Location-Independent Communication for Mobile Agents: a Two-Level Architecture

Peter Sewell[1] **Paweł T. Wojciechowski**[1]
Benjamin C. Pierce[2]

Abstract. We study communication primitives for interaction between mobile agents. They can be classified into two groups. At a low level there are *location dependent* primitives that require a programmer to know the current site of a mobile agent in order to communicate with it. At a high level there are *location independent* primitives that allow communication with a mobile agent irrespective of its current site and of any migrations. Implementation of these requires delicate distributed infrastructure. We propose a simple calculus of agents that allows implementations of such distributed infrastructure algorithms to be expressed as encodings, or compilations, of the whole calculus into the fragment with only location dependent communication. These encodings give executable descriptions of the algorithms, providing a clean implementation strategy for prototype languages. The calculus is equipped with a precise semantics, providing a solid basis for understanding the algorithms and for reasoning about their correctness and robustness. Two sample infrastructure algorithms are presented as encodings.

Table of Contents

[1] Computer Laboratory, University of Cambridge.
`{Peter.Sewell,Pawel.Wojciechowski}@cl.cam.ac.uk`
[2] Dept. of Computer & Information Science, University of Pennsylvania.
`bcpierce@cis.upenn.edu`

1 Introduction

Recent years have seen an explosion of interest in wide-area distributed applications, executing on intranets or on the global internet. A key concept for structuring such applications is *mobile agents*, units of executing code that can migrate between sites [CHK97]. Mobile agent programming requires novel forms of language and runtime support—for interaction between agents, responding to network failures and reconfigurations, binding to resources, managing security, etc. In this paper we focus on the first of these, considering the design, semantic definition, and implementation of communication primitives by which mobile agents can interact.

Mobile agent communication primitives can be classified into two groups. At a low level, there are *location dependent* primitives that require a 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 communicating program must explicitly track its new location. At a high level, there are *location independent* primitives that allow communication with a mobile agent irrespective of its current site and of any migrations of sender or receiver. Location independent primitives may greatly simplify the development of mobile applications, since they allow movement and interaction to be treated as separate concerns. Their design and implementation, however, raise several difficult issues. A distributed infrastructure is required for tracking migrations and routing messages to migrating agents. This infrastructure must address fundamental network issues such as failures, network latency, locality, and concurrency; the algorithms involved are thus inherently rather delicate and cannot provide perfect location independence. Moreover, applications may be distributed on widely different scales (from local to wide-area networks), may exhibit different patterns of communication and migration, and may demand different levels of performance and robustness; these varying demands will lead to a multiplicity of infrastructures, based on a variety of algorithms. These *infrastructure algorithms* will be exposed, via their performance and behaviour under failure, to the application programmer — some detailed understanding of an algorithm will be required for the programmer to understand its robustness properties under, for example, failure of a site.

The need for clear understanding and easy experimentation with infrastructure algorithms, as well as the desire to simultaneously support multiple infrastructures on the same network, suggests a two-level architecture—a low-level consisting of a single set of well-understood, location-dependent primitives, in terms of which a variety of high-level, location-independent communication abstractions may be expressed. This two-level approach enables one to have a standardized low-level runtime that is common to many machines, with divergent high-level facilities chosen and installed at run time. It also facilitates simple implementation of the location-independent primitives (cf. protocol stacks).

For this approach to be realistic, it is essential that the low-level primitives should be directly implementable above standard network protocols. The Internet Protocol (IP) supports asynchronous, unordered, point-to-point, unreliable packet delivery; it abstracts from routing. We choose primitives that are directly implementable using asynchronous, unordered, point-to-point, reliable messages. This abstracts away from a multitude of additional details—error correction, retransmission, packet fragmentation, etc.—while still retaining a clear relationship to the well-understood IP level. It also well suited to the process calculus presentation that we use below. More controversially, we also include agent migration among the low-level primitives. This requires substantial runtime support in individual network sites, but not sophisticated distributed algorithms—only one message need be sent per migration. By treating it as a low-level primitive we focus attention more sharply on the distributed algorithms supporting location-independent communication. We also provide low-level primitives for agent creation, for sending messages between agents at the same site, for generating globally unique names, and for local computation.

Many forms of high-level communication can be implemented in terms of these low-level primitives, for example synchronous and asynchronous message passing, remote procedure calls, multicasting to agent groups, etc. For this paper we consider only a single representative form: an asynchronous message-passing primitive similar to the low-level primitive for communication between co-located agents but independent of their locations and transparent to migrations.

This two-level framework can be formulated very cleanly using techniques from the theory of process calculi. Such a formulation permits a precise definition of both low and high levels, and allows distributed infrastructure algorithms to be treated rigorously as translations between calculi. The operational semantics of the calculi provides a precise and clear understanding of the algorithms' behaviour, aiding design, and ultimately, one may hope, supporting proofs of correctness and robustness. Our presentation draws on ideas first developed in Milner, Parrow, and Walker's π-calculus [MPW92,Mil92] and extended in the distributed join-calculus of Fournet et al [FGL$^+$96].

To facilitate experimentation, the *Nomadic Pict* project is implementing prototype mobile agent programming languages corresponding to our high- and low-level process calculi. The low-level language extends the compiler and run-time system of *Pict* [PT97,Tur96], a concurrent language based on the π-calculus, to support our primitives for agent creation, migration, and location-dependent communication. High-level languages, with particular infrastructures for location-independent communication, can then be obtained by applying user-supplied translations into the low-level language. In both cases, the full language available to the user remains very close to the process calculus presentation, and can be given rigorous semantics in a similar style. Analogous extensions could be given for other concurrent uniprocessor programming languages, such as Amber [Car86], Concurrent ML [Rep91], and Concurrent Haskell [JGF96].

In the next section we introduce the two calculi informally, discussing our primitives in detail and giving examples of common programming idioms. In §3 and §4 we then present two sample infrastructure algorithms — one using a centralised server and one using chains of forwarding pointers — as illustrations of the use of the calculi. The operational semantics of the calculi are defined precisely in §5, in terms of a reduction semantics. We conclude with some further discussion of related work, implementation, and future extensions. The paper develops ideas first presented in [SWP98] — that work introduced a slightly different calculus, using it to describe the forwarding-pointers infrastructure.

2 The Calculi

In this section our two levels of abstraction are made precise by giving two corresponding process calculi, the low- and high-level *Nomadic π-calculi*. Their design involves a delicate trade-off — the distributed infrastructure algorithms that we want to express involve non-trivial local computation within agents, yet for the theory to be tractable (particularly, for operational congruences to have tractable characterisations) the calculi must be kept as simple as possible. The primitives for agent creation, agent migration and inter-agent communication that we consider do not suffice to allow the required local computation to be expressed clearly, so we integrate them with those of an asynchronous π-calculus [HT91,Bou92]. The other computational constructs that will be needed, e.g. for finite maps, can then be regarded as lightweight syntactic sugar for π-processes.

The low- and high-level calculi are introduced in §2.1 and §2.2 respectively, followed by some examples and programming idioms in §2.3. In this section the operational semantics of the calculi are described informally — the precise reduction semantics will be given in §5. For simplicity, the calculi are presented without typing or basic values (such as integers and booleans). Type systems are briefly discussed in §6.3.

2.1 Low-Level Calculus

We begin with an example. Below is a term of the low-level calculus showing how an applet server can be expressed. It can receive (on the channel named *getApplet*) requests for an applet; the requests contain a pair (bound to a and s) consisting of the name of the requesting agent and the name of its site.

$$\begin{array}{l} *\mathit{getApplet}?(a\ s) \rightarrow \\ \quad \textbf{agent}\ b = \\ \quad\quad \textbf{migrate to}\ s \rightarrow (\langle a@s\rangle \mathit{ack}!b \mid B) \\ \quad \textbf{in} \\ \quad\quad 0 \end{array}$$

When a request is received the server creates an applet agent with a new name bound to b. This agent immediately migrates to site s. It then sends an acknowledgement to the requesting agent a (which is assumed to also be on site s) containing its name. In parallel, the body B of the applet commences execution.

The example illustrates the main entities represented in the calculus: sites, agents and channels. *Sites* should be thought of as physical machines or, more accurately, as instantiations of the Nomadic Pict runtime system on machines; each site has a unique name. This paper does not explicitly address questions of site failure, network failure and reconfiguration, or security. Sites are therefore unstructured; neither network topology nor administrative domains are represented in the formalism. *Agents* are units of executing code; an agent has a unique name and a body consisting of some term; at any moment it is located at a particular site. *Channels* support communication within agents, and also provide targets for inter-agent communication—an inter-agent message will be sent to a particular channel within the destination agent. Channels also have unique names.

The inter-agent message $\langle a@s\rangle ack!b$ is characteristic of the low-level calculus. It is location-dependent—if agent a is in fact on site s then the message b will be delivered, to channel ack in a; otherwise the message will be discarded. In an implementation at most one inter-site message is sent.

Names As in the π-calculus, names play a key rôle. We take an infinite set $\mathcal{N}$ of names, ranged over by a, b, c, s, and x. Formally, all names are treated identically; informally, a and b will be used for agent names, c for a channel name, and s for a site name. (A type system would allow these distinctions to be enforced.) The calculus allows new names (of agents and channels) to be created dynamically.

Names are *pure*, in the sense of Needham [Nee89]; they are not assumed to contain any information about their creation. They can therefore be implemented by any mechanism that allows globally-unique bit strings to be created locally, e.g. by appending sequence numbers to IP addresses, or by choosing large random numbers.

Values We allow the communication of first-order values, consisting of names and tuples.

$u, v ::= x$	name
$\quad [v_1 \mathinner{..} v_n]$	tuple ($n \geq 0$)

Patterns As is the π-calculus, values are deconstructed by pattern matching on input. Patterns have the same form as values, with the addition of a wildcard.

$p ::= \_$	wildcard
$\quad x$	name pattern
$\quad (p_1 \mathinner{..} p_n)$	tuple pattern ($n \geq 0$, no repeated names)

Process terms The main syntactic category is that of *process terms*, ranged over by P, Q. We will introduce the low-level primitives in groups.

agent $a = P$ **in** Q	agent creation
migrate to $s \rightarrow P$	agent migration

The execution of the construct **agent** $a = P$ **in** Q spawns a new agent on the current site, with body P. After the creation, Q commences execution, in parallel with the rest of the body of the spawning agent. The new agent has a unique name which may be referred to both in its body and in the spawning agent (i.e.

a is binding in P and Q). Agents can migrate to named sites — the execution of **migrate to** $s \rightarrow P$ as part of an agent results in the whole agent migrating to site s. After the migration, P commences execution in parallel with the rest of the body of the agent.

$P \mid Q$	parallel composition
0	nil

The body of an agent may consist of many process terms in parallel, i.e. essentially of many lightweight threads. They will interact only by message passing.

new c **in** P	new channel name creation
$c!v$	output v on channel c in the current agent
$c?p \rightarrow P$	input from channel c
$*c?p \rightarrow P$	replicated input from channel c
if $u = v$ **then** P **else** Q	value equality testing

To express computation within an agent, while keeping a lightweight semantics, we include π-calculus-style interaction primitives. Execution of **new** c **in** P creates a new unique channel name; c is binding in P. An output $c!v$ (of value v on channel c) and an input $c?p \rightarrow P$ in the same agent may synchronise, resulting in P with the names in the pattern p replaced by corresponding parts of v. A replicated input $*c?p \rightarrow P$ behaves similarly except that it persists after the synchronisation, and so may receive another value. In both $c?p \rightarrow P$ and $*c?p \rightarrow P$ the names in p are binding in P. The conditional allows any two values to be tested for equality.

iflocal $\langle a \rangle c!v \rightarrow P$ **else** Q	test-and-send to agent a on current site

Finally, the low-level calculus includes a single primitive for interaction between agents. The execution of **iflocal** $\langle a \rangle c!v \rightarrow P$ **else** Q in the body of an agent b has two possible outcomes. If agent a is on the same site as b, then the message $c!v$ will be delivered to a (where it may later interact with an input) and P will commence execution in parallel with the rest of the body of b; otherwise the message will be discarded, and Q will execute as part of b. The construct is analogous to test-and-set operations in shared memory systems — delivering the message and starting P, or discarding it and starting Q, atomically. It can greatly simplify algorithms that involve communication with agents that may migrate away at any time, yet is still implementable locally, by the runtime system on each site.

As in the π-calculus, names can be *scope-extruded* — here channel and agent names can be sent outside the agent in which they were created. For example, if the body of agent a is

$$\begin{array}{l} \textbf{agent } b = \\ \quad \textbf{new } d \textbf{ in} \\ \quad\quad \textbf{iflocal } \langle a \rangle c!d \rightarrow 0 \textbf{ else } 0 \\ \textbf{in} \\ \quad c?x \rightarrow x! \end{array}$$

then channel name d is created in agent b. After the output message $c!d$ has been sent from b to a (by **iflocal**) and has interacted with the input $c?x \to x!$ there will be an output $d!$ in agent a.

We require a clear relationship between the semantics of the low-level calculus and the inter-machine messages that would be sent in an implementation. To achieve this we allow communication between outputs and inputs on a channel only if they are *in the same agent* — messages can be sent from one agent to another only by **iflocal**. Intuitively, there is a distinct π-calculus-style channel for each channel name in every agent. For example, if the body of agent a is

$$\begin{array}{l}
\textbf{agent } b = \\
\quad \textbf{new } d \textbf{ in} \\
\qquad\quad d? \to 0 \\
\qquad \mid \textbf{iflocal } \langle a \rangle c!d \to 0 \textbf{ else } 0 \\
\textbf{in} \\
\quad c?x \to x!
\end{array}$$

then after some reduction steps a contains an output on d and b contains an input on d, but these cannot react. At first sight this semantics may seem counter-intuitive, but it reconciles the conflicting requirements of expressiveness and simplicity of the calculus. An implementation would create the mailbox datastructure — a queue of pending outputs or inputs — required to implement a channel as required; it could be garbage collected when empty.

Summarizing, the terms of the low-level calculus are:

$P, Q ::=$	$\textbf{agent } a = P \textbf{ in } Q$	agent creation
	$\textbf{migrate to } s \to P$	agent migration
	$P \mid Q$	parallel composition
	0	nil
	$\textbf{new } c \textbf{ in } P$	new channel name creation
	$c!v$	output v on channel c in the current agent
	$c?p \to P$	input from channel c
	$*c?p \to P$	replicated input from channel c
	$\textbf{if } u = v \textbf{ then } P \textbf{ else } Q$	value equality testing
	$\textbf{iflocal } \langle a \rangle c!v \to P \textbf{ else } Q$	test-and-send to agent a on current site

Note that the only primitive which involves network communication is **migrate**, which requires only a single message to be sent, asynchronously, between machines. Distributed implementation of the low-level calculus is therefore straightforward, requiring no non-trivial distributed algorithms. It could be done either above a reliable datagram layer or above TCP, using a lightweight layer that opens and closes streams as required.

Two other forms of location-dependent output will be useful in writing encodings, and are expressible in the calculus given.

$\langle a \rangle c!v$	output to agent a on the current site
$\langle a@s \rangle c!v$	output to agent a on site s

The execution of an output $\langle a \rangle c!v$ in the body of an agent b will either deliver the message $c!v$ to agent a, if agent b is on the same site as a, or will silently discard the message, if not. The execution of an output $\langle a@s \rangle c!v$ in the body of an agent will either deliver the message $c!v$ to agent a, if agent a is on site s, or will silently discard the message, if not. We regard these as syntactic sugar for

$$\textbf{iflocal}\ \langle a \rangle c!v \rightarrow 0\ \textbf{else}\ 0$$

and

$$\textbf{agent}\ b = (\textbf{migrate to}\ s \rightarrow (\textbf{iflocal}\ \langle a \rangle c!v \rightarrow 0\ \textbf{else}\ 0))\ \textbf{in}\ 0$$

(where b is fresh) respectively. In an implementation, the first is implementable locally; the second requires only one asynchronous network message. Note that one could optimize the case in which the second is used on site s itself by trying **iflocal** first:

$$\begin{array}{l}
\textbf{iflocal}\ \langle a \rangle c!v \rightarrow \\
\quad 0 \\
\textbf{else} \\
\quad \textbf{agent}\ b = (\textbf{migrate to}\ s \rightarrow (\textbf{iflocal}\ \langle a \rangle c!v \rightarrow 0\ \textbf{else}\ 0))\ \textbf{in}\ 0
\end{array}$$

2.2 High-Level Calculus

The high-level calculus is obtained by extending the low-level calculus with a single location-independent communication primitive:

$\langle a@? \rangle c!v$	location-independent output to agent a

The intended semantics of an output $\langle a@? \rangle c!v$ is that its execution will reliably deliver the message $c!v$ to agent a, irrespective of the current site of a and of any migrations.

2.3 Examples and Idioms

We give some syntactic sugar and programming idioms that will be used in the translations. Most are standard π-calculus idioms; some involve distributed communication.

Syntactic sugar Empty tuples and tuple patterns will generally be elided, writing $c!$ and $c? \rightarrow P$ for $c![]$ and $c?() \rightarrow P$. Multiple new channel bindings will be coalesced, writing **new** c, c' **in** P for **new** c **in new** c' **in** P. Let-declarations will be used, writing **let** $p = v$ **in** P for **new** c **in** $c!v \mid c?p \rightarrow P$ (where c is a name not occurring free in v or P).

Procedures Within a single agent one can express 'procedures' as simple replicated inputs. Below is a first attempt at a pair-server, that receives values x on channel *pair* and returns two copies of x on channel *result*, together with a single invocation of the server.

$$\begin{array}{l}\textbf{new}\ \mathit{pair}, \mathit{result}\ \textbf{in} \\ \quad\ \ *\mathit{pair}?x \rightarrow \mathit{result}![x\ x] \\ \quad \mid \mathit{pair}!v \\ \quad \mid \mathit{result}?z \rightarrow \ldots z \ldots\end{array}$$

This pair-server can only be invoked sequentially—there is no association between multiple requests and their corresponding results. A better idiom is below, in which new result channels are used for each invocation.

$$\begin{array}{l}\textbf{new}\ \mathit{pair}\ \textbf{in} \\ \quad\ \ *\mathit{pair}?(x\ r) \rightarrow r![x\ x] \\ \quad \mid \textbf{new}\ \mathit{result}\ \textbf{in}\ \mathit{pair}![v\ \mathit{result}] \mid \mathit{result}?z \rightarrow \ldots z \ldots \\ \quad \mid \textbf{new}\ \mathit{result}\ \textbf{in}\ \mathit{pair}![w\ \mathit{result}] \mid \mathit{result}?z \rightarrow \ldots z \ldots\end{array}$$

The example can easily be lifted to remote procedure calls between agents. We show two versions, firstly for location-dependent RPC between static agents and secondly for location-independent RPC between agents that may be migrating. In the first, the server becomes

$$\begin{array}{l}\textbf{new}\ \mathit{pair}\ \textbf{in} \\ \quad *\mathit{pair}?(x\ r\ b\ s) \rightarrow \langle b@s\rangle r![x\ x]\end{array}$$

which returns the result using location-dependent communication to the agent b on site s received in the request. If the server is part of agent a_1 on site s_1 it would be invoked from agent a_2 on site s_2 by

$$\begin{array}{l}\textbf{new}\ \mathit{result}\ \textbf{in} \\ \quad\ \ \langle a_1@s_1\rangle \mathit{pair}![v\ \mathit{result}\ a_2\ s_2] \\ \quad \mid \mathit{result}?z \rightarrow \ldots z \ldots\end{array}$$

If agents a_1 or a_2 can migrate this can fail. A more robust idiom is easily expressible in the high-level calculus—the server becomes

$$\begin{array}{l}\textbf{new}\ \mathit{pair}\ \textbf{in} \\ \quad *\mathit{pair}?(x\ r\ b) \rightarrow \langle b@?\rangle r![x\ x]\end{array}$$

which returns the result using location-independent communication to the agent b. If the server is part of agent a_1 it would be invoked from agent a_2 by

$$\begin{array}{l}\textbf{new}\ \mathit{result}\ \textbf{in} \\ \quad\ \ \langle a_1@?\rangle \mathit{pair}![v\ \mathit{result}\ a_2] \\ \quad \mid \mathit{result}?z \rightarrow \ldots z \ldots\end{array}$$

Locks, methods and objects An agent consisting of a parallel composition of replicated inputs, such as

$$\begin{array}{l} \quad *method1?arg \rightarrow \ldots \\ \mid *method2?arg \rightarrow \ldots \end{array}$$

is analogous to an object with methods *method1* and *method2*. Mutual exclusion between the bodies of the methods can be enforced by using a lock channel:

$$\begin{array}{l} \textbf{new } lock \textbf{ in} \\ \quad\quad lock! \\ \quad \mid *method1?arg \rightarrow \\ \quad\quad\quad lock? \rightarrow \\ \quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad lock! \\ \quad \mid *method2?arg \rightarrow \\ \quad\quad\quad lock? \rightarrow \\ \quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad lock! \end{array}$$

Here the lock is free if there is an output on channel *lock* and not free otherwise. State that is shared between the methods can be conveniently kept as the value of the output on the lock channel:

$$\begin{array}{l} \textbf{new } lock \textbf{ in} \\ \quad\quad lock!initialState \\ \quad \mid *method1?arg \rightarrow \\ \quad\quad\quad lock?state \rightarrow \\ \quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad lock!state' \\ \quad \mid *method2?arg \rightarrow \\ \quad\quad\quad lock?state \rightarrow \\ \quad\quad\quad\quad \ldots \\ \quad\quad\quad\quad lock!state'' \end{array}$$

For more detailed discussion of object representations in process calculi, the reader is referred to [PT94].

Finite maps The algorithms given in the following two sections involve finite maps — in the first, there is a daemon maintaining a map from agent names to site names; in the second, there are daemons maintaining maps from agent names to lock channels. The translations make use of the following constructs:

$c!\textbf{emptymap}$	output the empty map on channel c
$\textbf{lookup } a \textbf{ in } m \textbf{ with}$ $\quad \textbf{found}(p) \rightarrow P$ $\quad \textbf{notfound} \rightarrow Q$	look up a in map m
$\textbf{let } m' = (m \textbf{ with } a \mapsto v) \textbf{ in } P$	add a new binding

Our calculi are sufficiently expressive to allow these to be expressed directly, in a standard π-calculus style — we regard the constructs as syntactic sugar for the three process terms below. In the second and third the names x, *found*, and *notfound* are assumed not to occur free in P, Q, or a.

$$
\begin{array}{lll}
c!\mathbf{emptymap} & \stackrel{def}{=} & \mathbf{new}\ m\ \mathbf{in} \\
 & & \quad c!m \\
 & & \mid *m?(x\,\mathit{found}\ \mathit{notfound}) \to \mathit{notfound}! \\[1em]
\mathbf{lookup}\ldots & \stackrel{def}{=} & \mathbf{new}\ \mathit{found},\ \mathit{notfound}\ \mathbf{in} \\
 & & \quad m![a\,\mathit{found}\ \mathit{notfound}] \\
 & & \mid \mathit{found}?p \to P \\
 & & \mid \mathit{notfound}? \to Q \\[1em]
\mathbf{let}\ldots & \stackrel{def}{=} & \mathbf{new}\ m'\ \mathbf{in} \\
 & & \quad *m'?(x\,\mathit{found}\ \mathit{notfound}) \to \\
 & & \qquad \mathbf{if}\ x = a\ \mathbf{then} \\
 & & \qquad\quad \mathit{found}!v \\
 & & \qquad \mathbf{else} \\
 & & \qquad\quad m![x\,\mathit{found}\ \mathit{notfound}] \\
 & & \mid P
\end{array}
$$

These represent a finite map as a channel on which there is a process that receives lookup requests. Requests consist of a triple of a key and two result channels; the process returns a value on the first if the lookup succeeds, and otherwise signals on the second.

3 A Simple Infrastructure Translation

In this section and the following one we present two infrastructure algorithms, expressed as translations. The first is one of the simplest algorithms possible, highly sequential and with a centralized server daemon; the second is one step more sophisticated, with multiple daemons maintaining forwarding-pointer chains. The algorithms have been chosen to illustrate our approach, and the use of the calculi — algorithms that are widely applicable to actual mobile agent systems would have to be yet more delicate, both for efficiency and for robustness under partial failure. Even the simplest of our algorithms, however, requires delicate synchronization that (the authors can attest) is easy to get wrong; expressing them as translations between well-defined calculi provides a solid basis for discussion and algorithm design.

The algorithm presented in this section involves a central daemon that keeps track of the current sites of all agents and forwards any location-independent messages to them. The daemon is itself implemented as an agent which never migrates; the translation of a program then consists roughly of the daemon agent in parallel with a compositional translation of the program. For simplicity we consider only programs that are initiated as single agents, rather than many agents initiated separately on different sites. (Programs may, of course, begin by creating other agents that immediately migrate). The precise definition is given in Figures 1 and 2. Figure 2 defines a top-level translation $[\![\,]\!]$. For each term P of the high-level calculus, considered as the body of an agent named a and initiated at site s, the result $[\![P]\!]_{a,s}$ of the translation is a term of the low-level calculus. The definition of $[\![\,]\!]$ involves the body *Daemon* of the daemon agent and an auxiliary compositional translation $[\![P]\!]_a$, defined phrase-by-phrase, of P considered as part of the body of agent a. Both are given in Figure 1.

Let us look first at the daemon. It contains three replicated inputs, on the *register*, *migrating*, and *message* channels, for receiving messages from the encodings of agents. The daemon is essentially single-threaded — the channel *lock* is used to enforce mutual exclusion between the bodies of the replicated inputs, and the code preserves the invariant that at any time there is at most one output on *lock*. The *lock* channel is also used to maintain the site map — a finite map from agent names to site names, recording the current site of every agent. The body of each replicated input begins with an input on *lock*, thereby acquiring both the lock and the site map.

Turning to the compositional translation $[\![\_]\!]_a$, only three clauses are not trivial — for the location-independent output, agent creation, and agent migration primitives. We discuss each, together with their interactions with the daemon, in turn.

Location-independent output A location-independent output in an agent a is implemented simply by using a location-dependent output to send a request to the daemon D, at its site SD, on its channel *message*:

$$[\![\langle b@?\rangle c!v]\!]_a = \langle D@SD\rangle \mathit{message}![b\ c\ v]$$

The corresponding replicated input on channel *message* in the daemon

$$\begin{array}{l}
\mid *\mathit{message}?(a\ c\ v) \rightarrow \\
\quad \mathit{lock}?m \rightarrow \\
\quad \textbf{lookup}\ a\ \textbf{in}\ m\ \textbf{with} \\
\quad\quad \textbf{found}(s) \rightarrow \\
\quad\quad\quad\quad \langle a@s\rangle \mathit{deliver}![c\ v] \\
\quad\quad\quad \mid \mathit{dack}? \rightarrow \mathit{lock}!m \\
\quad\quad \textbf{notfound} \rightarrow 0
\end{array}$$

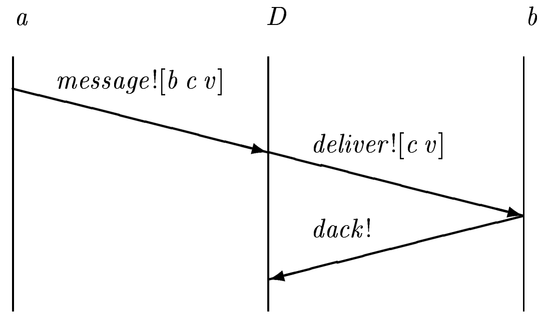
first acquires the lock and current site map m, then looks up the target agent's site in the map and sends a location-dependent message to the *deliver* channel of that agent. It then waits to receive an acknowledgement (on the *dack* channel) from the agent before relinquishing the lock. This prevents the agent migrating before the *deliver* message arrives. Note that the **notfound** branch of the lookup will never be taken, as the algorithm ensures that all agents register before messages can be sent to them. The inter-agent communications involved

$$
\begin{array}{ll}
[\![\langle b@?\rangle c!v]\!]_a & = \langle D@SD\rangle message![b\ c\ v] \\
[\![\textbf{agent}\ b = P\ \textbf{in}\ Q]\!]_a & = currentloc?s \rightarrow \\
& \quad \textbf{agent}\ b = \\
& \qquad *deliver?(c\ v) \rightarrow (\langle D@SD\rangle dack! \mid c!v) \\
& \qquad \mid \langle D@SD\rangle register![b\ s] \\
& \qquad \mid ack? \rightarrow (\langle a@s\rangle ack! \mid currentloc!s \mid [\![P]\!]_b) \\
& \quad \textbf{in} \\
& \qquad ack? \rightarrow (currentloc!s \mid [\![Q]\!]_a) \\
[\![\textbf{migrate to}\ s \rightarrow P]\!]_a & = currentloc?\_ \rightarrow \\
& \qquad \langle D@SD\rangle migrating!a \\
& \quad \mid ack? \rightarrow \\
& \qquad \textbf{migrate to}\ s \rightarrow \\
& \qquad\quad \langle D@SD\rangle migrated!s \\
& \qquad\quad \mid ack? \rightarrow (currentloc!s \mid [\![P]\!]_a) \\
[\![0]\!]_a & = 0 \\
[\![P \mid Q]\!]_a & = [\![P]\!]_a \mid [\![Q]\!]_a \\
[\![c?p \rightarrow P]\!]_a & = c?p \rightarrow [\![P]\!]_a \\
[\![*c?p \rightarrow P]\!]_a & = *c?p \rightarrow [\![P]\!]_a \\
[\![\textbf{iflocal}\ \langle b\rangle c!v \rightarrow P\ \textbf{else}\ Q]\!]_a & = \textbf{iflocal}\ \langle b\rangle c!v \rightarrow [\![P]\!]_a\ \textbf{else}\ [\![Q]\!]_a \\
[\![\textbf{new}\ c\ \textbf{in}\ P]\!]_a & = \textbf{new}\ c\ \textbf{in}\ [\![P]\!]_a \\
[\![\textbf{if}\ u = v\ \textbf{then}\ P\ \textbf{else}\ Q]\!]_a & = \textbf{if}\ u = v\ \textbf{then}\ [\![P]\!]_a\ \textbf{else}\ [\![Q]\!]_a
\end{array}
$$

$$
\begin{array}{l}
Daemon = \textbf{new}\ lock\ \textbf{in} \\
\qquad lock!\textbf{emptymap} \\
\quad \mid *register?(a\ s) \rightarrow \\
\qquad lock?m \rightarrow \\
\qquad \textbf{let}\ m' = (m\ \textbf{with}\ a \mapsto s)\ \textbf{in} \\
\qquad\quad lock!m' \mid \langle a@s\rangle ack! \\
\quad \mid *migrating?a \rightarrow \\
\qquad lock?m \rightarrow \\
\qquad \textbf{lookup}\ a\ \textbf{in}\ m\ \textbf{with} \\
\qquad\quad \textbf{found}(s) \rightarrow \\
\qquad\qquad \langle a@s\rangle ack! \\
\qquad\quad \mid migrated?s' \rightarrow \\
\qquad\qquad \textbf{let}\ m' = (m\ \textbf{with}\ a \mapsto s')\ \textbf{in} \\
\qquad\qquad lock!m' \mid \langle a@s'\rangle ack! \\
\qquad\quad \textbf{notfound} \rightarrow 0 \\
\quad \mid *message?(a\ c\ v) \rightarrow \\
\qquad lock?m \rightarrow \\
\qquad \textbf{lookup}\ a\ \textbf{in}\ m\ \textbf{with} \\
\qquad\quad \textbf{found}(s) \rightarrow \\
\qquad\qquad \langle a@s\rangle deliver![c\ v] \\
\qquad\quad \mid dack? \rightarrow lock!m \\
\qquad\quad \textbf{notfound} \rightarrow 0
\end{array}
$$

Fig. 1. A Simple Translation: the compositional translation and the daemon

in delivery of a single location-independent output are illustrated below.



Creation In order for the daemon's site map to be kept up to date, agents must register with the daemon, telling it their site, both when they are created and after they migrate. Each agent records its current site internally as an output on its *currentloc* channel. This channel is also used as a lock, to enforce mutual exclusion between the encodings of all agent creation and migration commands within the body of the agent.

The encoding of an agent creation in an agent a

$$
\begin{array}{ll}
[\![\mathbf{agent}\ b = P\ \mathbf{in}\ Q]\!]_a = & \mathit{currentloc}?s \rightarrow \\
& \mathbf{agent}\ b = \\
& \quad\ \ {*}\mathit{deliver}?(c\ v) \rightarrow (\langle D@SD\rangle \mathit{dack}! \mid c!v) \\
& \quad \mid \langle D@SD\rangle \mathit{register}![b\ s] \\
& \quad \mid \mathit{ack}? \rightarrow (\langle a@s\rangle \mathit{ack}! \mid \mathit{currentloc}!s \mid [\![P]\!]_b) \\
& \mathbf{in} \\
& \quad \mathit{ack}? \rightarrow (\mathit{currentloc}!s \mid [\![Q]\!]_a)
\end{array}
$$

first acquires the lock and current site s of a, and then creates the new agent b. The body of b sends a *register* message to the daemon and waits for an acknowledgement. It then sends an acknowledgement to a, initializes the lock for b and allows the encoding of the body P of b to proceed. Meanwhile, in a the lock is kept until the acknowledgement from b is received. The body of b is put in parallel with the replicated input

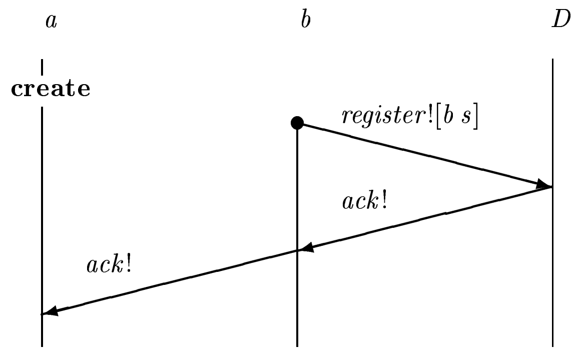
$$
{*}\mathit{deliver}?(c\ v) \rightarrow (\langle D@SD\rangle \mathit{dack}! \mid c!v)
$$

which will receive forwarded messages for channels in b from the daemon, send an acknowledgement back, and deliver the value locally to the appropriate channel.

The replicated input on *register* in the daemon

$$
\begin{array}{l}
\mid {*}\mathit{register}?(a\ s) \rightarrow \\
\quad\ \mathit{lock}?m \rightarrow \\
\quad\ \mathbf{let}\ m' = (m\ \mathbf{with}\ a \mapsto s)\ \mathbf{in} \\
\quad\quad \mathit{lock}!m' \mid \langle a@s\rangle \mathit{ack}!
\end{array}
$$

first acquires the lock and current site map, replaces the site map with an updated map, thereby relinquishing the lock, and sends an acknowledgement to the registering agent. The inter-agent communications involved in a single agent creation are illustrated below.



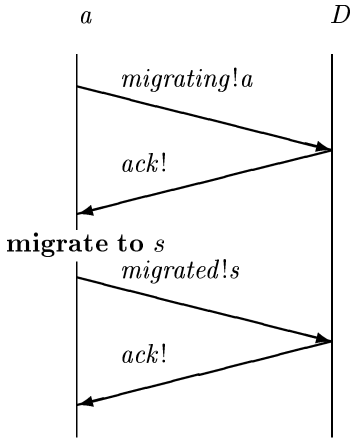
Migration The encoding of a **migrate** in agent a

$$
\begin{aligned}
[\![\textbf{migrate to } s \to P]\!]_a = {} & \mathit{currentloc}?\_ \to \\
& \quad \langle D@SD\rangle \mathit{migrating}!a \\
& \quad \mid \mathit{ack}? \to \\
& \qquad \textbf{migrate to } s \to \\
& \qquad\quad \langle D@SD\rangle \mathit{migrated}!s \\
& \qquad\quad \mid \mathit{ack}? \to (\mathit{currentloc}!s \mid [\![P]\!]_a)
\end{aligned}
$$

first acquires the lock for a (discarding the current site data). It then sends a *migrating* message to the daemon, waits for an *ack*, migrates to its new site s, sends a *migrated* message to the daemon, waits again for an *ack*, and releases the lock (with the new site s). The replicated input on *migrating* in the daemon

$$
\begin{aligned}
& \mid *\mathit{migrating}?a \to \\
& \quad \mathit{lock}?m \to \\
& \quad \textbf{lookup } a \textbf{ in } m \textbf{ with} \\
& \qquad \textbf{found}(s) \to \\
& \qquad\quad \langle a@s\rangle \mathit{ack}! \\
& \qquad\quad \mid \mathit{migrated}?s' \to \\
& \qquad\qquad \textbf{let } m' = (m \textbf{ with } a \mapsto s') \textbf{ in} \\
& \qquad\qquad \mathit{lock}!m' \mid \langle a@s'\rangle \mathit{ack}! \\
& \qquad \textbf{notfound} \to 0
\end{aligned}
$$

first acquires the lock and current site map, looks up the current site of a and sends an *ack* to a at that site. It then waits to receive the new site, replaces the site map with an updated map, thereby relinquishing the lock, and sends an acknowledgement to a at its new site. The inter-agent communications involved in a single migration are shown below.



The top level Putting the daemon and the compositional encoding together, the top level translation, defined in Figure 2, creates the daemon agent, installs the

$$
\begin{array}{l}
[\![P]\!]_{a,s} = \textbf{new } \mathit{register}, \mathit{migrating}, \mathit{migrated}, \mathit{message}, \mathit{dack}, \mathit{deliver}, \mathit{ack}, \mathit{currentloc} \textbf{ in} \\
\qquad\qquad \textbf{agent } D = \mathit{Daemon} \textbf{ in} \\
\qquad\qquad\quad \textbf{let } SD = s \textbf{ in} \\
\qquad\qquad\qquad\quad * \mathit{deliver}?(c\ v) \rightarrow (\langle D@SD \rangle \mathit{dack}! \mid c!v) \\
\qquad\qquad\qquad \mid \langle D@SD \rangle \mathit{register}![a\ s] \\
\qquad\qquad\qquad \mid \mathit{ack}? \rightarrow (\mathit{currentloc}!s \mid [\![P]\!]_a)
\end{array}
$$

where the **new**-bound names, SD, and D, do not occur in P.

Fig. 2. A Simple Translation: the top level

replicated input on *deliver* for a, registers agent a to be at site s, initializes the lock for a, and starts the encoding of the body $[\![P]\!]_a$.

4 A Forwarding-Pointers Infrastructure Translation

In this section we give a more distributed algorithm, in which daemons on each site maintain chains of forwarding pointers for agents that have migrated. It removes the single bottleneck of the centralised-server solution in the preceding section; it is thus a step closer to algorithms that may be of wide practical use. The algorithm is more delicate; expressing it as a translation provides a more rigorous test of the framework.

As before, the translation consists of a compositional encoding of the bodies of agents, given in Figure 3, daemons, defined in Figure 4, and a top-level translation putting them together, given in Figure 5. The top-level translation of a program, again initially a single agent, creates a daemon on each site mentioned by the agent. These will each maintain a collection of forwarding pointers for all agents that have migrated away from their site. To keep the pointers current, agents synchronize with their local daemons on creation and migration. 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. A typed version of the encoding would involve a translation of types with clauses

$$\begin{aligned} [\![\mathbf{Agent}]\!] &= [\mathbf{Agent\ Site\ Agent}] \\ [\![\mathbf{Site}]\!] &= [\mathbf{Site\ Agent}] \end{aligned}$$

We generally use lower case letters for site and agent names occurring in the source program and upper case letters for sites and agents introduced by its encoding.

Looking first at the compositional encoding, in Figure 3, each agent uses a *currentloc* channel as a lock, as before. It is now also used to store both the site where the agent is and the name of the daemon on that site. The three interesting clauses of the encoding, for location-independent output, creation, and migration, each begin with an input on *currentloc*. They are broadly similar to those of the simple translation.

Turning to the body of a daemon, defined in Figure 4, it is parametric in a pair s of the name of the site S where it is and the daemon's own name DS. It has four replicated inputs, on its *register*, *migrating*, *migrated*, and *message* channels. Some partial mutual exclusion between the bodies of these inputs is enforced by using the *lock* channel. The data stored on the *lock* channel now maps the name of each agent that has ever been on this site to a lock channel (e.g. *Bstate*) for that agent. These agent locks prevent the daemon from attempting to forward messages to agents that may be migrating. Each stores the site and daemon (of that site) where the agent was last seen by this daemon — i.e. either this site/daemon, or the site/daemon to which it migrated to from here. The use of agent locks makes this algorithm rather more concurrent than the previous one — rather than simply sequentialising the entire daemon, it allows daemons to process inputs while agents are migrating, so many agents can be migrating away from the same site, concurrently with each other and with delivery of messages to other agents at the site.

Location-independent output A location-independent output $\langle b@?\rangle c!v$ in agent A is implemented by requesting the local daemon to deliver it. (Note that A may migrate away before the request is sent to the daemon, so the request must be of the form $\langle DS@S\rangle message![b\ c\ v]$, not of the form $\langle DS\rangle message![b\ c\ v]$.)

The *message* replicated input of the daemon gets the map m from agent names to agent lock channels. If the destination agent B is not found, the message

$$
\begin{array}{ll}
\llbracket \langle b@?\rangle c!v \rrbracket_A & = \mathit{currentloc}?(S\ DS) \rightarrow \\
& \qquad \langle DS@S\rangle \mathit{message}![b\ c\ v] \\
& \qquad \mid \mathit{currentloc}![S\ DS] \\
\llbracket \textbf{agent}\ b = P\ \textbf{in}\ Q \rrbracket_A & = \mathit{currentloc}?(S\ DS) \rightarrow \\
& \quad \textbf{agent}\ B = \\
& \qquad \textbf{let}\ b = [B\ S\ DS]\ \textbf{in} \\
& \qquad\quad \mathit{currentloc}![S\ DS] \\
& \qquad\quad \mid \langle DS\rangle \mathit{register}!B \\
& \qquad\quad \mid \mathit{ack}? \rightarrow (\langle A@S\rangle \mathit{ack}! \mid \llbracket P \rrbracket_B) \\
& \quad \textbf{in} \\
& \qquad \textbf{let}\ b = [B\ S\ DS]\ \textbf{in} \\
& \qquad \mathit{ack}? \rightarrow (\mathit{currentloc}![S\ DS] \mid \llbracket Q \rrbracket_A) \\
\llbracket \textbf{migrate to}\ u \rightarrow P \rrbracket_A & = \mathit{currentloc}?(S\ DS) \rightarrow \\
& \textbf{let}\ (U\ DU) = u\ \textbf{in} \\
& \quad \textbf{if}\ [S\ DS] = [U\ DU]\ \textbf{then} \\
& \qquad (\mathit{currentloc}![U\ DU] \mid \llbracket P \rrbracket_A) \\
& \quad \textbf{else} \\
& \qquad\quad \langle DS\rangle \mathit{migrating}!A \\
& \qquad \mid \mathit{ack}? \rightarrow \\
& \qquad\qquad \textbf{migrate to}\ U \rightarrow \\
& \qquad\qquad\qquad \langle DU\rangle \mathit{register}!A \\
& \qquad\qquad\quad \mid \mathit{ack}? \rightarrow \\
& \qquad\qquad\qquad\quad \langle DS@S\rangle \mathit{migrated}![A\ [U\ DU]] \\
& \qquad\qquad\qquad \mid \mathit{ack}? \rightarrow (\mathit{currentloc}![U\ DU] \mid \llbracket P \rrbracket_A) \\
\llbracket \textbf{iflocal}\ \langle b\rangle c!v \rightarrow P\ \textbf{else}\ Q \rrbracket_A & = \textbf{let}\ (B\ \_\ \_) = b\ \textbf{in} \\
& \quad \textbf{iflocal}\ \langle B\rangle c!v \rightarrow \llbracket P \rrbracket_A\ \textbf{else}\ \llbracket Q \rrbracket_A \\
\llbracket 0 \rrbracket_A & = 0 \\
\llbracket P \mid Q \rrbracket_A & = \llbracket P \rrbracket_A \mid \llbracket Q \rrbracket_A \\
\llbracket c?p \rightarrow P \rrbracket_A & = c?p \rightarrow \llbracket P \rrbracket_A \\
\llbracket *c?p \rightarrow P \rrbracket_A & = *c?p \rightarrow \llbracket P \rrbracket_A \\
\llbracket \textbf{new}\ c\ \textbf{in}\ P \rrbracket_A & = \textbf{new}\ c\ \textbf{in}\ \llbracket P \rrbracket_A \\
\llbracket \textbf{if}\ u = v\ \textbf{then}\ P\ \textbf{else}\ Q \rrbracket_A & = \textbf{if}\ u = v\ \textbf{then}\ \llbracket P \rrbracket_a\ \textbf{else}\ \llbracket Q \rrbracket_A
\end{array}
$$

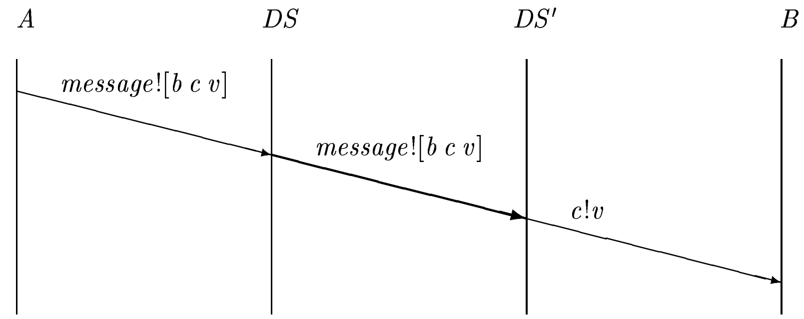
Fig. 3. A Forwarding-Pointers Translation: the compositional translation

```
Daemon_s =  let (S DS) = s in
            new lock in
              lock!emptymap
            | *register?B → lock?m → lookup B in m with
                  found(Bstate) →
                    Bstate?(_ _) →
                      Bstate![S DS]
                    | lock!m
                    | ⟨B⟩ack!
                  notfound →
                    new Bstate in
                      Bstate![S DS]
                    | let m' = (m with B ↦ Bstate) in lock!m'
                    | ⟨B⟩ack!
            | *migrating?B → lock?m → lookup B in m with
                  found(Bstate) →
                    Bstate?(_ _) →
                      lock!m
                    | ⟨B⟩ack!
                  notfound → 0
            | *migrated?(B (U DU)) → lock?m → lookup B in m with
                found(Bstate) →
                    lock!m
                  | Bstate![U DU]
                  | ⟨B@U⟩ack!
                notfound → 0
            | *message?((B U DU) c v) → lock?m → lookup B in m with
                  found(Bstate) →
                      lock!m
                    | Bstate?(R DR) →
                        iflocal ⟨B⟩c!v →
                          Bstate![R DR]
                        else
                            ⟨DR@R⟩message![[B U DU] c v]
                          | Bstate![R DR]
                  notfound →
                      lock!m
                    | ⟨DU@U⟩message![[B U DU] c v]
```

Fig. 4. A Forwarding-Pointers Translation: the Daemon

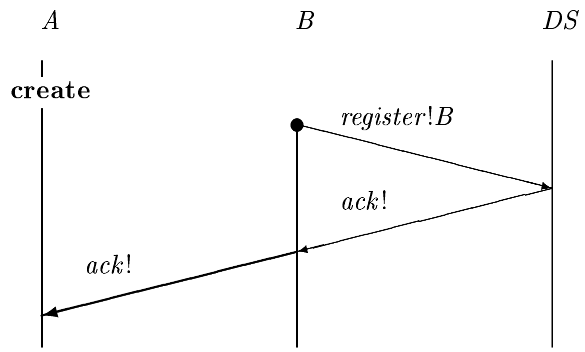
is forwarded to the daemon DU on the site U where B was created. Otherwise, if B is found, the agent lock *Bstate* is grabbed, obtaining the forwarding pointer $[R\ DR]$ for B. Using **iflocal**, the message is then either delivered to B, if it is here, or to the daemon DR, otherwise. Note that the *lock* is released before the agent lock is requested, so the daemon can process other inputs even if B is currently migrating.

A single location-independent output, forwarded once between daemons, involves inter-agent messages as below. (Communications that are guaranteed to be between agents on the same site are drawn with thin arrows.)



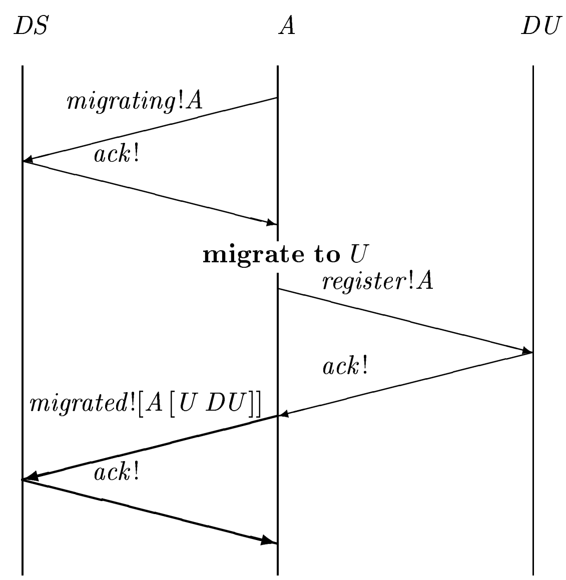
Creation The compositional encoding for **agent** is similar to that of the encoding in the previous section. It differs in two main ways. Firstly the source language name b of the new agent must be replaced by the actual agent name B tupled with the names S of this site and DS of the daemon on this site. Secondly, the internal forwarder, receiving on *deliver*, is no longer required: the final delivery of messages from daemons to agents is now always local to a site, and so can be done using **iflocal**. An explicit acknowledgement (on *dack* in the simple translation) is likewise unnecessary.

A single creation involves inter-agent messages as below.



Migration Degenerate migrations, of an agent to the site it is currently on, must now be identified and treated specially; otherwise the Daemon can deadlock. An agent A executing a non-degenerate migration now synchronises with the daemon DS on its starting site S, then migrates, registers with the daemon DU on its destination site U, then synchronises again with DS. In between the first and last synchronisations the agent lock for A in daemon DS is held, preventing DS from attempting to deliver messages to A.

A single migration involves inter-agent messages as below.



Local communication The translation of **iflocal** must now extract the real agent name B from the triple b, but is otherwise trivial.

The top level The top-level translation of a program P, given in Figure 5, dynamically creates a daemon on each site mentioned in P. Each site name si is re-bound to the pair $[si\, DSi]$ of the site name together with the respective daemon name. A top-level agent A is created and initialised; the agent name a is re-bound to the triple $[A\, S1\, DS1]$ of the low-level agent name A together with the initial site and daemon names.

5 Reduction Semantics

The informal descriptions of the primitives in §2 can be made precise by giving them an operational semantics. We adopt a *reduction semantics*, defining the atomic state-changes that a system of agents can undergo by reduction axioms with a structural congruence, following the style of [BB92,Mil92].

The process terms of the calculi in §2.1,2.2 only allow the source code of the body of a single agent to be expressed. During computation, this agent may evolve into a system of many agents, distributed over many sites. The reduction relation must be between the possible states of these systems, not simply between terms of the source calculi; we express such states as *configurations* Γ, P. Here

$$
\begin{array}{l}
[\![P]\!]_{a,s1..sn} = \textbf{new } \mathit{register}, \mathit{migrating}, \mathit{migrated}, \mathit{message}, \mathit{ack}, \mathit{currentloc}, \mathit{lock}, \\
\qquad\qquad\qquad \mathit{daemondaemon}, \mathit{nd} \textbf{ in} \\
\qquad\qquad\qquad\quad *\mathit{daemondaemon}?S \rightarrow \\
\qquad\qquad\qquad\qquad \textbf{agent } D = \\
\qquad\qquad\qquad\qquad\quad \textbf{migrate to } S \rightarrow (\mathit{Daemon}_{[S\ D]} \mid \langle a@s1\rangle \mathit{nd}![S\ D]) \\
\qquad\qquad\qquad\qquad \textbf{in } 0 \\
\qquad\qquad\qquad \mid \mathit{daemondaemon}!s1 \mid \mathit{nd}?s1 \rightarrow \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad\quad \mathit{daemondaemon}!sn \mid \mathit{nd}?sn \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \textbf{let } (S1\, DS1) = s1 \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{agent } A = \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let } a = [A\, S1\, DS1] \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{currentloc}!s1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \langle DS1\rangle \mathit{register}!A \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{ack}? \rightarrow [\![P]\!]_A \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } 0
\end{array}
$$

where P is initiated on site $s1$, the free site names in P are $s1..sn$, and the **new**-bound names, $S1$, $DS1$, and A do not occur in P.

Fig. 5. A Forwarding-Pointers Translation: the top level

Γ is a *location context* that gives the current site of any free agent names; P is a term of the (low- or high-level) calculus extended with two new forms.

$@_a P$	P as part of agent a
new $a@s$ **in** P	new agent name a, currently at site s

Configurations may involve many agents in parallel. The form $@_a P$ denotes the process term P as part of the body of agent a, so for example $@_a P \mid @_b Q$ denotes P as part of the body of a in parallel with Q as part of the body of b. It will be convenient to allow the parts of the body of an agent to be syntactically separated, so e.g. $@_a P_1 \mid @_b Q \mid @_a P_2$ denotes $P_1 \mid P_2$ as part of a in parallel with Q as part of b. Configurations must record the current sites of all agents. For free agent names this is done by the location context Γ; for the others, the form **new** $a@s$ **in** P declares a new agent name a, which is binding in P, and records that agent a is currently at site s.

We now give the detailed definitions. Process terms are taken up to alpha-conversion throughout. Structural congruence $\equiv$ includes the axiom

$$@_a (P \mid Q) \equiv @_a P \mid @_a Q$$

allowing the parts of an agent a to be syntactically separated or brought together, and the axiom

$$@_a \textbf{ new } c \textbf{ in } P \equiv \textbf{new } c \textbf{ in } @_a P \qquad \text{if } c \neq a$$

allowing channel binders to be extruded past $@_a$. It is otherwise similar to a standard structural congruence for an asynchronous π-calculus, with scope extrusion both for the new channel binder **new** c **in** P and for the new agent binder **new** $a@s$ **in** P. In full, it is the least congruence satisfying the following axioms.

$$\begin{array}{rcll}
P & \equiv & P \mid 0 & \\
P \mid Q & \equiv & Q \mid P & \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \\
P \mid \mathbf{new}\ c\ \mathbf{in}\ Q & \equiv & \mathbf{new}\ c\ \mathbf{in}\ P \mid Q & \text{if } c \text{ not free in } P \\
P \mid \mathbf{new}\ a@s\ \mathbf{in}\ Q & \equiv & \mathbf{new}\ a@s\ \mathbf{in}\ P \mid Q & \text{if } a \text{ not free in } P \\
@_a (P \mid Q) & \equiv & @_a P \mid @_a Q & \\
@_a\ \mathbf{new}\ c\ \mathbf{in}\ P & \equiv & \mathbf{new}\ c\ \mathbf{in}\ @_a P & \text{if } c \neq a
\end{array}$$

A configuration is a pair Γ, P, where the location context Γ is a finite partial function from $\mathcal{N}$ to $\mathcal{N}$, intuitively giving the current site of any free agent names in P, and P is a term of the (low- or high-level) extended calculus. The initial configuration, for a program P of the (low- or high-level) unextended calculus, to be considered as the body of an agent a created on site s, is:

$$\{a \mapsto s\},\ @_a P$$

We are concerned only with configurations that can arise by reduction of initial configurations for well-typed programs. In these, any particle (i.e., **agent**, **migrate**, output, input, **if**, or **iflocal**) will be under exactly one @ operator, specifying the agent that contains it. (In this paper we do not give a type system, and so leave this informal.) Other configurations have mathematically well-defined reductions but may not be easily implementable or desirable, for example

$$\Gamma,\ @_a (c?b \rightarrow @_b P)$$

receives an agent name and then adds P to the body of that agent.

We define a partial function match, taking a value and a pattern and giving (where it is defined) a finite substitution from names to values.

$$\begin{array}{rcl}
\text{match}(v, _) & = & \{\} \\
\text{match}(v, x) & = & \{x \mapsto v\} \\
\text{match}([v_1 \mathinner{..} v_m], (p_1 \mathinner{..} p_m)) & = & \text{match}(v_1, p_1) \cup \ldots \cup \text{match}(v_m, p_m) \\
\text{match}(v, (p_1 \mathinner{..} p_m)) & & \text{undefined, if } v \text{ is not of the form } [v_1 \mathinner{..} v_m]
\end{array}$$

The natural definition of the application of a substitution from names to values to a process term P is also a partial operation, as the syntax does not allow arbitrary values in all the places where free names can occur. We write $\{v/p\}P$ for the result of applying the substitution match(v, p) to P. This may be undefined either because match(v, p) is undefined, or because match(v, p) is a substitution but the application of that substitution to P is undefined.

The reduction axioms for the low-level calculus are as follows.

$$
\begin{array}{lll}
\Gamma, @_a \textbf{agent } b = P \textbf{ in } Q & \longrightarrow \Gamma, \textbf{new } b@\Gamma(a) \textbf{ in } (@_b P | @_a Q) & \\
\Gamma, @_a \textbf{migrate to } s \to P & \longrightarrow (\Gamma \oplus a \mapsto s), @_a P & \\
\Gamma, @_a \textbf{iflocal } \langle b \rangle c!v \to P \textbf{ else } Q & \longrightarrow \Gamma, @_b c!v | @_a P & \text{if } \Gamma(a) = \Gamma(b) \\
 & \longrightarrow \Gamma, @_a Q & \text{if } \Gamma(a) \neq \Gamma(b) \\
\Gamma, @_a (c!v | c?p \to P) & \longrightarrow \Gamma, @_a \{v/p\}P & \\
\Gamma, @_a (c!v | *c?p \to P) & \longrightarrow \Gamma, @_a (\{v/p\}P | *c?p \to P) & \\
\Gamma, @_a \textbf{if } u = v \textbf{ then } P \textbf{ else } Q & \longrightarrow \Gamma, @_a P & \text{if } u = v \\
 & \longrightarrow \Gamma, @_a Q & \text{if } u \neq v
\end{array}
$$

The rules mentioning potentially-undefined expressions $\Gamma(x)$ or $\{v/p\}P$ in their side-condition or conclusion have an implicit additional premise that these are defined. Such premises should be automatically satisfied in derivations of reductions of well-typed programs.

Note that the only inter-site communication in an implementation will be for the **migrate** reduction, in which the body of the migrating agent a must be sent from its current site to site s.

The high-level calculus has the additional axiom below, for delivering location-independent messages to their destination agent.

$$\Gamma, @_a \langle b@? \rangle c!v \longrightarrow \Gamma, @_b c!v$$

Reduction is closed under structural congruence, parallel, **new** c **in** _ and **new** $a@s$ **in** _, as specified by the rules below.

$$\frac{Q \equiv P \quad \Gamma, P \longrightarrow \Gamma', P' \quad P' \equiv Q'}{\Gamma, Q \longrightarrow \Gamma', Q'} \qquad \frac{\Gamma, P \longrightarrow \Gamma', P'}{\Gamma, P \mid Q \longrightarrow \Gamma', P' \mid Q}$$

$$\frac{(\Gamma, a \mapsto s), P \longrightarrow (\Gamma, a \mapsto s'), P'}{\Gamma, \textbf{new } a@s \textbf{ in } P \longrightarrow \Gamma', \textbf{new } a@s' \textbf{ in } P'} \qquad \frac{\Gamma, P \longrightarrow \Gamma', P' \quad c \notin \mathrm{dom}(\Gamma)}{\Gamma, \textbf{new } c \textbf{ in } P \longrightarrow \Gamma', \textbf{new } c \textbf{ in } P'}$$

6 Discussion

We conclude by discussing alternative approaches for the description of mobile agent infrastructures, related distributed process calculi, implementation, and future work.

6.1 Infrastructure Description

In this paper we have identified two levels of abstraction, precisely formulated them as process calculi, and argued that distributed infrastructure algorithms for mobile agents can usefully be expressed as translations between the calculi. Such translations should be compared with the many other possible ways of describing the algorithms — we briefly consider diagrammatic, pseudocode, and automata based approaches.

The *diagrams* used in §3,4 convey basic information about the algorithms — the messages involved in isolated transactions — but they are far from complete descriptions and can be misleading. The correctness of the algorithms depends on details of synchronisation and locking that are precisely defined by the translation but are hard to express visually.

For a *psuedocode* description to provide a clear (if necessarily informal) description of an algorithm the constructs of the psuedocode must themselves have clear intuitive semantics. This may hold for psuedocodes based on widespread procedural languages, such as Pascal. Infrastructure algorithms, however, involve constructs for agent creation, migration and communication. These do not have a widespread, accepted, semantics — a number of rather different semantic choices are possible — so more rigorous descriptions are required for clear understanding.

Automata-based descriptions have been widely used for precise specification of distributed algorithms, for example in the text of Lynch [Lyn96]. Automata do not allow agent creation and migration to be represented directly, so for working with a mobile agent algorithm one would either have to use a complex encoding or consider only an abstraction of the algorithm — a non-executable model, rather than an executable complete description.

The modelling approach has been followed by Amadio and Prasad in their work on IP mobility [AP98]. They consider idealizations of protocols from IPv6 proposals for mobile host support, expressed in a variant of CCS, and prove correctness results. There is a trade-off here: the idealizations can be expressed in a simpler formal framework, greatly simplifying correctness proofs, but they are further removed from implementation, inevitably increasing the likelihood that important details have been abstracted away.

Few current proposals for mobile agent systems support any form of location-independence. Those that do include the Distributed Join Language [FGL$^+$96,Joi98], the MOA project of the Open Group Research Institute [MLC98], and the Voyager system of ObjectSpace [Obj97]. The distributed join language is at roughly the same level of abstraction as the high-level Nomadic π-calculus. It provides location-independent communication, with primitives similar to the outputs and replicated inputs used here. The MOA project associates a locating scheme to each agent; chosen from querying a particular site (updated on each migration), searching along a pre-defined itinerary, and following forwarding pointers. Voyager provides location-independent asynchronous and synchronous messages, and multicasts. Migrating objects leave trails of forwarders behind them; entities that communicate with these objects are sent updated addresses to be cached. Forwarders are garbage-collected; the garbage collection involves heartbeat messages. More precise descriptions of the algorithms used in these systems do not appear to have been published, making it difficult for the application programmer to predict their performance and robustness.

6.2 Related Calculi

In recent years a number of process calculi have been introduced in order to study some aspect of distributed and mobile agent computation. They include:

- The π_l calculus of Amadio and Prasad [AP94], for modelling the failure semantics of Facile [TLK96].
- The Distributed Join Calculus of Fournet et al [FGL$^+$96], intended as the basis for a mobile agent language.
- The language of located processes and the Dπ calculus of Riely and Hennessy, used to study the semantics of failure [RH97,RH98] and typing for control of resource use by mobile agents [HR98b,HR98a].
- The calculus of Sekiguchi and Yonezawa [SY97], used to study various primitives for code and data movement.
- The dpi calculus of Sewell [Sew97a,Sew98], used to study a subtyping system for locality enforcement of capabilities.
- The Ambient calculus of Cardelli and Gordon [CG98], used for modelling security domains.
- The Seal calculus of Vitek and Castagna [VC98], focussing on protection mechanisms including revocable capabilities.

There is a large design space of such calculi, with very different primitives being appropriate for different purposes, and with many semantic choices. A thorough comparison and discussion of the design space is beyond the scope of this paper — a brief discussion can be found in [Sew99]; here we highlight only some of the main design choices:

Hierarchy We have adopted a two-level hierarchy, of agents located on sites. One might consider tree-structured mobile agents with migration of subtrees, e.g. as in [FGL$^+$96]. The added expressiveness may be desirable from the programmer's point of view, but it requires somewhat more complex infrastructure algorithms — migrations of an agent can be caused by migrations of their parents — so we neglect it in the first instance.

Unique Naming The calculi of §2 ensure that agents have unique names, in contrast, for example, to the Ambients of [CG98]. Inter-agent messages are therefore guaranteed to have a unique destination.

Communication In earlier work [SWP98] the inter-agent communication primitives were separated from the channel primitives used for local computation. The inter-agent primitives were

$\langle a@?\rangle!v$	location-independent output of v to agent a
$\langle a@s\rangle!v$	location-dependent output
$?p \rightarrow P$	input at the current agent

These give a conceptually simpler model, with messages sent to agents rather than to channels at agents, but to allow encodings to be expressed it was necessary to add variants and local channels. This led to a rather large calculus and somewhat awkward encodings.

6.3 Implementation

In order to experiment with infrastructure algorithms, and with applications that use location-independent communication, we have implemented an experimental programming language, *Nomadic Pict*. The Nomadic Pict implementation is based on the Pict compiler of Pierce and Turner [PT97]. It is a two-level language, corresponding to the calculi presented in this paper. The low level extends Pict by providing direct support for agent creation, migration and location-dependent communication. The high level supports location-independent communication by applying translations — the compiler takes as input a program in the high-level language together with an encoding of each high-level primitive into the low-level language. It type-checks and applies the encoding; the resulting low-level intermediate code can be executed on a relatively straightforward distributed run-time system. The two encodings given have both been successfully type-checked and executed.

Typing In this paper the calculi have been presented without typing. The Nomadic Pict implementation inherits from Pict its rather expressive type system. For reasoning about infrastructure encodings a simple type system for the calculi would be desirable, with types

$$T ::= \mathit{Site} \mid \mathit{Agent} \mid \updownarrow T \mid [T \mathinner{..} T] \mid X \mid \exists X.T$$

for site and agent names, channels carrying values of type T, tuples, and existential polymorphism.

The calculi allow a channel name to escape the agent in which it is declared and be used subsequently both for input and output within other agents. The global/local typing of [Sew97a,Sew98] could be used to impose tighter disciplines on channels that are intended to be used only locally, preventing certain programming errors.

Input/Output and Traders Up to this point we have considered only communications that are internal to a distributed computation. External input and output primitives can be cleanly provided in the form of special agent names, so that from within the calculus inputs and outputs are treated exactly as other communications. For example, for console I/O one might have a fictitious console agent on each site, together with globally-known channel names *getchar* and *putchar*. Messages sent to these would be treated specially by the local run-time system, leading to idioms such as

$$\mathbf{new}\ a\ \mathbf{in}\ \langle \mathit{console}\rangle \mathit{putchar}![c\ a] \,|\, (a? \rightarrow P)$$

for synchronous output of a character c to the local console, and

$$\mathbf{new}\ a\ \mathbf{in}\ \langle \mathit{console}\rangle \mathit{getchar}!a \,|\, (a?x \rightarrow P)$$

for synchronous input of a character, to be bound to x, from the local console.

In realistic systems there will be a rich collection of input/output resources, differing from site to site, so agents may need to acquire resources dynamically. Moreover, in realistic systems agents will be initiated separately on many sites; if they are to interact some mechanism must be provided for them to acquire each other's names dynamically. To do this in a lexically-scoped manner we envisage each site maintaining a *trader*, a finite map from strings to values that supports registration and lookup of resources. Agents would typically obtain the trader name associated with a site at the same time as obtaining the site name. For traders to be type-sound a type Dynamic [ACPP91] is required.

6.4 Future Work

This paper provides only a starting point — much additional work is required on algorithms, semantics, and implementation.

- The choice of infrastructure algorithm(s) for a given application will depend strongly on many characteristics of the application and target network, especially on the expected statistical properties of communication and migration. In wide area applications, sophisticated distributed algorithms will be required, allowing for dynamic system reconfigurations such as adding new sites to the system, migrating parts of the distributed computation before shutting down some machines, tracing locations of different kinds of agents, and implementing tolerance of partial failures. The space of feasible algorithms and the trade-offs involved require detailed investigation.
- Turning to semantics, in order to state correctness properties (in the absence of failures) a theory of observational equivalence is required. Such a theory was developed for an idealised Pict in [Sew97b]; it must be generalized to the distributed setting and supported by coinductive proof techniques.
- Finally, to investigate the behaviour of infrastructure algorithms in practice, and to assess the usefulness of our high-level location-independent primitives in applications, the implementation must be developed to the point where it is possible to experiment with non-trivial applications.

The calculi of §2 make the unrealistic assumption that communications and sites are reliable. This is implausible, even for local area networks of moderate size, so usable infrastructure algorithms must be robust under some level of failure. To express such algorithms some notion of time must be introduced into the low-level calculus, to allow timeouts to be expressed, yet the semantics must be kept tractable, to allow robustness properties to be stated and proved.

One might also consider other high-level communication primitives, such as location-independent multicast, and agent primitives, such as tree-structured agents. More speculatively, the two levels of abstraction that we have identified may be a useful basis for work on security properties of mobile agent infrastructures — to consider whether a distributed infrastructure for mobile agents is secure one must first be able to define it precisely, and have a clear understanding of how it is distributed on actual machines.

Acknowledgements The authors would like to thank Ken Moody and Asis Unyapoth for discussions and comments. Sewell was supported by EPSRC grants GR/K 38403 and GR/L 62290, Wojciechowski by the Wolfson Foundation, and Pierce by Indiana University and by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects.*

References

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems,*, 13(2):237–268, April 1991.

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of* 14^{th} *FST and TCS Conference, FST-TCS'94. LNCS 880*, pages 205–216. Springer-Verlag, 1994.

[AP98] Roberto M. Amadio and Sanjiva Prasad. Modelling IP mobility. In *Proceedings of CONCUR '98: Concurrency Theory. LNCS 1466*, pages 301–316, September 1998.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[Car86] Luca Cardelli. Amber and the amber machine. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages, LNCS 242*, pages 21–70, 1986.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98, LNCS 1378*, pages 140–155, March 1998.

[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] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[HR98a] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Workshop on High-Level Concurrent Languages*, 1998. Full version as University of Sussex technical report CSTR 98/02.

[HR98b] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Workshop on Mobile Object Systems, (satellite of ECOOP '98)*, 1998. Full version as University of Sussex technical report CSTR 98/03.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of ECOOP '91, LNCS 512*, pages 133–147, July 1991.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of the* 23^{rd} *ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[Joi98] The join calculus language, 1998. Implementations available from `http://pauillac.inria.fr/join/unix/eng.htm`.

[Lyn96] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MLC98] D. S. Milojicic, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *USENIX COOTS '98, Santa Fe*, April 1998.

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

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

[Obj97] ObjectSpace. Voyager core technology user guide, version 2.0 beta 1. Available from `http://www.objectspace.com/`, 1997.

[PT94] Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, November 1994.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[Rep91] John Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation*, pages 293–259. SIGPLAN, ACM, June 1991.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of ICALP '97. LNCS 1256*, pages 471–481. Springer-Verlag, July 1997.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, January 1998.

[Sew97a] Peter Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, University of Cambridge, August 1997. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[Sew97b] Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405, 1997.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98, LNCS 1443*, pages 695–706, 1998.

[Sew99] Peter Sewell. A brief introduction to applied π, January 1999. Lecture notes for the Mathfit Instructional Meeting on Recent Advances in Semantics and Types for Concurrency: Theory and Practice, July 1998. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[SWP98] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages, Chicago*, May 1998.

[SY97] Tatsurou Sekiguchi and Akinori Yonezawa. A calculus with code mobility. In Howard Bowman and John Derrick, editors, *Formal Methods for Open Object-based Distributed Systems (Proceedings of FMOODS '97)*, pages 21–36. IFIP, Chapman and Hall, July 1997.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *Proceedings of CONCUR '96. LNCS 1119*, pages 278–298. Springer-Verlag, August 1996.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[VC98] Jan Vitek and Guiseppe Castagna. Towards a calculus of mobile computations. In *Workshop on Internet Programming Languages, Chicago*, May 1998.