Nomadic Pict: Correct Communication Infrastructure for Mobile Computation

Asis Unyapoth and Peter Sewell[*]
Computer Laboratory, University of Cambridge
{Asis.Unyapoth,Peter.Sewell}@cl.cam.ac.uk

Abstract

This paper addresses the design and verification of infrastructure for mobile computation. In particular, we study language primitives for communication 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 any migrations. Implementation of the high level requires delicate distributed infrastructure algorithms. In earlier work with Wojciechowski and Pierce we made the two levels precise as process calculi, allowing such algorithms to be expressed as encodings of the high level into the low level; we built NOMADIC PICT, a distributed programming language for experimenting with such encodings. In this paper we turn to semantics, giving a definition of the core language and proving correctness of an example infrastructure. This requires novel techniques: we develop equivalences that take migration into account, and reasoning principles for agents that are temporarily immobile (eg. waiting on a lock elsewhere in the system).

1 Introduction

Background: Mobility and Location Independence Mobile computations – units of executing computation that can migrate between machines – are predicted to be an important enabling technology for future distributed systems [Car97, CHK97]. To write applications involving mobility one would like high-level *location independent* (LI) communication facilities, allowing the parts of an application to interact without explicitly tracking each other's movements. Such primitives have been provided by several languages, including Facile [TLK96], and Distributed Join [FGL$^+$96]. Standard network technologies, however, directly support only location-dependent (LD) communication, so to provide location independence one needs a distributed infrastructure algorithm. The languages cited above have particular algorithms hard-coded into their implementations, but in the wide-area case this is problematic:

- infrastructure algorithm design must be application-specific — any given one will only have satisfactory performance for some range of migration and communication behaviour; it should be matched to the expected properties (and robustness and security demands) of applications, and of the underlying network;
- the algorithms needed are delicate and error-prone; they are hard to reason about.

To allow more flexibility, a wide-area programming language should provide a low-level abstraction that makes distribution and network communication clear; with higher levels – including location independence – expressed using the modularisation facilities of the language.

Nomadic π and Pict Following the above, in earlier work with Wojciechowski and Pierce [SWP98, Woj00], we designed *Nomadic Pict*, a distributed programming language intended as a vehicle for exploring infrastructure for mobility. It builds on the Pict language of Pierce and Turner [PT00], a concurrent, though not distributed, language based on the asynchronous π-calculus [MPW92]. Pict supports fine-grain concurrency and the communication of asynchronous messages. Low-level Nomadic Pict adds primitives for programming mobile computations: agent creation, migration of agents between sites, and communication of location-dependent asynchronous messages between agents. High-level Nomadic Pict adds location-independent communication; we can express an arbitrary infrastructure for implementing this as a user-defined translation into the low-level language. Mobility allows infrastructures to be deployed dynamically. The language has been implemented by Wojciechowski [Woj00, WS00], and used for prototyping a wide range of infrastructures, from a simple centralised-server solution to federated algorithms supporting disconnection, suited for different applications. Our earlier work showed how the two levels can be cleanly based on corresponding high- and low-level process calculi. The operational semantics of the calculi provided a clear informal understanding of the algorithms' behaviour, which aided our design work.

---

[*] Supported by a Royal Thai Government Scholarship and a Royal Society University Research Fellowship respectively.

Problem: Semantics and Verification Our focus here is on developing semantics and proof techniques to allow formal correctness proofs for such infrastructure algorithms. If systems involving location independence are widely deployed, the behaviour of these algorithms will be critical. They are highly concurrent – as we can attest, it is hard to ensure the absence of race conditions, deadlocks and other errors. The algorithms are small enough, though, to make verification plausible.

New semantic technology is required, going beyond earlier work on π-calculi and distributed algorithms, both to deal with the new entities – sites and mobile agents – and to capture the subtle reasons why the algorithms are correct. This technology is worth developing in its own right: it is a step towards a semantically-founded view of richer wide-area distributed systems, where one wants proofs of robustness properties in the presence of failure and malicious attack.

Outline In this paper we give the rigorous development of a fragment of Nomadic Pict, with semantics and proof techniques, that suffices for verification of an example algorithm. For lack of space many details are omitted; they will appear in the first author's forthcoming PhD thesis. We begin in Sections 2 and 3 by introducing the language and a simple example infrastructure algorithm. This recapitulates material from [SWP98], adding a type system. In Section 4 we discuss the operational semantics, again building on [SWP98] (which gave only an untyped reduction semantics) and in Section 5 the techniques required for stating and proving correctness. We must:

1. extend the standard π-calculus reduction and labelled transition semantics to deal with agent mobility, location-dependent communication, and a rich type system;
2. consider *translocating* versions of behavioural equivalences (bisimulation [MPW92] and expansion [SM92] relations) that are preserved by certain spontaneous migrations;
3. prove congruence properties of some of these, to allow compositional reasoning;
4. deal with partially-committed choices, and hence state the main correctness result in terms of *coupled simulation* [PS92];
5. identify properties of agents that are *temporarily immobile*, waiting on a lock somewhere in the system; and,
6. as we are proving correctness of an encoding, we must analyse the possible reachable states of the encoding applied to an arbitrary high-level source program – introducing an intermediate language for expressing the key states, and factoring out as many 'house-keeping' reduction steps as possible.

A correctness proof for our example is given in Section 6. Finally, Section 7 concludes with further discussion.

2 Language

In this section we describe the language informally, beginning with an example program in the low-level language showing how an applet server can be expressed.

$$\begin{array}{l}
\mathbf{*}\mathit{getApplet}\,\mathbf{?}[a\ s] \rightarrow\\
\quad \mathbf{create}^{\mathtt{m}}\ b =\\
\quad\quad \mathbf{migrate\ to}\ s \rightarrow\\
\quad\quad\quad (\langle a@s'\rangle \mathit{ack}\,\mathbf{!}b \mid B)\\
\quad \mathbf{in}\ \mathbf{0}
\end{array}$$

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 the site for the applet to go to. 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 be on site s') containing its name. In parallel, the body B of the applet commences execution.

The example illustrates the main entities of the language: 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 network failure and reconfiguration, or of security. Sites are therefore unstructured; neither network topology nor administrative domains are represented in the language. *Agents* are units of executing code; an agent has a unique name and a body consisting of some Nomadic Pict process; 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. New agents and channels can be created dynamically. The language is built above asynchronous messaging, both within and between sites; in the current implementation inter-site messages are sent on TCP connections, created on demand, but our algorithms do not depend on the message ordering that could be provided by TCP.

The inter-agent message $\langle a@s'\rangle \mathit{ack}\,\mathbf{!}b$ is characteristic of the low-level language. 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 the implementation at most one inter-site message is sent.

Types Typing infrastructure algorithms requires an expressive type system. We take types

T ::=	B	base type
	$\mid [T_1 \ldots T_n]$	tuple
	$\mid \hat{}^{I}T$	channel name
	$\mid \{X\}\,T$	existential
	$\mid X$	type variable
	$\mid$ `Site`	site name
	$\mid$ $\mathtt{Agent}^{Z}$	agent name

where B might be `int`, `bool` etc. Existentials are needed as an infrastructure must be able to forward messages of any type (see the `message` and `deliver` channels in Fig. 1). For more precise typing, and to aid reasoning, channel and agent types are refined by annotating them with *capabilities*, ranged over by I and Z respectively. As in [PS96], channels can be used for input only `r`, output only `w`, or both `rw`; these induce a subtype order. In addition, agents are either static `s`, or mobile `m` [Sew98, CGG99].

Values and patterns Channels allow the communication of first-order values: names, constants t, tuples and existential packages. Patterns p are of similar shapes as value.

$$
\begin{array}{rcl}
v & ::= & t \mid x \mid [v_1 \dots v_n] \mid \{T\}\, v \\
p & ::= & \_ \mid x \mid [p_1 \dots p_n] \mid \{X\}\, p
\end{array}
$$

The value grammar is extended with some basic functions, including equality tests, to give *expressions*, ranged over by ev.

Processes The syntax of the low-level language is as follows.

$$
\begin{array}{ll}
P ::= & \mathbf{create}^{Z}\ a = P\ \mathbf{in}\ Q \\
& \mid \mathbf{migrate\ to}\ s \rightarrow P \\
& \mid \mathbf{iflocal}\ \langle a\rangle c\,\mathbf{!}\,v\ \mathbf{then}\ P\ \mathbf{else}\ Q \\
& \left.\begin{array}{l}
\mid \mathbf{0} \mid P|Q \mid \mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ P \\
\mid c\,\mathbf{!}\,v \mid c\,\mathbf{?}\,p \rightarrow P \mid \mathbf{*}c\,\mathbf{?}\,p \rightarrow P \\
\mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid \mathbf{let}\ p = ev\ \mathbf{in}\ P
\end{array}\right\}\pi \\
\\
& \mid \langle a\rangle c\,\mathbf{!}\,v \mid \langle a@s\rangle c\,\mathbf{!}\,v \qquad\qquad \text{sugar}
\end{array}
$$

Executing the construct $\mathbf{create}^{Z}\ b = P\ \mathbf{in}\ Q$ spawns a new agent (with mobility capability Z and body P) on the current site. After the creation, Q commences execution, in parallel with the rest of 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 (b is binding in P and Q). Agents can migrate to named sites—the execution of $\mathbf{migrate\ to}\ s \rightarrow P$ 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.

The body of an agent may consist of many process terms in parallel, i.e. essentially of many threads. We include π-calculus style interaction primitives. Execution of $\mathbf{new}\ c : \hat{}^{I}T\ \mathbf{in}\ P$ creates a new unique channel name for carrying values of type T (and accessible in I mode); c is binding in P. An output $c\,\mathbf{!}\,v$ (of value v on channel c) and an input $c\,\mathbf{?}\,p \rightarrow P$ in the same agent may synchronise, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. Note that outputs do not have continuation processes – this is an asynchronous calculus. A replicated input $\mathbf{*}c\,\mathbf{?}\,p \rightarrow P$ behaves similarly except that it persists after the synchronisation, and so might receive another value. In $c\,\mathbf{?}\,p \rightarrow P$, $\mathbf{*}c\,\mathbf{?}\,p \rightarrow P$ and $\mathbf{let}\ p = ev\ \mathbf{in}\ P$ the names in p are binding in P.

Finally, the low-level calculus includes a single primitive for interaction between agents. The execution of $\mathbf{iflocal}\ \langle a\rangle c\,\mathbf{!}\,v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ in the body of agent b has two possible outcomes. If the agent a is on the same site as agent b then the message $c\,\mathbf{!}\,v$ 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 not be delivered 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 systems on each site. We can express two other useful constructs in the language introduced so far: $\langle a\rangle c\,\mathbf{!}\,v$ and $\langle a@s\rangle c\,\mathbf{!}\,v$ attempt to deliver $c\,\mathbf{!}\,v$ to agent a, on the current site and on s, respectively. They fail silently if a is not where it is expected to be and so are usually used only where a is predictable.

In the execution of **iflocal** a new channel name can escape the agent where it was created, to be used elsewhere for output and/or input. Synchronisation of a local output $c\,\mathbf{!}\,v$ and an input $c\,\mathbf{?}\,x \rightarrow P$ only occurs within an agent, however. Consider for example the process below, executing as the body of an agent a.

$$
\begin{array}{l}
\mathbf{create}^{\mathtt{m}}\ b = \\
\quad c\,\mathbf{?}\,x \rightarrow (x\,\mathbf{!}\,3 | x\,\mathbf{?}\,n \rightarrow \mathbf{0}) \\
\mathbf{in} \\
\quad \mathbf{new}\ d : \hat{}^{\mathtt{rw}}\mathtt{int}\ \mathbf{in} \\
\qquad \mathbf{iflocal}\ \langle b\rangle c\,\mathbf{!}\,d\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ \mathbf{0} \\
\qquad | d\,\mathbf{!}\,7
\end{array}
$$

It has a reduction for the creation of agent b, a reduction for the **iflocal** that delivers the output $c\,\mathbf{!}\,d$ to b, and then a local synchronisation of this output with the input on c. Agent a then has body $d\,\mathbf{!}\,7$ and agent b has body $d\,\mathbf{!}\,3 | d\,\mathbf{?}\,n \rightarrow \mathbf{0}$. Only the latter output on d can synchronise with b's input $d\,\mathbf{?}\,n \rightarrow \mathbf{0}$. For each channel name there is therefore effectively a π-calculus-style channel in each agent. The channels are distinct, in that outputs and inputs can only interact if they are in the same agent. At first sight this semantics may seem counter-intuitive, but it reconciles the conflicting requirements of expressiveness and simplicity of the calculus.

The high-level language

$$
\begin{array}{rcl}
P & ::= & \dots \\
& & \langle a@?\rangle c\,\mathbf{!}\,v
\end{array}
$$

is obtained by extending the low-level language with a single location-independent communication primitive $\langle a@?\rangle c\,\mathbf{!}\,v$, whose intended semantics is that its execution will reliably deliver the message $c\,\mathbf{!}\,v$ to agent a, irrespective of the current site of a and of any migrations. The low-level communication primitives are also available for interaction with application agents whose locations are predictable.

Located Processes The basic process terms given above 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. To denote such systems, we define *located processes*

$$
LP ::= @_a P \mid LP|LQ \mid \mathbf{new}\ x : T@s\ \mathbf{in}\ LP
$$

Here the body of an agent a may be split into many parts, for example written $@_a P_1 | \dots | @_a P_n$. The construct $\mathbf{new}\ x : T@s\ \mathbf{in}\ LP$ declares a new name x (binding in LP); if this is an agent name, with $T = \mathtt{Agent}^{Z}$, we have an annotation $@s$ giving the name s of the site where the agent is currently located. Channels, on the other hand, are not located – if $T = \hat{}^{I}T'$ then the annotation is omitted.

3 An Example Infrastructure

In this section we present an infrastructure algorithm, expressed as a translation, based on the simplest algorithm

```
Daemon  ≝    *message? {X} [a c v] →
               lock?m →
                 lookup[Agent^s Site] a in m with
                   found(s) → new dack : ^rw[] in ⟨a@s⟩deliver! {X} [c v dack] | dack?[] → lock!m
                   notfound → 0
             | *register?[b s rack] →
               lock?m → let[Agent^s Site] m′ = (m with b ↦ s) in lock!m′ | ⟨b@s⟩rack![]
             | *migrating?[a mack] →
               lock?m →
                 lookup[Agent^s Site] a in m with
                   found(s) → new migrated : ^rw[Site ^w[]] in
                     ⟨a@s⟩mack![migrated]
                     | migrated?[s′ ack] → let m′ = (m with a ↦ s′) in lock!m′ | ⟨a@s′⟩ack![]
                   notfound → 0

Φ_aux   ≝   register : ^rw[Agent^s Site ^w[]],   migrating : ^rw[Agent^s ^w[^w[Site ^w[]]]],
            message : ^rw {X} [Agent^s ^wX X],   deliver : ^rw {X} [^wX X ^w[]],
            D : Agent^s@SD,   lock : ^rwMap[Agent^s Site],   currentloc : ^rwSite
```

Figure 1: The Central Server Daemon and the Interface Context

from [SWP98]. It is a central-forwarding-server algorithm, with a single daemon that keeps track of the current sites of all agents and forwards any location-independent messages to them. The original algorithm has been modified in several ways to simplify the correctness proof:

- type annotations have been added and checked with the Nomadic Pict type checker [Woj00] (although this does not check the static/mobile subtyping);
- the algorithm is more serialised;
- fresh channels are used for transmitting acknowledgements, making such channels linear [KPT96]; and
- the translation is extended to arbitrary located processes (not just source programs containing a single agent).

The daemon is itself implemented as a static agent; the translation $\mathcal{C}_\Phi[\![LP]\!]$ of a located process $LP =$ **new** Δ **in** $@_{a_1}P_1 \mid \ldots \mid @_{a_n}P_n$ (well-typed with respect to Φ) then consists roughly of the daemon agent in parallel with a compositional translation $[\![P_i]\!]_{a_i}$ of each source agent:

$$\begin{aligned}\mathcal{C}_\Phi[\![LP]\!] \stackrel{\text{def}}{=} \; & \textbf{new } \Delta, \Phi_{aux} \textbf{ in} \\ & @_D(\ldots|\mathit{Daemon}) \\ & \mid \textstyle\prod_{i\in\{1\ldots n\}} @_{a_i}(\ldots \mid [\![P_i]\!]_{a_i})\end{aligned}$$

(we omit various initialisation code, and will often elide type contexts Φ). The body of the daemon and selected clauses of the compositional translation are shown in Figures 1 and 2. They interact using channels of an *interface context* Φ_{aux}, also defined in Figure 1, which in addition declares lock channels and the daemon name D. It uses a map type constructor, which (together with the map operations) can be translated into the core language.

The daemon consists of three replicated inputs, on the `message`, `register`, and `migrating` channels, ready to receive messages from the encodings of agents. It is at a fixed site SD. Part of the initialisation code places *Daemon* in parallel with an output on `lock` which carries a reference to a *site map*: a finite map from agent names to site names, recording the current site of every agent. The single-threaded nature of the daemon is ensured by using `lock` to enforce mutual exclusion between the three replicated inputs – each of them begins with an input on `lock`, thereby acquiring both the lock and the site map, and does not relinquish the lock until the daemon finishes with the request. The code preserves the invariant that at any time there is at most one output on `lock`.

Turning to the compositional translation $[\![.]\!]$, it is defined by induction on type derivations.Only three clauses are non-trivial: for the location-independent output, agent creation and agent migration primitives. For the rest, $[\![.]\!]$ acts homomorphically. We discuss only LI output and creation; migration is similar.

Location-Independent Output An LI output in an agent a (of message $c!v$ to agent b) is implemented simply by using a location-dependent output $\langle b@?\rangle c!v$ to send the message to channel `message` at the daemon, as an existential package with a triple $[b\ c\ v]$. Reacting to this, the daemon acquires its lock and looks up b's site in the acquired site map. It then creates a fresh channel `dack` and forwards the message in LD mode (together with `dack`) to the `deliver` channel of agent b. In each agent the `deliver` channel is handled by a `Deliverer` process, as in Figure 2. This reacts to `deliver` messages by emitting a local $c!v$ message and acknowledging the daemon (again using LD communication) via `dack`. Meanwhile no agent may migrate before the `deliver` message arrives, since the daemon is single-threaded and waits for such an acknowledgement before releasing `lock`. Note that the **notfound** branch of the daemon's lookup will never be taken as the algorithm ensures that all agents register before messages can be sent to them.

Creation In order for the daemon's site map to be kept up to date, agents must register with the daemon, telling it

$$
\begin{array}{lcl}
\llbracket \langle b@?\rangle c\,\texttt{!}\,v \rrbracket_a & \overset{\text{def}}{=} & \langle D@SD\rangle \texttt{message!}\ \{T\}\ [b\ c\ v] \qquad \text{where } c : \texttt{\^{}}^{I}T \\[1em]
\llbracket \textbf{create}^{Z}\ b = P\ \textbf{in}\ Q \rrbracket_a & \overset{\text{def}}{=} & \texttt{currentloc?}s \to \textbf{new}\ \texttt{pack} : \texttt{\^{}}^{\texttt{rw}}[], \texttt{rack} : \texttt{\^{}}^{\texttt{rw}}[]\ \textbf{in} \\
 & & \quad \textbf{create}^{Z}\ b = \\
 & & \qquad \langle D@SD\rangle \texttt{register!}[b\ s\ \texttt{rack}] \\
 & & \qquad \mid \texttt{rack?}[] \to \\
 & & \qquad\quad \textbf{iflocal}\ \langle a\rangle \texttt{pack!}[]\ \textbf{then}\ \texttt{currentloc!}s \mid \llbracket P \rrbracket_b \mid \texttt{Deliverer}\ \textbf{else}\ \mathbf{0} \\
 & & \quad \textbf{in} \\
 & & \qquad \texttt{pack?}[] \to \big(\texttt{currentloc!}s \mid \llbracket Q \rrbracket_a\big) \\
\quad \text{where} \quad \texttt{Deliverer} & \overset{\text{def}}{=} & \texttt{*deliver?}\ \{X\}\ [c\ v\ \texttt{dack}] \to (\langle D@SD\rangle \texttt{dack!}[] \mid c\,\texttt{!}\,v) \\[1em]
\llbracket \textbf{migrate to}\ s \to P \rrbracket_a & \overset{\text{def}}{=} & \texttt{currentloc?\_} \to \textbf{new}\ \texttt{mack} : \texttt{\^{}}^{\texttt{rw}}[\texttt{\^{}}^{\texttt{w}}[\texttt{Site}\ \texttt{\^{}}^{\texttt{w}}[]]]\ \textbf{in} \\
 & & \quad \langle D@SD\rangle \texttt{migrating!}[a\ \texttt{mack}] \\
 & & \quad \mid \texttt{mack?}[\texttt{migrated}] \to \\
 & & \qquad \textbf{migrate to}\ s \to \\
 & & \qquad\quad \textbf{new}\ \texttt{ack} : \texttt{\^{}}^{\texttt{rw}}[]\ \textbf{in} \\
 & & \qquad\qquad \langle D@SD\rangle \texttt{migrated!}[s\ \texttt{ack}] \mid \texttt{ack?}[] \to \big(\texttt{currentloc!}s \mid \llbracket P \rrbracket_a\big) \\[1em]
\llbracket P|Q \rrbracket_a & \overset{\text{def}}{=} & \llbracket P \rrbracket_a \mid \llbracket Q \rrbracket_a
\end{array}
$$

Figure 2: The Compositional Encoding (selected clauses)

their site, both when they are created and when 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 (in Figure 2) first acquires the local lock and current site s and then creates the new agent b, as well as channels `pack` and `rack`. The body of b sends a `register` message to the daemon, supplying `rack`; the daemon uses `rack` to acknowledge that it has updated its site map. After the acknowledgement is received from the daemon, b sends an acknowledgement to a using `pack`, initialises the local lock of b with s, installs a `Deliverer`, and allows the encoding of the body P of b to proceed. Meanwhile, the local lock of a and the encoding of the continuation process Q are blocked until the acknowledgement via `pack` is received.

4 Semantic Definition

Type System We highlight only the non-standard aspects of the type system. Firstly, we work with *located type contexts* Γ, which include data specifying the site where each declared agent is located; the operational semantics updates this when agents move.

$$\Gamma ::= \bullet \mid \Gamma, X \mid \Gamma, x : \texttt{Agent}^{Z}@s \mid \Gamma, x : T \quad T \neq \texttt{Agent}^{Z}$$

A type context is *closed* if it declares no type variables or term variables of base type, and *extensible* if all term variables are of agent or channel types, and therefore may be new-bound.

The main judgements, for well-formedness of a basic process as part of agent a, and for located processes, have the forms $\Gamma \vdash_a P$ and $\Gamma \vdash LP$; there is also a judgement $\Gamma \vdash x@z$ giving the location z of x. Subtyping is as in [PS96] but with also $\texttt{Agent}^{\texttt{s}} \leq \texttt{Agent}^{\texttt{m}}$; there is a standard subsumption rule

$$\frac{\Gamma \vdash e \in S \quad \Gamma \vdash S \leq T}{\Gamma \vdash e \in T}$$

The most interesting rules are below.

$$\frac{\begin{array}{c}\Gamma \vdash a \in \texttt{Agent}^{\texttt{m}} \\ \Gamma \vdash s \in \texttt{Site} \\ \Gamma \vdash_a P\end{array}}{\Gamma \vdash_a \textbf{migrate to}\ s \to P} \qquad \frac{\begin{array}{c}a \neq b \\ \Gamma, b : \texttt{Agent}^{Z} \vdash_b P \\ \Gamma, b : \texttt{Agent}^{Z} \vdash_a Q\end{array}}{\Gamma \vdash_a \textbf{create}^{Z}\ b = P\ \textbf{in}\ Q}$$

$$\frac{\begin{array}{c}\Gamma \vdash a, b \in \texttt{Agent}^{\texttt{s}} \\ \Gamma \vdash s \in \texttt{Site} \\ \Gamma \vdash c \in \texttt{\^{}}^{\texttt{w}}T \\ \Gamma \vdash v \in T\end{array}}{\Gamma \vdash_a \langle b@s\rangle c\,\texttt{!}\,v} \qquad \frac{\Gamma \vdash_z P}{\Gamma \vdash @_z P}$$

Operational Semantics To capture our informal understanding of the language in as lightweight a way as possible, we give a reduction semantics. It is defined with a structural congruence and reduction axioms, extending that for the π-calculus [Mil93]. Reductions are over *configurations*, which are pairs $\Gamma \Vdash LP$ of a located typing context Γ and a located process LP. The most interesting axioms for the low-level language are given in Figure 3.

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

$$\Gamma \Vdash @_a \langle b@?\rangle c\,\texttt{!}\,v \quad \longrightarrow \quad \Gamma \Vdash @_b c\,\texttt{!}\,v$$

$$
\begin{array}{lll}
\Gamma \Vdash @_a\mathbf{create}^Z\ b = P\ \mathbf{in}\ Q & \longrightarrow & \Gamma \Vdash \mathbf{new}\ b : \mathtt{Agent}^Z@s\ \mathbf{in}\ (@_b P \mid @_a Q) \quad \text{where } \Gamma \vdash a@s \\
\Gamma \Vdash @_a\mathbf{migrate\ to}\ s \to P & \longrightarrow & (\Gamma \oplus a \mapsto s) \Vdash @_a P \\
\Gamma \Vdash @_a\ (c\mathbf{!}v | c\mathbf{?}p \to P) & \longrightarrow & \Gamma \Vdash @_a \mathsf{match}(p, v)P \\
\Gamma \Vdash @_a\mathbf{iflocal}\ \langle b \rangle c\mathbf{!}v\ \mathbf{then}\ P\ \mathbf{else}\ Q & \longrightarrow & \Gamma \Vdash @_a P \mid @_b c\mathbf{!}v \qquad \text{where } \Gamma \vdash a@s \wedge \Gamma \vdash b@s \\
\Gamma \Vdash @_a\mathbf{iflocal}\ \langle b \rangle c\mathbf{!}v\ \mathbf{then}\ P\ \mathbf{else}\ Q & \longrightarrow & \Gamma \Vdash @_a Q \qquad \text{where } \Gamma \vdash a@s \wedge \Gamma \vdash b@s' \wedge s \neq s'
\end{array}
$$

Figure 3: Selected Reduction Rules

$$
\frac{}{\Gamma \Vdash_a\ c\mathbf{!}v \xrightarrow{c\mathbf{!}v} @_a\mathbf{0}}
\qquad
\frac{\Gamma \vdash c \in \hat{}^{\mathtt{r}}T \quad \Gamma, \Delta \vdash v \in T \quad \mathsf{dom}(\Delta) \subseteq \mathsf{fv}(v) \quad \Delta \text{ is extensible.}}{\Gamma \Vdash_a\ c\mathbf{?}p \to P \xrightarrow[\Delta]{c\mathbf{?}v} @_a \mathsf{match}(p, v)P}
$$

$$
\frac{\Gamma \Vdash_a\ P \xrightarrow[\Delta]{c\mathbf{!}v} LP \quad \Gamma \Vdash_a\ Q \xrightarrow[\Delta]{c\mathbf{?}v} LQ}{\Gamma \Vdash_a\ P \mid Q \xrightarrow{\tau} \mathbf{new}\ \Delta\ \mathbf{in}\ LP \mid LQ}
\qquad
\frac{(\Gamma, x : T) \Vdash_a\ P \xrightarrow[\Delta]{c\mathbf{!}v} LP \quad x \in \mathsf{fv}(v) \quad x \neq c}{\Gamma \Vdash_a\ \mathbf{new}\ x : T\ \mathbf{in}\ P \xrightarrow[\Delta, x:T]{c\mathbf{!}v} LP}
$$

$$
\frac{}{\Gamma \Vdash_a\ \mathbf{migrate\ to}\ s \to P \xrightarrow{\mathsf{migrate\ to}\ s} @_a P}
\qquad
\frac{(\Gamma, a : \mathtt{Agent}^{\mathtt{m}}@s) \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s'} LQ}{\Gamma \Vdash \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ LP \xrightarrow{\tau} \mathbf{new}\ a : \mathtt{Agent}^{\mathtt{m}}@s'\ \mathbf{in}\ LQ}
$$

Figure 4: Selected LTS Rules

for delivering location-independent messages to their destination agent.

The reduction semantics describes only the internal behaviour of processes – for compositional reasoning we need also a typed labelled transition semantics, expressing how processes can interact with their environment. Transitions are defined inductively on process structure, without the structural congruence. The transition relations have the following forms, for basic and located process:

$$\Gamma \Vdash_a\ P \xrightarrow[\Delta]{\alpha} LP \qquad \Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$$

Here the *unlocated labels* α are of the following forms:

τ	internal computation
migrate to s	migrate to the site s
$c\mathbf{!}v$	send value v along channel c
$c\mathbf{?}v$	receive value v from channel c

The *located labels* β are of the form τ or $@_a\alpha$ for $\alpha \neq \tau$. Private names (together with their types, which may be annotated with an agent's current site) may be exchanged in communication and are made explicit in the transition relation by the extruded context Δ. Selected rules are given in Figure 4.

Theorem 4.1 (Reduction/LTS Correspondence) For any well-formed located type context Γ and located process LP such that $\Gamma \vdash LP$, we have: $\Gamma \Vdash LP \longrightarrow \Gamma' \Vdash LQ$ if and only if either

- $\Gamma \Vdash LP \xrightarrow{\tau} LQ$ with $\Gamma' = \Gamma$, or
- $\Gamma \Vdash LP \xrightarrow{@_a \mathsf{migrate\ to}\ s} LQ$ with $\Gamma' = \Gamma \oplus a \mapsto s$.

Theorem 4.2 (Subject Reduction) For any well-formed closed located type context Γ, if $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ then $\Gamma, \Delta \vdash LQ$.

5 Semantic Techniques

In this section we describe the tools used for stating and proving correctness. We are expressing distributed infrastructure algorithms as encodings from a high-level language to its low-level fragment, so the behaviour of a source program and its encoding can be compared directly with some notion of *operational equivalence* – our main theorem will be roughly of the form

$$\forall P \,.\, P \simeq \mathcal{C}\,[\![P]\!] \tag{†}$$

where P ranges over well-typed programs of the high-level language (P may use LI communication whereas $\mathcal{C}\,[\![P]\!]$ will not). Now, what equivalence $\simeq$ should we take? The stronger it is, the more confidence we gain that the encoding is correct. At first glance, one might take some form of weak bisimulation since (modulo divergence) it is finer than most notions of testing [dH84, Sew97] and is easier to work with. However, as in Nestmann's work on choice encodings [NP96], (†) would not hold, as the encoding $\mathcal{C}\,[\![P]\!]$ involves *partial commitment* of some nondeterministic choices. An example is given in §6. We therefore take $\simeq$ to be an adaptation of *coupled simulation* [PS92] to our language. This is a slightly coarser relation, but it is expected to be finer than any reasonable notion of observational equivalence for Nomadic π (modulo questions of divergence and fairness).

Translocating Equivalences To prove (†) we need compositional techniques, allowing separate parts of the protocols to be treated separately. In particular, we need operational congruences (both equivalences and preorders) that are preserved by program contexts involving parallel composition and new-binding. In Nomadic π the behaviour of LD communications depends on the relative location of agents: if a and b are at the same site then the LD message $@_b\langle a@s\rangle c\texttt{!}v$ reduces to (and in fact is weakly equivalent to) the local output $@_a c\texttt{!}v$, whereas if they are at different sites then the LD message is weakly equivalent to $\mathbf{0}$. A parallel context, eg. $[.]|@_a$**`migrate to`** s, can migrate the agent a, so to obtain a congruence we need refined equivalences, taking into account the possibility of such changes of agent location caused by the environment. Allowing arbitrary relocations would give too strong a notion, though. We introduce *translocating* relations that are parameterised by a set of agents that the environment may move.

Channel communication introduces further problems since it allows extrusion of new agent names to and from the environment. Consider an output of a new-bound agent name a to the environment. Other components in the environment may then send messages to a, but cannot migrate it, so when checking a translocating equivalence we do not need to consider relocation of a. On the other hand, a new agent name received from the environment by an input process is the name of an agent created in the environment, so (if created with the mobile capability) it may be migrated at any time.

We define a *translocating strong simulation* $\mathcal{S}$ to be a relation over located processes, indexed by pairs of a type context Γ and a set of names M, with $(LP, LQ) \in \mathcal{S}_\Gamma^M$ implying

- $\Gamma \vdash LP$ and $\Gamma \vdash LQ$.
- $M \subseteq \mathsf{movable}(\Gamma)$.
- For any δ valid for (Γ, M), if $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ then for some LQ' we have $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}_{\Gamma(\delta\oplus\beta),\Delta}^{M \uplus_\beta \mathsf{movable}(\Delta)}$.

Here $\mathsf{movable}(\Gamma)$ is the set of names of type $\texttt{Agent}^{\texttt{m}}$ in Γ and a valid *relocator* δ for (Γ, M) is a partial function from M to the site names of Γ. We write $\Gamma\delta$ for the result of applying δ to Γ. The set $M \uplus_\beta S$ is $M \cup S$ whenever β is an input and is M otherwise.

A symmetric *strong translocating bisimulation*, denoted $\dot\sim$, and weak version can be defined analogously.

We prove congruence results for both strong and weak translocating bisimulation, stating the result only for the strong. It uses a further auxiliary definition: the set $\mathsf{mayMove}(LP)$ is the set of agents in LP syntactically containing **`migrate to`**.

Theorem 5.1 (Translocating Congruence)
Given a closed located type context Γ, Θ with Θ extensible, if

- $LP \mathrel{\dot\sim}_{\Gamma,\Theta}^{M_P} LP'$ and $LQ \mathrel{\dot\sim}_{\Gamma,\Theta}^{M_Q} LQ'$, and
- $\mathsf{mayMove}(LQ, LQ') \subseteq M_P$ and $\mathsf{mayMove}(LP, LP') \subseteq M_Q$, and
- $M \stackrel{\text{def}}{=} M_P \cap M_Q \cap \mathsf{agentIn}(\Gamma)$

then

$$\textbf{new}\ \Theta\ \textbf{in}\ (LP \mid LQ) \mathrel{\dot\sim}_\Gamma^M \textbf{new}\ \Theta\ \textbf{in}\ (LP' \mid LQ')\,.$$

Intuitively, the first premise ($LP \mathrel{\dot\sim}_{\Gamma,\Theta}^{M_P} LP'$) of the theorem must allow all the potential agent movements of LQ and LQ', and symmetrically.

Expansion To construct the coupled simulation, we use an *expansion* relation $\dot\succeq$ [NP96] and the "up to" technique of [SM92], adapted with translocation, to allow elimination of target processes that are in intermediate/housekeeping stages. The definitions are omitted. We depend on a congruence result, analogous to that above, for expansion.

Temporary Immobility At many points in the execution of an encoded program, it is intuitively clear that an agent cannot migrate – while waiting for an acknowledgement from the daemon, or for either `currentloc` or `lock` to be released in the agent or daemon. To capture such an intuition, we consider derivatives of a process LP — if an input action on a lock channel l always precedes any (observable) migration action then LP can be said to be temporarily immobile, blocked by l. Care must be taken, however, to ensure that the lock l is not released by the environment. This can be made precise by the following definitions.

As in the case of translocating equivalences, we need to consider the possibility of agents being moved by the environment.

Definition 5.1 (Translocating Path)
A *translocating path* of LP_0 wrt (Γ, M) is a sequence

$$\xrightarrow[\Delta_1]{\beta_1} \ldots \xrightarrow[\Delta_n]{\beta_n}$$

for which there exist $LP_1, \ldots, LP_n$ and $\delta_0, \ldots, \delta_{n-1}$ such that for each $i \in 0 \ldots n-1$:

- δ_i is a valid relocator for $(\hat\Gamma, \hat M)$, where

$$\begin{array}{rcl} \hat\Gamma & \stackrel{\text{def}}{=} & \Gamma, \Delta_1, \ldots, \Delta_i \\ \hat M & \stackrel{\text{def}}{=} & M \uplus_{\beta_1} \mathsf{movable}(\Delta_1) \ldots \uplus_{\beta_i} \mathsf{movable}(\Delta_i),\text{ and} \end{array}$$

- $((\Gamma\delta_0, \Delta_1)\delta_1\beta_1, \Delta_2 \ldots \beta_i, \Delta_i)\delta_i \Vdash LP_i \xrightarrow[\Delta_{i+1}]{\beta_{i+1}} LP_{i+1}$.

Definition 5.2 (Temporary Immobility)
Given a closed located type context Γ, a located process LP with $\Gamma \vdash LP$, and a translocating index $M \subseteq \mathsf{agentIn}(\Gamma)$, LP is *temporarily immobile* under lock l wrt (Γ, M) if, for all translocating paths

$$\xrightarrow[\Delta_1]{\beta_1} \ldots \xrightarrow[\Delta_n]{\beta_n}$$

of LP wrt (Γ, M) which do not contain an input action $\beta_i = @_a c\texttt{?}v$ with $l \in \mathsf{fv}(c, v)$, the following hold for all $i \leq n$, b, c, v and s:

- $\beta_i = @_b c\texttt{!}v$ implies $l \notin \mathsf{fv}(\beta_i)$; and

- $\beta_i \neq @_b\mathsf{migrate\ to}\ s$.

Consider for example the process below.

$$\begin{aligned} LQ \ \stackrel{\text{def}}{=}\ & \mathbf{new}\ \Omega_{aux}\ \mathbf{in} \\ & \quad @_D Daemon \\ & \ \mid @_a(\llbracket P \rrbracket_a \mid \mathtt{currentloc!}s \mid \mathtt{Deliverer}) \end{aligned}$$

Here agent a cannot migrate until the daemon lock `lock` is successfully acquired, so LQ is temporarily immobile under `lock` with respect to any type-correct (Γ, M) that does not admit environmental relocation of a, ie. with $a \notin M$. Assume further that a is at s and that the daemon is forwarding an LI message to a, ie. the above is in parallel with

$$LP \ \stackrel{\text{def}}{=}\ @_D\langle a@s\rangle\mathtt{deliver!}[c\ v\ \mathtt{ack}]$$

This parallel composition, with a surrounding new-binder for `lock`, expands to

$$\begin{aligned} & \mathbf{new}\ \mathtt{lock} : {}^{\mathtt{rw}}\mathtt{Map}[\mathtt{Agent}^{\mathtt{s}}\ \mathtt{Site}]\ \mathbf{in} \\ & \quad LQ \mid @_a\mathtt{deliver!}[c\ v\ \mathtt{ack}] \end{aligned}$$

The proof of this expansion relies on the fact that the reductions of LP cannot release `lock`, so a cannot migrate, and hence the reductions of LP are deterministic, successfully delivering the message to a at s. It uses the following lemma.

Lemma 5.1
Given that LQ is temporarily immobile under l with respect to Γ, Δ and M, with Δ extensible and $l \in \mathsf{dom}(\Delta)$, if $\Gamma, \Delta \Vdash LP_1 \xrightarrow[M]{\mathsf{det}} LP_2$ then

$$\mathbf{new}\ \Delta\ \mathbf{in}\ LP_1 \mid LQ \quad \dot{\succeq}_{\Gamma}^{M\cap\mathsf{dom}(\Gamma)} \quad \mathbf{new}\ \Delta\ \mathbf{in}\ LP_2 \mid LQ$$

where LP_1 τ-deterministically reduces to LP_2 wrt (Γ, M), written $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LP_1$, if it has a reduction and, for any valid relocator δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP_1'$ implies $\beta = \tau$ and $LP_1 \dot{\sim}_{\Gamma}^{M} LP_1'$.

Proofs of temporary immobility can be hard, since they involve quantification over derivatives. However we may apply "up to" techniques to simplify processes under transitions.

Finite maps and functional computation We also prove the correctness of the encoding of finite maps (and so can omit maps from the basic calculus), and use uniform receptiveness [San99] to derive expansions from computation steps that are essentially functional.

Coupled Simulation Coupled simulation [PS92] relaxes the bisimulation clauses somewhat. A pair $(\mathcal{S}_1, \mathcal{S}_2)$, of type-context-indexed relations, is a *coupled simulation* if:

- $\mathcal{S}_1$ and $(\mathcal{S}_2)^{-1}$ are weak simulations (*not* translocating).
- if $(LP, LQ) \in (\mathcal{S}_1)_\Gamma$ then there exists LQ' such that $\Gamma \Vdash LQ \stackrel{\tau}{\Longrightarrow} LQ'$ and $(LP, LQ') \in (\mathcal{S}_2)_\Gamma$.
- if $(LP, LQ) \in (\mathcal{S}_2)_\Gamma$ then there exists LP' such that $\Gamma \Vdash LP \stackrel{\tau}{\Longrightarrow} LP'$ and $(LP', LQ) \in (\mathcal{S}_1)_\Gamma$.

Two processes LP, LQ are *coupled similar* wrt Γ, written $LP \fallingdotseq_\Gamma LQ$, if they are related by both components of some coupled simulation.

Intuitively "LQ coupled simulates LP" means that "LQ is at most as committed as LP" with respect to internal choices and that LQ may internally evolve to a state LQ' where it is at least as committed as LP, i.e. where LP coupled simulates LQ'.

In this paper, coupled simulation will be used for relating whole systems, which cannot be placed in any program context. For this reason, we do not need to incorporate translocation into the definition above.

6 Correctness of the Infrastructure

This section outlines the strategies taken in order to prove the correctness of the example encoding, using the techniques from §5.

Partial Commitment Our example infrastructure introduces many τ steps, each of which induces an intermediate state — a target level term which is not a literal translation of any source level term. Some of these steps are deterministic *house-keeping* steps; they can be reduced to (and related by expansions to) normal forms. Some, however (migration steps and acquisitions of the daemon lock or of local agent locks), are *partial commitment steps.* They involve nondeterministic internal choices and lead to partially committed states – target level terms which are not bisimilar to any source level term, but must be related to them by coupled simulation.

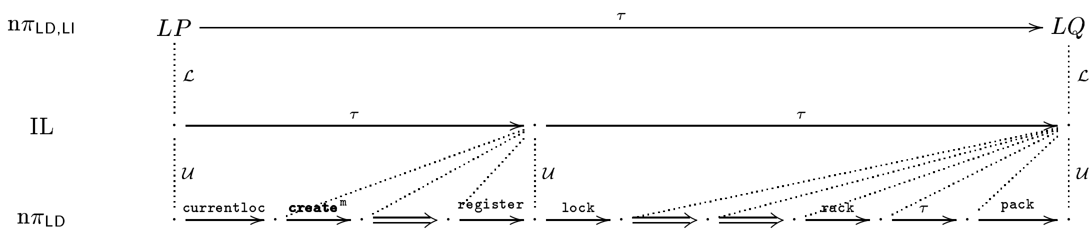
As an example, consider the encoding $\mathcal{C}\llbracket LP \rrbracket$ of an agent a which sends message $c!v$ to agent b at the current site of a, and in parallel visits the sites s_1 and s_2 (in any order).

$$LP \stackrel{\text{def}}{=} @_a\big(\langle a\rangle c!v \mid \mathbf{migrate\ to}\ s_1 \mid \mathbf{migrate\ to}\ s_2\big)$$

Assume a and b are initially at the same site. If the **migrate to** s_1 process in $\mathcal{C}\llbracket LP \rrbracket$ successfully acquires the local lock (a partial commitment step) the resulting state does not correspond exactly to any state of LP – we know that a will eventually end up in s_2, yet, as the first migration has not taken place, the message from a will reach b.

Intermediate Language We factor the construction of the main coupled simulation (between source program and its encoding) by introducing an *intermediate language* IL, with states ranged over by *Sys*. This helps us manage the complexity of the state-space of the encoding, by:

1. reducing the size of the coupled simulation relations, omitting states which reduce by house-keeping steps to certain normal forms (which have no house-keeping steps); and
2. dealing with states in which many agents may be partially committed simultaneously; and



where $LP = @_a\mathbf{create}^{m}\ b = P\ \mathbf{in}\ Q$ and $LQ = \mathbf{new}\ b\ \mathbf{in}\ @_aQ|@_bP$.

Figure 5: Relationships Between Source, Intermediate, and Target

3. capturing some invariants, eg. that the daemon's site-map is correct, in a type system for IL.

The cost is that the typing and labelled transition rules for IL must be defined; for lack of space we shall only outline some of the details.

Each term of the intermediate language therefore represents a normal form of target-level derivatives, possibly in a partially committed state. It describes the state of the daemon as well as that of the encoded agent. The syntax is:

$$Sys \quad ::= \quad \mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$$

Each term $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is parameterised by Δ, a located type context corresponding to all names dynamically created during the execution of the program, and $\boldsymbol{D}$ and $\boldsymbol{A}$, the state of the daemon and of the agents. Δ is binding in $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ and is therefore subject to alpha-conversion. The latter two parameters are described in more detail below:

- The state $\boldsymbol{D}$ of the daemon is described by the following syntax:

  $$\begin{array}{lcl} \boldsymbol{D} & ::= & [map\ \mathsf{mesgQ}] \\ \mathsf{mesgQ} & ::= & \prod_{i\in I} \mathsf{mesgReq}(\{T_i\}\ [a_i\ c_i\ v_i]) \end{array}$$

  Each daemon state $[map\ \mathsf{mesgQ}]$ consists of a site map map, expressed as a list of pairs, and an unordered queue of message forwarding requests mesgQ. A message forwarding request $\mathsf{mesgReq}(\{T\}\ [a\ c\ v])$ requires the daemon to forward $c!v$ to the agent a, where T is the type of v.

- The state $\boldsymbol{A}$ of the agents is a partial function mapping agent names to agent states. Each agent state, represented as $[P\ \boldsymbol{E}]$, consists of a *main body* P and a *pending state* $\boldsymbol{E}$. The syntax of $\boldsymbol{E}$ is given below:

  $$\begin{array}{lcl} \boldsymbol{E} & ::= & \mathsf{FreeA}(s) \mid \mathsf{RegA}(b\ Z\ s\ P\ Q) \\ & & \mid \mathsf{MtingA}(s\ P) \mid \mathsf{MtedA}(s\ P) \end{array}$$

  If an agent a has pending state $\mathsf{FreeA}(s)$, the local lock of a is free and is ready to initiate a **create** or **migrate to** process from its main body. Otherwise, a is in a partially committed state, with a pending execution of $\mathbf{create}^{Z}\ b = P\ \mathbf{in}\ Q$ (when its state is $\mathsf{RegA}(b\ Z\ s\ P\ Q)$) or $\mathbf{migrate\ to}\ s \to P$ (when its state is $\mathsf{MtingA}(s\ P)$ or $\mathsf{MtedA}(s\ P)$). In $\mathsf{FreeA}(s)$ and $\mathsf{RegA}(b\ Z\ s\ P\ Q)$, s denotes the current site of a, internally recorded and maintained by the agent itself.

  In $\mathsf{RegA}(b\ Z\ s\ P\ Q)$, the name b is bound in P and Q and is subject to alpha-conversion.

Informally, each transition of a system originates either from an agent or the daemon. A process from the main body of an agent may be executed immediately if it is either an **iflocal**, **if**, **let** or a pair of an output and a (replicated) input on the same channel. The result of such an execution (governed by $n\pi_{\mathsf{LD,LI}}$ LTS rules) is placed in parallel with other processes in the main body, except for execution of an LI output $\langle b\rangle c!v$, which results in the message forwarding request $\mathsf{mesgReq}(\{T\}\ [b\ c\ v])$ being added to the message queue of the daemon (T is the type of v). These steps correspond exactly to those taken by source- and target-level terms. A process $\mathbf{create}^{Z}\ b = P\ \mathbf{in}\ Q$ or $\mathbf{migrate\ to}\ s \to P$ from the main body of a may proceed (in fact initiate) if the local lock is free, ie. the pending state is $\mathsf{FreeA}(s')$. The result of such initiation turns the pending state to $\mathsf{RegA}(b\ Z\ s'\ P\ Q)$ or $\mathsf{MtingA}(s\ P)$ respectively. Translating into target-level terms, an agent in such a state has successfully acquired its local lock and sent a registration or migrating request to the daemon.

A system with registration request $\mathsf{RegA}(b\ Z\ s\ P\ Q)$ is executed in a single reduction step, corresponding in the target-level to acquiring the daemon lock, updating the site map and sending the acknowledgement to b. After completion, the declaration $b : \mathtt{Agent}^{Z}@s$ is placed at the top level and, at the same time, the site map is extended with the new entry (b, s). The new agent b with state $[P\ \mathsf{FreeA}(s)]$ now commences its execution, and so does its parent. Figure 5 gives the correspondences between steps in the source, intermediate and the target languages in the creation case. In the figure, some τ communication steps are annotated with the command or the name of the channel involved.

Likewise, a system with a message forwarding request $\mathsf{mesgReq}(\{T\}\ [b\ c\ v])$ is executed in a single reduction step, corresponding in the target-level to acquiring the daemon lock, looking up the site of b, delivering the message, and receiving an acknowledgement from b. After completion, the message $c!v$ is added to the main body of b.

Serving a migrating request $\mathsf{MtingA}(s\ P)$ from an agent a, however, involves two steps. The first step acquires the daemon lock, initialising the request and turning the pending state of a to $\mathsf{MtedA}(s\ P)$. In the second step, the agent a migrates to s (hence changes the top-level declaration) and the site map updates a with the entry (a, s). The first step corresponds in the target-level to acquiring the daemon lock, looking up the site of a in the site map, and sending an acknowledgement, permitting a to migrate. The second step corresponds to a migrating to s and sending an acknowledgement back to the daemon, which updates its site map and then sends the final acknowledgement to a, allowing it to proceed.

Factoring the proof The infrastructure encoding is factored into the composition of a *loading* encoding $\mathcal{L}$, mapping source terms to corresponding systems in the intermediate language, and an *unloading* encoding $\mathcal{U}$, mapping systems in the intermediate language to their corresponding target terms.

$$\begin{tikzcd}
\mathrm{n}\pi_{\mathsf{LD,LI}} \arrow[r, "\mathcal{L}[\![\cdot]\!]"] \arrow[dr, "\mathcal{C}[\![\cdot]\!]"'] & \mathrm{IL} \arrow[d, "\mathcal{U}[\![\cdot]\!]"'] \\
& \mathrm{n}\pi_{\mathsf{LD}}
\end{tikzcd}$$

Note that our encoding is not *uniform* [Pal97], as it introduces a centralised daemon at top level. This means that our reasoning must largely be about the whole system, dealing with interactions between encoded agents and the daemon. We cannot use simple induction on source program syntax.

We prove the coupled simulation over programs which are well-typed with respect to a *valid system context*: a type context in which all agents are declared as static (in order to use the standard definition of coupled simulation) and channels are not used for sending or receiving agent names (in order to make sure the daemon has a record of all agents in the system). Dynamically created new-bound agents may be mobile, of course.

We use two functions mapping intermediate language states back into the source language. The *undo* and *commit* decoding functions, $\mathcal{D}^\flat$ and $\mathcal{D}^\sharp$ respectively, undo and complete partially committed migrations (it suffices to have both functions commit creations and LI messages, as these are somewhat confluent).

$$\begin{tikzcd}
\mathrm{n}\pi_{\mathsf{LD,LI}} & \mathrm{IL} \arrow[l, "\mathcal{D}^\flat[\![\cdot]\!]"', "\mathcal{D}^\sharp[\![\cdot]\!]"]
\end{tikzcd}$$

The main lemmas can now be stated.

Lemma 6.1 (Syntactic Factorisation)
For any LP well-typed wrt a valid system context Φ

- $\mathcal{C}_\Phi [\![LP]\!] \equiv \mathcal{U} [\![\mathcal{L}_\Phi [\![LP]\!]]\!]$, and
- $LP \equiv \mathcal{D}^\flat [\![\mathcal{L}_\Phi [\![LP]\!]]\!] \equiv \mathcal{D}^\sharp [\![\mathcal{L}_\Phi [\![LP]\!]]\!]$.

Lemma 6.2 (Semantic Correctness of IL)
For any Sys well-formed wrt Φ, $\mathcal{U} [\![Sys]\!] \dot{\succeq}^{\emptyset}_{\Phi} Sys$.

The proof of this uses expansion up to expansion to relate each well-formed term in the intermediate language with its corresponding target term. Here we heavily employ the congruence properties of translocating expansion for factoring out program contexts which are not involved in housekeeping reductions of the target terms. Temporary immobility is used whenever we need to guarantee that LD messages to partially-committed agents are safely delivered.

The following two lemmas relate intermediate language states to source terms, by weak simulation relations using either the undo or commit decodings. Their proofs are relatively straightforward.

Lemma 6.3 ($\mathcal{D}^\flat$ is a strict simulation)
For any Sys well-formed wrt Φ, if $\Phi \Vdash Sys \xrightarrow[\Xi]{\beta} Sys'$ then $\Phi \Vdash \mathcal{D}^\flat [\![Sys]\!] \xrightarrow[\Xi]{\hat{\beta}} \mathcal{D}^\flat [\![Sys']\!]$.

Lemma 6.4 (${\mathcal{D}^\sharp}^{-1}$ is a progressing simulation)
For any Sys well-formed wrt Φ, if $\Phi \Vdash \mathcal{D}^\sharp [\![Sys]\!] \xrightarrow[\Xi]{\beta} LP$ then there exists a well-formed state Sys' such that $LP \equiv \mathcal{D}^\sharp [\![Sys']\!]$ and $\Phi \Vdash Sys \overset{\beta}{\underset{\Xi}{\Longrightarrow}} Sys'$.

These results are combined to give a direct relation between the source and the target terms (using weak simulation up to expansion), proving that a source term LP and its translation $\mathcal{C} [\![LP]\!]$ are related by a coupled simulation.

Theorem 6.1 (Encoding Correctness)
For any LP well-formed wrt a valid system context Φ, $LP \leftrightharpoons_\Phi \mathcal{C}_\Phi [\![LP]\!]$.

The use of coupled simulation makes this a rather strong result, but it does not take the external I/O of a whole program into account. That could be done by tuning a notion of testing [dH84] to this setting, generalising [Sew97] from Pict to the distributed case.

7 Conclusion

Related Work A wide range of other aspects of distributed and mobile computation have been studied via particular process calculi, eg. in [FGL$^+$96, CG98, RH98, Sew98, VC98] among others – space prevents a detailed comparison, but see [Sew00].

There is a large body of semantic work on concurrent and distributed algorithms. Crudely, it can be subdivided into work taking an automata-theoretic approach and work on encodings of high-level primitives. The former includes [AP98, JNW98], addressing Mobile IP, and [Mor99], which studies an infrastructure providing a similar abstraction to that of this paper. All involve more-or-less idealised models of algorithms rather than directly executable code. The latter includes encodings of choice [NP96], π/join communication [FG96], and authenticated communication [AFG00], all in terms of some code that in principle is executable. There is a trade-off here: the idealised models can be expressed in a simpler formal framework, greatly simplifying correctness proofs, but they are further removed from implementation, increasing the likelihood that important details have been abstracted away. This is discussed further in [Woj00]. Much of the latter work uses correctness results stated in terms of full abstraction wrt. some barbed congruence. Here, as the target language is a sublanguage of the source, we could state a more direct correspondence between the behaviour of source and target. Verification of mobile communication infrastructures has also been considered in the Mobile Unity setting [MR97].

Summary and Future Work We have addressed the distributed infrastructure algorithms required for location-independent communication between mobile agents. We have developed semantics and proof techniques for proving correctness of such algorithms, expressed as translations from high-level to low-level Nomadic Pict. The techniques were illustrated by a proof that an example algorithm is correct wrt. coupled simulation. This algorithm, though non-trivial, is one of the simplest possible. We believe that more sophisticated algorithms can be dealt with using the same techniques, albeit with new intermediate languages (tailored to particular algorithms).

By expressing infrastructure algorithms as Nomadic Pict encodings, we have descriptions of them that are:

- executable – one can rapidly prototype both algorithms and applications written above them in the high-level language; and
- concise – with the details of concurrency, locking, name-generation etc. made clear; and
- precise – with a semantics that we can use for formal reasoning and that gives a solid understanding of the primitives for informal reasoning.

More generally, the work is a step towards a semantically-founded view of richer wide-area distributed systems – here we dealt with the combination of migration and location-dependent communication; ultimately one must also simultaneously address failure and malicious attack.

References

[ACM96] ACM. *23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)*, 1996.

[AFG00] Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of POPL '00*. ACM, 2000.

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

[Car97] Luca Cardelli. Global computation. *ACM SIGPLAN Notices*, 32(1):66–68, January 1997.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.

[CGG99] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP '99*, volume 1644 of *LNCS*, pages 230–239. Springer, July 1999.

[CHK97] D. Chess, C. G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In *Mobile Object Systems - Towards the Programmable Internet*, LNCS, pages 25–47. Springer-Verlag, Berlin Germany, 1997.

[CON96] *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *LNCS*, Pisa, Italy, August 1996. Springer-Verlag.

[dH84] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, November 1984.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96* [ACM96], pages 372–385.

[FGL$^+$96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. [CON96], pages 406–421.

[JNW98] Daniel Jackson, Yuchung Ng, and Jeannette Wing. A Nitpick analysis of mobile IPv6. Technical Report CMU-CS-98-113, Computer Science Department, CMU, March 1998.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of POPL '96* [ACM96], pages 358–371.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. volume 94 of *Series F: Computer and System Sciences*. Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mor99] Luc Moreau. Distributed Directory Service and Message Router for Mobile Agents. Technical Report ECSTR M99/3, University of Southampton, 1999.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.

[MR97] P. J. McCann and G.-C. Roman. Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts. In *Proceedings of COORDINATION 97, LNCS 1282*, 1997.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. [CON96], pages 179–194.

[Pal97] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM, January 1997.

[PS92] J. Parrow and P. Sjodin. Multiway synchronization verified with coupled simulation. In *Proceedings CONCUR 92, LNCS 630*, 1992.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proc. LICS '93*: 376–385.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT press, May 2000.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL '98*, January 1998.

[San99] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256.

[Sew97] 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 pi-calculus. In *Proceedings of ICALP '98*, volume 1443 of *LNCS*, pages 695–706. Springer, July 1998.

[Sew00] Peter Sewell. Applied π – a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, August 2000.

[SM92] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In *Proceedings CONCUR 92, LNCS 630*, 1992.

[SWP98] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Proceedings of the Workshop on Internet Programming Languages (Chicago)*, May 1998. Full version appeared in LNCS 1686.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. [CON96], pages 278–298.

[VC98] Jan Vitek and Giuseppe Castagna. Towards a calculus of secure mobile computations. In *IEEE Workshop on Internet Programming Languages*, Chicago, Illinois, May 1998. Full version appeared in LNCS 1686.

[Woj00] Paweł T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Computer Laboratory, University of Cambridge, 2000. Available as Technical Report 492, June 2000.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April–June 2000.