Location Independence for Mobile Agents

Peter Sewell Paweł T. Wojciechowski Benjamin C. Pierce
University of Cambridge Indiana University
{Peter.Sewell,Pawel.Wojciechowski}@cl.cam.ac.uk pierce@cs.indiana.edu

1 Introduction

In recent years there has been an explosion of interest in wide-area distributed applications, executing on intranets or on the global internet [CHK97]. A key concept for structuring such applications is that of *mobile agents*, i.e. units of executing code that can migrate between sites. Mobile agents require novel programming language support and raise new problems, of interaction between agents, of robustness under network failure and reconfiguration, of binding to resources, and of security. In this paper we study the first of these, considering the design, semantic definition and implementation of communication primitives by which mobile agents can interact.

Communication primitives can be classified into two groups. At a low level there are *location dependent* primitives that require an application programmer to know the current site of a mobile agent in order to communicate with it. If a party to such communications migrates then the application program must explicitly track its new location. At a high level there are *location independent* primitives (mentioned, e.g., in [MGW97]) that allow communication with a mobile agent irrespective of its current site and of any migrations. Location independent primitives have the potential to greatly simplify the construction of applications. Their design is problematic, however – a distributed infrastructure is required, executing algorithms for tracking migrations and delivering messages to migrating agents. These algorithms are highly concurrent. Moreover, their behaviour is exposed, via performance and network failure/reconfiguration, to the application programmer. There is therefore a considerable advantage to be gained from expressing the specifications and implementations of location independent primitives in a form with clear and precise semantics.

In Section 2 we propose a simple calculus of agents, building on [MPW92, FGL$^+$96, Sew98], that can migrate between sites and can interact by both location dependent and independent message passing. It has a precise reduction semantics. Implementations of the location independent primitives can be expressed as encodings, or compilations, of the whole calculus into the fragment with only location dependent communication (henceforth, the D-fragment). This allows:

- The algorithms to be stated and understood precisely.
- A clean implementation strategy for prototype languages – implementations can be factored into an implementation of the D-fragment (which requires no distributed infrastructure) together with an encoding realized as one phase of a compiler.
- Reasoning. One would like to have *correctness results*, in the absence of failures, and *robustness results*, showing how the infrastructure degrades with limited failures. A functioning agent infrastructure will critically depend on these properties. The algorithms appear to be sufficiently subtle to make such proofs worthwhile, yet are small enough to make them tractable.

- Heterogeneity. The potential applications of mobile agent technology are extremely varied, involving agents with very different patterns of migration and communication and requiring different failure guarantees. For example, one can consider an active-badge tracker (with frequent migrations within a local network), an active-badge server (migrating only to avoid machine reboots) and a remote database interrogator (with a single migration into a network under a different administration). Any given infrastructure algorithm will have satisfactory behaviour only for some of these patterns of behaviour (of agents and of the network), and therefore only for some applications. It is therefore important to associate algorithms with common patterns of behaviour for which they are satisfactory, and to be able to provide a heterogeneous infrastructure above a standard base language.

We conclude in Section 3 by illustrating a sample encoding that uses a forwarding-pointers algorithm. Work on a prototype implementation, and on correctness and robustness results, is ongoing.

2 An Agent Calculus

For the implementation strategy above to be realistic it is essential that the D-fragment should be directly implementable using the communication primitives provided by actual networks. The fundamental communication primitive provided by most current network technology is, at the IP level, asynchronous, unordered, unreliable packet delivery to machines (strictly, to IP addresses). The calculus is designed so that each reduction step of the D-fragment requires at most one message (asynchronous, unordered, of arbitrary size) to be sent between sites. It is thus directly implementable above a lightweight reliable datagram protocol above IP, or above TCP by using a lightweight layer that opens and closes streams as required. We are thus abstracting from details of retransmission, of packet fragmentation, of stream connections etc., while retaining a clear relationship between the reduction semantics and the behaviour of an actual implementation.

For simplicity, we consider messages sent to agents themselves. In practice more structured communication, to entities within agents, would be required, e.g. method invocations of concurrent objects, remote procedure calls, distributed join calculus communication [FGL$^+$96], distributed π-calculus channels or multicasts. These are largely orthogonal to location independence – they are encodable, at least conceptually, in a calculus with location independent message passing, though usable implementations may require optimizations that are incompatible with this factorisation.

We take a *site* to be an instantiation of a runtime system on a machine; sites have unique names. We take an *agent* to be a unit of executing code; agents have unique names and are, at any moment, located at particular sites. Letting $a, b, \ldots$ range over names (of sites and of agents), the code of an agent is a term of the grammar

$$P ::= 0 \;\Big|\; P_1 \mid P_2 \;\Big|\; \textbf{create } a \textbf{ is } P_1.P_2 \;\Big|\; \textbf{migrate to } s.P \;\Big|\; \overline{a}v \;\Big|\; (y).P \;\Big|\; \overline{a@s}v \;\Big|\; \textbf{tryhere } \overline{a}v.P_1 \textbf{ else } P_2$$

In **create** a **is** $P_1.P_2$ the a binds in P_1 and P_2; in $(y).P$ the y binds in P. Free name sets and substitution will be written $\mathrm{fn}(P)$ and $\{v/x\}P$; we work up to alpha conversion of bound names.

Nil and parallel The empty agent is written 0; the parallel composition of two parts of an agent is written $P \mid Q$.

Creation and migration The construct **create** a **is** $P_1.P_2$ allows a new agent to be spawned on the current site, with code P_1. After the creation P_2 commences execution as part of the spawning agent. Both P_1 and P_2 can refer to the name a of the spawned agent. The construct **migrate to** $s.P$ allows the agent containing it to migrate to site s; after the migration P commences execution.

Location independent communication The output $\overline{a}v$ sends a message v to the agent named a, irrespective of the site of a. The input $(y).P$ waits for such a message v to arrive, then executes P with v replacing the formal parameter y.

Location dependent communication The output $\overline{a@s}v$ is similar to $\overline{a}v$ except that the site s of the destination agent must be given explicitly – if a *is* located at s then the message v will be delivered to it, otherwise the output will fail silently.

Test and send If agent a is located at the current site the construct **tryhere** $\overline{a}v.P_1$ **else** P_2 will deliver v to a and then execute P_1, otherwise it will execute P_2. This primitive is only required for some algorithms.

The D-fragment is given by the subgrammar without $\overline{a}v$, the purely location independent fragment (henceforth, the I-fragment) is given by the subgrammar without $\overline{a@s}v$. To define the operational semantics we extend the calculus to allow configurations of many agents, each with combined code and state, to be expressed. We take primitives for declaring a term P to be part of an agent a, for binding an agent name a and declaring it to be at site s, and for a message that has reached its destination agent:

$$P ::= \ldots \;\Big|\; @_a P \;\Big|\; (\mathbf{new}\ a@s)P \;\Big|\; v$$

In $(\mathbf{new}\ a@s)P$ the a binds in P. Certain term formation rules are required, to forbid e.g. $(y).@_y P$; we omit their definition. A term $@_a P$, where P is from the first syntax, denotes a program that is initially a single agent a. We take *location contexts* Γ to be finite partial functions from names to names, giving the sites of free agent names. The *reduction relation* is given by the axioms

$$\begin{array}{lcll}
\Gamma,\ @_a\mathbf{create}\ b\ \mathbf{is}\ P.Q & \longrightarrow & \Gamma,\ (\mathbf{new}\ b@s)(@_b P \mid @_a Q) & \text{if } \Gamma(a)=s\\
\Gamma,\ @_a\mathbf{migrate\ to}\ s.P & \longrightarrow & (\Gamma \oplus a \mapsto s),\ @_a P & \\
\Gamma,\ @_a\overline{b}v & \longrightarrow & \Gamma,\ @_b v & \\
\Gamma,\ @_a(v \mid (y).P) & \longrightarrow & \Gamma,\ @_a\{v/y\}P & \\
\Gamma,\ @_a\overline{b@s}v & \longrightarrow & \Gamma,\ @_b v & \text{if } \Gamma(b)=s\\
\Gamma,\ @_a\overline{b@s}v & \longrightarrow & \Gamma,\ 0 & \text{if } \Gamma(b)\neq s\\
\Gamma,\ @_a\mathbf{tryhere}\ \overline{b}v.P\ \mathbf{else}\ Q & \longrightarrow & \Gamma,\ @_b v \mid @_a P & \text{if } \Gamma(a)=\Gamma(b)\\
\Gamma,\ @_a\mathbf{tryhere}\ \overline{b}v.P\ \mathbf{else}\ Q & \longrightarrow & \Gamma,\ @_a Q & \text{if } \Gamma(a)\neq\Gamma(b)
\end{array}$$

together with closure under structural congruence, parallel and new:

$$\frac{Q \equiv P \quad \Gamma, P \longrightarrow \Gamma', P' \quad P' \equiv Q'}{\Gamma, Q \longrightarrow \Gamma', Q'} \qquad \frac{\Gamma, P \longrightarrow \Gamma', P'}{\Gamma, P \mid Q \longrightarrow \Gamma', P' \mid Q} \qquad \frac{(\Gamma, a \mapsto s), P \longrightarrow (\Gamma, a \mapsto s'), P'}{\Gamma, (\mathbf{new}\ a@s)P \longrightarrow \Gamma, (\mathbf{new}\ a@s')P'}$$

where the structural congruence $\equiv$ is the least congruence relation over terms satisfying the following.

$$\begin{array}{rclrclr}
P \mid Q & \equiv & Q \mid P & P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & P \mid 0 \equiv P\\
@_a(P \mid Q) & \equiv & @_a P \mid @_a Q & P \mid (\mathbf{new}\ a@s)Q & \equiv & (\mathbf{new}\ a@s)(P \mid Q) & \text{if } a \notin \mathrm{fn}(P)\\
 & & & (\mathbf{new}\ a@s)(\mathbf{new}\ b@t)P & \equiv & (\mathbf{new}\ b@t)(\mathbf{new}\ a@s)P & \text{if } a \notin \{b,t\} \wedge b \neq s
\end{array}$$

Note that the reduction axioms allow a location dependent output $\overline{b@s}v$ to silently lose the message v if agent b is not at site s. One would expect to prove that this can never occur for the encoding of a program from the I-fragment. (The originating agent of such an output may itself migrate, so other possible semantics, e.g. returning an error message to the originating agent, require complex algorithms. These could be incorporated into semantic definitions given as encodings into the D-fragment.)

As presented, the calculus is rather far from being a practical programming language – it does not allow any values except names to be communicated and does not have primitives for non-trivial computation within an agent. The choice of these primitives is largely orthogonal to location independence – in the sample encoding of Section 3, and in our prototype implementation, we are extending the Pict programming language of Pierce and Turner [PT97], based on the π-calculus of Milner, Parrow and Walker [MPW92]. The agent primitives could be added to other languages.

This paper does not explicitly address questions of security, or of network failure/reconfiguration. It does, however, identify a level of abstraction that may be a useful basis for such work – to consider whether a distributed infrastructure for mobile agents is secure or robust one must first be able to define it precisely, and have a clear understanding of how it is distributed on actual machines.

3 Example: A Forwarding Pointers Infrastructure

We conclude by sketching an encoding using a highly sequentialised forwarding-pointers algorithm. It is intermediate in complexity between the simplest (and most unsatisfactory) centralised-server solutions and algorithms that may be of practical use – the intention is to illustrate that such encodings can be precisely stated in a tractable way, rather than to propose an innovative algorithm.

The core of the encoding is given in Figure 1. It has a daemon at each site that maintains a collection of forwarding pointers; an agent that is migrating from s to s' synchronises with the daemons daemon_s and $\mathit{daemon}_{s'}$. Location independent communications are implemented via the daemons, using the forwarding pointers where possible. If a daemon has no pointer for the destination agent of a message then it will forward the message to the daemon on the site where the destination agent was created; to make this possible an agent name is encoded by a triple of an agent name and the site and daemon of its creation. Similarly, a site name is encoded by a pair of a site name and the daemon name for that site.

To express the encoding we extend the calculus of Section 2 with primitives loosely taken from Pict (the encoding should be extended homomorphically). We take types Site of site names, $\mathit{Agent}\ T$ of agents receiving values of type T, tuples $[T_1..T_n]$, existential polymorphic types $[\#X T_1..T_n]$, variants $\{\text{label}_1 \triangleright T_1..\text{label}_n \triangleright T_n\}$, recursive types $\mathit{rec}\, X = T$, local (to within an agent) channels $\updownarrow T$ carrying T, and finite maps $\mathsf{Map}\ T\ T'$ from T to T'. We take primitives for communication on local channels: for output $\overline{x}v$, input $x(y).P$, replicated input $!x(y).P$, and new local channel creation $\textbf{new}\ c : \updownarrow T$. The extended calculus is not given a precise definition here – the target language of the encoding might therefore be regarded as a form of pseudocode, albeit one that describes concurrency and synchronisation rather more explicitly than usual. Typing environments are also elided. The typing rules are standard except for the use of global/local typing [Sew98] to prohibit the sending of local pointers (in this setting, local channel names) between agents, and for the use of name equality testing over $[\#X\ \mathit{Agent}(X)]$.

$$
\begin{array}{lll}
\llbracket 0 \rrbracket_{a,here,buf,ack} & = & 0 \\
\llbracket P \mid Q \rrbracket_{a,here,buf,ack} & = & \llbracket P \rrbracket_{a,here,buf,ack} \mid \llbracket Q \rrbracket_{a,here,buf,ack}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \textbf{create } b : \mathit{Agent}\,T \textbf{ is } P.Q \rrbracket_{a,here,buf,ack} = \mathit{here}([S\;\; D_S]). \\
\qquad\qquad\qquad (\textbf{create } B : \mathit{AGENT}\, \llbracket T \rrbracket \textbf{ is} \\
\qquad\qquad\qquad\quad (\textbf{new } \mathit{here}' : \updownarrow[\mathit{Site}\;\; \mathit{Daemon}],\; \mathit{buf}' : \updownarrow \llbracket T \rrbracket,\; \mathit{ack}' : \updownarrow [],\; x : \updownarrow [] \\
\qquad\qquad\qquad\quad \textbf{let } b = [B\;\; S\;\; D_S] \textbf{ and } [A\;\; \_\;\; \_] = a \textbf{ in} \\
\qquad\qquad\qquad\qquad (\overline{D_S @ S}\{\text{Register} \rhd [\llbracket T \rrbracket\;\; B]\} \mid \overline{\mathit{here}'}[S\;\; D_S] \mid \overline{x}[] \\
\qquad\qquad\qquad\qquad \mid\, !x().(w).(\overline{x}[] \mid \textbf{case } w \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\text{Data} \rhd v\} \to \overline{\mathit{buf}'}v \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\text{Ack} \rhd\} \to \overline{\mathit{ack}'}[] \quad \textbf{endcase}) \\
\qquad\qquad\qquad\qquad \mid \mathit{ack}'().(\overline{A @ S}\{\text{Ack} \rhd []\} \mid \llbracket P \rrbracket_{b,here',buf',ack'})) \\
\qquad\qquad\qquad\quad). \\
\qquad\qquad\qquad \textbf{let } b = [B\;\; S\;\; D_S] \textbf{ in } \mathit{ack}().(\overline{\mathit{here}}[S\;\; D_S] \mid \llbracket Q \rrbracket_{a,here,buf,ack}))
\end{array}
$$

$$
\begin{array}{l}
\llbracket \textbf{migrate to } u.P \rrbracket_{a,here,buf,ack} = \mathit{here}([S\;\; D_S]). \\
\qquad\qquad\qquad \textbf{let } [U\;\; D_U] = u \textbf{ and } [A\;\; \_\;\; \_] = a \textbf{ in} \\
\qquad\qquad\qquad\quad \overline{D_S @ S}\{\text{Embark} \rhd [\llbracket T \rrbracket\;\; A]\} \mid \mathit{ack}().(\textbf{migrate to } U. \\
\qquad\qquad\qquad\qquad (\overline{D_U @ U}\{\text{Register} \rhd [\llbracket T \rrbracket\;\; A]\} \\
\qquad\qquad\qquad\qquad \mid \mathit{ack}().(\overline{D_S @ S}\{\text{Migrated} \rhd [\llbracket T \rrbracket\;\; A\;\; [U\;\; D_U]]\} \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{ack}().(\overline{\mathit{here}}[U\;\; D_U] \mid \llbracket P \rrbracket_{a,here,buf,ack})) \\
\qquad\qquad\qquad\qquad)) \qquad\qquad\qquad\qquad \text{where } a : \mathit{Agent}\,T \text{ in the source}
\end{array}
$$

$$
\begin{array}{lll}
\llbracket \overline{b}v \rrbracket_{a,here,buf,ack} & = & \mathit{here}([S\;\; D_S]).(\overline{D_S @ S}\{\text{Message} \rhd [\llbracket T \rrbracket\;\; b\;\; v]\} \mid \overline{\mathit{here}}[S\;\; D_S]) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \text{where } b : \mathit{Agent}\,T \text{ in the source} \\
\llbracket \overline{b @ u}v \rrbracket_{a,here,buf,ack} & = & \textbf{let } [U\;\; \_] = u \textbf{ and } [B\;\; \_\;\; \_] = b \textbf{ in } \overline{B @ U}\{\text{Data} \rhd v\} \\
\llbracket (y).P \rrbracket_{a,here,buf,ack} & = & \mathit{buf}(y).\llbracket P \rrbracket_{a,here,buf,ack}
\end{array}
$$

$$
\begin{array}{l}
\llbracket \textbf{tryhere } \overline{b}v.P \textbf{ else } Q \rrbracket_{a,here,buf,ack} = \textbf{let } [B\;\; \_\;\; \_] = b \textbf{ in} \\
\qquad\qquad\qquad \textbf{tryhere } \overline{B}\{\text{Data} \rhd v\}\, .\llbracket P \rrbracket_{a,here,buf,ack} \textbf{ else } \llbracket Q \rrbracket_{a,here,buf,ack}
\end{array}
$$

$$
\begin{array}{l}
\mathit{daemon}_s \overset{\mathit{def}}{=} \textbf{let } [S\;\; D_S] = s \textbf{ in } \textbf{new } z : \updownarrow(\mathsf{Map}\;[\#X\;\; \mathit{AGENT}(X)]\; \updownarrow[\mathit{Site}\;\; \mathit{Daemon}]) \\
\quad \overline{z}\mathsf{Map.empty} \mid\, !z(\mathit{Pointers}).(w).\textbf{case } w \textbf{ of} \\
\quad \{\text{Register} \rhd [X\;\; B]\} \to \textbf{case } (\mathsf{Map.lookup}\; \mathit{Pointers}\; [X\;\; B]) \textbf{ of} \\
\qquad \{\text{Found} \rhd p\} \to p(\_).(\overline{p}[S\;\; D_S] \mid \overline{z}\mathit{Pointers} \mid \overline{B @ S}\{\text{Ack} \rhd []\}) \\
\qquad \{\text{NotFound} \rhd\} \to \textbf{new } p : \updownarrow[\mathit{Site}\;\; \mathit{Daemon}] \\
\qquad\quad (\overline{p}[S\;\; D_S] \mid \overline{z}(\mathsf{Map.add}\; \mathit{Pointers}\; [X\;\; B]\; p) \mid \overline{B @ S}\{\text{Ack} \rhd []\}) \quad \textbf{endcase} \\
\quad \{\text{Embark} \rhd [X\;\; B]\} \to \textbf{case } (\mathsf{Map.lookup}\; \mathit{Pointers}\; [X\;\; B]) \textbf{ of} \\
\qquad \{\text{Found} \rhd p\} \to p(\_).(\overline{z}\mathit{Pointers} \mid \overline{B @ S}\{\text{Ack} \rhd []\}) \\
\qquad \{\text{NotFound} \rhd\} \to 0 \quad \textbf{endcase} \\
\quad \{\text{Migrated} \rhd [X\;\; B\;\; [U\;\; D_U]]\} \to \textbf{case } (\mathsf{Map.lookup}\; \mathit{Pointers}\; [X\;\; B]) \textbf{ of} \\
\qquad \{\text{Found} \rhd p\} \to \overline{z}\mathit{Pointers} \mid \overline{p}[U\;\; D_U] \mid \overline{B @ U}\{\text{Ack} \rhd []\} \\
\qquad \{\text{NotFound} \rhd\} \to 0 \quad \textbf{endcase} \\
\quad \{\text{Message} \rhd [X\;\; [B\;\; U\;\; D_U]\;\; v]\} \to \textbf{case } (\mathsf{Map.lookup}\; \mathit{Pointers}\; [X\;\; B]) \textbf{ of} \\
\qquad \{\text{Found} \rhd p\} \to \overline{z}\mathit{Pointers} \mid p([R\;\; D_R]).\textbf{tryhere } \overline{B}\{\text{Data} \rhd v\}\, .\overline{p}[R\;\; D_R] \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } (\overline{D_R @ R}\{\text{Message} \rhd [X\;\; [B\;\; U\;\; D_U]\;\; v]\} \mid \overline{p}[R\;\; D_R]) \\
\qquad \{\text{NotFound} \rhd\} \to \overline{z}\mathit{Pointers} \mid \overline{D_U @ U}\{\text{Message} \rhd [X\;\; [B\;\; U\;\; D_U]\;\; v]\} \quad \textbf{endcase} \\
\quad \textbf{endcase}
\end{array}
$$

In the target A, B are agent names, R, S, U are site names, D_R, D_S, D_U are the associated daemon names, *buf*, *ack* are local channels, and *here*, x, z, p are local channels used as locks.

Figure 1: A Forwarding-Pointers Distributed Infrastructure

The semantic function $[\![\, P \,]\!]_{a,here,buf,ack}$ is parameterised by the name a of the agent that P is part of, and by local channel names $here, buf, ack$, used respectively to store the agent's current site and daemon name, to buffer messages received by the agent and to buffer acknowledgements received by the agent. The daemons are agents (that never migrate) of type *Daemon*, i.e.

$$\begin{array}{l} rec\, Z = Agent\{\text{Register} \triangleright [\#X \;\; AGENT(X)] \quad \text{Embark} \triangleright [\#X \;\; AGENT(X)] \\ \quad\quad \text{Migrated} \triangleright [\#X \;\; AGENT(X) \;\; [Site \;\; Z]] \quad \text{Message} \triangleright [\#X \;\; [AGENT(X) \;\; Site \;\; Z] \;\; X] \,\} \end{array}$$

where $AGENT(X) = Agent\{\text{Data} \triangleright X \;\; \text{Ack} \triangleright []\}$. The encoding of types is homomorphic except for $[\![\, Agent\; T \,]\!] = [AGENT([\![\, T \,]\!]) \;\; Site \;\; Daemon]$ and $[\![\, Site \,]\!] = [Site \;\; Daemon]$. Putting this all together, a program $@_b P$, consisting of an agent b accepting values of type T, initiated at site s_1 with free site names $s_1, \ldots, s_n$, is encoded by

$$\begin{array}{rcl} Q & = & \sigma\big(@_{D_{s_1}} daemon_{s_1} \mid \ldots \mid @_{D_{s_n}} daemon_{s_n}\big) \\ & & \mid @_a \sigma\big(\mathbf{new}\;\, here : \updownarrow[Site \;\; Daemon],\; x : \updownarrow[]\; (\overline{here} s_1 \mid [\![\, \mathbf{create}\; b : Agent\, T \;\mathbf{is}\; P.0 \,]\!]_{a,here,x,x})\big) \end{array}$$

where $a, x, here$ and the D_{s_i} are fresh names and σ is the substitution replacing each s_i by $[s_i \;\, D_{s_i}]$ and a by $[a \;\, s_1 \;\, D_{s_1}]$. The encoding is then with respect to a location context Γ mapping a to site s_1 and each D_{s_i} to site s_i. In an implementation the daemons could either be built into each runtime system or, more flexibly, could be migrated on demand from a daemon server, to any site that wished to participate in this infrastructure.

The simplest correctness result for this infrastructure would be that $@_b P$ is *observationally equivalent* to Q. Identifying the appropriate notion of observational equivalence for the extended calculus is not trivial, but appears to be feasible – a generalisation of [Sew97], in which this was considered for an idealised non-distributed Pict, may be required.

Acknowledgements The authors would like to thank Ken Moody for many stimulating discussions, and to acknowledge support from EPSRC grant GR/K 38403 and the Wolfson Foundation.

References

[CHK97] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems – Towards the Programmable Internet. LNCS 1222*, pages 25–48, 1997.

[FGL+96] C. Fournet, G. Gonthier, J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996. Available from `http://pauillac.inria.fr/~fournet/`.

[MGW97] D. S. Milojicic, S. Guday, and R. Wheeler. Old wine in new bottles: Applying OS process migration technology to mobile agents. In *ECOOP Workshop on Mobile Object Systems 97*, June 1997. To appear in LNCS. Available from `http://cuiwww.unige.ch/~ecoopws/ws97/abstracts.html`.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[PT97] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press. Available from `http://www.cs.indiana.edu/hyplan/pierce/pierce/ftp/`, 1997.

[Sew97] P. Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer-Verlag, 1997. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98. LNCS*, July 1998. To appear. Technical report available from `http://www.cl.cam.ac.uk/users/pes20/`.