# Resource Access Control in Systems of Mobile Agents

(Extended Abstract)

Matthew Hennessy [1]

*School of Cognitive and Computing Sciences*
*Univ. of Sussex*
*Brighton, UK*
matthewh@cogs.susx.ac.uk

James Riely [2]

*Department of Computer Science*
*North Carolina State Univ.*
*Raleigh, NC, USA*
riely@csc.ncsu.edu

**Abstract**

We describe a typing system for a distributed π-calculus which guarantees that distributed agents cannot access the *resources* of a system without first being granted the *capability* to do so. The language studied allows agents to move between distributed *locations* and to augment their set of capabilities via communication with other agents. The type system is based on the novel notion of a *location type*, which describes the set of resources available to an agent at a location. Resources are themselves equipped with capabilities, and thus an agent may be given permission to send data along a channel at a particular location without being granted permission to read data along the same channel. We also describe a *tagged* version of the language, where the capabilities of agents are made explicit in the syntax. Using this tagged language we define *access violations* as runtime errors and prove that well-typed programs are incapable of such errors.

[1] Research funded by CONFER II and EPSRC project GR/K60701.
[2] Research funded by NSF grant EIA-9805604 and EPSRC project GR/K60701.

# 1 Introduction

*Mobile computation*, where independent agents roam widely distributed networks in search of resources and information, is fast becoming a reality. A number of programming languages, APIs and protocols have recently emerged which seek to provide high-level support for mobile agents. These include Java [24], Odyssey [11], Aglets [15], Voyager [20] and the latest revisions of the Internet protocol [21,1]. In addition to these commercial efforts, many prototype languages have been developed and implemented within the programming language research community — examples include Linda [5,6], Facile [12], Obliq [4], Infospheres [8], and the join calculus [9]. In this paper we address the issue of resource access control for such languages.

Central to the paradigm of mobile computation are the notions of agent, resource and location. *Agents* are effective entities that perform computation and interact with other agents. Interaction is achieved using shared *resources* such as memory cells, M-structures, objects (with shared methods and state) or communication channels. The use of the term "mobile" implies that agents are bound to particular *locations* and that this binding may vary over time, *i.e.* agents can *move*. Resources, on the other hand, are often fixed to a single location, although proxies and mirrors may be set up in order to distribute their contents.

In *open* distributed systems, such as the internet, it is unwise to assume that all agents are benign, and thus a certain amount of effort must be spent to ensure that vital resources are protected from unauthorized access. This can be accomplished by using a system of *capabilities* and by predicating resource access on possession of the appropriate capability. It is unreasonable, however, to expect that *every* use of *every* resource in a network be thus verified dynamically; such a requirement surely would degrade performance unacceptably. Thus it is attractive to develop static analyses, or *typing systems* that guarantee controlled access to network resources.

We present a typed language for mobile agents which allows fine control over the use of resources in a network. We also define a *tagged* version of the language in which agents explicitly carry the sets of capabilities which they have acquired. Using this tagged language, we capture resource access violations as *runtime errors* and show that well-typed terms are incapable of such errors.

The language studied in this paper, called D$\pi$, is a distributed variant of the $\pi$-calculus [19], and thus the *resources* of interest are *channels* which support binary communication between agents. We take agents to be located *threads*, which are simply terms of the ordinary polyadic $\pi$-calculus [18], extended with primitives for *movement* between locations and for the creation of *new* locations.

The type system is based on the notion of *location types* of the form:

$$\mathsf{loc}\{\kappa_1, ..., \kappa_n\}$$

where each $\kappa_i$ is a *location capability*. These may take the form of primitive capabilities, typical examples being go, the ability to move to the location, and newc, the ability to create a new local channel; or a capability associated with a particular

channel, $a$:A. Here A represents a set of *channel capabilities* which are of the form

- $\mathsf{r}\langle \mathrm{T} \rangle$, the capability to receive values $V$ from a channel and then to use each $V$ with *at most* the capabilities specified by the type T, or
- $\mathsf{w}\langle \mathrm{T} \rangle$, the capability to send values $V$ into a channel, as long as that agent has, on each $V$ sent, *at least* the capabilities specified by T.

Agents may restrict access to a resource by controlling the type of the channel over which the name of the resource is sent. Thus if an agent sends the name of a location, $\ell$, over a channel of type $\mathsf{res}\{\mathsf{r}\langle \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}\rangle\}$, then the recipient gains capabilities on the channels $a$ and $b$ at $\ell$, as specified by the capability sets A and B respectively. Instead, when the same name is communicated over a channel of type $\mathsf{res}\{\mathsf{r}\langle \mathsf{loc}\{a{:}\mathrm{A}'\}\rangle\}$, the recipient gains access only to channel $a$ at $\ell$, with permissions determined by $\mathrm{A}'$.

The remainder of this extended abstract is organized as follows. In the next section we define the language D$\pi$ and its reduction semantics; we also describe several examples that highlight some of the main features of the language. The following section gives a description of the typing system and states a Subject Reduction Theorem; we show the application of the typing system to an example program, a *cell server*. In Section 4 we give an informal outline the Type Safety Theorem, which formalizes the idea that well-typed networks can not misuse resources. We end with a brief comparison with related work.

In this extended abstract all proofs are omitted, as are many other details, including some definitions. The reader is referred to the published technical report [14] for a full account.

## 2 The Language

A typical D$\pi$ network is the following:

$$\ell[\![P]\!] \mid (\nu_k a{:}\mathrm{A})\,(\ell[\![Q]\!] \mid k[\![R]\!])$$

There are three agents running in parallel: $\ell[\![P]\!]$ and $\ell[\![Q]\!]$ running at location $\ell$ and $k[\![R]\!]$ running at location $k$. Moreover $Q$ and $R$ share a private channel $a$, declared at location $k$. The syntax for the agents is a mild extension of that of the $\pi$-calculus; *structured values* may be exchanged along channels, and there is a new command for *code movement* '$\mathsf{go}\,\ell.P$', which causes the agent to move to location $\ell$ and then execute $P$.

The syntax of the language is given in Table 2, where letters *a–m* range over the set *Name* of *names*, *x–z* range over the disjoint set *Var* of *variables*, and *u–w* range over *identifiers* in $\mathit{Name} \cup \mathit{Var}$. The syntax for types T will be explained in the following section. Types appear in the binders for variables and names: the input construct '$u?(X{:}\mathrm{T})\,Q$' binds each of the variables in the pattern $X$; the restriction constructs '$(\nu e{:}\mathrm{T})\,P$' and '$(\nu_k e{:}\mathrm{T})\,N$' bind the name $e$.

The typing system will distinguish the location of resources, leading us to define

**Table 1** Syntax of patterns $X$, values $U$, threads $P$ and networks $M$

| | | | | | |
|---|---|---|---|---|---|
| $X,Y$ | $::= x$ | Variable | $U,V$ | $::= u$ | Id |
| | $\mid (X_1, .., X_n)$ | Tuple | | $\mid (U_1, .., U_n)$ | Tuple |
| $P,Q,R$ | $::= \mathsf{stop}$ | Termination | $M,N$ | $::= \mathbf{0}$ | Empty |
| | $\mid P \mid Q$ | Composition | | $\mid M \mid N$ | Composition |
| | $\mid (\nu e{:}\mathrm{T})\, P$ | Restriction | | $\mid (\nu_k e{:}\mathrm{T})\, N$ | Restriction |
| | $\mid \mathsf{go}\, u.P$ | Movement | | $\mid k[\![P]\!]$ | Agent |
| | $\mid u!\langle V\rangle P$ | Output | | | |
| | $\mid u?(X{:}\mathrm{T})\, P$ | Input | | | |
| | $\mid *P$ | Replication | | | |
| | $\mid \mathsf{if}\ U = V\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | Matching | | | |

dependent tuple types. In examples, we use the notation

$$u[v_1, .., v_n] \stackrel{def}{=} (u, v_1, .., v_n)$$

to indicate that the identifiers $v_i$ refer to resources at location $u$.

The reduction semantics is defined as a reduction relation between networks; thus judgments are of the form $M \longrightarrow M'$ where $M$ and $M'$ are (closed) network terms, *i.e.* terms which contain no free occurrences of variables. The semantics is given in Table 2 using two relations; a structural equivalence $\equiv$ and a primitive reduction relation $\longmapsto$. The main relation of interest is:

$$(\longrightarrow) \stackrel{def}{=} (\equiv \cdot \longmapsto \cdot \equiv)$$

The primitive reduction relation is defined to be the least precongruence on networks that satisfies the axioms given in Table 2. Most of the rules are familiar from the $\pi$-calculus, with a few changes to accommodate the fact that agents are explicitly located. The main new rule is that for code movement, $(\mathsf{r\text{-}move})$, which allows an agent to move from one location to another, say from $\ell$ to $k$: $\ell[\![\mathsf{go}\, k.P]\!] \longmapsto k[\![P]\!]$. The rule $(\mathsf{r\text{-}comm})$ for communication allows two agents running at the same location $\ell$ to exchange a value $V$ along a common channel $a$:

$$\ell[\![a!\langle V\rangle\, P]\!] \mid \ell[\![a?(X)\, Q]\!] \longmapsto \ell[\![P]\!] \mid \ell[\![Q\{^V\!/_X\}]\!]$$

It is worth emphasizing that the agents must be co-located for communication to occur; agents that wish to communicate on a remote channel must first move to the remote location using the asynchronous "move" operation. Nevertheless we can easily implement a form of remote asynchronous output by using $\ell.a!\langle V\rangle$ to denote $\mathsf{go}\,\ell.\, a!\langle V\rangle\,\mathsf{stop}$. In our reduction semantics we then have:

$$k[\![\ell.a!\langle V\rangle]\!] \mid \ell[\![a?(X)\, Q]\!] \longmapsto\longmapsto \ell[\![Q\{^V\!/_X\}]\!]$$

The purpose of the structural equivalence is to abstract from the static structure of terms, *i.e.* from the irrelevant details of the syntactic relation between composition $(P \mid Q)$, restriction $((\nu e)\, P)$ and location $(\ell[\![P]\!])$. The structural equivalence

**Table 2** Reduction

$$
\begin{array}{rl}
\text{(s-extr)} & N \mid (\nu e)\, M \equiv (\nu e)\,(N \mid M) \quad \text{if } e \notin \mathrm{fn}(N) \\
\text{(s-garbage}_1\text{)} & (\nu_\ell e{:}\mathrm{T})\,\mathbf{0} \equiv \mathbf{0} \\
\text{(s-garbage}_2\text{)} & \ell[\![\mathsf{stop}]\!] \equiv \mathbf{0} \\
\text{(s-split)} & \ell[\![P \mid Q]\!] \equiv \ell[\![P]\!] \mid \ell[\![Q]\!] \\
\text{(s-itr)} & \ell[\![*P]\!] \equiv \ell[\![P]\!] \mid \ell[\![*P]\!] \\
\text{(s-new)} & \ell[\![(\nu e{:}\mathrm{T})\,P]\!] \equiv (\nu_\ell e{:}\mathrm{T})\,\ell[\![P]\!] \quad \text{if } e \neq \ell \\
\text{(r-move)} & \ell[\![\mathsf{go}\,k.\,P]\!] \longmapsto k[\![P]\!] \\
\text{(r-comm)} & \ell[\![a!\langle V\rangle\, P]\!] \mid \ell[\![a?(X)\,Q]\!] \longmapsto \ell[\![P]\!] \mid \ell[\![Q\{^{V}/_{X}\}]\!] \\
\text{(r-eq}_1\text{)} & \ell[\![\mathsf{if}\ U = U\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \longmapsto \ell[\![P]\!] \\
\text{(r-eq}_2\text{)} & \ell[\![\mathsf{if}\ U = V\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] \longmapsto \ell[\![Q]\!] \quad \text{if } U \neq V
\end{array}
$$

is defined to be the least congruence over networks that satisfies the commutative monoid laws for composition and the axioms given in Table 2. In addition to the standard axiom for name extrusion (s-extr), the structural equivalence includes axioms that allow restriction and composition to be lifted from threads to networks. The most important of these is the rule (s-split) which allows an agent to split into two independent agents ($\ell[\![P \mid Q]\!] \equiv \ell[\![P]\!] \mid \ell[\![Q]\!]$). The rule (s-garbage$_2$) allows for garbage collection of terminated agents, whereas (s-itr) provides a standard interpretation of iteration. Note that when a channel name is extracted from a thread using (s-new) ($\ell[\![(\nu e{:}\mathrm{T})\,P]\!] \equiv (\nu_\ell e{:}\mathrm{T})\,\ell[\![P]\!]$) it is necessary to note the location where the name is defined. This in fact determines the syntactic form for channel restriction at the network level. In $(\nu_\ell a{:}\mathrm{A})\,M$ the channel $a$ defined at location $\ell$ and its scope is restricted to the network $M$.

**Example 2.1 (A Cell)** A simple system consisting of a user and a cell may be described as follows:

$$
\begin{array}{rl}
\mathrm{Net}_1 \Leftarrow & \ell[\![\mathrm{Cell}(v)]\!] \mid h[\![\mathrm{User}]\!] \\
\mathrm{Cell}(n) \Leftarrow & (\nu s)\, s!\langle n\rangle \\
& \quad \mid *g?(z) \quad s?(v)\ (s!\langle v\rangle \mid z.ret!\langle v\rangle) \\
& \quad \mid *p?(z,x)\ s?(v)\ (s!\langle x\rangle \mid z.ack!\langle\rangle) \\
\mathrm{User} \Leftarrow & \ell.p!\langle h, 2\rangle \mid ack?()\,(g!\langle h\rangle \mid ret?(x)\ print!\langle x\rangle)
\end{array}
$$

The cell has an internal channel $s$ in which the contents is stored and two public channels (or methods) for accessing the contents, $p$ for putting values into the cell and $g$ for getting the current contents; to make the example more accessible we assume the existence of some primitive values such as integers. The get method receives a return address from the user, which is assumed to be a location, reads the current contents and sends it along the channel *ret* at the callers site. The get method acts in a similar manner; it receives a value and a return address, updates the contents and sends an acknowledgement along the channel *ack* at the return address.

According to our reduction semantics the user and the cell may interact twice,

after which the *print* channel at the users site $\ell$ will have the value 2 available on it.

**Example 2.2 (A refined Cell)** The cell in the previous example has the disadvantage that it may only be used by users which have the two (global) methods $ret, ack$ available at their sites. Here we improve on this by using structured values:

$$\begin{aligned}
\mathrm{Net}_2 &\Leftarrow \ell[\![\mathrm{Cell}(n)]\!] \mid h[\![\mathrm{User}]\!] \\
\mathrm{Cell}(n) &\Leftarrow (\nu s)\, s!\langle n\rangle \\
&\qquad \mid *g?(z[y]) \quad s?(v)\ (s!\langle v\rangle \mid z.y!\langle v\rangle) \\
&\qquad \mid *p?(z[y],x)\ s?(v)\ (s!\langle x\rangle \mid z.y!\langle\rangle) \\
\mathrm{User} &\Leftarrow (\nu r_1)\, \ell.p!\langle h[r_1],2\rangle \mid r_1?()\,(\nu r_2)\,\big(g!\langle h[r_2]\rangle \mid r_2?(x)\ print!\langle x\rangle\big)
\end{aligned}$$

On the get method, for example, the cell receives a structured value consisting of a location, bound to $z$, and a channel $y$ *at that location*, reads the current contents and sends it along the newly acquired channel. When the cell is defined in this manner the user may generate new channels $r_1$, $r_2$, local to its site $h$, for the purpose of communicating with the cell. This interaction strategy on makes the cell less dependent on global assumptions.

**Example 2.3 (A Cell Server)** A server for generating new cells may be defined as $serv[\![\mathrm{S}]\!]$ where S is given by:

$$\mathrm{S} \Leftarrow *req?(z[y])\ (\nu cell)\ z.y!\langle cell\rangle \mid \mathrm{go}\, cell.Cell(2)$$

Upon receiving a new request, the server creates a new cell location *cell*, spawns the cell code at that location, initialized to 2, and then sends the name of the cell location to the user. A typical user would take the form $h[\![\mathrm{cU}]\!]$, where:

$$\mathrm{cU} \Leftarrow (\nu r)\ serv.req!\langle h[r]\rangle \mid r?(z)\ \mathrm{U}(z)$$

Many variations of cell servers can be described in our language. For example the following code describes a server which spawns a new cell at a location specified by the user; moreover the put and get methods are no longer global, but private to the new cell and the calling user:

$$\begin{aligned}
\mathrm{cS}' &\Leftarrow *req?(z[x])\ \mathrm{go}\, z.\,(\nu)\, p,g\ (\mathrm{Cell}(2) \mid x!\langle p,g\rangle) \\
\mathrm{cU}' &\Leftarrow (\nu r)\ serv.req!\langle h[r]\rangle \mid r?(p,g)\,\mathrm{U}'(p,g)
\end{aligned}$$

**Example 2.4 (Routed Forwarding)** Here we write a program $Forwarder(h[in], d[s])$ which establishes a connection between the local channel *in* and the (possibly remote) channel $s$. By "connection" we mean that messages sent into *in* should eventually find their way to the service channel $s$ at destination $d$. Such a program is trivial to write in D$\pi$:

$$*in?(x)\ \mathrm{go}\, d.\, s!\langle x\rangle$$

The unpleasant part of the problem specification is that we are not allowed to assume that there is a direct connection from the current location to $d$. Instead, the program must consult the local method $route(d)$ which returns the name of the neighboring location that is closest to $d$, *i.e.* somewhere between the current location and $d$. To make the program readable, we assume some additional syntactic

conventions, including recursive definitions and let-expressions.

$$
\begin{array}{lll}
\mathit{Forwarder}(h[in], d[s]) & \Leftarrow & \mathsf{if}\ h = d\ \mathsf{then} \\
& & \quad *in?(x)\ \ s!\langle x\rangle \\
& & \mathsf{else} \\
& & \quad \mathsf{let}\ n \leftarrow \mathit{route}(d) \\
& & \quad \mathsf{in}\ \ \mathsf{go}\, n.\, (\nu c)\ \ \mathit{Forwarder}(n[c], d[s]) \\
& & \qquad\qquad\qquad \mid \mathsf{go}\, h.\, *in?(x)\ \ \mathsf{go}\, n.\, c!\langle x\rangle \\
& & \mathsf{endif}
\end{array}
$$

When the *Forwarder* is started, it checks to see if the destination $d$ is the same as the current location $h$. If $h$ and $d$ are the same, then there is no need for routing, and the program can simply set up a forwarding process from $in$ to $s$: '$*in?(x)\ \ s!\langle x\rangle$'. If $h$ and $d$ are different, then the name of a neighbor $n$ is retrieved, where $n$ is between $h$ and $d$ on the network. Then a new copy of the code is started at $n$, and a forward process is set up between $in$ and $n$.

## 3 Typing

An informal description of the types for D$\pi$ was given in the introduction. Formally they are a subset of the *pre-types* defined in Table 3 which satisfy some consistency constraints. These pre-types belong to three distinct syntactic categories:

- location types, K, L, of the form $\mathsf{loc}\{\widetilde{\kappa}\}$, where $\kappa_i$ are location capabilities.
- channel types, A, B, C, of the form $\mathsf{res}\{\widetilde{\alpha}\}$, where $\alpha_i$ are channel capabilities.
- transmission types, S, T, which can be of the form K for locations, A for local resources, $\widetilde{\mathrm{T}}$ for tuples, or $\mathrm{K}[\widetilde{\mathrm{A}}]$ for dependent tuples with non-local resources.

Location and channel types are identified up to reordering of capabilities; in fact, they may be viewed simply as sets of capabilities. We also routinely drop brackets when they are empty.

The types come equipped with a subtyping relation, also defined in Table 3. For location pre-types we have $\mathrm{K} <: \mathrm{L}$ if for every capability $\lambda \in \mathrm{L}$ there exists a capability $\kappa \in \mathrm{K}$ which is "at least as good", *i.e.* $\kappa <: \lambda$. Here the location capabilities $\kappa$ and $\lambda$ are compared inductively using the associated types, *e.g.* $a{:}\mathrm{A} <: a{:}\mathrm{B}$ if $\mathrm{A} <: \mathrm{B}$. Subtyping for channels is just as for locations: $\mathrm{A} <: \mathrm{B}$ if for every capability $\beta \in \mathrm{B}$ there exists a capability $\alpha \in \mathrm{A}$ such that $\alpha <: \beta$. But the subtyping relation on channel capabilities is more interesting:

$$
\begin{array}{rcl}
\mathsf{r}\langle \mathrm{S}\rangle & <: & \mathsf{r}\langle \mathrm{T}\rangle \ \text{ if } \ \mathrm{S} <: \mathrm{T} \\
\mathsf{w}\langle \mathrm{S}\rangle & <: & \mathsf{w}\langle \mathrm{T}\rangle \ \text{ if } \ \mathrm{T} <: \mathrm{S}
\end{array}
$$

As one should expect from [22], the read capability is covariant, whereas the write capability is contravariant. Thus a receiver can always take *fewer* capabilities than specified, whereas a sender can always send *more*.

**Table 3** Pre-Types

Capabilities:

$$\begin{array}{rcl} \kappa & ::= & \mathsf{go} \mid \mathsf{newc} \\ & \mid & a{:}\mathrm{A} \\ \alpha & ::= & \mathsf{r}\langle \mathrm{T} \rangle \\ & \mid & \mathsf{w}\langle \mathrm{T} \rangle \end{array}$$

Subtyping:

$$\begin{array}{rcll} \kappa <: \kappa & & & \\ a{:}\mathrm{A} <: a{:}\mathrm{B} & & \text{if } \mathrm{A} <: \mathrm{B} & \\ \mathsf{r}\langle \mathrm{S} \rangle <: \mathsf{r}\langle \mathrm{T} \rangle & & \text{if } \mathrm{S} <: \mathrm{T} & \\ \mathsf{w}\langle \mathrm{S} \rangle <: \mathsf{w}\langle \mathrm{T} \rangle & & \text{if } \mathrm{T} <: \mathrm{S} & \end{array}$$

Pre-Types:

$$\begin{array}{rcl} \mathrm{K} & ::= & \mathsf{loc}\{\widetilde{\kappa}\} \\ \mathrm{A} & ::= & \mathsf{res}\{\widetilde{\alpha}\} \\ \mathrm{T} & ::= & \mathrm{K} \mid \mathrm{A} \mid (\mathrm{T}_1, .., \mathrm{T}_n) \\ & \mid & \mathrm{K}[\mathrm{A}_1, .., \mathrm{A}_n] \end{array}$$

$$\begin{array}{rcll} \mathrm{K} <: \mathrm{L} & & \text{if } \forall \lambda \in \mathrm{L} \colon \exists \kappa \in \mathrm{K} \colon \kappa <: \lambda & \\ \mathrm{A} <: \mathrm{B} & & \text{if } \forall \beta \in \mathrm{B} \colon \exists \alpha \in \mathrm{A} \colon \alpha <: \beta & \\ \widetilde{\mathrm{S}} <: \widetilde{\mathrm{T}} & & \text{if } \forall i \colon \mathrm{S}_i <: \mathrm{T}_i & \\ \mathrm{K}[\widetilde{\mathrm{A}}] <: \mathrm{L}[\widetilde{\mathrm{B}}] & & \text{if } \mathrm{K} <: \mathrm{L} \text{ and } \widetilde{\mathrm{A}} <: \widetilde{\mathrm{B}} & \end{array}$$

**Definition 3.1 (Types)**

(i) *A location pre-type* K *is a type if* $a{:}\mathrm{A} \in \mathrm{K}$ *and* $a{:}\mathrm{A}' \in \mathrm{K}$ *imply* $\mathrm{A} = \mathrm{A}'$.

(ii) *A channel pre-type* A *is a type if:*

$$\begin{array}{rcl} \mathsf{r}\langle \mathrm{T} \rangle \in \mathrm{A} & \textit{and} & \mathsf{r}\langle \mathrm{T}' \rangle \in \mathrm{A} \ \textit{imply}\ \mathrm{T} = \mathrm{T}' \\ \mathsf{w}\langle \mathrm{S} \rangle \in \mathrm{A} & \textit{and} & \mathsf{w}\langle \mathrm{S}' \rangle \in \mathrm{A} \ \textit{imply}\ \mathrm{S} = \mathrm{S}' \\ \mathsf{r}\langle \mathrm{T} \rangle \in \mathrm{A} & \textit{and} & \mathsf{w}\langle \mathrm{S} \rangle \in \mathrm{A} \ \textit{imply}\ \mathrm{S} <: \mathrm{T} \end{array}$$

(iii) *Pre-types of the form* $\widetilde{\mathrm{T}}$ *and* $\mathrm{K}[\widetilde{\mathrm{A}}]$ *are types if their constituent components are types.* □

Thus location types are allowed at most one capability for each channel. Channel types are also constrained to have at most one read and one write capability. The final constraint on channel types is a consistency requirement. It prevents agents from "fabricating" capabilities. For example, it prevents an agent from sending a value at type $\mathsf{loc}\{a{:}\mathrm{A}\}$ and then receiving the same value at type $\mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}$.

Readers familiar with [22] will notice that Pierce and Sangiorgi's channel types — "PS" types — are also representable in our type system (ignoring recursion). The PS read type $[\mathrm{T}]^{\mathsf{r}}$ is identified with $\mathsf{res}\{\mathsf{r}\langle \mathrm{T} \rangle\}$, the PS write type $[\mathrm{T}]^{\mathsf{w}}$ is identified with $\mathsf{res}\{\mathsf{w}\langle \mathrm{T} \rangle\}$, and the PS read/write type $[\mathrm{T}]^{\mathsf{rw}}$ is identified with $\mathsf{res}\{\mathsf{w}\langle \mathrm{T} \rangle, \mathsf{r}\langle \mathrm{T} \rangle\}$, which we abbreviate by $\mathsf{rw}\langle \mathrm{T} \rangle$. For these PS types, our definition of subtyping coincides with that of Pierce and Sangiorgi.

Our channel types include many types that are not definable using the system of Pierce and Sangiorgi, however. For example, the type

$$\mathrm{C} = \mathsf{res}\{\mathsf{r}\langle \mathsf{loc}\{a{:}\mathrm{A}\} \rangle, \mathsf{w}\langle \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\} \rangle\}$$

is not expressible as a PS type. Nonetheless, it is easy to see how such types arise when agents are granted different permissions on the names in a network.

In addition, our subtyping relation induces a partial *meet* operator '$\sqcap$'. No such operator exists for PS types — consider the types $[[]^{\mathsf{r}}]^{\mathsf{r}}$ and $[[]^{\mathsf{rw}}]^{\mathsf{w}}$.

**Table 4** A Type System

Values:

$$\frac{\Gamma(u) <: \mathrm{K}}{\Gamma \vdash_w u{:}\mathrm{K}} \qquad \frac{\Gamma(w) <: \mathsf{loc}\{u{:}\mathrm{T}\}}{\Gamma \vdash_w u{:}\mathrm{T}} \qquad \frac{\Gamma \vdash_w U_i{:}\mathrm{T}_i \ \ (\forall i)}{\Gamma \vdash_w \widetilde{U}{:}\widetilde{\mathrm{T}}} \qquad \frac{\Gamma \vdash_w u{:}\mathrm{K} \quad \Gamma \vdash_u \widetilde{v}{:}\widetilde{\mathrm{A}}}{\Gamma \vdash_w (u,\widetilde{v}){:}\mathrm{K}[\widetilde{\mathrm{A}}]}$$

Threads:

$$\frac{\Gamma \vdash_w u{:}\mathsf{res}\{\mathsf{w}\langle\mathrm{T}\rangle\} \quad \Gamma \vdash_w V{:}\mathrm{T} \quad \Gamma \vdash_w P}{\Gamma \vdash_w u!\langle V\rangle P} \qquad \frac{\Gamma \vdash_w u{:}\mathsf{res}\{\mathsf{r}\langle\mathrm{T}\rangle\} \quad \mathrm{fv}(X) \text{ disjoint } \mathrm{fv}(\Gamma) \quad \Gamma \sqcap \{_w X{:}\mathrm{T}\} \vdash_w Q}{\Gamma \vdash_w u?(X{:}\mathrm{T})\,Q}$$

$$\frac{\Gamma \vdash_w U{:}\mathrm{S} \quad \Gamma \vdash_w V{:}\mathrm{T} \quad \Gamma \sqcap \{_w U{:}\mathrm{T}\} \sqcap \{_w V{:}\mathrm{S}\} \vdash_w P \quad \Gamma \vdash_w Q}{\Gamma \vdash_w \mathsf{if}\ U = V\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\frac{\Gamma \vdash_w u{:}\mathsf{loc}\{\mathsf{go}\} \quad \Gamma \vdash_u P}{\Gamma \vdash_w \mathsf{go}\,u.P} \qquad \frac{k \notin \mathrm{fn}(\Gamma) \quad \Gamma \sqcap \{k{:}\mathrm{K}\} \vdash_w P}{\Gamma \vdash_w (\nu k{:}\mathrm{K})\,P} \qquad \frac{\Gamma \vdash_w w{:}\mathsf{loc}\{\mathsf{newc}\} \quad a \notin \mathrm{fn}(\Gamma) \quad \Gamma \sqcap \{_w a{:}\mathrm{A}\} \vdash_w P}{\Gamma \vdash_w (\nu a{:}\mathrm{A})\,P} \qquad \frac{\Gamma \vdash_w P \quad \Gamma \vdash_w Q}{\Gamma \vdash_w \mathsf{stop}, P \mid Q, *P}$$

Networks:

$$\frac{\Gamma \vdash_k k{:}\mathsf{loc} \quad \Gamma \vdash_k P}{\Gamma \vdash k[\![P]\!]} \qquad \frac{\Gamma \vdash_k k{:}\mathsf{loc} \quad \ell \notin \mathrm{fn}(\Gamma) \quad \Gamma \sqcap \{\ell{:}\mathrm{L}\} \vdash M}{\Gamma \vdash (\nu_k \ell{:}\mathrm{L})\,M} \qquad \frac{\Gamma \vdash_k k{:}\mathsf{loc}\{\mathsf{newc}\} \quad a \notin \mathrm{fn}(\Gamma) \quad \Gamma \sqcap \{_k a{:}\mathrm{A}\} \vdash M}{\Gamma \vdash (\nu_k a{:}\mathrm{A})\,M} \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash \mathbf{0}, M \mid N}$$

The primary judgments of the type system are of the form $\Gamma \vdash M$ where $\Gamma$ is a *type environment* and $M$ is a network term. Type environments are taken to be to be maps from identifiers to *open location types*, which have the form $\mathsf{loc}\{\widetilde{u}{:}\widetilde{\mathrm{T}}\}$. The typing system is given in Table 3 and uses auxiliary judgments for threads, identifiers and values. For threads, judgments have the form $\Gamma \vdash_w P$, indicating that the thread $P$ is well-typed to run at location $w$, where $w \in \mathrm{dom}(\Gamma)$. This in turn uses judgments of the form $\Gamma \vdash_w V{:}\mathrm{T}$, which indicates that the value $V$ is well formed at $w$ and has at least the capabilities specified by T.

In this extended abstract we do not explain the various rules in detail. Instead, we briefly look at some examples.

At the thread level to deduce that $\mathsf{go}\,u.P$ is well-typed to run at $w$, that is $\Gamma \vdash_w \mathsf{go}\,u.P$, we need to establish $\Gamma(u)$ is a location with go capability and that $P$ is well-typed to run at $u$, *i.e.* $\Gamma \vdash_u P$. At the network level to deduce that $u[\![P]\!]$ is well-typed, $\Gamma \vdash u[\![P]\!]$, we need to show that $u$ is a location and $P$ is well-typed to run at $u$, *i.e.* $\Gamma \vdash_u P$.

At the thread level to deduce $\Gamma \vdash_w u?(X{:}\mathrm{T})\,Q$ we must establish that $u$ can be assigned type $\mathsf{res}\{\mathsf{r}\langle\mathrm{T}\rangle\}$ at location $w$, $\Gamma \vdash_w u{:}\mathsf{res}\{\mathsf{r}\langle\mathrm{T}\rangle\}$, and that $Q$ is well-typed to run at $w$. But in showing the later, we may augment the environment $\Gamma$ with the information that $X$ is of type T, that is we must show $\Gamma \sqcap \{_w X{:}\mathrm{T}\} \vdash_w Q$. The formal definition of this environment extension uses the *partial meet* operator $\sqcap$,

mentioned above. Since the pattern $X$ may include structured values, the definition of environment extension is somewhat non-standard. For example, if $X$:T is $(x, z[y])$:$(\mathrm{B}, \mathsf{loc}\{a{:}\mathrm{A}'\}[\mathrm{C}])$ then:

$$\{{}_w X{:}\mathrm{T}\} = \{w{:}\mathsf{loc}\{x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}', y{:}\mathrm{C}\}\}$$

If further $\Gamma$ is $\{w{:}\mathsf{loc}\{a{:}\mathrm{A}\}\}$, then $\Gamma \sqcap \{{}_w X{:}\mathrm{T}\}$ denotes $\{w{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}', y{:}\mathrm{C}\}\}$. The same notation is used in the rules for restriction.

To deduce $\Gamma \vdash_w$ if $u = v$ then $P$ else $Q$, where in $\Gamma$ both $u$ and $v$ have location types — say $\Gamma(u) <: \mathrm{K}$ and $\Gamma(v) <: \mathrm{L}$ — then it is necessary to establish that $Q$ is well-typed to run at $w$, $\Gamma \vdash_w Q$, and that $P$ is well-typed to run at $w$, relative to an augmented version of $\Gamma$ in which both $u$ and $v$ have inherited each others type information: $\Gamma \sqcap \{u{:}\mathrm{L}, v{:}\mathrm{K}\} \vdash_w P$. It is worth noting that the Routed Forwarding example of the previous section cannot be typed using the standard typing rule for matching, which requires $\Gamma \vdash_w P$; other examples are discussed in the full version of the paper.

The main result of this section is the following:

**Theorem 3.2 (Subject reduction)** *If* $\Gamma \vdash M$ *and* $M \longrightarrow M'$ *then* $\Gamma \vdash M'$. □

**Example 3.3 (A Typed Cell Server)** As an example of the use of these types to control access to capabilities, consider again the cell server from Example 2.3, this time annotated with types.

$$\mathrm{S} \Leftarrow *req?(z[y])\ (\nu cell{:}\mathrm{L}_{\mathsf{cell}})\ z.y!\langle cell\rangle \mid \mathsf{go}\, cell.\mathit{Cell}(0)$$

where "Cell(0)" represents the code for the cell initialized to 0.

Let us use the abbreviations for PS types introduced on page 8. The *allocation type* $\mathrm{L}_{\mathsf{cell}}$ of the cell location *cell* can then be written:

$$\begin{aligned} \mathrm{L}_{\mathsf{cell}} &= \mathsf{loc}\{\mathsf{go}, \mathsf{newc}, g{:}\mathsf{rw}\langle \mathrm{T}_{\mathsf{g}}\rangle, p{:}\mathsf{rw}\langle \mathrm{T}_{\mathsf{p}}\rangle\} \\ \mathrm{T}_{\mathsf{g}} &= \mathsf{loc}\{\mathsf{go}\}[\mathsf{w}\langle \mathsf{int}\rangle] \\ \mathrm{T}_{\mathsf{p}} &= (\mathsf{loc}\{\mathsf{go}\}[\mathsf{w}\langle\rangle], \mathsf{int}) \end{aligned}$$

Location *cell* must be given at least the type $\mathrm{L}_{\mathsf{cell}}$ in order for the cell code to typecheck. Note that the channels $g$ and $p$ must be declared with both read and write capabilities as the server reads from them and a user must be able to write to them. The cell requires only the write capability on the response channels it receives on $p$ and $g$.

The user's capabilities on the cell are determined by the *transmission type* $\mathrm{T}_{\mathsf{req}}$ of channel *req* (which must have type $\mathsf{rw}\langle \mathrm{T}_{\mathsf{req}}\rangle$). If one takes

$$\begin{aligned} \mathrm{T}_{\mathsf{req}} &= \mathsf{loc}\{\mathsf{go}\}[\mathsf{w}\langle \mathrm{L}'_{\mathsf{cell}}\rangle] \\ \mathrm{L}'_{\mathsf{cell}} &= \mathsf{loc}\{\mathsf{go}, g{:}\mathsf{w}\langle \mathrm{T}_{\mathsf{g}}\rangle, p{:}\mathsf{w}\langle \mathrm{T}_{\mathsf{p}}\rangle\} \end{aligned}$$

then this type ensures that a cell user cannot "redefine" the methods $p$ or $g$ (by intercepting messages sent on these channels), nor can it create new channels at the cell location. We should point out that this typing also affords some level of protection

to the user. The response channel $r$ is sent to the server with write capability only; thus the server may not intercept other messages that the user may wish to receive on $r$. Perhaps more important, the user's location is sent without the privilege to create new channels there, keeping the server from performing any computation at the users location.

To emphasize the restrictions imposed by these capabilities consider the following user:

$$\mathrm{U} \Leftarrow (\nu r)\ \mathit{serv}.\mathit{req}!\langle h[r]\rangle \mid r?(z)\ \mathrm{U}'(z)$$

U requests a cell using the response channel $r$. Then the network $\mathit{serv}[\![\mathrm{S}]\!] \mid h[\![\mathrm{U}]\!]$ can reduce to

$$\mathit{serv}[\![\mathrm{S}]\!] \mid (\nu \mathit{cell}{:}\mathrm{L}_{\mathsf{cell}})\ h[\![\mathrm{U}'(\mathit{cell})]\!] \mid \mathit{cell}[\![\mathrm{Cell}]\!]$$

If $\mathrm{T}_{\mathsf{req}}$ is as above, then one can be sure that the agent $\mathrm{U}'(\mathit{cell})$ has restricted access to *cell* in this network. For example, if $\mathrm{U}'$ has the form

$$\mathrm{U}'(\mathit{cell}) \Leftarrow \mathsf{go}\,\mathit{cell}.p?(X)\ldots$$

then U will be untypable.

The user may pass on to its clients the capabilities it has received for the cell, or restrictions of them. For example if $\mathrm{U}'$ has the form

$$\mathrm{U}'(\mathit{cell}) \Leftarrow \mathit{req}_{\mathit{low}}?(z[y])\ z.y!\langle \mathit{cell}\rangle \mid \mathit{req}_{\mathit{high}}?(z[y])\ z.y!\langle \mathit{cell}\rangle \mid \ldots$$

then the capabilities sent to *low* and *high* priority clients can be controlled by the types of the channels $\mathit{req}_{\mathit{low}}$ and $\mathit{req}_{\mathit{high}}$. For example if $\mathit{req}_{\mathit{low}}$ has the type $\mathsf{loc}\{\mathsf{go}\}[\mathsf{w}\langle\mathsf{loc}\{\mathsf{go}, g{:}\mathsf{w}\langle \mathrm{T}_{\mathsf{g}}\rangle\}\rangle]$ then low priority clients will not have any access to the put method at the cell.

## 4 Type Safety

Due to lack of space in this extended abstract we can only we briefly outline the Type Safety theorem for D$\pi$.

We first define a tagged version of the language, where threads are explicitly annotated with the permissions/capabilities they have accumulated for locations. The syntax of threads and values is unchanged from that of Table 2; only the network level is affected, and here only the clause for agents. Each agent of the original language $\ell[\![P]\!]$ is tagged with a *closed* type environment $\Gamma$ which represents the capabilities (or permissions) of the agent. For example, the agent

$$\ell[\![P]\!]_{\{\ell:\mathsf{loc}\{a:\mathrm{A},b:\mathrm{B}\},k:\mathsf{loc}\{a:\mathrm{A}'\}\}}$$

has knowledge of resources $a$ and $b$ at $\ell$ and of resource $a$ at $k$. In addition to recording the *names* of available resources, the tag also records the *permissions* that the agent has acquired for the use of that resource (the types A, B and A$'$). This additional information allows fine control in the definition of runtime error.

**Table 5** Runtime Errors

| | | | |
|---|---|---|---|
| (e-move) | | $\ell[\![\mathsf{go}\,k.P]\!]_\Gamma \xrightarrow{err}$ | if $\Gamma(k) \not<: \mathsf{loc}\{\mathsf{go}\}$ |
| (e-newc) | | $\ell[\![(\nu a)P]\!]_\Gamma \xrightarrow{err}$ | if $\Gamma(k) \not<: \mathsf{loc}\{\mathsf{newc}\}$ |
| (e-snd) | | $\ell[\![a!\langle V\rangle Q]\!]_\Gamma \xrightarrow{err}$ | if $\Gamma_\ell(V) \not<: \mathrm{wobj}(\Gamma(\ell,a))$ |
| (e-rcv) | | $\ell[\![a?(X{:}\mathrm{T})P]\!]_\Gamma \xrightarrow{err}$ | if $\mathrm{robj}(\Gamma(\ell,a)) \not<: \mathrm{T}$ |
| (e-comm) | $\ell[\![a!\langle V\rangle P]\!]_\Delta \mid \ell[\![a?(X{:}\mathrm{T})Q]\!]_\Gamma \xrightarrow{err}$ | | if $\mathrm{wobj}(\Delta(\ell,a)) \not<: \mathrm{robj}(\Gamma(\ell,a))$ |

$$\text{(e-new)}\ \frac{M \xrightarrow{err}}{(\nu e)M \xrightarrow{err}} \qquad \text{(e-str)}\ \frac{M \xrightarrow{err}}{M \mid N \xrightarrow{err}} \qquad \frac{M \equiv N \quad N \xrightarrow{err}}{M \xrightarrow{err}}$$

Next the reduction semantics of Table 2 is adapted to show how tags evolve over time. To avoid confusion, we write $M \longmapsto M'$ for tagged reduction. The only non-trivial change is to the rule (r-comm) which is revised to

$$\ell[\![a!\langle V\rangle P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})Q]\!]_\Delta \longmapsto \ell[\![P]\!]_\Gamma \mid \ell[\![Q\{^{V}/_{X}\}]\!]_{\Delta \sqcap \{\ell V:\mathrm{T}\}}$$

Note that here the receiver can accumulate new capabilities from the sender as $\Delta \sqcap \{\ell V{:}\mathrm{T}\}$ denotes the result of augmenting $\Delta$ with the information that the value $V$ at $\ell$ has acquired the capabilities described by T. As an example, let $\mathrm{T} = \mathsf{loc}\{\}[C]$ in the following tagged network:

$$\ell[\![a!\langle k[c]\rangle\ P]\!]_{\{\ldots,\, k:\mathsf{loc}\{b:\mathrm{B},c:\mathrm{C}\}\}} \mid \ell[\![a?(z[x]{:}\mathrm{T})\ Q]\!]_{\{\ldots,\, k:\mathsf{loc}\{d:\mathrm{D}\}\}}$$

After the communication the network is:

$$\ell[\![P]\!]_{\{\ldots,\, k:\mathsf{loc}\{b:\mathrm{B},c:\mathrm{C}\}\}} \mid \ell[\![Q\{^{k[c]}/_{z[x]}\}]\!]_{\{\ldots,\, k:\mathsf{loc}\{d:\mathrm{D},c:\mathrm{C}\}\}}$$

The receptor has gained extra capabilities through this communication, as mediated through the reception type T.

Next because of the presence of these tags we can easily define a notion of run-time error; informally $M \xrightarrow{err}$ means that somewhere in the (tagged network) $M$ a thread can use a resource in some manner which contradicts the explicit permissions it has accumulated over that resource. The formal definition is given in Table 4, where $\mathrm{robj}(\mathsf{res}\{\mathsf{r}\langle\mathrm{T}\rangle,\ldots\}) = \mathrm{T}$ and $\mathrm{wobj}(\mathsf{res}\{\mathsf{w}\langle\mathrm{T}\rangle,\ldots\}) = \mathrm{T}$.

The final step in the formalization is to extend the typing system of Table 3 to tagged networks, $\Gamma \Vdash M$. This is achieved by adding the following rule, where $\Gamma <: \Delta$ if $\Gamma(w) <: \Delta(w)$ for every $w$ in $\mathrm{dom}(\Delta)$.

$$\frac{\Delta \vdash_k k{:}\mathsf{loc} \quad \Delta \vdash_k P}{\Gamma \Vdash k[\![P]\!]_\Delta}\ \Gamma <: \Delta$$

Within this framework we can prove the following results:

- SUBJECT REDUCTION FOR TAGGED NETWORKS: For all tagged networks $N$, $\Gamma \Vdash N$ and $N \longmapsto N'$ then $\Gamma \Vdash N'$

- TYPE SAFETY FOR TAGGED NETWORKS: For all tagged networks $N$, $\Gamma \Vdash N$ implies $N \stackrel{err}{\not\longrightarrow}$
- STRONG EQUIVALENCE OF TAGGED AND UNTAGGED REDUCTION: For every well-typed (untagged) network, $\Gamma \vdash M$, we can define a canonical well-typed tagged network $\Gamma \Vdash \mathrm{tag}_\Gamma(M)$ that is strongly bisimilar to $M$.

These results, together with Theorem 3.2, imply that a well-typed network $\Gamma \vdash M$ is strongly bisimilar to the tagged network $\mathrm{tag}_\Gamma(M)$ and that this explicitly tagged network will never raise a runtime error, *i.e.* no agent will ever misuse a resource during its execution.

## 5 Related Work

There are numerous languages now in the literature for describing distributed systems; Dπ is perhaps closest in spirit to [9,2,23,3] which also take as their point of departure the π-calculus, although with each there are significant differences. For example in the join calculus [9] message routing is *automatic* as the restricted syntax ensures that all channels have a unique location at which they are serviced. In Dπ, to send a message to a remote location, an agent must first spawn a sub-agent which moves to that location; locations are more *visible* in Dπ. In addition, several of these languages [9,23,3] adopt *location movement* as the mechanism for agent mobility. Location movement allows groups of running threads to be moved about the network asynchronously (*i.e.* without each thread performing an explicit go); for further discussion, see the full version.

Many channel-based typing systems for π-calculi and related languages have been proposed. For example in [22], Pierce and Sangiorgi define a type system for the π-calculus with read and write capabilities on channels. Sewell [23] generalizes the type system of [22] to distinguish between *local* communication, which can be efficiently implemented, and *non-local* communication. Fournet *et al*. [10] have developed an *ML*-style typing system for the join calculus where channels are allowed a certain amount of polymorphism. Amadio [2] has presented a type system that guarantees that channel names are defined at exactly one location, whereas the type system of Kobayashi *et al*. [17] ensures that some channels are used linearly.

The work closest to ours is that of de Nicola, Ferrari and Pugliese [7]. Their goals are the same as ours, but the specifics of their solution are quite different. They work with a variant of Linda [6] with multiple "tuple spaces". Tuple spaces correspond to locations in our setting, and tuples (named data) correspond to resources. The type system of [7] controls access to tuple spaces, rather than to specific tuples, and thus provides coarser-grained control of resource access than that provided by our typing system.

Static analyses for proving various security properties of programs have also been proposed by several authors; two recent references are [16,13].

## Acknowledgements

We would like to thank INRIA Sophia Antipolis for their hospitality while conducting this research. We have benefited from conversations with Alan Jeffrey, Peter Sewell and Luca Cardelli, among others.

## References

[1] R. Amadio and S. Prasad. Modelling IP mobility. Internal Report 244, Laboratoire d'Informatique de Marseille, 1997.

[2] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proc. FOSSACS'98, International Conference on Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[4] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. A preliminary version appeared in Proceedings of the 22nd ACM Symposium on Principles of Programming.

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

[6] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science, pages 66–76. Springer-Verlag, 1995.

[7] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[8] K.M. Chandy *et al*. A world-wide distributed system using java and the internet. In *IEEE International Symposium on High Performance Distributed Computing*. IEEE, August 1996.

[9] C. Fournet, G. Gonthier, J.J. Levy, L. Marganget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *CONCUR: Proceedings of the International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, August 1996. Springer-Verlag.

[10] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing la ml for the join-calculus. In *CONCUR: Proceedings of the International Conference on Concurrency Theory*, Lecture Notes in Computer Science, Warsaw, August 1997. Springer-Verlag.

[11] General Magic Inc. Agent technology. hhtp://www.genmagic.com/html/agent_overview.html, 1997.

[12] A. Giacalone, P. Mishra, and S. Prasad. A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[13] N. Heintz and J.G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

[14] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Computer Science Technical Report 2/98, University of Sussex, 1998. Available from http://www.cogs.susx.ac.uk/.

[15] IBM Corp. The IBM aglets workbench. http://www.trl.ibm.co.jp/aglets/, 1996.

[16] G. Karjoth, D.B. Lange, and M. Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4), 1997.

[17] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Paris, January 1996. ACM Press.

[18] R. Milner. The polyadic $\pi$-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.

[20] ObjectSpace Inc. Objectspace voyager. http://www.objectspace.com/voyager, 1997.

[21] C. Perkins. IP mobility support. RFC 2002, 1996.

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

[23] P. Sewell. Global/local subtyping for a distributed $\pi$-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, August 1997.

[24] Sun Microsystems Inc. Java home page. http://www.javasoft.com/, 1995.