02]. Dπ is a slight extension of the π-calculus which includes processes executing at named locations,

along with a single primitive to allow migration of code. The type systems proposed for Dπ are specifically tailored to study the problem of resource access control. Each location in the system is given a record type which contains the list of resources in existence at that location and their permitted modes of use. This is modelled using, a variation on the I/O subtyping of Pierce and Sangiorgi, by identifying resources with channels. The use of subtyping enables processes to offer fine-grained control of their resources by only distributing location names at types which offer restricted capabilities. An interesting feature of the type systems here is that they make use of a limited form of dependent types in that channel names, or resources, are language level entities, which also may appear in location types. Correspondingly, the type inference rules and proofs of type safety can become complex.

A very similar framework is developed for the coordination language Klaim in [NFPV00]. Here again there are explicit localities and both data and processes can be transfered between them, although there is a differentiation between logical locations and physical sites. The type system checks that migrating processes have the correct permissions to perform the *co-ordination* operations of read, write and execute.

2.4 Behavioural types in concurrent systems

The notion of type often used in type systems for the π-calculus are static in nature. By this we mean that the existence of a channel, or resource, of a given type takes no account of how the availability of that resource may change over time.

In a similar vein to the approaches used for TAL and typed intermediate languages, it has been proposed [RV00, Yos96, Pun97] that behavioural types which reflect the dynamics of the resources are appropriate. The common feature here is that types are built using some notion of labelled graph and type checking is done with respect to these graphs. The nodes of the graph represent states of the system and the labelled transitions the changes in state as the resource named in the label is accessed. Various notions of safety property are proposed for such type systems. An interesting one is given, for example, in [RV00]: the goal of this paper is to study type safety in a system in which concurrent objects methods are not persistently available. Behavioural types are proposed as a means of tackling this problem and the notion of run-time safety which types guarantee is that whenever an object method is called then this method will eventually become available.

2.5 Generalised systems

Finally, we mention recent work on attempts to provide a general formalisation of resource access control policies. Igarashi and Kobayashi, [IK02], provide a formal definition of what they refer to as a *resource usage analysis problem* by making an analogy with the definition of a flow analysis problem [NNH99]. The definition is as follows: suppose each expression of a program is labelled with labels drawn from some set $\mathcal{L}$. Let $\mathcal{L}^*$ denote finite strings of such labels. A resource usage analysis problem is defined to be the problem of

1. computing a function $\mathsf{use} : \mathcal{L} \to 2^{\mathcal{L}^*}$ such that $l_1 l_2 \ldots l_n \in \mathsf{use}(l)$ implies any value generated by an expression labelled l may be accessed by primitives labelled with $l_1, l_2, \ldots, l_n$ (in that order).

2. checking whether $\mathsf{use}(l)$ contains only valid access sequences (according to some policy)

This is a general means of stating a resource access control problem which seems to cover all known instances. Igarashi and Kobayashi then go on to present a generic type system for a small functional language with primitives for resource manipulation. They also provide a usage inference algorithm to infer types, rather than relying on programmer annotation. This makes their system rather attractive but it is unclear at this stage how powerful it is. It is also difficult to compare it against the solutions discussed in

previous subsections, as they rely on programmer annotation and consequently allow for more properties to be verified.

3 Mobility Control

As stated in the Introduction here we confine our attention to a discussion of the ambient calculi, and for simplicity we only discuss *monadic* versions. Moreover we assume the reader is familiar with at least its reduction semantics of the basic version, namely Mobile Ambients, [CG98b]; discussion of this may be found in Deliverable D1.1.1.

Although here we are primarily interested in ambients as a focus for discussion on the use of types to control mobility it will be useful to first discuss the basic framework of typing ambients. This is the topic of the first subsection, which is followed by one on mobility types. We then have two further subsections on interesting variations on Mobile Ambients, namely *Safe Ambients* and *Boxed Ambients*.

3.1 Introduction

The pure calculus of Mobile Ambients (MA) [CG98b] with only the mobility primitives is a completely untyped calculus of foundational character, analogous to untyped λ-calculus, π-calculus, etc. But as soon as it is completed with the communication primitives, a type system is used to prevent trivial runtime errors, as will be made clear below.

Following Cardelli and Gordon's seminal work [CG99], all the ambient calculi that have been introduced over the years come equipped with type systems. In this section we review a representative sample of the most fruitful research directions that have been pursued in the area, exemplified by summary descriptions of a small number of systems.

The usual presentations of ambient calculi rely, apriori, on typing to keep the (untyped) syntax and operational semantics as simple as possible; instead of having two categories of variables along with two corresponding versions of the communication primitives, one for ambient names and the other for capabilities, they only define one kind of construct and variable, thus allowing meaningless terms to be defined, or to be produced by communication of wrong categories. For example, one may obtain an ambient whose name is an action, $\langle \mathsf{in}\ m \rangle \,|\, (x)\,.\,x[P] \ \rightarrow\ (\mathsf{in}\ m)[P]$, or vice-versa an action consisting of an ambient name, $\langle m \rangle \,|\, (x)\,.\,x\,.\,P \ \rightarrow\ m\,.\,P$.

Such anomalies are then eliminated by a simple type system where the two syntactic categories become two message types: ambients and capabilities. The fact that ambient calculi, though inheriting from process calculi the usual constructs for process composition and communication, do not have (named) channels, has an important consequence: since the reduction rules do not contain types, the above undesired reductions may only be ruled out by ensuring that input and output constructs of different types can never coexist in a situation where they may synchronise with each other.

In most ambient calculi communication may only occur locally within an ambient; each ambient may thus be viewed as an anonymous local channel available to all its processes. Each ambient has therefore to be assigned one type, the so-called topic of conversation, and all processes within it may only communicate messages of this type. Adopting, for the sake of uniformity, a slightly different notation from that of Cardelli and Gordon's, we indicate by $\mathsf{amb}(T)$ the type of ambients where the topic of conversation is T, and by $\mathsf{proc}(T)$ the type of processes that perform communication of type T. The basic rules, for process types, are then given in the first part of Figure 2. The first two, (T-AMB) and (T-PAR) are found (possibly with appropriate variations) in most type systems for ambients. Note that in the conclusion of the former the process $m[P]$ may be assigned any (process) type, since an ambient-process cannot perform any communication and may therefore be safely put in any ambient.

Process types:

$$\frac{\Gamma \vdash m : \mathsf{amb}(T) \quad \Gamma \vdash P : \mathsf{proc}(T)}{\Gamma \vdash m[P] : \mathsf{proc}(S)} \text{(T-AMB)}$$

$$\frac{\Gamma \vdash P : \mathsf{proc}(T) \quad \Gamma \vdash Q : \mathsf{proc}(T)}{\Gamma \vdash P \,|\, Q : \mathsf{proc}(T)} \text{(T-PAR)}$$

$$\frac{\Gamma, x{:}W \vdash P : \mathsf{proc}(W)}{\Gamma \vdash (x{:}W)\,.\,P : \mathsf{proc}(W)} \text{(T-INPUT)}$$

$$\frac{\Gamma \vdash M : W}{\Gamma \vdash \langle M \rangle : \mathsf{proc}(W)} \text{(T-OUTPUT)}$$

$$\frac{(\Gamma \text{ is a well-formed env.})}{\Gamma \vdash 0 : \mathsf{proc}(T)} \text{(T-NULL)}$$

Capabilities:

$$\frac{\Gamma \vdash M : \mathsf{amb}(S)}{\Gamma \vdash \mathsf{in}\ M : \mathsf{cap}(T)} \text{(T-IN)}$$

$$\frac{\Gamma \vdash M : \mathsf{amb}(S)}{\Gamma \vdash \mathsf{out}\ M : \mathsf{cap}(T)} \text{(T-OUT)}$$

$$\frac{\Gamma \vdash M : \mathsf{amb}(T)}{\Gamma \vdash \mathsf{open}\ M : \mathsf{cap}(T)} \text{(T-OPEN)}$$

$$\frac{\Gamma \vdash M_1 : \mathsf{cap}(T) \quad \Gamma \vdash M_2 : \mathsf{cap}(T)}{\Gamma \vdash M_1.M_2 : \mathsf{cap}(T)} \text{(T-PATH)}$$

$$\frac{\Gamma \vdash M : \mathsf{cap}(T) \quad \Gamma \vdash P : \mathsf{proc}(T)}{\Gamma \vdash M.P : \mathsf{proc}(T)} \text{(T-PREF)}$$

Figure 2: Typing for Mobile Ambients

The type of a process P is of course determined by P's input and output operations, which must be all of the same type; The resulting rules for communication, (T-IN) and (T-OUT) are therefore natural. Note also that s a starting point for typing derivations a null process axiom, (T-NULL) is required; we leave unstated the well-formedness requirement on type environments.

There are a further set of rules for inferring capability types. Consider, as an example, the case of the open action, which dissolves an ambient's boundary and merges its content with the parent's one. The two ambients involved in the action must have the same topics of conversation. The type of open m must therefore record the internal topics of m, and that requires capability types to be of the form $\mathsf{cap}(T)$; see the rule (T-OPEN).

On the other hand the in and out actions, not triggering any communication, may be assigned any capability type, as in the rules (T-IN), (T-OUT).

Since ambient calculi allow communication of ambient names and capabilities (or paths), but not of processes, the message type W can only be of the forms $\mathsf{amb}(T)$ and $\mathsf{cap}(T)$. The definitions of type expressions are therefore mutually recursive. As with types for the π-calculus a basis is needed to start the inductive construction of types and in order to keep the calculus pure, such a basic type is suitably chosen as a topic called shh, characterising ambients where no communication occurs. Types T of topics are thus not totally coincident with message types W, since shh cannot be a message type, i.e., it cannot type an input variable or an output expression.

Summarising, the type syntax for Mobile ambients is given in Figure 3. Examples of types are:

- $\mathsf{amb}(\mathsf{shh})$, the type of silent ambients;

$\mathsf{proc}(T)$ *type of processes performing communication of type* T

$W ::=$ *message types*
- $\mathsf{amb}(T)$ names of ambients where the topics of conversation is T
- $\mathsf{cap}(T)$ capabilities compatible with the topics T

$T, S ::=$ *types of topics (also called exchange types, or communication types)*
- shh no communication
- W communication of messages of type W

Figure 3: Types for Pure Ambients

- $\mathsf{amb}(\mathsf{amb}(\mathsf{shh}))$, the type of ambients where silent ambient names may be exchanged;
- $\mathsf{cap}(\mathsf{amb}(\mathsf{shh}))$, the type of actions that may be performed by a process that communicates silent ambient names; etc.

Figure 2 also contains rules for prefixing paths of capabilities to processes, (T-PATH) and (T-PREF); note that in the latter, in order to type $M.P$ the type of M and that of P must have the same conversation. The other typing rules needed to complete the system, including those for restriction and replication, are straightforward and omitted.

It is worth remarking on the different roles played by ambient types and process types, in any type system for ambients. Ambients are names, therefore an ambient's typing always immediately results from an assumption. On the other hand a process is a term of an arbitrary syntactic complexity, which is reflected in the complexity of the typing derivation, from assumptions concerning names and variables.

In general, a typing m : *AmbProp* means that the property *AmbProp* is assumed to hold for any occurrence of a subterm of the form $m[P]$ within the term representing the global program. If such term is well typed, then we may be sure that the property actually holds and will keep holding during the computation; of course this assumes that Subject Reduction holds, which is usually the case.

3.2 Mobility Types for Mobile Ambients

Before discussing mobility types it is worthwhile digressing a little to explain the use of *groups* in ambient types.

3.2.1 Groups and Group Restriction

A refinement of the archetypal system sketched in the previous subsection is presented in [CGG00], where the notion of a *group* of ambients is introduced.

Syntactically, groups are merely names G occurring as further constituents of ambient types; again using a slightly different notation from the original, we use $\mathsf{amb}(G, T)$ for the type of ambients of topics T and group G. The syntax does not prevent the existence of distinct types for distinct ambients "belonging" to the same group. The typing rules reported in the previous subsection are unaffected (except for the obvious replacement of $\mathsf{amb}(T)$ with the new form).

The important thing, however, is that the syntax of the calculus itself is modified, with the introduction of the restriction on groups, $\nu G.P$. This kind of restriction, which concerns ambient *types* instead of ambient names, is very useful for the preservation of name secrecy; it enables limits to be put on the scope extrusion of ambient names.

The new construct does not affect the operational semantics; the relevant reduction rule is

$$P \to Q \implies \nu G\,.\,P \to \nu G\,.\,Q$$

But it makes types a more intrinsic part of the calculus. Group names might be viewed as type variables of a *kind* $\mathbb{G}$ (the class of all possible groups), which, being the only kind, may be omitted. The corresponding natural typing rule is:

$$(\text{T-G} - \text{RES})\ \frac{\Gamma, G : \mathbb{G} \vdash P : \mathsf{proc}(T) \qquad G \notin \mathit{freegroups}(T)}{\Gamma \vdash \nu G\,.\,P : \mathsf{proc}(T)}$$

However, now environments in judgements need to be sequences and not sets, and group names may be used in environment assumptions only after being explicitly declared. In a different and less formal style, where environments are sets, and group names need not be explicitly declared, the rule could be written as:

$$(\text{T-G} - \text{RES})\ \frac{\Gamma \vdash P : \mathsf{proc}(T) \qquad G \notin \mathit{freegroups}(T) \qquad G \textit{ does not occur in } \Gamma}{\Gamma \vdash \nu G\,.\,P : \mathsf{proc}(T)}$$

In the original calculus MA, if a well-typed process $P = \nu m{:}\mathsf{amb}(\mathsf{shh}).(\langle M\rangle \mid Q)$ is present in a system represented by a larger term, the scope of the restriction, initially limited to P, may be dynamically extended through communication and mobility, and the name m, initially not known outside P, may be transmitted arbitrarily far. Thus an initially secret name may be inadvertently given away by bad programming.

On the contrary, a well-typed process of the form $\nu G.\nu m{:}\mathsf{amb}(G, \mathsf{shh}).P$, in whatever context it is placed, is guaranteed not to give away the name m, since group restriction, by hiding an essential constituent – the group G – of the *type* of the secret name, forbids in any other process even the definition of a well-typed input construct $(x{:}\mathsf{amb}(?, \mathsf{shh})).Q$ capable of receiving that name. At the same time, the possibility of scope extrusion also for group names ensures that group restriction "does not impede mobility of ambients that are enclosed in the initial scope of fresh groups but later move away" [CGG00].

Finally, observe that a process of the form

$$\nu G\,.\,\nu n{:}\mathsf{amb}(G, T)\,.\,\langle n\rangle$$

is not well typed: the restriction on the group name G cannot be performed, since G occurs in the type $\mathsf{proc}(\mathsf{amb}(G, T))$ of the process one would like to restrict. Neither is the process

$$\nu G\,.\,m[\,\nu n{:}\mathsf{amb}(G, \mathsf{shh})\,.\,\langle n\rangle\,]$$

Although the type of $m[\,\nu n{:}\mathsf{amb}(G, \mathsf{shh})\,.\,\langle n\rangle\,]$ may be freely chosen so as not to contain G, any such typing may only be derived with respect to an environment where G occurs in the type assumed for m.

3.2.2 Mobility types

As first argued by the authors in [CGG00], since types are supposed to maintain key invariants of programs, and ambient calculi are explicitly designed to describe mobile code, type systems for ambients should naturally be able to express properties related to mobility.

$\mathbf{G}, \mathbf{H} ::=$	$\{G_1, \ldots, G_k\}$	finite sets of group names
$T, S \ ::=$		types of topics, i.e., types of communication
	shh	no communication
	W	communication of messages of type W
$W \ ::=$		message types
	Amb	ambient types
	Cap	capability types
Proc $::=$	$\mathsf{proc}(\frown\mathbf{G},^\circ\mathbf{H}, T)$	processes performing communication of type T, able to drive ambients in and out of ambients of groups $\mathbf{G}$, and to open ambients of groups $\mathbf{H}$
Amb $::=$	$\mathsf{amb}(G,\frown\mathbf{G},^\circ\mathbf{H}, T)$	ambients of group G, having T as topics of conversation, allowed to be driven in and out of ambients of groups $\mathbf{G}$, and wherein ambients of groups $\mathbf{H}$ may be opened
Cap $::=$	$\mathsf{cap}(\frown\mathbf{G},^\circ\mathbf{H}, T)$	capabilities that may be exercised by processes which: perform communication of type T, drive ambients in and out of ambients of groups $\mathbf{G}$, open ambients of groups $\mathbf{H}$.

The triple $\frown\mathbf{G},^\circ\mathbf{H}, T$ is called an *effect* and may be collectively denoted by F.

Figure 4: Mobility Types for Mobile Ambients

In [CGG00] a type system for MA is defined, which tracks ambient opening and ambient movement. In this system we find the actual reason for the introduction of groups: the need to express ambient properties regarding openings and movements with respect to other ambients, while avoiding *dependent types*, i.e., types dependent on values, notoriously delicate to handle.

For example, if through a typing one wants to declare that the ambient m has the property of being able to enter the ambient n, one should write $m : \mathit{CanEnter}(n)$, where the type depends on the value n. With groups one can avoid the need for such type expressions. They can act as intermediaries between types and values. The above property can be reformulated as the more general statement that the ambient m may enter the ambients of group G and that n is an ambient of G. One ends up with the the two typings $m : \mathit{CanEnter}\,(G),\ n : \mathsf{amb}(G)$.

The types introduced in [CGG00] and the properties they denote, are summarised in Figure 4. The typing rules naturally express the informal meanings already discussed. Typical examples of the rules are

$$(\text{T-IN} - \text{OUT})\ \frac{\Gamma \vdash n : \mathsf{amb}(G, F) \quad G \in \mathbf{G}}{\Gamma \vdash \mathsf{in/out}\ n : \mathsf{cap}(\frown\mathbf{G},^\circ\mathbf{H}, T)} \qquad (\text{T-OPEN})\ \frac{\Gamma \vdash n : \mathsf{amb}(G,\frown\mathbf{G},^\circ\mathbf{H}, T) \quad G \in \mathbf{H}}{\Gamma \vdash \mathsf{open}\ n : \mathsf{cap}(\frown\mathbf{G},^\circ\mathbf{H}, T)}$$

where in the assumptions we omitted the well-formedness requirement for the effect $\frown\mathbf{G},^\circ\mathbf{H}, T$.

The premise $G \in \mathbf{H}$ in the OPEN rule requires that if an ambient is openable then all ambients of its group may be opened inside it. This seemingly odd condition stems from the fact that, if an ambient n may be opened within m, processes internal to n must have the same type – and thus the same capabilities – as those in m. Among such capabilities, there is of course the one of opening n itself (i.e., all ambients of its

group)!

This has the pleasant side-effect that a type $\mathsf{amb}(G,{}^{\frown}\mathbf{G},{}^{\circ}\mathbf{H},T)$ specifies whether the ambient is openable or not, depending on whether G is or is not contained in $\mathbf{H}$. The form of the rules, however, also has the consequence that, recursively, an ambient must have the same capabilities of any ambient where it can stay, at any level of nesting, within a chain of openable ambients.

Numerous other type disciplines for MA, and its variations, have been proposed. The original MA calculus, however, equipped with the type system we have just sketched, and supplemented by a modal spatial and temporal logic [CG00] for expressing program properties, has become one of the well-established paradigms for mobile computing, and a natural basis for any study in this area.

3.3 Safe Ambients

The calculus of Safe Ambients (SA) [LS00] represented a first important variation of MA, with the introduction of *co-actions* (or *co-capabilities*). They were added with the explicit purpose of allowing the definition of a type system capable of controlling mobility and eliminating the possibility of the so-called *grave interferences*.

The type system is based on the same general scheme as that for MA. However with regards to mobility, since it does not use groups, it may only express the binary property of being immobile or not: ${}^{\mathsf{I}}\mathsf{amb}(T)$ is the type of immobile ambients of topics T, while ${}^{\mathsf{B}}\mathsf{amb}(T)$ is the type of unconstrained ambients. This is in contrast with the more expressive MA group based type system where immobility only is a particular case among many possible mobility types: an ambient is immobile if the ${}^{\frown}$ component of its type is empty.

Subject reduction guarantees that if an ambient m is typed as immobile, no process $m[P]$ will ever come out of the enclosing ambient or enter a sibling ambient, though the enclosing ambient may be free to move (immobility is relative to the surrounding ambient). Of course, an unconstrained ambient does not necessarily move; it just can not be guaranteed to be immobile.

The distinguishing feature of the SA type system, which differentiates it from most other ambient systems, is however the expression of the *single-threadness* property.

A *single-threaded* (ST) ambient is one where "at any moment at most *one* process may have the control thread and may therefore use a capability" [LS00]; a ST process is the corresponding notion for processes, so that a term of the form $m[P]$, with m ST, is well-typed if and only if P is ST. More precisely, an ST ambient or process is one in which at any reduction step there is, at the top level, at most one unguarded action or co-action, i.e., an action or co-action in initial prefix position, ready for execution.

This does not forbid that other actions in internal sub-ambients to be concurrently performed; for example, if an open is executed, at least the matching $\overline{\mathsf{open}}$[1] must be consumed!. Moreover, as stated above, only the execution of a capability represents a step of the thread of computation. An input or output operation, although a computational activity, is not in itself considered a step of an individual thread, since it is not a "control" action affecting an ambient's movement or dissolution. It may however find itself inserted in a thread, between or after actions, as in $\mathsf{out}\, m.(x).\mathsf{in}\, n.P$ or in $\mathsf{in}\, n.\langle M\rangle$, in which case, as we will see, it may transfer the thread.

In an ST ambient, therefore, communication between processes (not holding the control) is allowed concurrently with the control thread, as for example in the term

$$[\,\mathsf{out}\, m.\mathsf{in}\, n \mid \langle k\rangle \mid (x).x[P]\,]$$

where the thread is defined by the process that drives the ambient out of m into n. Communication between the other two processes is performed concurrently (simulated by interleaving), but it is not considered to give rise to a distinct thread.

[1] $\overline{\mathsf{open}}$, $\overline{\mathsf{in}}$, $\overline{\mathsf{out}}$ are usually pronounced as co-open, co-in, co-out, respectively.

Amb ::=	$^{\mathsf{B}}\mathsf{amb}(T)$	mobile and openable ambients of topics T, they however can not be opened within ambients of type $^{\mathsf{I}}\mathsf{amb}(T)$
	$^{\mathsf{I}}\mathsf{amb}(T)$	ambients of topics T, which can not be moved nor opened, and wherein no ambient may be opened
	$^{\mathsf{ST}}\mathsf{amb}^{\uparrow}(Tt)$	ST ambients whose opening exposes a process taking the thread
	$^{\mathsf{ST}}\mathsf{amb}^{\dagger}(Tt)$	ST ambients whose opening exposes a process not taking the thread

where *Tt*, the communication type of ST ambients, is:

Tt ::=	shh	no communication
	$^{\uparrow}W$	communication of type W; messages carry the thread
	$^{\dagger}W$	communication of type W; messages do not carry the thread

Process and capability types have corresponding forms:

$$\begin{array}{rcl} \mathit{Proc} & ::= & {}^{\mathsf{B}}\mathsf{proc}(T) \mid {}^{\mathsf{I}}\mathsf{proc}(T) \mid {}^{\mathsf{ST}}\mathsf{proc}^{J}(T) \\ \mathit{Cap} & ::= & {}^{\mathsf{B}}\mathsf{cap}(T) \mid {}^{\mathsf{I}}\mathsf{cap}(T) \mid {}^{\mathsf{ST}}\mathsf{cap}^{J}(T) \end{array} \qquad \text{with } J ::= \uparrow \mid \dagger$$

where $^{\mathsf{ST}}\mathsf{proc}^{J}(T)$ is the type of ST processes holding or not holding the thread, according to whether $J =\uparrow$ or $J = \dagger$; $^{\mathsf{ST}}\mathsf{cap}^{J}(T)$ is the type of an action that, when performed, leaves its continuation with thread right J.

Figure 5: Types for Safe Ambients

Within a ST ambient the thread may pass from one sequential process to another through the opening of an ambient, whereby a newly exposed process catches the control, as in the following example adapted from [LS00]:

$$n[\,\mathsf{open}\; m.\langle M\rangle \mid m[\overline{\mathsf{open}}\; m.\mathsf{in}\; k.\mathsf{in}\; h]\,] \mid k[\,h[\ldots]\,] \;\rightarrow\; n[\,\langle M\rangle \mid \mathsf{in}\; k.\mathsf{in}\; h\,] \mid k[\,h[\ldots]\,]$$

In the ambient n the single thread is initially held by the sequential process performing the open. After this action the process only has to perform an output, and therefore relinquishes the thread to the process $\mathsf{in}\; k.\mathsf{in}\; h$, which will drive n into k and h.

The thread may also pass from one process to another through communication. An input-output operation can not give rise to a thread but it may transmit one, as in the following example again taken from [LS00]:

$$n[\,\mathsf{in}\; h.\langle M\rangle \mid (x).x.Q\,] \mid h[\,\overline{\mathsf{in}}\; h.R\,] \;\rightarrow\; \rightarrow\; h[\,R \mid n[M.Q]\,]$$

The thread of n is initially held by $\mathsf{in}\; h.\langle M\rangle$, which after executing the in passes the capability M to the process $(x).x.Q$. The latter, by turning itself into $M.Q$, immediately activates M itself and thus takes hold of the thread.

This discussion motivates the structure of types for SA, which is given in Figure 5.

During a computation, the condition for an ST process to be the owner of the thread is only loosely defined. At a given step an ST process holds the thread if it has one unguarded capability (i.e., it is ready to perform a control instruction), or if it is an output process passing the thread, or if it had the thread in

the previous step (i.e., it is the derivative of a process holding the thread in the previous reduction step) and no other parallel ST process now holds the thread. In the other cases, it may be assumed as having or not having the thread, with the constraints that at most one among all the parallel components of a ST process may own the thread, and that the process P consisting of the whole content of a ST ambient $m[P]$ must own the thread.

Thus, for example, in a process $P = (x).0 \,|\, (y).(z).0$ either the component $(x).0$ or the component $(y).(z).0$ or none of them may be assumed to be the thread owner. However, if P in turn is in parallel with a process definitely owning the thread, say $\mathsf{out}\, m$, this forces the third alternative ("none of them"). If on the other hand P is the whole content of a ST ambient m, as in the term $m[P]$, the choice must be on one of the first two possible assumptions.

The typing rules, as usual, formally reflect the above intentions. The fundamental rule that links the assumed ambient's behaviour to its content is the same as in MA:

$$(\text{T-BI} - \text{AMB})\ \frac{\Gamma \vdash m :^{bi} \mathsf{amb}(T) \quad \Gamma \vdash P :^{bi} \mathsf{proc}(T)}{\Gamma \vdash m[P] : \mathit{Proc}}\ (\mathit{bi} = \mathsf{B}, \mathsf{I})$$

$$(\text{T-ST} - \text{AMB})\ \frac{\Gamma \vdash m : {}^{\mathsf{ST}}\mathsf{amb}^{J}(\mathit{Tt}) \quad \Gamma \vdash P : {}^{\mathsf{ST}}\mathsf{proc} \uparrow (\mathit{Tt})}{\Gamma \vdash m[P] : \mathit{Proc}}$$

The absence of explicit subtyping with a subsumption rule means that each kind of process may only stay in the corresponding kind of ambient; for example a process typed as ST or immobile (that is, good for staying in immobile ambients) can not go into a basic ambient. There is however in the system an implicit subtyping: the simultaneous applicability of different rules and the usual polymorphism of $\mathsf{0}$ allow the assignment of different types to a term. In particular, a process typed as ST may, under certain conditions (no free recursion variables, process not holding the thread, unthreaded or silent communication), also be typed as basic or immobile.

Also observe that in the rule ST-AMB the process P, consisting of the whole content of the ST ambient m, and in particular with no other concurrent processes in m, must have the (single) thread. This does not contradict the fact that m may be an ambient not catching the thread on opening. When m is opened, P – which has the thread – must execute a $\overline{\mathsf{open}}$, but its continuation, exposed to the surrounding ambient, need not hold the thread; for example, it may create an ambient $n[Q]$, or perform unthreaded communication, or just die. In such cases the control is kept by the process originally in the enclosing ambient.

Singleness is naturally expressed by the rules for parallel composition, where at most one of the components may hold the thread:

$$(\text{T-ST} - \text{PAR}1)\ \frac{\Gamma \vdash P_1 : {}^{\mathsf{ST}}\mathsf{proc} \uparrow (\mathit{Tt}) \quad \Gamma \vdash P_2 : {}^{\mathsf{ST}}\mathsf{proc} \dagger (\mathit{Tt})}{\Gamma \vdash P_1 \,|\, P_2 : {}^{\mathsf{ST}}\mathsf{proc} \uparrow (\mathit{Tt})}$$

$$(\text{T-ST} - \text{PAR}2)\ \frac{\Gamma \vdash P_1 : {}^{\mathsf{ST}}\mathsf{proc} \dagger (\mathit{Tt}) \quad \Gamma \vdash P_2 : {}^{\mathsf{ST}}\mathsf{proc} \uparrow (\mathit{Tt})}{\Gamma \vdash P_1 \,|\, P_2 : {}^{\mathsf{ST}}\mathsf{proc} \uparrow (\mathit{Tt})}$$

$$(\text{T-ST} - \text{PAR}3)\ \frac{\Gamma \vdash P_1 : {}^{\mathsf{ST}}\mathsf{proc} \dagger (\mathit{Tt}) \quad \Gamma \vdash P_2 : {}^{\mathsf{ST}}\mathsf{proc} \dagger (\mathit{Tt})}{\Gamma \vdash P_1 \,|\, P_2 : {}^{\mathsf{ST}}\mathsf{proc} \dagger (\mathit{Tt})}$$

An output process $\langle M \rangle$, if it is a process with thread-carrying communication, must hold the thread, i.e., it must be the continuation of a control process, now ready to pass the capability M to some other process which is going to immediately use it and thus to catch in turn the thread. With unthreaded communication, the output may or may not have the thread, depending on the process of which it is the continuation.

To type an input process $(x).P$, on the other hand, the premise requires that its continuation P does or does not hold the thread, depending on whether or not the message carries it. An input process waiting for a thread-carrying message is of course not (yet) holding the thread. The rules therefore are:

$$(\text{T-ST}-\text{OUTP1})\ \frac{\Gamma \vdash M : W}{\Gamma \vdash \langle M \rangle : {}^{\mathsf{ST}}\mathsf{proc}^{\uparrow}({}^{\uparrow}W)} \qquad (\text{T-ST}-\text{OUTP2})\ \frac{\Gamma \vdash M : W}{\Gamma \vdash \langle M \rangle : {}^{\mathsf{ST}}\mathsf{proc}^{J}({}^{\dagger}W)}$$

$$(\text{T-ST}-\text{INP})\ \frac{\Gamma, x{:}W \vdash P : {}^{\mathsf{ST}}\mathsf{proc}^{J}({}^{J}W)}{\Gamma \vdash (x{:}W).P : {}^{\mathsf{ST}}\mathsf{proc}^{K}({}^{J}W)}$$

The other possible thread-passing event is described by one of the open rules: when a ST ambient is opened within another ST ambient, the thread is kept or relinquished by the old process, depending on whether or not the newly exposed process catches it:

$$\text{ST-OPEN1}\ \frac{\Gamma \vdash m : {}^{\mathsf{ST}}\mathsf{amb}^{J}(Tt)}{\Gamma \vdash \mathsf{open}\, m : {}^{\mathsf{ST}}\mathsf{cap}^{J^{-1}}(Tt)} \quad \text{where } \uparrow^{-1} \text{ is } \dagger \text{ and conversely.}$$

The reason why this rule, if expressed in words as above, sounds tautological, is that it actually *defines* the meaning of the qualifier J of types ${}^{\mathsf{ST}}\mathsf{amb}^{J}(Tt)$.

The rules for the other constructs are generally more standard, though many of them present different cases corresponding to the different kinds of ambients (mobile, immobile, single-threaded). For example, the in and out primitives can never be inserted in a process staying in an immobile ambient; if they are inserted in a process in a ST ambient, by definition they hold the thread:

$$\text{BI-INOUT}\ \frac{\Gamma \vdash m : \mathit{Amb}}{\Gamma \vdash \mathsf{in}/\mathsf{out}\, m : {}^{\mathsf{B}}\mathsf{cap}(T)} \qquad \text{ST-INOUT}\ \frac{\Gamma \vdash m : \mathit{Amb}}{\Gamma \vdash \mathsf{in}/\mathsf{out}\, m : {}^{\mathsf{ST}}\mathsf{cap}^{\uparrow}(Tt)}$$

The corresponding co-actions, on the other hand, being exercised not in the driven ambients but in those where they are moved to or from, may also stay in immobile ambients:

$$\text{BI-COINOUT}\ \frac{\Gamma \vdash m :^{bi} \mathsf{amb}(T)}{\Gamma \vdash \overline{\mathsf{in}}/\overline{\mathsf{out}}\, m :^{bi} \mathsf{cap}(S)} \qquad \text{ST-COINOUT}\ \frac{\Gamma \vdash m : {}^{\mathsf{ST}}\mathsf{amb}^{J}(Tt)}{\Gamma \vdash \overline{\mathsf{in}}/\overline{\mathsf{out}}\, m : {}^{\mathsf{ST}}\mathsf{cap}^{\uparrow}(Tt')}$$

In conclusion, the type system for SA is rather complex, as it expresses and checks properties not easily found in other systems for ambients. Immobile and ST types allow the definition of algebraic laws for a behavioural equivalence consisting of a barbed congruence. This is only possible because of the synchronous character of the movement interactions, which is obtained with the introduction of co-actions, and is a distinguishing feature of SA.

3.4 Boxed Ambients

Boxed Ambients (BA) [BCC01] are another ambient calculus derived from MA, where the open primitive is dropped, and its absence is compensated for by the introduction of communication across one ambient boundary, between parent and children. Beside the usual communication primitives of MA, upward and downward input-output constructs are provided; primitives for downward communication require as argument the name of the (child) ambient addressed, while those for the upward direction do not.

Inter-ambient communication is achieved by synchronising a local operation with an across-the-boundary operation. For example, a message may be passed from child to parent, that is from a process in an ambient

m to a process in the ambient enclosing m, by synchronising a local output in m with a downward input in the parent, or by an upward output in m with a local input in the parent. Reverse communication is achieved in an analogous fashion. Also local communication is synchronous, in the sense that outputs may have non-null continuations.

This extended communication mechanism leads to a significant change to the type system. There are now two possible kinds of topics of conversation for both ambients and processes, the local topics, and the parent's topics; moreover they do not have to be the same. In this way, the parent may exchange values of its local type with its children, which are required to all have the same upward topics, but in addition the parent may exchange values of different types with different children, using its downward primitives.

For instance, using extra atomic types just for the sake of the example, suppose b is an ambient whose topics is bool, which encloses ambients r of topics real, s of topics string and t of topics toy, as in $b[P \mid n[Q] \mid s[S] \mid t[R]]$. The four different possibilities of communication may be exemplified as follows:

- $P = (x).P_1,\ \ Q = \langle \mathsf{true} \rangle^{\uparrow}.Q_1,\ S = \langle \mathsf{false} \rangle^{\uparrow}.S_1, \ldots$
  processes Q, S, R, located in the children ambients, pass booleans to P, located in the parent;
- $P = \langle \mathsf{true} \rangle.P_1,\ \ Q = (x)^{\uparrow}.Q_1,\ S = (y)^{\uparrow}.S_1, \ldots$
  parent passes booleans to children;
- $P = (x)^r.(y)^s \mid (z)^t.P_1,\ \ Q = \langle 3.14 \rangle,\ S = \langle \text{“hi Mom!”} \rangle,\ R = \langle \mathsf{ball} \rangle$
  children pass respectively reals, strings and toys to parent;
- $P = \langle 2.71 \rangle^r.\langle \text{“hi Sally!”} \rangle^s \mid \langle \mathsf{ball} \rangle^t,\ \ Q = (x).Q_1,\ S = (y).S_1,\ R = (z).R_1$
  parent passes respectively reals, strings and toys to children.

Ambient and process types thus have two components. On the other hand, owing to the absence of open, capability types have only one component. However this component has a completely different meaning then that used in the other forms of ambients, MA and SA. In those systems $\mathsf{cap}(T)$ is the type of the open capabilities which open ambients of topics T; as we saw, the type T must be the same as that of the ambient where the capability is exercised.

Here there is no opening, hence no "openable topics" as a component of the capability type; instead, communication with the parent requires that an ambient m is only allowed to stay within – and therefore to be driven into – those ambients whose (local) topics is the same as m's upward communication type. While in MA and SA the type $\mathsf{cap}(T)$ records the topics T of the ambients m occurring in $\mathsf{open}\ m$ constructs, in BA it must record the topics T of the ambients into which the in and out operations may drive the ambients in which they are exercised.

The types required for Safe Ambients are given in Figure 6, and we hope that there structure has been sufficiently motivated by the above discussion. Note that ambient and process types do not need to record children's topics, as this information is available locally to each child.

The intended use of capability types, also described above, are reflected in the introduction rules for $\mathsf{cap}(T)$. If within m the local topics of conversation is T, while within m's parent the topics is S, then the action $\mathsf{in}\ m$, which turns its own ambient into a child of m, must be performed by a process upward-talking T. Similarly $\mathsf{out}\ m$, which turns its own ambient into a sibling of m, must be performed by a process upward-talking S. Moreover, silent communication, i.e., absence of input and output, is compatible with any type of communication. So a silent process may safely participate in a conversation on whatever topics. The rules therefore are:

$$\frac{\Gamma \vdash m : \mathsf{amb}(T,S)}{\Gamma \vdash \mathsf{in}\ m : \mathsf{cap}(T)} \qquad \frac{\Gamma \vdash m : \mathsf{amb}(T,S)}{\Gamma \vdash \mathsf{out}\ m : \mathsf{cap}(S)} \qquad \frac{\Gamma \vdash m : \mathsf{amb}(T,S)}{\Gamma \vdash \mathsf{in/out}\ m : \mathsf{cap}(\mathsf{shh})}$$

T, S ::=		types of topics, i.e., types of communication
	shh	no communication
	W	communication of messages of type W
W ::=		message types
	Amb	ambient types
	Cap	capability types
Proc ::=	$\mathsf{proc}(T, S)$	processes performing local communication of type T and upward communication of type S, able to drive ambients into ambients of topics S
Amb ::=	$\mathsf{amb}(T, S)$	ambients having T as local topics of conversation, allowed to stay within ambients of topics S
Cap ::=	$\mathsf{cap}(S)$	capabilities that can drive ambients into ambients of topics S

Figure 6: Types for Boxed Ambients

As already remarked, superposition between rules – in this case between the third rule and the first two – is a form of implicit subtyping. In all ambient calculi, therefore, $\mathsf{shh} \leq T$ is a natural form of subtyping, which should be implicitly or explicitly present in the type system. In BA the explicit formulation is chosen and the rules are rewritten as:

$$\text{IN}\ \frac{\Gamma \vdash m : \mathsf{amb}(T, S) \quad T' \leq T}{\Gamma \vdash \mathsf{in}\ m : \mathsf{cap}(T')} \qquad \text{OUT}\ \frac{\Gamma \vdash m : \mathsf{amb}(T, S) \quad S' \leq S}{\Gamma \vdash \mathsf{out}\ m : \mathsf{cap}(S')}$$

The rules for prefix and ambient are easily obtained from the previous informal definitions:

$$\text{PREF}\ \frac{\Gamma \vdash M : \mathsf{cap}(S) \quad \Gamma \vdash P : \mathsf{proc}(T, S)}{\Gamma \vdash M.P : \mathsf{proc}(T, S)} \qquad \text{AMB}\ \frac{\Gamma \vdash m : \mathsf{amb}(T, S) \quad \Gamma \vdash P : \mathsf{proc}(T, S)}{\Gamma \vdash m[P] : \mathsf{proc}(S, S')}$$

where the component S of the capability type is the process' upward topic. The fundamental rule AMB is only modified with respect to MA in order to take into account upward communication. From the typing of m in the premise, S is the local topics of the m-enclosing ambient, which by definition is the ambient enclosing the process $m[P]$; hence S is the type of local conversation $m[P]$ may be engaged in.

The other rules for processes are standard, but to be able to exploit the relation $\mathsf{shh} \leq T$ one has to extend subtyping to process types and introduce for them a subsumption rule:

$$\text{PROC-SUBTYPING:}\ \mathsf{proc}(\mathsf{shh}, S) \leq \mathsf{proc}(T, S) \qquad \text{PROC-SUBSUM:}\ \frac{\Gamma \vdash P : \mathit{Proc} \quad \mathit{Proc} \leq \mathit{Proc}'}{\Gamma \vdash P : \mathit{Proc}'}$$

Observe that subtyping does not apply to upward communication: an upward-silent process is not a process that may safely drive its surrounding ambient into an arbitrary ambient, since it may take with it a parallel process of an incompatible type. The rules for the six forms of input-output are as expected from the definition.

In BA, owing to the two-level input-output, mobility is strictly connected with, and constrained by, communication. Unlike the basic MA and SA type systems, the type system we have briefly sketched here for BA already addresses mobility. The subtyping $\mathsf{shh} \leq T$ is further exploited in a refinement of the system called *moded typing*, where the silent and non-silent phases of process execution are expressed at a finer grain, and typed mobility is enhanced.

A significant extension of the type system is presented in [MS02], with the addition of groups, following the paradigm of [CGG00]. They allow, as in [CGG00], separation of concerns about communication and mobility, which become two aspects "orthogonal to each other, and interacting only in well determined places" [MS02]. Subtyping is explicitly introduced both on communication and mobility, with suitable subsumption rules, and is for the first time extensively used in a type system for ambients, to achieve greater expressivity and flexibility.

There is an interesting difference between the use of mobility types for BA and those for MA, as presented in [CGG00]. In the latter the types record which ambients an ambient may *cross*, i.e., which destination an ambient may be driven into by an in action, or of which source an ambient may be driven out by an out action. The mobility types for BA, on the other hand, also in the case of out keeps track of which destination the ambient is driven into. There is considerable scope for using these types to record further "mobility information" and the topic is under active research. See for example [MS02].

4 Information Control

4.1 Introduction

Here the problem is to ensure secure information flow within distributed systems. The general scenario is that data, resources, information, in general knowledge, can have varying degrees of sensitivity, and mechanisms should be in place to ensure that this knowledge is only gained by appropriate principals.

This has been studied extensively in the literature. For example [BL75] laid down some general principles for ensuring secure information flow. These were later developed into a *lattice model of information flow* in [Den77] which could be used to express *information flow policies*. Static analysis techniques were then developed for ensuring that programs, written in a simple programming language, did not violate these policies.

One criticism of this and related work, voiced in [VSI96], is that there was no formal treatment of the soundness of the static analysis. That is, no proof was given that the static analysis ensured that all information flows in a program are in accord with the underlying information flow policy. However soundness can be formally stated using the concept of *non-interference*, [GM82]. This is a very general idea which can be applied in many different scenarios, although as we will see, there are variations in its application.

Although we are primarily interested in distributed systems it is perhaps best to first see a scenario in which these ideas are have been formally worked out successfully. This is the topic of the next three subsections. There then follows a discussion of the extent to which they have been extended to distributed systems.

4.2 Information flow in a simple sequential language

Let us consider the sequential language **While** described in Figure 7. Here x ranges over a set of variables, e over some language of arithmetic expressions, and for simplicity we also use arithmetic expressions as booleans, with 0 playing the role of $\mathbf{true}$ and all other integers $\mathbf{false}$.

Here the sensitivity of data is represented by the fact that certain variables are deemed to hold *high-level data* and others *low-level data*. High-level principals have access to all variables while low-level principals

$$C ::= \quad \mathsf{skip} \mid x := e \mid C; C \mid \\ \mathsf{if}\ e\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ e\ \mathsf{do}\ C$$

Figure 7: A Simple Imperative Language **While**

only have access to the low-level subset, and intuitively these low-level principals should never become aware of changes in the high-level data. In order to explain how this might or might not happen it is usual to annotate the variables with the security level of its data. Thus for example in the simple program

$$x_{\mathrm{L}} ::= y_{\mathrm{H}} + 1$$

the value of a low-level variable x is directly affected by that of the high-level variable, y. This is deemed to be an (unacceptable) direct flow of information from high to low, as a low-level principal, having access to the low-level variable x, can receive some high-level knowledge, some information about the value of the high-level variable y. *Indirect* information flow is more interesting. Consider the following three programs:

$$\mathsf{if}\ y_{\mathrm{H}} = 0\ \mathsf{then}\ x_{\mathrm{L}} ::= 1\ \mathsf{else}\ x_{\mathrm{L}} ::= 2$$
$$\mathsf{if}\ y_{\mathrm{H}} = 0\ \mathsf{then}\ x_{\mathrm{L}} ::= x_{\mathrm{L}} + 1\ \mathsf{else}\ x_{\mathrm{L}} ::= x_{\mathrm{L}} + 2$$
$$\mathsf{if}\ y_{\mathrm{H}} = y'_{\mathrm{H}}\ \mathsf{then}\ x_{\mathrm{L}} ::= 1\ \mathsf{else}\ \mathsf{skip}$$

In the first a low-level principal can detect whether or not the value of the high-level variable y is equal to 0; it is sufficient to see if the final value of the low-level variable is 1 or 2. In the second the same high-level information can be obtained by comparing the initial value of x_{L} with the final value. In the third example some more indirect high-level information can be ascertained, namely whether two high-level variables contain the same value; again the low-level principal must have access to the initial and final values of the low-level variable x_{L}.

In each of these three cases the indirect high-to-low information flow arises because the flow of control in the program depends on the values of high-level variables, and the flow is such that different choices affects the values of low-level variables.

A static analysis of programs can easily be designed to catch such undesirable information flows, but as was pointed out in [VSI96] such an analysis can also be elegantly captured using a type system.

4.3 A Type System

Here we outline the bones of a type system for ensuring that there is no direct or indirect information flow in a sequential program written in the language from Figure 7. Although it is more usual to define these using programs in which occurrences of variables are tagged with their security levels, as in the examples of the previous sub-section, for consistency with the mobility control type systems of the previous section here we work with respect to *security type environments*; a security type environment Γ is simply a mapping from

Commands:

$$\frac{\Gamma \vdash e : \sigma, \;\; x : \sigma}{\Gamma \vdash x ::= e : \sigma}\ (\textsc{t-assign}) \qquad \frac{\Gamma \vdash C_1 : \sigma, \;\; C_2 : \sigma, \;\; b : \sigma}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ : \sigma}\ (\textsc{t-choice})$$

$$\frac{\Gamma \vdash C_1 : \sigma, \;\; C_2 : \sigma}{\Gamma \vdash C_1 : C_2 : \sigma}\ (\textsc{t-seq}) \qquad \frac{\Gamma \vdash C : \sigma, \quad \Gamma \vdash b : \sigma}{\Gamma \vdash \mathsf{while}\ b\ \mathsf{do}\ C\ : \sigma}\ (\textsc{t-while})$$

Expressions:

$$\frac{\Gamma(x) = \textsc{h}, \;\; x \in e}{\Gamma \vdash e : \textsc{h}}\ (\textsc{t-high}) \qquad \frac{x \in e \;\; \text{implies} \;\; \Gamma(x) = \textsc{l}}{\Gamma \vdash e : \textsc{l}}\ (\textsc{t-low})$$

Subtyping:

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma'}\ \sigma <: \sigma'\ (\textsc{sub-exp}) \qquad \frac{\Gamma \vdash C : \sigma}{\Gamma \vdash C : \sigma'}\ \sigma' <: \sigma\ (\textsc{sub-com})$$

Figure 8: A Type System

variables to their security levels. In more complicated languages it could also incorporate other information such as the type of values which can be stored or indeed resource access information.

Underlying the static analysis for information flow are two principles.

- To eliminate *direct* information flow we must ensure that in every assignment

  $$x := e$$

  the security level of x must be at least as high as that of the highest level read from, during the evaluation of e.

This requirement requires us to calculate the highest security level read from when evaluating an expression, which in our type system is checked via their judgement

$$\Gamma \vdash e : \sigma$$

In Figure 8 we give two (informal) rules for this judgement, (T-HIGH) and (T-LOW), and a subtyping rule (SUB-EXP); although simpler rules can be used for our language these exemplify more readily the general principle.

- In each possible choice point, which in our simple language **While** are

  $$\mathsf{if}\ b\ \mathsf{then}\ *\ \mathsf{else}\ *$$
  $$\mathsf{while}\ b\ \mathsf{do}\ *$$

the program fragments to be inserted into occurrences of $(*)$ can only write to variables whose security level is at least that read from, during the evaluation of b.

In our type system this is ensured by the judgements for commands:

$$\Gamma \vdash C : \sigma$$

which checks that when executing the command C variables at level *at least* σ are written to. The main rules are (T-CHOICE) and (T-WHILE), together with the subtyping rule (SUB-COM).

Thus, assuming that Γ is in accord with the tagging of variables, we can deduce

$$\Gamma \vdash \mathsf{if}\ x_{\mathrm{L}} = 0\ \mathsf{then}\ x_{\mathrm{L}} ::= 1\ \mathsf{else}\ y_{\mathrm{H}} ::= 2 \quad : \mathrm{L}$$

but we can not deduce

$$\mathsf{if}\ y_{\mathrm{H}} = 0\ \mathsf{then}\ x_{\mathrm{L}} ::= 1\ \mathsf{else}\ x_{\mathrm{L}} ::= 2 \quad : \sigma$$

for any σ. The basic reason is that we can only deduce

$$\Gamma \vdash y_{\mathrm{H}} = 0 \quad : \sigma$$

when σ is H and

$$\Gamma \vdash x_{\mathrm{L}} ::= 1 \quad : \sigma$$

when σ is L, and so the rule (T-CHOICE) can not be applied.

4.4 Non-interference: A first attempt

One can argue that the type-checking rules in Figure 8 incorporate the principles outlined in [Den77] for eliminating high-to-low information flow, but how can we ensure that all such flows are eliminated from well-typed programs? Following [VSI96] we can express the inability of low-level principals to use well-typed programs to discover any high-level information as a form of *non-interference*, [GM82].

Let us assume an operational semantics for our language, in the form of one step reductions

$$(C, s) \rightarrow (C', s')$$

where s represents a *store*, a mapping of variables to values. If skip represents a terminated program then

$$(C, s) \rightarrow^* (\mathsf{skip}, s')$$

means that when C is executed in the initial store s it will terminate, with the final store s'.

A low-level principal is able to examine the contents of low-level variables in the store, both before a program is run and after it has terminated. Let us say that

$$\Gamma \vdash s \approx_{\mathrm{L}} s'$$

whenever $s(x) = s'(x)$ for every x such that $\Gamma(x) = \mathrm{L}$; intuitively a low-level principal can not tell the difference between any two stores such that $s \approx_{\mathrm{L}} s'$. But running a program should also not provide any information about high-level variables. One way of formalising this is to say that the outcome of running a program is independent of the contents of high-level variables.

Definition 4.1 [Non-interference] A program C is interference-free, relative to Γ, if for all stores such that $\Gamma \vdash s_1 \approx_{\mathrm{L}} s_2$

$$(C, s_1) \rightarrow^* (\mathsf{skip}, s_1') \ \text{ and } \ (C, s_2) \rightarrow^* (\mathsf{skip}, s_2') \ \text{ implies } \ \Gamma \vdash s_1' \approx_{\mathrm{L}} s_2'.$$

Then we can prove that the type system eliminates all high-to-low information flow by establishing the following result:

Theorem 4.2 *If* $\Gamma \vdash C : \mathrm{L}$ *then* C *is interference-free, relative to* Γ.

4.5 Non-interference Revisited

We have just seen a scenario in which a type system has been used to eliminate high-to-low information flow; a survery of extensions to other program constructs may be found in [SM02]. But here let us re-examine some of the in-built assumptions, which may not hold in a distributed systems.

The definition of non-interference given in Definition 4.1 assumes that the low-level principal can only inspect the store, in fact the low-level variables in the store, before the program starts and on termination; indeed this latter point is essential. As pointed out in [BC02] the program

$$(\mathsf{while}\ y_{\mathrm{H}} = 0\ \mathsf{do\ skip}\ ); x_{\mathrm{L}} ::= 1 \tag{1}$$

is well-typed and therefore interference free; whenever the program terminates the value of the low-level variable x is 1. However clearly the fact that the program terminates contains some high-level information, namely that the value of the high-level variable y is not 0.

In [BC02] the authors go on to argue that in a multi-threading setting programs such as (1) above lead to non-trivial leaks of information and should be ruled out. Formally a stronger notion of non-interference is required. This could either be based on the ability of low-level principles to detect the termination of programs, or, as in [BC02], on their ability to examine the low-level store *during* the computation of the program. A modified type system is then given, an improvement on that in [SV98], for ensuring this stronger version of non-interference. They then go on to argue that scheduling policies in multi-threaded systems can affect information leakage and suggest further type rules for checking these policies.

The semantics of our language assumes that the store is always present, willing to receive updates and return values. However viewed via an implementation into a distributed language, the store is simply another programmable resource, perhaps accessible and modifiable by other users, who may be either high or low level principals. In this scenario a definition of non-interference would also have to take into account the role of this store; for example whether it is available for use.

According to Definition 4.1 the program

$$(x_{\mathrm{L}} ::= 2); (y_{\mathrm{H}} ::= y_{\mathrm{H}} + 1); (x_{\mathrm{L}} ::= 3)$$

is interference-free. But formally this depends on the readiness of the store to accept interactions at the high-level variable y. However the fact that the low-level value of x changes from 2 to 3 contains some high-level information, the presence of a high-level process servicing the high-level variable y.

As a final example consider the extension of our language with a simple non-deterministic construct, a generalisation of the if b then $\ldots$ else $\ldots$ construct. Consider the program:

$$\begin{array}{l} \mathbf{true} \rightarrow x_{\mathrm{L}} ::= 1 \\ \qquad + \\ \mathbf{true} \rightarrow \mathsf{while}\ y_{\mathrm{H}} = 0\ \mathsf{do\ skip}\ \ ;\ x_{\mathrm{L}} ::= 2 \end{array}$$

Here there is a choice between two computation paths and the possible values which end up in the low-level variable x depends on which path is taken. But also the *set of possible values* depends on high-level information; if the high-level variable y contains the value 0 the only possible value for x is 1 while if it is not 0 then both 1 and 2 are possibilities. Thus if low-level principles are allowed to discover the range of possible outcomes this program contains an unwanted information leak. In the literature this is referred to as a *possibilitic* security leak and static analysis methods have been developed for them; see [Man00] for work in this area.

4.6 Distributed Systems

From these examples it is clear that the proper treatment of information flow depends on the language used, and also on the perceived abilities of low-level principles to make observations. In papers such as [FG95, RS97] is is argued that classical concepts of process equivalences, originally developed for process algebras, should be used to clarify the role of observations in the definition of non-interference. Indeed in the work from [BC02] cited above a variation on *bisimulation equivalence* is used to formalise the ability of low-level principals to view the sequence of store changes made by a program execution. In the just cited papers, [FG95, RS97], a variety of formulations of non-interference is suggested, based on well-known process equivalences, such as *trace equivalence*, *failure equivalence*, and *bisimulation equivalence*. However types are not used to ensure non-interference; instead conditions are imposed on the operational semantics of processes, and these conditions can in principle be checked semi-automatically by software verification systems, such as that reported in [FG97]. Moreover these conditions have only been developed for relatively simple process languages based on CCS and CSP, of limited use for the Mikado project.

However there is a growing body of work based around the π-calculus which

- uses process equivalences to express non-interference
- uses type systems to ensure non-interference.

Here we review a small but representative sample.

In [Aba99], the *spi calculus*, an extension of the π-calculus with encryption and decryption primitives, is equipped with a type system which encapsulates informal principles for ensuring secrecy in communication protocols. Both data and communication channels are labelled **public** or **private**, similar to our tags of L and H and the type system enforces requirements such as the inability of public channels to carry private data. The preservation of secrets is expressed in terms of a definition of non-interference which uses *may testing equivalence*. Further work in this direction is reported in [GJ02].

In [Con02] security types are developed for Djoin [FG96], a distributed language similar to the π-calculus; this again implicitly views messages as having security tags associated with them. Here non-interference is expressed with respect to a much stronger equivalence, called *weak barbed congruence*. It is in languages such as Djoin where non-interference has to take into account the presence or absence of high-level processes; we have already seen above the implicit assumption, in treating the sequential language **While**, that the high-level memory is always available for use.

In [HR00, Hen00] the π-calculus itself is endowed with security types. In the former a uniform treatment is attempted in which both resource access control and information control can be defined; information control is again expressed in terms of non-interference relative to *may testing equivalence*. However in the latter reference the types used are further restricted to ensure non-interference relative to *must testing equivalence*.

Finally an extension of the π-calculus is considered in [HY02]. A sophisticated system of types are used, which include *linear*, *affine*, *receptive* and *behavioural* annotations and consequently typechecking is quite complicated. However the payoff is considerable as non-interference is established with respect to a form of *weak barbed congruence*. Moreover non-interference for imperative languages such as our **While** can be recovered from the natural translation of **While** into their extension of the π-calculus.

References

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

[BC02] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281-1:109–130, 2002.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01 Proc. of the 4th Int. Conference on Theoretical Aspects of Computer Science*, number 2215 in Lecture Notes in Computer Science, pages 38–63. Springer-Verlag, 2001.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical report MTR-2997, MITRE Corporation, 1975.

[CG98a] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[CG98b] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[CG99] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *26th Annual Symposium on Principles of Programming Languages*, pages 79–92, 1999.

[CG00] L. Cardelli and A. D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 365–377. ACM Press, 2000.

[CGG00] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *IFIP TCS*, pages 333–347, 2000.

[Con02] Sylvain Conchon. Modular information flow analysis for process calculi. In Iliano Cervesato, editor, *Proceedings of the Foundations of Computer Security Workshop (FCS 2002)*, Copenhagen, Denmark, JULY 2002.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *In the Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston*, pages 184–198, 2000.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *In Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, 1999.

[Den77] D. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, 1977.

[FG95] Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1), 1995.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *23th Annual Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[FG97] Riccardo Focardi and Roberto Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23, 1997.

[GJ02] A.D. Gordon and A.S.A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *CSFW 12*, pages 79–91. IEEE Press, 2002.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and privacy*, 1982.

[Hen00] Matthew Hennessy. The security picalculus and non-interference. Technical report 2000:05, University of Sussex, 2000. Available from `http://www.cogs.susx.ac.uk/`.

[HR00] Matthew Hennessy and James Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In U. Montanari, J. Rolim, and E. Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427, Geneva, Switzerland, 9–15 July 2000. Springer-Verlag. Full version to appear in ACM Transactions on Programming Languages.

[HR02] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[HY02] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *29th Annual Symposium on Principles of Programming Languages*. ACM, January 2002.

[IK00] A. Igarashi and N. Kobayashi. Type reconstruction for linear pi-calculus with i/o subtyping. *Information and computation*, 161:1–44, 2000.

[IK02] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 331–342, 2002.

[Lam74] B.W Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

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

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Man00] Heiko Mantel. Possibilistic Definitions of Security – An Assembly Kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, UK, July 3–5 2000. IEEE Computer Society.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *In the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta*, pages 25–35, 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[Mil99] R. Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[MS02] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings CONCUR 02*, number 1644 in Lecture Notes in Computer Science, 2002.

[NFPV00] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, 1996.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. Extended abstract in LICS '93.

[Pun97] Franz Puntigam. Coordination requirements expressed in types for active objects. In *Proc.* 11^{th} *ECOOP*, volume 1241 of *LNCS*, pages 367–388. Springer-Verlag, 1997.

[RS97] P.Y.A. Ryan and S.A. Schneider. Process algebra and non-interference. In *CSFW 12*. IEEE, 1997.

[RV00] António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488. SPRINGER, 2000.

[Sch00] Fred Scheider. Enforceable security policies. *ACM transactions on information and system security*, 2(4), 2000.

[SM02] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communication*, 2002. To appear.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, January 1998.

[SW01] Davide Sangiorgi and David Walker. *The* pi*-calculus: A Theory of mobile processes*. Cambridge University Press, 2001.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[Wal00] David Walker. A type system for expressive security properties. In *In the Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston*, pages 254–267, 2000.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Proc. of* 16^{th} *FST/TCS*, volume 1180 of *LNCS*, pages 371–386. Springer-Verlag, 1996.