Nomadic Pict: Programming Languages, Communication Infrastructure Overlays, and Semantics for Mobile Computation

PETER SEWELL
University of Cambridge
PAWEŁ T. WOJCIECHOWSKI
Poznań University of Technology
and
ASIS UNYAPOTH
Government Information Technology Services, Thailand

Mobile computation, in which executing computations can move from one physical computing device to another, is a recurring theme: from OS process migration, to language-level mobility, to virtual machine migration. This article reports on the design, implementation, and verification of overlay networks to support reliable communication between migrating computations, in the Nomadic Pict project. We define two levels of abstraction as calculi with precise semantics: a low-level Nomadic π calculus with migration and location-dependent communication, and a high-level calculus that adds location-independent communication. Implementations of location-independent communication, as overlay networks that track migrations and forward messages, can be expressed as translations of the high-level calculus into the low. We discuss the design space of such overlay network algorithms and define three precisely, as such translations. Based on the calculi, we design and implement the Nomadic Pict distributed programming language, to let such algorithms (and simple applications above them) to be quickly prototyped. We go on to develop the semantic theory of the Nomadic π calculi, proving correctness of one example overlay network. This requires novel equivalences and congruence results that take migration into account, and reasoning principles for agents that are temporarily immobile (e.g., waiting on a lock elsewhere in the system). The whole

This work was funded in part by EPSRC grants GR/L62290, GR/N24872, GR/T11715, EP/C510712, EP/F036345, and EP/H005633, a Wolfson Foundation Scholarship for P. T. Wojciechowski, a Royal Thai Government Scholarship for A. Unyapoth, and a Royal Society University Research Fellowship for P. Sewell.
Authors' addresses: P. Sewell, Computer Laboratory, University of Cambridge, J. J. Thomson Avenue, Cambridge CB3 0FD, UK; email: Peter.Sewell@cl.cam.ac.uk; P. T. Wojciechowski, Institute of Computing Science, Poznań University of Technology, Piotrowo 2, Poznań 60-965, Poland; email: Pawel.T.Wojciechowski@cs.put.poznan.pl; A.Unyapoth, Government Information Technology Services, National Science and Technology Development Agency, Ministry of Science and Technology, Thailand; email: asis@gits.net.th.

DOI 10.1145/1734206.1734209 http://doi.acm.org/10.1145/1734206.1734209

stands as a demonstration of the use of principled semantics to address challenging system design problems.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages

General Terms: Algorithms, Design, Languages, Theory, Verification

ACM Reference Format:
Sewell, P., Wojciechowski, P. T., and Unyapoth, A. 2010. Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. ACM Trans. Program. Lang. Syst. 32, 4, Article 12 (April 2010), 63 pages.
DOI = 10.1145/1734206.1734209 http://doi.acm.org/10.1145/1734206.1734209

1. INTRODUCTION

Mobile computation, in which executing computations can move (or be moved) from one physical computing device to another, has been a recurring focus of research, spanning disparate communities. The late 1970s and the 1980s saw extensive work on process migration, largely in the setting of operating system support for local-area distributed computation, using migration for load-balancing, checkpointing, etc. This was followed in the late 1990s by work on programming language support for mobility, largely in the mobile agent community, aiming at novel wide-area distributed applications. The late 1990s also saw work on semantics, using the tools of process calculi and operational semantics. In parallel, there has been a great deal of interest in the related areas of *mobile code*, popularized by Java applets, in which executable (but not yet executing) code can be moved, and in *mobile devices*, such as smartphones, PDAs, and the other devices envisaged in ubiquitous computing, which provide applications for both mobile computation and mobile code. Recently, the late 2000s have seen a renewed interest in mobile computation, now driven by the rise of virtualization systems, such as VMWare and Xen, which support migration of client OS images. These are finally realizing the prospect of commercial *commodity computation*, in which management of services and applications can be decoupled from physical machines in a datacenter, and in which flexible markets for computational resources can emerge.

Building systems with mobile computation, whether it be at the hypervisor, OS process, or programming language level, raises challenging problems, ranging from security concerns to interaction between changing versions of the infrastructure. In this article we focus on one of these problems: that of providing reliable communication between migrating computations, with messages being delivered correctly even if the sending and receiving computation migrate. Such high-level *location-independent* communication may greatly simplify the development of mobile applications, allowing movement and interaction to be treated as separate concerns. To provide reliable communication in the face of migration, above the low-level *location-dependent* communication primitives of the existing Internet Protocol (IP) network, one essentially has to build an overlay network, to track migrations and route application messages to migrating computations. 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. Lastly, these infrastructure algorithms will be to some extent exposed, via their performance and behavior under failure, to the application programmer; some 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 runtime.

For this approach to be realistic, it is essential that the low-level primitives should be directly implementable above standard network protocols. The IP network 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. More substantially, we also include migration of running computations 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 creation of running computations, for sending messages between computations 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 article we consider only a single representative form: an asynchronous message-passing primitive, similar to the low-level primitive for communication between colocated computations, but independent of their locations, and transparent to migrations.

This two-level framework can be formulated cleanly using techniques from the theory of process calculi. We precisely define the low and high levels of abstraction as process calculi, the *Nomadic π calculi*, equipped with operational semantics and type systems. The overlay networks implementing the high level in terms of the low can then be treated rigorously as translations between these calculi. The semantics of the calculi provides a precise and clear understanding of the algorithms' behavior, aiding design, and supporting proofs of correctness. Our calculi draw on ideas first developed in Milner et al.'s π calculus [Milner et al. 1992; Milner 1992] and extended in the *Pict* language of Pierce and Turner [Pierce and Turner 2000; Turner 1996], the distributed join-calculus of Fournet et al. [1996], and the JoCaml programming language [Conchon and Le Fessant 1999].

To facilitate experimentation, we designed and implemented a *Nomadic Pict* programming language based on our calculi. The low-level language extends the compiler and runtime system of Pict, a concurrent language based on the π calculus, to support our primitives for computation 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.

We begin in Section 2 by introducing the Nomadic π calculi, discussing their primitives and semantics, and giving examples of common programming idioms.

In Section 3 we present a first example overlay network, expressed as a semantics-preserving translation of the high-level Nomadic π calculus into the low-level calculus. This is a central forwarding server, relatively simple but still requiring subtle locking to ensure correctness.

In Section 4 we give a brief overview of the design space for such overlay networks, presenting a range of basic techniques and distributed algorithms informally, and discussing their scalability and fault-tolerance properties with respect to possible applications.

For two of these more elaborate overlay algorithms, one using forwarding-pointer chains (broadly similar to that used in the JoCaml implementation) and one using query servers with caching, we give detailed definitions as Nomadic π calculus translations, in Section 5 and Section 6 (and in online Appendix C) respectively.

In Section 7 (with further details in online Appendices D, E, and F) we describe the design and implementation of the Nomadic Pict programming language, which lets us build executable distributed prototypes of these and many other overlay algorithms, together with simple example applications that make use of them.

We then return to semantics, to prove correctness of such overlay networks. In Section 8 we flesh out the semantic definition of the Nomadic π calculi and their basic metatheory: type preservation, safety, and correspondence between reduction and labelled transition semantics, and in Section 9 we develop operational reasoning techniques for stating and proving correctness. We

(1) extend the standard π calculus reduction and labeled transition semantics to deal with computation mobility, location-dependent communication, and a rich type system;
(2) consider *translocating* versions of behavioral relations (bisimulation [Milner et al. 1992] and expansion [Sangiorgi and Milner 1992] 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* [Parrow and Sjödin 1992];
(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 analyze 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.

We apply these to the Central Forwarding Server overlay of Section 3, describing a full correctness proof in Section 10. Finally, we discuss related work in Section 11 and conclude in Section 12.

This article thus gives a synoptic view of the results of the Nomadic Pict project, covering calculi, semantics, overlay network design, programming language design and implementation, proof techniques, and overlay network verification. Elements of this have previously appeared in conferences: the initial calculi of Sewell, Wojciechowski, and Pierce [1998, 1999]; the programming language implementation and example algorithms by Wojciechowski and Sewell [Wojciechowski and Sewell 1999, Wojciechowski 2001, 2006]; and an outline of the metatheory and algorithm verification of Unyapoth and Sewell [2001]. Further details of the implementation and algorithms, and of the semantics and proof, can be found in the Ph.D. theses of Wojciechowski and Unyapoth respectively [Wojciechowski 2000b; Unyapoth 2001]. The implementation is available online [Wojciechowski 2010].

Nomadic Pict was originally thought of in terms of computation mobility at the programming-language level, and the terminology of the body of the article is chosen with that in mind (we speak of mobile agents and language threads). Later work on the Acute programming language [Sewell et al. 2007] developed this point of view: Acute has slightly lower-level constructs than low-level Nomadic Pict for checkpointing running multithreaded computations, using which we built a small Acute library providing the low-level Nomadic Pict primitives; overlay-network implementations of the high-level Nomadic Pict abstraction could be expressed as ML-style modules above that. The underlying ideas may also be equally applicable to mobility at the virtual-machine OS image level, as we argued in a position paper [Sewell and Wojciechowski 2008] in the Joint HP-MSR Research Workshop on *The Rise and Rise of the Declarative Datacentre*.

2. THE NOMADIC π CALCULI

In this section we introduce the abstractions of the low- and high-level Nomadic π calculi.

The main entities are *sites* s and *agents* a. Sites represent physical machines or, more accurately, instantiations of the Nomadic Pict runtime system on physical machines; each site has a unique name.

Agents are units of running computation. Each agent has a unique name and a body consisting of some Nomadic Pict concurrent process P; at any moment it is located at a particular site. An agent can *migrate*, at any point, to any other site (identified by name), new agents can be *created* (with the system synthesizing a new unique name, bound to a lexically scoped identifier) and agents can *interact* by sending messages to each other.

A key point in the design of the low-level calculus is to make it easy to understand the behavior of the system in the presence of partial failure. To do so, we choose interaction primitives that can be directly implemented above the real-world network (the Sockets API and TCP or UDP), without requiring a sophisticated distributed infrastructure. Our guiding principle is that each reduction step of the low-level calculus should be implementable using at most one intersite asynchronous communication.

To provide an expressive language for local computation within each agent body, while keeping the calculus concise, we include the constructs of a standard asynchronous π calculus. The Nomadic Pict concurrent process of an agent body can involve parallel composition, new channel creation, and asynchronous messaging on those channels within the agent.

In the rest of this section we give the syntax of processes, with accompanying definitions of values, patterns, and types, and the key points of their reduction semantics. The full semantics is defined in Section 8 and online Appendices A and B.

2.1 Processes of the Low-Level Calculus

The syntax of the low-level calculus is as follows, grouped into the three agent primitives, two useful communication forms that are expressible as syntactic sugar, and the local asynchronous π calculus.

P, Q	$::=$	$\texttt{create}^Z\ a = P\ \texttt{in}\ Q$		spawn agent a with body P, on local site
	$\vert$	$\texttt{migrate to}\ s \to P$		migrate current agent to site s
	$\vert$	$\texttt{iflocal}\ \langle a \rangle c\,\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q$		send $c\,\texttt{!}v$ to agent a if it is co-located here, and run P, otherwise run Q
	$\vert$	$\langle a \rangle c\,\texttt{!}v$	(sugar)	send $c\,\texttt{!}v$ to agent a if it is co-located here
	$\vert$	$\langle a@s \rangle c\,\texttt{!}v$	(sugar)	send $c\,\texttt{!}v$ to agent a if it is at site s
	$\vert$	$\mathbf{0}$		empty process
	$\vert$	$P \vert Q$		parallel composition of processes P and Q
	$\vert$	$\texttt{new}\ c : \hat{}^I T\ \texttt{in}\ P$		declare a new channel c
	$\vert$	$c\,\texttt{!}v$		output of v on channel c in current agent
	$\vert$	$c\,\texttt{?}p \to P$		input on channel c in current agent

$|$ ***** c**?**$p \rightarrow P$ replicated input
$|$ **if** v **then** P **else** Q conditional
$|$ **let** $p = ev$ **in** P local declaration

Executing the construct **create**^{Z} $b = P$ **in** Q spawns a new agent, with body P, on the current site. 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 with b, both in its body and in the spawning agent (b is binding in P and Q). The Z is a mobility capability, either `s`, requiring this agent to be static, or `m`, allowing it to be mobile. We return to this when we discuss the type system.

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.

There is a single primitive for interaction between agents, allowing an atomic delivery of an asynchronous message between two agents that are colocated on the same site. The execution of **iflocal** $\langle a \rangle c$**!**v **then** P **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**!**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. This 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 a single site.

Two other useful constructs can be expressed as sugar: $\langle a \rangle c$**!**v and $\langle a@s \rangle c$**!**v attempt to deliver c**!**v (an output of v on channel c), 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 in a context where a is predictable. The first is implementable simply as **iflocal** $\langle a \rangle c$**!**v **then 0 else 0**; the second as **create**$^{\mathtt{m}}$ $b =$ **migrate to** $s \rightarrow$ $\langle a \rangle c$**!**v **in 0**, for a fresh name b that does not occur in s, a, c, or v.

Turning to the π calculus constructs, the body of an agent may be empty (**0**) or a parallel composition $P|Q$ of processes.

Execution of **new** c : $\hat{}^I T$ **in** P creates a new unique channel name for carrying values of type T; c is binding in P. The I is a capability: as in Pierce and Sangiorgi [1996], channels can be used for input only `r`, output only `w`, or both `rw`; these induce a subtype order.

An output c**!**v (of value v on channel c) and an input c**?**$p \rightarrow P$ in the same agent may synchronize, resulting in P with the appropriate parts of the value v bound to the formal parameters in the pattern p. Note that, as in other asynchronous π calculi, outputs do not have continuation processes. A replicated input *****c**?**$p \rightarrow P$ behaves similarly except that it persists after the synchronization, and so might receive another value.

Finally, we have conditionals **if** v **then** P **else** Q, and local declarations **let** $p = ev$ **in** P, assigning the result of evaluating a simple value expression

ev to a pattern p. In $c\,\texttt{?}\,p \rightarrow P$, $\texttt{*}c\,\texttt{?}\,p \rightarrow P$ and $\texttt{let}\ p = ev\ \texttt{in}\ P$ the names in pattern p are binding in P. If $\texttt{if}\ v\ \texttt{then}\ P$ is sugar for $\texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ \mathbf{0}$.

For a simple example program in the low-level calculus, consider the following applet server.

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

It can receive (on the channel named *getApplet*) requests for an applet. This is a replicated input ($\texttt{*}\mathit{getApplet}\,\texttt{?}\,[a\ s] \rightarrow \ldots$) so the server persists and can repeatedly grant requests. The requests contain a pair (bound to the tuple $[a\ s]$ of 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 acknowledgment to the requesting agent a (which here is assumed to be on site s') containing its name. In parallel, the body B of the applet commences execution.

2.2 Processes of the High-Level Calculus

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

$$
\begin{array}{rcll}
P & ::= & \ldots & \\
 & | & \langle a@?\rangle c\,\texttt{!}\,v & \text{send } c\,\texttt{!}\,v \text{ to agent } a \text{ whereever it is}
\end{array}
$$

The intended semantics is that this will reliably deliver the message $c\,\texttt{!}\,v$ to agent a, irrespective of the current site of a and of any migrations. The high-level calculus includes all the low-level constructs, so those low-level communication primitives are also available for interaction with application agents whose locations are predictable. We write $\mathrm{n}\pi_{\mathsf{LD}}$ for the processes of the low-level calculus, with location-dependent communication only, and $\mathrm{n}\pi_{\mathsf{LD,LI}}$ for the processes of the high-level calculus, with location-dependent and location-independent communication.

2.3 Values and Patterns

Channels allow the communication of first-order values: constants t, names x (including channel names c, agent names a, and site names s), tuples, and packages $\{T\}\,v$ of existential types, containing a witness type T and a value v. Patterns p are of similar shapes as values, but are subject to the condition that the names x and type variables X that they bind are all distinct.

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

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

2.4 Types

Typing infrastructure algorithms requires a moderately expressive type system. We take types

$$
\begin{array}{lll}
T ::= & B & \text{base type} \\
\mid & [T_1 \ldots T_n] & \text{tuple} \\
\mid & \hat{\ }^I T & \text{channel name} \\
\mid & \{X\}\, T & \text{existential} \\
\mid & X & \text{type variable} \\
\mid & \texttt{Site} & \text{site name} \\
\mid & \texttt{Agent}^Z & \text{agent name}
\end{array}
$$

where B might be `int`, `bool`, etc., taken from a set $\mathcal{T}$ of base types, and X is taken from a set $\mathcal{TV}$ of type variables. Existentials are needed as an infrastructure must be able to forward messages of any type (see the `message` and `deliver` channels in Figure 2 later). For more precise typing, and to support the proof techniques we develop in Section 9, channel and agent types are refined by annotating them with *capabilities*, ranged over by I and Z respectively. Channel capabilities were described in Section 2.2: channels can be used for input only `r`, output only `w`, or both `rw`. In addition, agents are either static `s`, or mobile `m` [Sewell 1998; Cardelli et al. 1999].

2.5 Outline of the Reduction Semantics

Located processes and located type contexts. The basic process terms given earlier 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*

$$LP,\, LQ ::= @_a P \mid LP|LQ \mid \mathbf{new}\; x : \texttt{Agent}^Z@s\; \mathbf{in}\; LP \mid \mathbf{new}\; x : \hat{\ }^I T\; \mathbf{in}\; LP$$

Here the body of an agent a may be split into many parts, written $@_a P_1|\ldots|@_a P_n$. The construct $\mathbf{new}\; x : \texttt{Agent}^Z@s\; \mathbf{in}\; LP$ declares a new agent name x (binding in LP); since this is an agent name, 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 the construct $\mathbf{new}\; x : \hat{\ }^I T\; \mathbf{in}\; LP$ declares a new channel name (binding in LP) and the annotation is omitted.

Correspondingly, we add location information to type contexts. *Located type contexts* Γ include data specifying the site where each declared agent is located; the operational semantics updates this when agents move.

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

For example, the following located type context declares two sites, s and s', and a channel c, which can be used for sending or receiving integers. It also declares a mobile agent a, located at s, and a static agent b, located at s'.

$$s : \texttt{Site},\ s' : \texttt{Site},\ c : \hat{\ }^{\texttt{rw}}\texttt{Int},\ a : \texttt{Agent}^{\texttt{m}}@s,\ b : \texttt{Agent}^{\texttt{s}}@s'$$

Pattern matching. When an input process receives a value v along a channel, it needs to deconstruct v, producing a substitution to be applied to its

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

Fig. 1. Selected reduction rules.

continuation process. As usual, this is done with an auxiliary partial function for *matching*, mapping pairs of patterns and values to name substitutions, whenever they are of the same shape.

$$
\begin{array}{lll}
\mathsf{match}(\_, v) & \stackrel{\text{def}}{=} & \{\} \\
\mathsf{match}(x, v) & \stackrel{\text{def}}{=} & \{v/x\} \\
\mathsf{match}([p_1 \ldots p_n], [v_1 \ldots v_n]) & \stackrel{\text{def}}{=} & \mathsf{match}(p_1, v_1) \cup \cdots \cup \mathsf{match}(p_n, v_n) \\
\mathsf{match}(\{X\}\ p, \{T\}\ v) & \stackrel{\text{def}}{=} & \{T/X\}\ \cup\ \mathsf{match}(p, v) \\
\mathsf{match}(p, v) & \stackrel{\text{def}}{=} & \bot \quad \text{(undefined)} \quad \text{otherwise}
\end{array}
$$

Reductions. To capture our informal understanding of the calculus 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 [Milner 1993]. Reductions are over *configurations*, which are pairs $\Gamma \Vdash LP$ of a located type context Γ and a located process LP. We use a judgement $\Gamma \vdash a@s$, meaning that an agent a is located at s in the located type context Γ. We shall give some examples of reductions, illustrating the new primitives, before giving the formal definition of reduction later, in Section 8 and Appendix B. The most interesting axioms for the low-level calculus are given in Figure 1.

An agent a can spawn a new mobile agent b, with body P, and continues with Q. The new agent is located at the same site as a (say s, with $\Gamma \vdash a@s$). The agent b is initially bound and the scope is over the process Q in a and the whole of the new agent.

$$
\begin{array}{l}
\Gamma \Vdash @_a(R \mid \mathbf{create}^{\mathtt{m}}\ b = P\ \mathbf{in}\ Q) \\
\rightarrow\ \Gamma \Vdash @_a R \mid \mathbf{new}\ b : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ (@_a Q \mid @_b P)
\end{array}
$$

When an agent a migrates to a new site s, we simply update the located type context.

$$
\begin{array}{l}
\Gamma \Vdash @_a(R \mid \mathbf{migrate\ to}\ \ s \rightarrow Q) \\
\rightarrow\ \Gamma \oplus a \mapsto s \Vdash @_a(R \mid Q)
\end{array}
$$

A **new**-bound agent may also migrate; in this case, we simply update the location annotation.

$$
\begin{array}{l}
\Gamma \Vdash @_a R \mid \mathbf{new}\ b : \mathtt{Agent}^{\mathtt{m}}@s'\ \mathbf{in}\ @_b\mathbf{migrate\ to}\ \ s \rightarrow Q \\
\rightarrow\ \Gamma \Vdash @_a R \mid \mathbf{new}\ b : \mathtt{Agent}^{\mathtt{m}}@s\ \mathbf{in}\ @_b Q
\end{array}
$$

An agent a may send a location-dependent message to an agent b if they are on the same site. The message, once delivered, may then react with an input in b. Assuming that $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$.

$$\begin{array}{ll} & \Gamma \Vdash @_a(\texttt{iflocal}\ \langle b\rangle c\,\texttt{!}[]\ \texttt{then}\ P\ \texttt{else}\ Q) \mid @_b(c\,\texttt{?}[] \to R) \\ \to & \Gamma \Vdash @_a P \mid @_b(c\,\texttt{!}[] \mid c\,\texttt{?}[] \to R) \\ \to & \Gamma \Vdash @_a P \mid @_b R \end{array}$$

If a and b are at different sites, say if $\Gamma \vdash a@s$ and $\Gamma \vdash b@s'$ for $s \neq s'$, then the message will get lost.

$$\begin{array}{ll} & \Gamma \Vdash @_a(\texttt{iflocal}\ \langle b\rangle c\,\texttt{!}[]\ \texttt{then}\ P\ \texttt{else}\ Q) \mid @_b(c\,\texttt{?}[] \to R) \\ \to & \Gamma \Vdash @_a Q \mid @_b(c\,\texttt{?}[] \to R) \end{array}$$

Synchronization of a local output $c\,\texttt{!}v$ and an input $c\,\texttt{?}x \to P$ only occurs within an agent, but 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. Consider for example the next process, executing as the body of an agent a.

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

It has a reduction for the creation of agent b, a reduction for the **iflocal** that delivers the output $c\,\texttt{!}d$ to b, and then a local synchronization of this output with the input on c. Agent a then has body $d\,\texttt{!}7$ and agent b has body $d\,\texttt{!}3 | d\,\texttt{?}n \to \mathbf{0}$. Only the latter output on d can synchronize with b's input $d\,\texttt{?}n \to \mathbf{0}$. 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. This provides a limited form of dynamic binding, with the semantics of a channel name (i.e., the set of partners that a communication on that channel might synchronize with) dependent on the agent in which it is used; it proves very useful in the infrastructure algorithms that we develop.

The high-level calculus has one additional axiom, allowing location-independent communication between agents.

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

This delivers the message $c\,\texttt{!}v$ to agent b irrespective of where b (and the sender a) are located. For example, next an empty-tuple message on channel c is delivered to an agent b with a waiting input on c.

$$\begin{array}{ll} & \Gamma \Vdash @_a(P \mid \langle b@?\rangle c\,\texttt{!}[]) \mid @_b(c\,\texttt{?}[] \to R) \\ \to & \Gamma \Vdash @_a P \mid @_b(c\,\texttt{!}[] \mid c\,\texttt{?}[] \to R) \end{array}$$

2.6 Discussion of Design Choices

The only inter-site communication required in an implementation of the low-level language is for the **migrate to** reduction, in which the body of the migrating agent a must be sent from its current site to site s. (For performance, one might also implement the location-dependent output $\langle a@s\rangle c\,\texttt{!}v$ directly, with a single inter-site message, rather than via the syntax desugaring into an agent creation and migration.)

This makes it easy to understand the behavior of the implementation in the presence of fail-stop site failure: if a site crashes, all agents are lost; and a migration from one site to another is guaranteed to succeed if those two sites do not fail. Elsewhere we develop distributed infrastructure algorithms that address site failure and/or disconnection [Wojciechowski 2000b, 2001]. They use an additional primitive for timeouts, which we do not include in the semantics in this article; our focus here is on the failure mode of message loss for location-dependent messages to agents that are not in the specified location.

One could also envisage extending the semantics with network topology information, so that link failure and network partitions could be modeled. As far as the operational semantics goes, that would be straightforward, but developing reasoning principles above the extended semantics would be a substantial task.

The inter-site messages that must be sent in an implementation (representations of migrating agents, and tuple-structured location-dependent messages) should be reliable in the face of intermittent network packet loss; our low-level semantics does not allow messages to be spontaneously discarded. They are also of unbounded size, and could often exceed the approximately 1500 bytes that can be sent in a UDP datagram over Ethernet without IP fragmentation. Hence, an implementation would send messages via TCP, not via UDP. This raises the question of whether the low-level calculus should guarantee that inter-site messages are received in the same order as they are sent. In favor, it would be easy to implement ordering guarantees, if all messages from one site to another are multiplexed on a single underlying TCP connection, and such guarantees may be useful for some distributed algorithms. Against this, the operational semantics would be much more complex, with queues of messages in the network, and reasoning principles above it would be correspondingly more complex. Moreover, if the low-level calculus guaranteed message ordering, it would be natural for the high-level calculus to also guarantee it. Implementing that, as agents migrate, would require more complex algorithms. Accordingly, we choose simple unordered asynchronous messages, in both the low- and high-level calculus.

A similar argument applies to the question of whether inter-site messages should be asynchronous or synchronous. If they are implemented above TCP, the implementation could conceivably acknowledge when each message is delivered to the destination Nomadic Pict runtime. This would add a nontrivial but modest communication cost (especially if messages are often relatively large, involving multiple TCP segments). However, the added semantic complexity would be large, and efficient implementations of synchronous messaging in the high-level calculus, between migrating agents, would be yet more complex. Accordingly, we stay with the asynchronous choice.

Another design choice is whether one allows agents to be nested. This might be desirable for a full-scale programming language design, but again would complicate reasoning, and would introduce many further choices as to how inter-agent communication happens across the nesting structure. We therefore stay with the simple choice described before, in which new agents are created as siblings, on the same site as their creator.

3. EXAMPLE INFRASTRUCTURE: CENTRAL FORWARDING SERVER ALGORITHM

In this section we present our first example distributed infrastructure, the *Central Forwarding Server (CFS)* algorithm. In subsequent sections we survey the algorithm design space and present two more algorithms in detail: a forwarding-pointers algorithm and a query server algorithm. In the last part of the article we develop semantic techniques and prove correctness of the CFS algorithm.

The problem that these algorithms solve is to implement the high-level calculus using the low-level primitives; specifically, to implement the high-level location-independent semantics

$$\Gamma \Vdash @_a \langle b@?\rangle c\,\mathtt{!}\,v \;\rightarrow\; \Gamma \Vdash @_b c\,\mathtt{!}\,v$$

that delivers a message to agent b irrespective of any migrations of agents a and b. To do so, they also use nontrivial implementations of the other high-level agent primitives, for example, adding some synchronizations around agent migrations and creations. The algorithms are expressed as translations of the high-level calculus into the low-level calculus.

The CFS algorithm translation is based on that in Sewell et al. [1998]. It involves a central daemon that keeps track of the current sites of all agents and forwards any location-independent messages to them. The daemon itself is 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. When a new agent is created, it has to register with the daemon, telling its site. Before an agent can migrate, it has to inform the daemon about its intent, and wait for an acknowledgment. After the migration, the agent tells the daemon it has finished moving and continues. Locks are used to ensure that an agent does not migrate away while a message forwarded by the daemon is on its way; this ensures that all messages forwarded from the daemon are delivered before the agent migrates away.

This is a relatively simple algorithm, rather sequential and with a centralized server daemon, but it still requires delicate synchronization that is easy to get wrong. Expressing it as a translation between well-defined low- and high-level languages provides a solid basis for discussion about design choices, and enables correctness proofs; the Nomadic Pict language implementation makes it possible to execute and use the algorithm in practice.

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

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

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

Fig. 2. The central server daemon and the interface context.

(we omit various initialization code, and will often elide type contexts Φ). For each term P_i of the source language $\mathrm{n}\pi_{\mathsf{LD,LI}}$, considered as the body of an agent named a_i, the result $[\![P_i]\!]_{a_i}$ of the translation is a term of the target language $\mathrm{n}\pi_{\mathsf{LD}}$. The body of the daemon and selected clauses of the compositional translation are shown in Figures 2 and 3. They interact using channels of an *interface context* Φ_{aux}, also defined in Figure 2, 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 original algorithm in Sewell et al. [1998] has been modified in the following ways to simplify the correctness proof.

—Type annotations have been added and checked with the Nomadic Pict type checker [Wojciechowski 2000b] (although this does not check the static/mobile subtyping).

—Fresh channels are used for transmitting acknowledgments, making such channels linear [Kobayashi et al. 1996]. This simplifies the proof of correctness, since communication along a linear channel yields an expansion.

$$[\![\langle b@?\rangle c!v]\!]_a \stackrel{\text{def}}{=} \langle D@SD\rangle \texttt{message!}\,\{T\}\,[b\ c\ v]$$

$$\begin{array}{l}
[\![\textbf{create}^Z\ b = P\ \textbf{in}\ Q]\!]_a \stackrel{\text{def}}{=} \\
\quad \texttt{currentloc?}s \rightarrow \textbf{new}\ \texttt{pack} : \hat{}^{\texttt{rw}}[],\ \texttt{rack} : \hat{}^{\texttt{rw}}[]\ \textbf{in} \\
\quad\quad \textbf{create}^Z\ b = \\
\quad\quad\quad \langle D@SD\rangle \texttt{register!}[b\ s\ \texttt{rack}] \\
\quad\quad\quad \mid \texttt{rack?}[] \rightarrow \textbf{iflocal}\ \langle a\rangle \texttt{pack!}[]\ \textbf{then} \\
\quad\quad\quad\quad\quad (\texttt{currentloc!}s \mid [\![P]\!]_b \mid \texttt{Deliverer}) \\
\quad\quad \textbf{in} \\
\quad\quad\quad \texttt{pack?}[] \rightarrow (\texttt{currentloc!}s \mid [\![Q]\!]_a) \\
\\
\quad\quad \text{where}\quad \texttt{Deliverer} \stackrel{\text{def}}{=} *\texttt{deliver?}\,\{X\}\,[c\ v\ \texttt{dack}] \rightarrow (\langle D@SD\rangle \texttt{dack!}[] \mid c!v)
\end{array}$$

$$\begin{array}{l}
[\![\textbf{migrate to}\ s \rightarrow P]\!]_a \stackrel{\text{def}}{=} \\
\quad \texttt{currentloc?_} \rightarrow \textbf{new}\ \texttt{mack} : \hat{}^{\texttt{rw}}[\hat{}^{\texttt{w}}[\texttt{Site}\ \hat{}^{\texttt{w}}[]]]\ \textbf{in} \\
\quad\quad \langle D@SD\rangle \texttt{migrating!}[a\ \texttt{mack}] \\
\quad\quad \mid \texttt{mack?}[\texttt{migrated}] \rightarrow \\
\quad\quad\quad \textbf{migrate to}\ s \rightarrow \textbf{new}\ \texttt{ack} : \hat{}^{\texttt{rw}}[]\ \textbf{in} \\
\quad\quad\quad\quad (\langle D@SD\rangle \texttt{migrated!}[s\ \texttt{ack}] \\
\quad\quad\quad\quad \mid \texttt{ack?}[] \rightarrow \texttt{currentloc!}s \mid [\![P]\!]_a)
\end{array}$$

$$\begin{array}{ll}
[\![\mathbf{0}]\!]_a & \stackrel{\text{def}}{=} \mathbf{0} \\
[\![P|Q]\!]_a & \stackrel{\text{def}}{=} [\![P]\!]_a \mid [\![Q]\!]_a \\
[\![c?p \rightarrow P]\!]_a & \stackrel{\text{def}}{=} c?p \rightarrow [\![P]\!]_a \\
[\![*c?p \rightarrow P]\!]_a & \stackrel{\text{def}}{=} *c?p \rightarrow [\![P]\!]_a \\
[\![c!v]\!]_a & \stackrel{\text{def}}{=} c!v \\
[\![\textbf{iflocal}\ \langle b\rangle c!v\ \textbf{then}\ P\ \textbf{else}\ Q]\!]_a & \stackrel{\text{def}}{=} \textbf{iflocal}\ \langle b\rangle c!v\ \textbf{then}\ [\![P]\!]_a\ \textbf{else}\ [\![Q]\!]_a \\
[\![\textbf{new}\ x : \hat{}^I T\ \textbf{in}\ P]\!]_a & \stackrel{\text{def}}{=} \textbf{new}\ x : \hat{}^I T\ \textbf{in}\ [\![P]\!]_a \\
[\![\textbf{if}\ v\ \textbf{then}\ P\ \textbf{else}\ Q]\!]_a & \stackrel{\text{def}}{=} \textbf{if}\ v\ \textbf{then}\ [\![P]\!]_a\ \textbf{else}\ [\![Q]\!]_a \\
[\![\textbf{let}\ p = ev\ \textbf{in}\ P]\!]_a & \stackrel{\text{def}}{=} \textbf{let}\ p = ev\ \textbf{in}\ [\![P]\!]_a
\end{array}$$

Fig. 3. The compositional encoding (selected clauses).

—We consider programs with many agents initiated separately on different sites, rather than only programs that are initiated as single agents (this more general translation is needed to make our coinductive proof techniques go through, analogous to strengthening of an induction hypothesis).

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 initialization 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. Finite maps, with lookup operation

$$\begin{array}{l}
\textbf{lookup}[T_1\ T_2]\ a\ \textbf{in}\ m\ \textbf{with} \\
\quad \textbf{found}(v) \rightarrow P \\
\quad \textbf{notfound} \rightarrow Q
\end{array}$$

and update operation (m **with** $a \mapsto v$), are expressed with a standard π calculus encoding [Unyapoth 2001, Section 6.5], so they do not need to be added as a primitive.

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 nontrivial: for the location-independent output, agent creation, and agent migration primitives. We discuss each one in turn, together with their interactions with the daemon. For the rest, $[\![.]\!]$ is homomorphic.

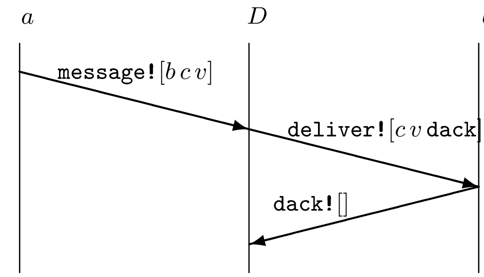
Location-independent output. A location-independent output $\langle b@?\rangle c\,!\,v$ in an agent a (of message $c\,!\,v$ to agent b) is implemented simply by requesting the central server daemon to deliver it; the request is sent to the daemon D, at its site SD, on its channel `message`, using a location-dependent output:

$$[\![\langle b@?\rangle c\,!\,v]\!]_a \stackrel{\text{def}}{=} \langle D@SD\rangle \texttt{message!}\{T\}\,[b\ c\ v]$$

The corresponding replicated input on channel `message` in the 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
```

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; the message also carries the name of a freshly created channel `dack`. It then waits to receive an acknowledgment (on the `dack` channel) from the agent before relinquishing the lock (with `lock!`m). This prevents the agent from migrating before the `deliver` message arrives, as the migration translation (that follows) also requires the lock. 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. In each agent the `deliver` message is handled by a `Deliverer` process (see Figure 3), which reacts to `deliver` messages by emitting a local $c\,!\,v$ message in parallel with sending the `dack` message to the daemon. The inter-agent communications involved in delivery of a single location-independent output are illustrated next.



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 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 of an agent creation in an agent a (in Figure 3)

$$
\begin{array}{l}
[\![\texttt{\textbf{create}}^{Z}\ b = P\ \texttt{\textbf{in}}\ Q]\!]_a \overset{\text{def}}{=} \\
\quad \texttt{currentloc}\,\texttt{?}s \rightarrow \texttt{\textbf{new}}\ \texttt{pack} : \texttt{\^{}rw}[],\ \texttt{rack} : \texttt{\^{}rw}[]\ \texttt{\textbf{in}} \\
\qquad \texttt{\textbf{create}}^{Z}\ b = \\
\qquad\quad \langle D@SD \rangle \texttt{register!}[b\ s\ \texttt{rack}] \\
\qquad\quad |\ \texttt{rack?}[] \rightarrow \texttt{\textbf{iflocal}}\ \langle a \rangle \texttt{pack!}[]\ \texttt{\textbf{then}} \\
\qquad\qquad (\texttt{currentloc!}s \mid [\![P]\!]_b \mid \texttt{Deliverer}) \\
\qquad \texttt{\textbf{in}} \\
\qquad\quad \texttt{pack?}[] \rightarrow (\texttt{currentloc!}s \mid [\![Q]\!]_a)
\end{array}
$$

$$
\text{where}\quad \texttt{Deliverer} \overset{\text{def}}{=} \texttt{*deliver?}\{X\}\,[c\ v\ \texttt{dack}] \rightarrow (\langle D@SD \rangle \texttt{dack!}[] \mid c\,\texttt{!}v)
$$

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 acknowledgment is received from the daemon, b sends an acknowledgment to a using `pack`, initializes 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 acknowledgment via `pack` is received.

The body of b is put in parallel with the replicated input

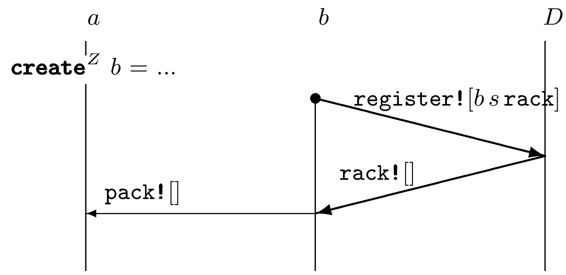
$$
\texttt{*deliver?}\{X\}\,[c\ v\ \texttt{dack}] \rightarrow \big(\langle D@SD \rangle \texttt{dack!}[] \mid c\,\texttt{!}v\big)
$$

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

The replicated input on `register` in the daemon

```
| *register?[b s rack] →
  lock?m →
    let[Agent^s Site] m′ = (m with b ↦ s) in
      (lock!m′ | ⟨b@s⟩rack![])
```

first acquires the lock and current site map, replaces the site map with an updated map, thereby relinquishing the lock, and sends an acknowledgment to the registering agent; the updated map records that a new agent b is located at site s. The inter-agent communications involved in a single agent creation are illustrated next.



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

```
[[migrate to s → P]]_a  ≝
  currentloc?_ → new mack : ^rw[^w[Site ^w[]]] in
    ⟨D@SD⟩migrating![a mack]
    | mack?[migrated] →
      migrate to s → new ack : ^rw[] in
        (⟨D@SD⟩migrated![s ack]
        | ack?[] → currentloc!s | [[P]]_a)
```

first acquires the output on `currentloc` at a (discarding the current site data). It then creates a fresh channel `mack`, sends a `migrating` message to the daemon with a tuple [a `mack`], and waits for an acknowledgment on `mack`.

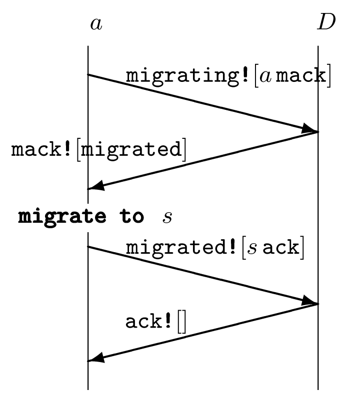
Reacting to the message on `migrating` message, the daemon

```
| *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
```

acquires its lock, looks up the current site of a in the acquired map m, creates a fresh channel `migrated`, and sends it (using an LD primitive) to a along channel `mack`. The daemon then waits to receive a message from `migrated`.

Once the waiting agent a receives a message from `mack`, it migrates to the new site s, then creates a fresh channel `ack` and sends a tuple [s `ack`] to the daemon via channel `migrated` (using an LD primitive). Meanwhile, the local lock and the encoding of the continuation process P is kept until the acknowledgment via `ack` is received from the daemon.

When the blocked daemon receives a message on `migrated`, it updates the site map, then relinquishes the lock and then sends an acknowledgment to a at its new site. The inter-agent communications involved in the migration of a single agent are illustrated next.



4. ALGORITHM DESIGN SPACE

Prospective applications may use some form of mobility for many different purposes, for example: to improve locality of computation; to support disconnected operation on mobile devices; to avoid transferring large volumes of data; to facilitate fault tolerance by moving computation from partially faulty machines; or to adapt to changes in the network characteristics and in the user environment. The different applications may have very different patterns of agent migration and communication, and require different performance and robustness properties. Agent migration would often be limited, for instance, to cases where agents migrate only once or twice, where migration is within a local-area network or between a few sites which are known in advance, where agents can only migrate to or from a central site, and between a mobile computer and the network, and so on.

In this section, we characterize some basic techniques and algorithms that can be useful for building such application-specific infrastructures, and assess their usefulness. We do not attempt to specify all the algorithms formally, so we use natural language descriptions. However, almost all algorithms have been implemented in Nomadic Pict, and the code is available with the language distribution [Wojciechowski 2010]. We also discuss informally the scalability and fault-tolerance properties of the algorithms. We do not attempt to give quantitative theoretical or empirical characterizations of the algorithms, because it would be too hard to take under consideration all the factors which exist in real systems; the range of possible migration and communication patterns is too great.

In the following sections, we describe two algorithms in more detail, presenting complete executable descriptions of the infrastructure in Nomadic Pict. They eliminate some of the drawbacks of the CFS algorithm in Section 3.

4.1 Background

We first discuss the space of all (deterministic) algorithms for location-independent message delivery to migrating entities. Awerbuch and Peleg [1995] (see also Mullender and Vitányi [1988]) stated the analogous problem of keeping track of mobile users in a distributed network. They consider two operations: "move", for a move of a user to a new destination, and "find", enabling one to contact a specified user at its current address. The problems of minimizing the communication overhead of these two operations appear to be in conflict. They examined two extreme strategies: full information and no information.

The *full-information* strategy requires every site in the network to maintain complete up-to-date information on the whereabouts of every user. This makes the "find" operation inexpensive. On the other hand, "move" operations are very expensive, since it is necessary to update information at every site. In contrast, the *no-information* approach does not assume any updates while migrating, thus the "move" operation has got a null cost. On the other hand, the "find" operation is very expensive because it requires global searching over the whole network. However, if a network is small and migrations frequent, the strategy can be useful. In contrary, the *full-information* strategy is appropriate for a near-static setting, where agents migrate relatively rarely, but frequently communicate with each other. Between these two extreme cases, there is space for designing intermediate strategies, that will perform well for any or some specific communication to migration pattern, making the costs of both "find" and "move" operations relatively inexpensive.

Awerbuch and Peleg [1995] describe a distributed directory infrastructure for online tracking of mobile users. They introduced the graph-theoretic concept of *regional matching*, and demonstrated how finding a regional matching with certain parameters enables efficient tracking of mobile users in a distributed network. The communication overhead of maintaining the distributed directory is within a polylogarithmic factor of the lower bound. This result is important in the case of mobile telephony and infrastructures which support mobile devices, where the infrastructure should perform well, considering *all* mobile users and their potential communication to migration patterns. These patterns can vary, depending on people, and can only be estimated probabilistically. The infrastructure should therefore support all migration and communication scenarios, and optimize those scenarios which are likely to happen more often (preferably it should adapt to any changes in behavior of mobile users dynamically). In mobile agent applications, however, the communication to migration pattern of mobile agents usually can be predicted precisely [Wojciechowski 2000b]. Therefore we can design algorithms which are optimal for these special cases and simpler than the directory infrastructure mentioned previously.

4.2 Central Server Algorithms

Central forwarding server algorithm. The server records the current site of every agent. Before migration an agent A informs the server and waits for ACK (containing the number of messages sent from the server to A). It then waits for all the messages due to arrive. After migration it tells the server it has finished moving. If B wants to send a message to A, B sends the message to the server, which forwards it. During migrations (after sending the ACK) the server suspends forwarding. A variant of this algorithm was described in Section 3.

Central query server algorithm. The server records the current site of every agent. If B wants to send a message to A, B sends a query (containing the message ID) to the server asking for the current site of A, gets the current site s of A, and sends the message to s. The name s can be used again for direct communication with A. If a message arrives at a site that does not have the recipient then a message is returned saying "you have to ask the name server again". Migration support is as before.

Home server algorithm. Each site s has a server that records the current site of some agents; usually the agents which were created on s. Agent names contain an address of the server which maintains their locations. On every migration agent A synchronizes with the server whose name is part of A's name. If B wants to send a message to A, B resolves A's name and contacts A's server. Other details are as before.

Discussion. If migrations are rare, and also in the case of stream communication or large messages, the Query Server seems the better choice. However, the Central Forwarding and Query Server algorithms do not scale well. If the number of agents is growing and communication and migration are frequent, the server can be a bottleneck. Home Servers can improve the situation. The infrastructure can work fine for small-to-medium systems, where the number of agents is small.

These algorithms do not support locality of agent migration and communication, that is, migration and communication involve the cost of contacting the server, which might be far away. If agents are close to the server, the cost of migration, search, and update is relatively low. Our algorithms clearly explore only a part of the design space; one can envisage, for example, splitting the servers into many parts (e.g., one dealing with agents created for each user). An exhaustive discussion is beyond the scope of this article.

In all three, the server is a single point of failure. In these and other algorithms, we can use some of the classical techniques of fault tolerance, for example, based on state checkpointing, message logging, and recovery. We can also replicate the server on different sites to enhance system availability and fault tolerance, such as using the *primary-backup* or *active* replication techniques (see, e.g., the tutorial by Guerraoui and Schiper [1996]). However, the implementation of a replicated system with replica crashes and unpredictable communication delay is a difficult task. The difficulty can be formally explained by theoretical impossibility results, such as the impossibility of solving consensus in an asynchronous system when processes can crash [Fischer et al. 1985]. These impossibility results can be overcome by strengthening the system model slightly [Chandra and Toueg 1996].

Mechanisms similar to Home Servers have been used in many systems which support *process migration*, such as Sprite [Douglis and Ousterhout 1991]. Caching has been used, for example, in LOCUS [Popek and Walker 1986], and V [Cheriton 1988], allowing operations to be sent directly to a remote process without passing through another site. If the cached address is wrong a home site of the process is contacted (LOCUS) or multicasting is performed (V). A variant of the Central Query Server algorithm, combined with Central Forwarding Server and data caching, will be described in detail in Section 6 and Appendix C; it also appeared in Wojciechowski and Sewell [2000].

4.3 Forwarding Pointers

Algorithm. There is a forwarding daemon on each site. The daemon on site s maintains a current guess about the site of agents which migrated from s. Every agent knows the initial home site of every agent (the address is part of an agent's name). If A wants to migrate from s_1 to s_2 it leaves a forwarding pointer at the local daemon. Communications follow all the forwarding pointers. If there is no pointer to agent A, A's home site is contacted. Forwarding pointers are preserved forever. This algorithm will be described in detail in Section 5.

Discussion. There is no synchronization between migration and communication as there was in centralized algorithms. A message may follow an agent which frequently migrates, leading to a race condition. The Forwarding Pointers algorithm is not practical if agents perform a large number of migrations to distinct sites (the chain of pointers grows, increasing the cost of search). Some "compaction" methods can be used to collapse the chain, for example, movement-based and search-based. In the former case, an agent would send backward a location update after performing a number of migrations; in the latter case, after receiving a number of messages (i.e., after a fixed number of "find" operations occurred).

Some heuristics can be further used such as search-update. A plausible algorithm can be as follows. On each site there is a daemon which maintains forwarding addresses (additionally to forwarding pointers) for all agents which ever visited this site. A *forwarding address* is a tuple $(timestamp, site)$ in which the site is the last known location of the agent and timestamp specifies the age of the forwarding address. Every message sent from agent B to A along the chain of forwarding pointers contains the latest available forwarding address of A. The receiving site may then update its forwarding address (and/or forwarding pointer) for the referenced agent, if required. Given conflicting guesses for the same agent, it is simple to determine which one is most recent using timestamps. When the message is eventually delivered to the current site of the agent, the daemon on this site will send an ACK to the daemon on the sender site, containing the current forwarding address. The address received replaces any older forwarding address but *not* the forwarding pointer (to allow updating the chain of pointers during any subsequent communication). A similar algorithm has been used in Emerald [Jul et al. 1988], where the new forwarding address is piggybacked onto the reply message in the object invocation. It is sufficient to maintain the timestamp as a counter, incremented every time the object moves.

A single site fail-stop in a chain of forwarding pointers breaks the chain. A solution is to replicate the location information in the chain on k consecutive sites, so that the algorithm is tolerant of a failure of up to $k - 1$ adjoint sites. Stale pointers should be eventually removed, either after waiting a sufficiently long time, or purged as a result of a distributed garbage collection. Distributed garbage collection would require detecting global termination of all agents that might ever use the pointer, therefore the technique may not always be practically useful. Alternatively, some weaker assumptions could be made and the agents decide arbitrarily about termination, purging the pointers beforehand.

4.4 Broadcast Algorithms

Data broadcast algorithm. Sites know about the agents that are currently present. An agent notifies a site on leaving and a forwarding pointer is left over until agent migration is finished. If agent B wants to send a message to A, B sends the message to all sites in a network. A site s discards or forwards the message if A is not at s (we omit details).

Query broadcast algorithm. As before but if agent B wants to send a message to A, B sends a query to all sites in a network asking for the current location of A. If site s receives the query and A is present at site s, then s suspends any migration of A until A receives the message from B. A site s discards or forwards the query if A is not at s.

Notification broadcast algorithm. Every site in a network maintains a current guess about agent locations. After migration an agent distributes in the network information about its new location. Location information is timestamped. Messages with stale location information are discarded. If site s receives a message whose recipient is not at s (because it has already migrated or the initial guess was wrong), it waits for information about the agent's new location. Then s forwards the message.

Discussion. The cost of communication in Query and Data Broadcasts is high (packets are broadcast in the network) but the cost of migration is low. Query Broadcast saves bandwidth if messages are large or in the case of stream communication. Notification Broadcast has a high cost of migration (the location message is broadcast to all sites) but the communication cost is low and similar to forwarding pointers with pointer chain compaction. In Data and Notification Broadcasts, migration can be fast because there is no synchronization involved (in Query Broadcast migration is synchronized with communication); the drawback is a potential for race conditions if migrations are frequent. Site failures do not disturb the algorithms. The simplest (partially) fault-tolerant algorithm could involve Data Broadcast with buffering of broadcast messages at target sites; however, two conditions should hold: buffers need to be infinite, and the broadcasting server needs to use *reliable broadcast* [Chandra and Toueg 1996].

Although we usually assume that the number of sites is too large to broadcast anything, we may allow occasional broadcasts within, for example, a local Internet domain, or local Ethernet. Broadcasts can be accomplished efficiently in bus-based multiprocessor systems. They are also used in radio networks. A realistic variant is to broadcast within a group of sites which belong to the itinerary of mobile agents that is known in advance. Broadcast has also been used in Emerald to find an object, if a node specified by a forwarding pointer is unreachable or has stale data. To reduce message traffic, only a site which has the specified object responds to the broadcast. If the searching daemon receives no response within a time limit, it sends a second broadcast requesting a positive or negative reply from all other sites. All sites not responding within a short time are sent a reliable, point-to-point message with the request. The Jini lookup and connection infrastructure [Arnold et al. 1999] uses multicast in the discovery protocol. A client wishing to find a Lookup Service sends out a known packet via multicast. Any Lookup Service receiving this packet will reply (to an address contained in the packet) with an implementation of the interface to the Lookup Service itself.

4.5 Agent-Based Broadcast

Algorithm. Agents are *grouped*, with the agents forming a group maintaining a current record about the site of every agent in the group. Agent names form a totally ordered set. We assume communication which takes place within a group only.

Before migration an agent A informs the other agents in the group about its intention and waits for ACKs (containing the number of messages sent to A). It then waits for all the messages due to arrive and migrates. After migration it tells the agents it has finished moving. Multicast messages to each agent within a group are reliably delivered in the order sent (using *first-in-first-out broadcast*). If B wants to send a message to A, B sends the message to site s which is A's current location. During A's migrations (i.e., after sending the ACK to A) B suspends sending any messages to A (in particular any migration requests). If two (or more) agents want to migrate at the same time there is a conflict which can be resolved as follows. Suppose A and C want to migrate. If B receives migration requests from A and C, it sends ACKs to both of them and suspends sending any messages to agents A and C (in particular any migration requests). If A receives a migration request from C after it has sent its own migration request it can either grant ACK to C (and C can migrate) or postpone the ACK until it has finished moving to a new site. The choice is made possible by ordering agent names. Thus, there is an invariant that at any time at most one agent can migrate in a given group.

Discussion. The advantage of this algorithm is that sites can be stateless (the location data are part of agent's state). The algorithm is suitable for frequent messages (or stream communication) between mobile agents and when migrations are rare.

Agents can be organized into *dynamic groups*, using the primitives of *group communication systems* (designed for nonmovable groups of distributed processes). The membership of a group can change over time, as agents *join* or *leave* the group, or as crashed (or suspected as crashed) agents are collectively *removed* from the group. The current set of agents that are members of a group is called the *group view*. Agents are added to and deleted from the group view via *view changes*, handled by a *membership* service.

Mobile agents forming a group can dynamically change sites. This creates a problem how to implement the join operation so that the agents joining a group will be able to localize the group. One solution is that migrating group agents could leave forwarding pointers that would be followed by agents joining the group to "catch up" with at least one group member. Another solution is to have one agent within a group: a *group coordinator*, which never migrates and can be used to contact the group. The intergroup communication algorithm could use either the pointers or coordination agents for delivering messages that cross group boundaries.

Other variants are also possible. For example, if agent migration would be limited to a fixed set of target sites that are known in advance, then the algorithms could broadcast only to such sites; the names of these sites could be encoded as part of agent's name.

4.6 Hierarchical Location Directory

Algorithm. A tree-like hierarchy of servers forms a location directory (similar to DNS). Each server in the directory maintains a current guess about the site of some agents. Sites belong to regions, each region corresponds to a subtree in the directory (in the extreme cases the subtree is simply a leaf-server for the smallest region, or the whole tree for the entire network). The algorithm maintains an invariant that for each agent there is a unique path of forwarding pointers which forms a single branch in the directory; the branch starts from the root and finishes at the server which knows the actual site of the agent (we call this server the "nearest"). Before migration an agent A informs the "nearest" server X_1 and waits for ACK. After migration it registers at a new "nearest" server X_2, tells X_1 it has finished moving, and waits for ACK. When it gets the ACK there is already a new path installed in the tree (this may require installing new and purging old pointers within the smallest subtree which contains X_1 and X_2). Messages to agents are forwarded along the tree branches. If B wants to send a message to A, B sends the message to the B's "nearest" server, which forwards it in the directory. If there is no pointer the server will send the message to its parent.

Discussion. Certain optimizations are plausible, for instance, if an agent migrates very often within some subtree, only the root of the subtree would contain the current location of the agent (the "move" operation would be cheaper). Moreau [2002] describes an algorithm for routing messages to migrating agents which is also based on distributed directory service. A proposal of Globe uses a hierarchical location service for worldwide distributed objects [van Steen et al. 1998]. The Hierarchical Location Directory scales better than Forwarding Pointers and Central Servers. Also, some kinds of fault can be handled more easily (see Awerbuch and Peleg [1995], and there is also a lightweight crash recovery in the Globe system [Ballintijn et al. 1999]).

4.7 Arrow Directory

Some algorithms can be devised for a particular communication pattern. For example, if agents do not require instant messaging, a simple mailbox infrastructure can be used, where senders send messages to static mailboxes and all agents periodically check mailboxes for incoming messages.

Demmer and Herlihy [1998] describe the Arrow Distributed Directory protocol for distributed shared object systems. The algorithm is devised for a particular object migration pattern; it assumes that the whole object is always sent to the object requester. The arrow directory imposes an optimal distributed queue of object requests, with no point of bottleneck.

The protocol was motivated by emerging *active network* technology, in which programmable network switches are used to implement customized protocols, such as application-specific packet routing.

Algorithm. The arrow directory is given by a minimum spanning tree for a network, where the network is modeled as a connected graph. Each vertex models a node (site), and each edge a reliable communication link. A node can send messages directly to its neighbors, and indirectly to non-neighbors along a path. The directory tree is initialized so that following arrows (pointers) from any node leads to the node where the object resides.

When a node wants to acquire exclusive access to the object, it sends a message *find* which is forwarded via arrows and sets its own arrow to itself. When the other node receives the message, it immediately "flips" the arrow to point back to the immediate neighbor who forwarded the message. If the node does not hold the object, it forwards the message. Otherwise, it buffers the message *find* until it is ready to release the object to the object requester. The node releases the object by sending it directly to the requester, without further interaction with the directory.

If two *find* messages are issued at about the same time, one will eventually cross the other's path and be "diverted" away from the object, following arrows towards the node (say v) where the other *find* message was issued. Then, the message will be blocked at v until the object reaches v, is accessed and eventually released.

5. EXAMPLE INFRASTRUCTURE: FORWARDING-POINTERS ALGORITHM

In this section we give a forwarding-pointers algorithm, in which daemons on each site maintain chains of forwarding pointers for agents that have migrated from their site. It removes the single bottleneck of the centralized-server solution in Section 3; it is thus a step closer to algorithms that may be of wide practical use. The algorithm is more delicate, so expressing it as a translation provides a more rigorous test of the framework.

The daemons are implemented as static agents; the translation $\mathcal{FP}_\Phi[\![LP]\!]$ of a located process $LP = \mathbf{new}\ \Delta\ \mathbf{in}\ @_{a_1}P_1 \mid \ldots \mid @_{a_n}P_n$, (well-typed with respect to Φ) then consists roughly of the daemon agent (one on each site s_j, named DS_j) in parallel with a compositional translation $[\![P_i]\!]_{a_i}$ of each source agent:

$$\begin{aligned}\mathcal{FP}_\Phi[\![LP]\!] \stackrel{\text{def}}{=}\ & \mathbf{new}\ \Delta, \Phi_{aux}\ \mathbf{in}\\ & @_{DS_1}(Daemon_{s_1} \mid \mathtt{lock!}m) \mid \ldots \mid @_{DS_m}(Daemon_{s_m} \mid \mathtt{lock!}m)\\ & \mid @_{a_1}[\![P_1]\!]_{a_1} \mid \ldots \mid @_{a_n}[\![P_n]\!]_{a_n}\end{aligned}$$

where m is a map such that $m(a) = [s_j\ DS_j]$ if $\Phi, \Delta \vdash a@s_j$. For each term P_i of the source language $\mathrm{n}\pi_{\mathsf{LD,LI}}$, considered as the body of an agent named a_i, the result $[\![P_i]\!]_{a_i}$ of the translation is a term of the target language $\mathrm{n}\pi_{\mathsf{LD}}$. As before, the translation consists of a compositional encoding of the bodies of agents, given in Figure 5, and daemons, defined in Figure 4. Note that in terms of the target language, each site name s_i is rebound to the pair $[s_i\ DS_i]$ of the site name together with the respective daemon name; the agent name a_i is rebound to the triple $[A_i\ s_i\ DS_i]$ of the low-level agent name A_i together with the initial site and daemon names. The low-level agent A_i is defined by the agent encoding; it contains the body P_i of agent a_i. Agents and daemons interact using channels of an *interface context* Φ_{aux}, also defined in Figure 4, which in addition declares lock channels and the daemon names $DS_1 \ldots DS_m$. It uses a map type constructor, which (together with the map operations) can be translated into the core language.

Daemons are created, one on each site. 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. There is a translation of types with clauses

$$[\![\mathtt{Agent}^Z]\!] \stackrel{\text{def}}{=} [\mathtt{Agent}^Z\ \mathtt{Site}\ \mathtt{Agent}^Z]$$
$$[\![\mathtt{Site}]\!] \stackrel{\text{def}}{=} [\mathtt{Site}\ \mathtt{Agent}^Z]$$

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 5, 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

```
Daemon_s
  def
   =  let [S DS] = s in
      *register?[B rack] → lock?m →
        lookup[Agent^s ^rw[Site Agent^s]] B in m with
          found(Bstate) →
            Bstate?[_ _] →
              Bstate![S DS] | lock!m | ⟨B⟩rack![]
          notfound →
            new Bstate : ^rw[Site Agent^s] in
              Bstate![S DS] | ⟨B⟩rack![]
              | let[Agent^s [Site Agent^s]] m' = (m with B ↦ Bstate) in
                lock!m'
      | *migrating?[B mack] → lock?m →
        lookup[Agent^s ^rw[Site Agent^s]] B in m with
          found(Bstate) →
            Bstate?[_ _] → (lock!m | ⟨B⟩mack![])
          notfound → 0
      | *migrated?[B [U DU] ack] → lock?m →
        lookup[Agent^s ^rw[Site Agent^s]] B in m with
          found(Bstate) →
            lock!m | ⟨B@U⟩ack![] | Bstate![U DU]
          notfound → 0
      | *message? {X} [[B U DU] c v] → lock?m →
        lookup[Agent^s ^rw[Site Agent^s]] B in m with
          found(Bstate) →
            lock!m
            | Bstate?[R DR] →
              iflocal ⟨B⟩c!v then Bstate![R DR]
                else ⟨DR@R⟩message! {X} [[B U DU] c v]
                  | Bstate![R DR]
          notfound → lock!m
            | ⟨DU@U⟩message! {X} [[B U DU] c v]

Φ_aux  def=  DS_1 : Agent^s@s_1, ..., DS_m : Agent^s@s_m,
             lock : ^rwMap[Agent^s ^rw[Site Agent^s]]
             register : ^rw[Agent^s ^w[]],
             migrating : ^rw[Agent^s ^w[]],
             migrated : ^rw[Agent^s [Site Agent^s] ^w[]],
             message : ^rw {X} [[Agent^s Site Agent^s] ^wX X],
             currentloc : ^rw[Site Agent^s]
```

Fig. 4. A forwarding-pointers translation: the daemon.

migration, each begin with an input on `currentloc`. They are broadly similar to those of the simple Central-Forwarding-Server translation in Section 3.

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

```
[[⟨b@?⟩c!v]]_A
  def= currentloc?[S DS] →
        iflocal ⟨DS⟩message! {T} [b c v]
          then currentloc![S DS]
          else currentloc![S DS]

[[create^Z b = P in Q]]_A
  def= currentloc?[S DS] →
        new pack : ^rw[], rack : ^rw[] in
        create^Z B =
          let b = [B S DS] in
            ⟨DS⟩register![B rack]
            | rack?[] → iflocal ⟨A⟩pack![] then
                currentloc![S DS] | [[P]]_B
        in
          let b = [B S DS] in
            pack?[] → (currentloc![S DS] | [[Q]]_A)

[[migrate to s → P]]_A
  def= currentloc?[S DS] →
        let [U DU] = s in
          if [S DS] = [U DU] then
            currentloc![U DU] | [[P]]_A
          else
            new mack : ^rw[] in
              ⟨DS⟩migrating![A mack]
              | mack?[] → migrate to U →
                new rack : ^rw[] in
                  ⟨DU⟩register![A rack]
                  | rack?[] → new ack : ^rw[] in
                    ⟨DS@S⟩migrated![A [U DU] ack]
                    | ack?[] → (currentloc!s | [[P]]_A)

[[iflocal ⟨b⟩c!v then P else Q]]_A
  def= let [B _ _] = b in
          iflocal ⟨B⟩c!v then [[P]]_A else [[Q]]_A
```

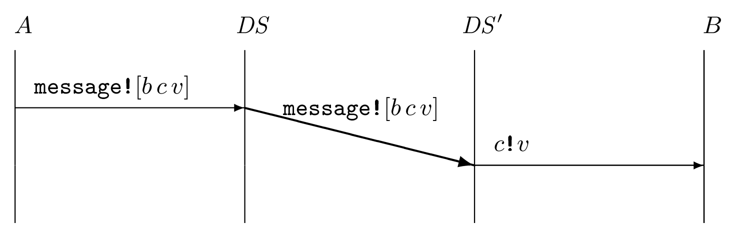
Fig. 5. A forwarding-pointers translation: the compositional encoding (selected clauses).

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; that is, either this site/daemon, or the site/daemon to which it migrated from here. The use of agent locks makes this algorithm rather more concurrent than the previous one; rather than simply sequentializing 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 $\langle b@?\rangle c\,\mathtt{!}v$ in agent A is implemented by requesting the local daemon to deliver it. (Note that A cannot migrate away before the request is sent to the daemon and a lock on `currentloc` is released.)

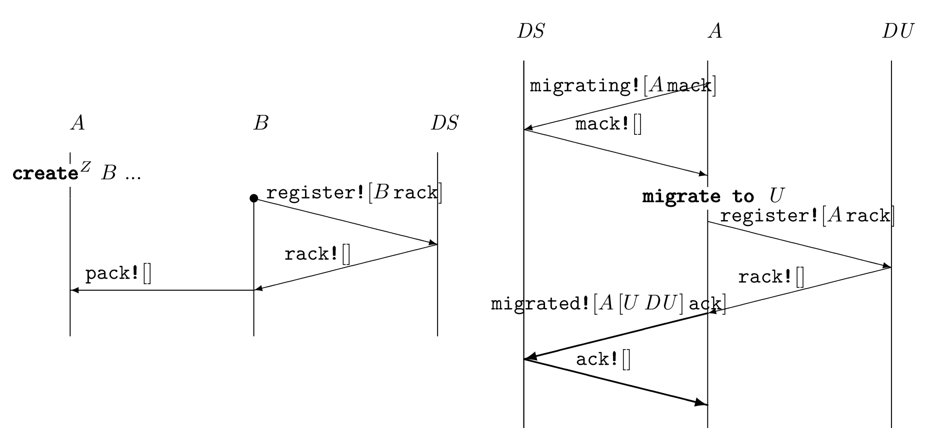
The message replicated input of the daemon gets the map m, from agent names to agent lock channels. If the destination agent B is not found, the message is forwarded to the daemon DU on the site U where B was created. Otherwise, if B is found, the agent lock Bstate is grabbed, obtaining the forwarding 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; it also prevents deadlock. In particular, in order to complete any migration of B the daemon should be able to process message migrated that requires to acquire lock.

A single location-independent output, forwarded once between daemons (if the target agent is not at the local site), involves inter-agent messages as shown next. (Communications that are guaranteed to be between agents on the same site are drawn with thin arrows.)



Creation. The compositional encoding for **create**^{Z} is similar to that of the encoding in Section 3. 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 acknowledgment (on dack in the simple translation) is likewise unnecessary.

A single creation involves inter-agent messages as on the left in the following diagram.



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 nondegenerate migration now synchronizes with the daemon DS on its starting site S, then migrates, registers with the daemon DU on its destination site U, then synchronizes again with DS. In between the first and last synchronizations 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 on the right in the preceding diagram.

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

6. EXAMPLE INFRASTRUCTURE: QUERY SERVER WITH CACHING ALGORITHM

In this final example we take a further step towards a realistic algorithm, demonstrating that nontrivial optimizations can be cleanly expressed within the Nomadic Pict framework.

The central forwarding server described in Section 3 is a bottleneck for all agent communication; further, all application messages must make two hops (and these messages are presumably the main source of network load). The forwarding-pointers algorithm described in Section 5 removes the bottleneck, but there application messages may have to make many hops, even in the common case. Adapting the central forwarding server so as to reduce the number of application-message hops required, we have the central query server algorithm first described in Section 4. It has a server that records the current site of every agent; agents synchronize with it on migrations. In addition, each site has a daemon. An application message is sent to the local daemon, which then queries the server to discover the site of the target agent; the message is then sent to the daemon on the target site. If the agent has migrated away, a notification is returned to the original daemon to try again. In the common case application messages will here take only one hop. The obvious defect is the large number of control messages between daemons and the server; to reduce these each site's daemon can maintain a cache of location data. The *Query Server with Caching (QSC)* [Wojciechowski and Sewell 2000] does this. When a daemon receives a misdelivered message, for an agent that has left its site, the message is forwarded to the server. The server both forwards the message on to the agent's current site and sends a cache-update message to the originating daemon. In the common case application messages are therefore delivered in only one hop.

The QSC encoding in Appendix C makes the algorithm precise, reusing the main design patterns from the encodings of Sections 4 and 3. Each class of agents maintains some explicit state as an output on a lock channel. The query server maintains a map from each agent name to the site (and daemon) where the agent is currently located. This is kept accurate when agents are created or migrate. Each daemon maintains a map from some agent names to the site (and daemon) that they guess the agent is located at. This is updated only when a message delivery fails. The encoding of each high-level agent records its current site (and daemon).

The algorithm is very asynchronous and should have good performance, with most application-level messages delivered in a single hop and none taking more than three hops (though 5 messages). The query server is involved only between a migration and the time at which all relevant daemons receive a cache update; this should be a short interval. Some additional optimizations are feasible, such as updating the daemon's cache more frequently.

The algorithm does, however, depend on reliable machines. The query server has critical state; the daemons do not, and so in principle could be reinstalled after a site crash, but it is only possible to reboot a machine when no other daemons have pointers (that they will use) to it. In a refined version of the protocol the daemons and the query server would use a store-and-forward protocol to deliver all messages reliably in spite of failures, and the query server would be replicated. In order to extend collaboration between clusters of domains (e.g., over a wide-area network), a federated architecture of interconnected servers must be adopted. In order to avoid long hops, the agents should register and unregister with the local query server on changing domains (see Wojciechowski [2006] for an example algorithm: the Federated Query Server with Caching).

7. NOMADIC PICT: THE PROGRAMMING LANGUAGE AND ITS IMPLEMENTATION

Nomadic Pict is a prototype distributed programming language, based on the Nomadic π calculus of Section 2 and on the Pict language of Pierce and Turner [2000]. Pict is a concurrent, though not distributed, language based on the asynchronous π calculus [Milner et al. 1992]. It supports fine-grain concurrency and the communication of asynchronous messages, extending the π calculus with a rich type system, a range of convenient forms for programming (such as function declarations) that can be compiled down to π calculus, and various libraries.

Low-level Nomadic Pict adds the Nomadic π calculus primitives for programming mobile computations from Section 2: agent creation, migration of agents between sites, and communication of location-dependent asynchronous messages between agents. In addition to these, Nomadic Pict adds timeouts, a facility for initiating communication between separate programs with a trader for type dynamic values, and labeled variant types. 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. The rest of the language is taken directly from Pict, with the front-end of the Nomadic Pict compiler based on the Pict compiler.

The language inherits a rich type system from Pict, including simple record types, higher-order polymorphism, simple recursive types, and subtyping. It has a partial type inference algorithm, and many type annotations can in practice be inferred by the compiler.

Names play a key rôle in the Nomadic Pict language, as in Nomadic π. New names of agents and channels can be created dynamically. These names are *pure*, in the sense of Needham [1989]; no information about their creation is visible within the language (in our current implementation they do contain site IDs, but could equally well be implemented by choosing large random numbers). Site names contain an IP address and TCP port number of the runtime system which they represent. Channel, agent, and site names are first-class values and they can be freely sent to processes which are located at other agents. As in the π calculus, names can be scope-extruded.

Programs in high-level Nomadic Pict are compiled in the same way as they are formally specified, by translating the high-level program into the low-level language. That in turn is compiled to a core language executed by the runtime. The core language is architecture-independent; its constructs correspond approximately to those of the low-level Nomadic π calculus, extended with value types and system function calls. The runtime system executes in steps, in each of which the closure of the agent at the front of the *agent queue* is executed for a fixed number of interactions. An agent closure consists of a *run queue*, of Nomadic π process/environment pairs waiting to be scheduled (round-robin), *channel queues* of terms that are blocked on internal or inter-agent communication, and an environment that records bindings of variables to channels and basic values. The process at the front of the run queue is evaluated according to the abstract machine designed for Pict [Turner 1996]. It ensures fair execution of the fine-grain parallelism in the language. The compiler and runtime are written in OCaml [Leroy 1995].

In Appendix D we give a more detailed overview of the language. To make the article self-contained, we include both the Nomadic-Pict-specific features and some aspects of Pict. We also describe some useful syntactic sugar and distributed programming programming idioms, such as Remote Procedure Calls (RPC) and distributed objects. The language implementation is described in Appendix E. For concreteness, the full syntax of the language is included as Appendix F. The implementation is available online, together with a tutorial, library documentation, and examples [Wojciechowski 2000a].

8. CORRECTNESS: NOMADIC π CALCULUS SEMANTIC DEFINITION

We now return to the calculus of Section 2. This section defines its semantics—the type system and operational semantics—and gives the basic metatheoretic results. Section 9 develops proof techniques over the semantics, which are then used in Section 10 to prove correctness of the Central Forwarding Server algorithm we gave in Section 3. Throughout we give outline proofs, highlighting the main points, and refer the reader to the Ph.D. thesis of Unyapoth [2001] for full details.

8.1 Type System

The type system is based on a standard simply typed π calculus, with channels carrying (possibly tuple-structured) first-order values. This is extended with input/output subtyping, as in Pierce and Sangiorgi [1996]: channel types have capabilities `r` (only input is allowed), `w` (output only), or `rw` (both), with `r` covariant and `w` contravariant. Additionally, the type of agent names has a capability m or s, with $\texttt{Agent}^{\texttt{s}} \leq \texttt{Agent}^{\texttt{m}}$; only the latter supports migration. There is a standard subsumption rule

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

The main judgements are $\Gamma \vdash_a P$, for well-formedness of a basic process as part of agent a, and $\Gamma \vdash LP$, for well-formedness of located processes. There is also a judgement $\Gamma \vdash x@z$, taking the location z of x from a located type context Γ. Sometimes we use unlocated type contexts, also written Γ, and there are standard rules for pattern and expression formation. The typing rules are given in full in Appendix A; a few of the most interesting rules are as follows.

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

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

The system also includes type variables and existential packages, deconstructed by pattern matching.

A type context is *extensible* if all term variables are of agent or channel types, and therefore may be new-bound.

8.2 Reduction Semantics

The reduction semantics was introduced informally in Section 2.5. Its formal definition involves structural congruence relations $P \equiv Q$ and $LP \equiv LQ$, defined in Appendix B.1, and a reduction relation $\Gamma \Vdash LP \rightarrow \Gamma' \Vdash LP'$ over pairs of located type contexts and located processes, defined in Appendix B.2.

8.3 Labeled Transition Semantics

The reduction semantics describes only the internal behavior of complete systems of located processes; for compositional reasoning we need also a typed labeled transition semantics, expressing how processes can interact with their environment. This lifts the development of corresponding reduction and labeled transition semantics in the π calculus [Milner 1992] to Nomadic π. Transitions are defined inductively on process structure, without the structural congruence. The transition relations have the following forms, for basic and located process:

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

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

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

$$\frac{}{\Gamma \Vdash_a \ \mathtt{migrate\ to}\ s \rightarrow P \xrightarrow{\mathsf{migrate\ to}\ s} @_a P}$$

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

Fig. 6. Selected LTS rules.

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

τ	internal computation
migrate to s	migrate to the site s
$c\,\mathtt{!}\,v$	send value v along channel c
$c\,\mathtt{?}\,v$	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 6, and the full definition in Appendix B.3.

Adding migrate to s to the standard input/output and τ labels is an important design choice, made for the following reasons.

—Consider a located process *LP* in some program context. If an agent a in *LP* migrates, the location context is consequently updated with a associated to its new site. This change of location context has an effect on both *LP* and its environment, since it can alter their execution paths (especially those involving location-dependent communication with a). Migration of an agent must therefore be thought of as a form of interaction with the environment.

—We observe, in the reduction rules, that the location context in the configuration *after* the transition can only be modified by migration of an agent. Including this migrating action allows the location context on the right-hand side to be omitted.

Execution of other agent primitives (i.e., **`create`** and **`iflocal`**) is regarded as internal computation, since it does not have an immediate effect on program contexts. In the case of **`create`**, the newly created agent remains unknown to the environment unless its name is extruded by an output action.

8.4 Basic Metatheory

In a typed semantics, the type system should prevent a mismatch between the value received and the shape expected in communication. However, matching a value and a pattern of the same type does not always yield a substitution. For example, taking Γ to be $x : [[]\ []]$, a pattern $[y\ z]$ may have type $[[]\ []]$ with respect to Γ, but $\mathsf{match}([y\ z], x)$ is undefined. A similar situation occurs when matching a name x of an existential type to an existential pattern $\{X\}\, p$. To prevent this, we define *ground* and *closed* type contexts as follows.

Definition 8.1 (*Ground Type Context*). A type context Γ is ground if, for all $x \in \mathsf{dom}(\Gamma)$, $\Gamma \vdash x \in T$ implies $T \neq [T_1 \ldots T_n]$ and $T \neq \{X\}\, S$, for any $T_1, \ldots, T_n, X, S$.

Definition 8.2 (*Closed Type Context*). A type context Γ is closed if it is ground and $\mathsf{fv}\,(\Gamma) \cap \mathcal{TV} = \emptyset$ and, for all $x \in \mathsf{dom}(\Gamma)$, $\Gamma \vdash x \in T$ implies $T \notin \mathcal{T}$.

It is easy to show that each name declared in a closed type context is either a site, an agent, or a channel.

We may now state the type preservation result.

THEOREM 8.1 (TYPE PRESERVATION). *For any well-formed closed located type context* Γ*, if* $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$ *then* $\Gamma, \Delta \vdash LQ$.

PROOF (SKETCH). An induction on the derivations of $\Gamma \Vdash_a P \xrightarrow[\Delta]{\alpha} LP$ and $\Gamma \Vdash LP \xrightarrow[\Delta]{\beta} LQ$. Γ needs to be closed so that matching a pattern with a value of the same type always yields a type-preserving substitution, whenever the transition involving matching occurs. □

THEOREM 8.2 (REDUCTION/LTS CORRESPONDENCE). *For any well-formed located type context* Γ *and located process* LP *such that* $\Gamma \vdash LP$*, we have:* $\Gamma \Vdash LP \to \Gamma' \Vdash LQ$ *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* $\Gamma' = \Gamma \oplus a \mapsto s$.

PROOF (SKETCH). We need to show this in two parts: that a reduction implies a silent transition or a migrate action, and vice versa. Each of the two parts is shown by an induction on reduction/transition derivations. The case where the silent transition of LP is derived by the communication rule needs the following lemma, which can easily be proved by an induction on transition derivations.

LEMMA 8.3.

—*If* $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c\,\mathbf{!}\,v} LQ$ *then* $LP \equiv \mathbf{new}\ \Delta, \Xi\ \mathbf{in}\ (@_a c\,\mathbf{!}\,v \mid LP')$ *for some* Δ *and* LP'*. Moreover,* $LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ LP'$.
—*If* $\Gamma \Vdash LP \xrightarrow[\Xi]{@_a c\,\mathbf{?}\,v} LQ$ *then, for some* Δ*,* p *and* LP'*,* Q*, with* $\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Xi) = \emptyset$*, either:*

—$LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a c\mathbf{?}p \rightarrow Q \mid LP')$ *and*
$LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathsf{match}(p, v)Q) \mid LP')$; *or*
—$LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a \mathbf{*}c\mathbf{?}p \rightarrow Q \mid LP')$ *and*
$LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathsf{match}(p, v)Q) \mid @_a \mathbf{*}c\mathbf{?}p \rightarrow Q \mid LP')$.

—*If* $\Gamma \Vdash LP \xrightarrow{@_a\mathsf{migrate\ to}\ s} LQ$ *then*

$$LP \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a\mathbf{migrate\ to}\ s \rightarrow P \mid LP')$$

for some Δ *and* LP', P. *Moreover,* $LQ \equiv \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a P \mid LP')$.

As in Theorem 8.1, Γ needs to be closed so that matching a pattern with a value of the same type always yields a type-preserving substitution, whenever the transition involving matching occurs. □

The next two lemmas ensure the absence of two kinds of runtime errors: mismatching of values exchanged in channel communication, and nonevaluable expressions.

LEMMA 8.4 (RUNTIME SAFETY: CHANNELS). *Given that* Γ *is a closed type context, and* $(\Gamma, \Delta)(c) = \hat{}^I T$, *we have:*

(1) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a c\mathbf{!}v \mid LP)$ *then* $I \leq \mathtt{w}$;
(2) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a c\mathbf{?}p \rightarrow P \mid LP)$ *then* $I \leq \mathtt{r}$;
(3) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a \mathbf{*}c\mathbf{?}p \rightarrow P \mid LP)$ *then* $I \leq \mathtt{r}$;
(4) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(c\mathbf{!}v \mid c\mathbf{?}p \rightarrow P) \mid LP)$ *then* $\mathsf{match}(p, v)$ *is defined; and*
(5) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(c\mathbf{!}v \mid \mathbf{*}c\mathbf{?}p \rightarrow P) \mid LP)$ *then* $\mathsf{match}(p, v)$ *is defined.*

LEMMA 8.5 (RUNTIME SAFETY: EXPRESSIONS). *Given that* Γ *is a closed type context, we have:*

(1) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q) \mid LP)$ *then* $v \in \{\mathbf{true}, \mathbf{false}\}$;
(2) *if* $\Gamma \vdash \mathbf{new}\ \Delta\ \mathbf{in}\ (@_a(\mathbf{let}\ p = ev\ \mathbf{then}\ P) \mid LP)$ *then* $\mathtt{eval}(ev)$ *and* $\mathsf{match}(p, \mathtt{eval}(ev))$ *are defined.*

9. CORRECTNESS: NOMADIC π CALCULUS SEMANTIC TECHNIQUES

In this section we describe the Nomadic π techniques used for stating and proving correctness. This is not specific to the particular CFS algorithm, although examples are taken from it. The next section describes the large-scale structure of the correctness proof, using these techniques.

Correctness statement. We are expressing distributed infrastructure algorithms as encodings from a high-level language to its low-level fragment, so the behavior 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

$$\forall P\ .\ P \simeq \mathcal{C}[\![P]\!] \qquad (\dagger)$$

where P ranges over well-typed programs of the high-level language (P may use location-independent 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 [de Nicola and Hennessy 1984] and is easier to work with; see also the discussion of Sewell [1997] on the choice of an appropriate equivalence for a Pict-like language. However, as in Nestmann and Pierce's work on choice encodings [1996], (†) would not hold, as the encodings $\mathcal{C}[\![P]\!]$ tend to involve *partial commitment* of some nondeterministic choices. In particular, migration steps and acquisitions of the daemon or agent locks involve nondeterministic internal choices, and lead to partially committed states: target-level terms which are not bisimilar to any source-level term. We therefore take $\simeq$ to be an adaptation of *coupled simulation* [Parrow and Sjödin 1992] 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 (again modulo questions of divergence and fairness). This is discussed further in Section 9.1.

Dealing with house-keeping steps. 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 the partial commitment steps just mentioned. Many, however, are deterministic *house-keeping* steps; they can be reduced to certain normal forms, and related to them by *expansions* (defined in Section 9.3). For example, consider the following fragment of code from the $\mathcal{C}$-encoding (after some reduction steps).

$$
\begin{array}{l}
\mathbf{new}\ \Phi_{aux}, m : \texttt{Map[Agent}^{\texttt{s}}\ \texttt{Site]}, \Delta\ \mathbf{in} \\
\quad @_D(\mathit{Daemon} \\
\qquad\quad \mid \mathbf{lookup}[\texttt{Agent}^{\texttt{s}}\ \texttt{Site}]\ a\ \mathbf{in}\ m\ \mathbf{with} \\
\qquad\qquad \mathbf{found}(s) \rightarrow \mathbf{new}\ \texttt{dack} : \texttt{\^{}}^{\texttt{rw}}[]\ \mathbf{in} \\
\qquad\qquad\quad \langle a@s\rangle \texttt{deliver!}\{X\}\ [c\ v\ \texttt{dack}] \mid \texttt{dack?}[] \rightarrow \texttt{lock!}m \\
\qquad\qquad \mathbf{notfound} \rightarrow \mathbf{0}) \\
\quad \mid @_a([\![P]\!]_a \mid \texttt{Deliverer} \mid \ldots) \\
\quad \mid @_{b_1}([\![Q_1]\!]_{b_1} \mid \ldots) \mid \ldots \mid @_{b_n}([\![Q_n]\!]_{b_n} \mid \ldots) \\
\text{where}\quad \texttt{Deliverer} \stackrel{\text{def}}{=} \texttt{*deliver?}\{X\}\ [c\ v\ \texttt{dack}] \rightarrow \big(\langle D@SD\rangle \texttt{dack!}[] \mid c\,\texttt{!}v\big)
\end{array}
$$

This is a state of the encoded whole system in which an agent has sent a message forwarding request (to agent a) to the daemon, and the daemon's request code has acquired the daemon lock, which contains the site map m. The subsequent steps performed by the daemon D, and by the `Deliverer` process in the agent a, are house-keeping steps. They include the map lookup operation, sending the message to the `Deliverer` process in a (with a location-dependent message to channel `deliver` there), and communication along the `dack` channel.

To prove these give rise to expansions requires a variety of techniques, some novel and some straightforward adaptations of earlier work.

—*Maps*. We use a π calculus encoding of finite maps, similar to the encoding of lists with persistent values [Milner 1993]. We prove that the encoding is correct, and that map lookup and update operations yield expansions.

— The location-dependent `deliver` message, sent to agent a, is guaranteed to arrive because a cannot migrate until the daemon lock is released by `lock!`m, which does not occur until agent a returns a `dack` to the daemon. To capture this, we introduce a notion of *temporarily immobile* located process: one in which no migration can take place until an input on a lock channel. This is discussed in Section 9.5.
  Certain reductions, such as the location-dependent message delivery step, are *deterministic*, as defined in Section 9.4. The key property of a temporarily immobile process is that such deterministic reductions still give rise to expansions when in parallel with temporarily immobile processes.
  Proving that processes are temporarily immobile involves a coinductive characterization and preservation results (under parallel and new-binders).
— The reaction of the `deliver` message and the `Deliverer` process, in agent a, is essentially functional. We adapt the notion of uniform receptiveness [Sangiorgi 1999], showing that the reaction induces an expansion by showing that the `deliver` channel is uniformly receptive: it always has a single replicated input in each agent, and no other input. The details are omitted here.
— The location-dependent `dack` message, from agent a to the daemon, is guaranteed to arrive for the simple reason that the daemon cannot migrate; it has the static type $\texttt{Agent}^{\texttt{s}}$. The reduction step is therefore deterministic, and hence induces an expansion.
— The `dack` acknowledgement channel is fresh for each request, so the daemon contains exactly one input and at most one output. It is straightforward to show that the communication is deterministic and hence gives an expansion.
— In all of the preceding techniques, we make essential use of congruence results for expansion to pull out the interesting part of the process, allowing the b_i agents and parts of the daemon to be neglected. The presence of agent mobility and location-dependent communication means these results must take account of the possible migrations of agents; in Section 9.2 we define *translocating* bisimulations that do so; translocating expansions are similar.

9.1 Partial Commitment and Coupled Simulation

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

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

Assuming a and b are initially at the same site, parts of the reduction graphs of LP and $\mathcal{C}[\![LP]\!]$ can be represented as in Figure 7. If the **migrate to** s_1 process in $\mathcal{C}[\![LP]\!]$ successfully acquires the local lock (a partial commitment step) the resulting process (LQ_{1p} in Figure 7) does not correspond exactly to any state of LP. LQ_{1p} cannot correspond to LP_1 since executing $\langle b\rangle c\,\texttt{!}\,v$ at this point means that $c\,\texttt{!}\,v$ will reach b (which is not the case for node LP_1); it cannot correspond to LP either, since we know that a will eventually end up in s_2.

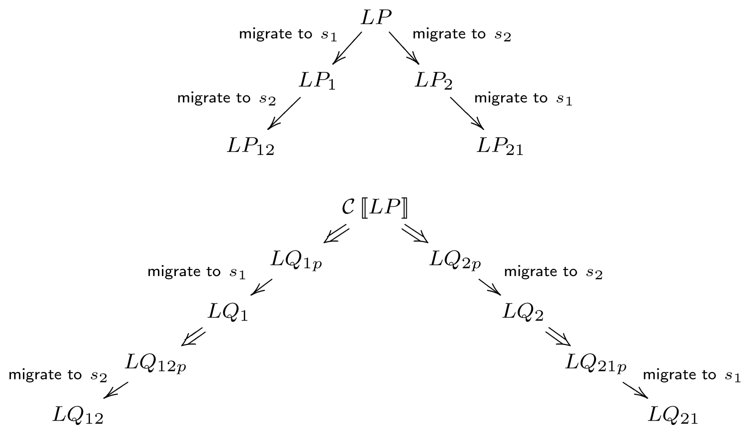


Fig. 7. An example of partial committed state.

To address this phenomenon, coupled simulation [Parrow and Sjödin 1992] 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 (the standard coinductive notion of simulation, indexed by located type contexts).
—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* with respect to Γ, written $LP \leftrightharpoons_\Gamma LQ$, 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*, that is, where *LP* coupled simulates LQ'.

In this article, 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 previous definition.

9.2 Translocating Equivalences and Congruence Result

To prove our main result (†) 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 behavior of location-dependent communications depends on the relative location of agents: if a and b are at the same site then the location-dependent message $@_b\langle a@s\rangle c\,!\,v$ reduces to (and in fact is weakly equivalent to) the local output $@_a c\,!\,v$, whereas if they are at different sites then the location-dependent message is weakly equivalent to **0**. A parallel context, for example

$[.]|@_a$**`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.

Relocators, ranged over by δ, can be applied to located type contexts in order to relocate agents in such contexts. A valid relocator for (Γ, M) is a type-respecting partial function from M to site names of Γ, formally defined next.

Definition 9.1 (*Valid Relocators*). A relocator δ is said to be *valid for* (Γ, M) if $\mathsf{dom}(\delta) \subseteq M$ and, for all $x \in M$, $\Gamma \vdash x \in \mathtt{Agent}^{\mathtt{m}}$ and $\Gamma \vdash \delta(x) \in \mathtt{Site}$.

We write $\Gamma\delta$ for the result of applying δ to Γ and $\Gamma\delta\beta$ for $(\Gamma\delta)\beta$.

Allowing arbitrary relocations would give too strong a notion, though. We introduce *translocating* relations that are parameterized by a set of agents that the environment may move.

Definition 9.2 (*Translocating Indexed Relation*). A *translocating indexed relation* is a binary relation between $\mathrm{n}\pi_{\mathsf{LD,LI}}$ processes, indexed by closed well-formed located type contexts Γ and sets $M \subseteq \mathsf{mov}(\Gamma)$, where $\mathsf{mov}(\Gamma)$ is the set of names of type $\mathtt{Agent}^{\mathtt{m}}$ in Γ:

$$\begin{aligned} \mathsf{mov}(\bullet) &\stackrel{\mathrm{def}}{=} \emptyset \\ \mathsf{mov}(\Gamma,\ X) &\stackrel{\mathrm{def}}{=} \mathsf{mov}(\Gamma) \\ \mathsf{mov}(\Gamma,\ x : T@z) &\stackrel{\mathrm{def}}{=} \begin{cases} \mathsf{mov}(\Gamma) \cup \{x\} & T = \mathtt{Agent}^{\mathtt{m}} \\ \mathsf{mov}(\Gamma) & \text{otherwise} \end{cases} \end{aligned}$$

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.

Therefore the translocating index of the bisimulation only needs to be updated when an input action occurs. For this we define the set $M_1 \uplus_\beta M_2$ to be $M_1 \cup M_2$ whenever β is an input, and to be M_1 otherwise.

$$M_1 \uplus_\beta M_2 \stackrel{\mathrm{def}}{=} \begin{cases} M_1 \cup M_2 & \exists a, c, v,\ \beta = @_a c\,\mathbf{?}v \\ M_1 & \text{otherwise} \end{cases}$$

The notion of translocating bisimulation can therefore be formalized as follows.

Definition 9.3 (*Translocating Simulations*).

(1) A translocating indexed relation $\mathcal{S}$ on $\mathrm{n}\pi_{\mathsf{LD,LI}}$ is a *translocating strong simulation* if $(LP, LQ) \in \mathcal{S}_\Gamma^M$ implies the following:
—$\Gamma \vdash LP$ and $\Gamma \vdash LQ$;
—$M \subseteq \mathsf{mov}(\Gamma)$; and

—For any relocator δ valid for (Γ, M), if $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ then there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ and $(LP', LQ') \in \mathcal{S}^{M \uplus_\beta \mathsf{mov}(\Delta)}_{\Gamma\delta\beta,\Delta}$.

$\mathcal{S}$ is called a *translocating strong bisimulation* if all of its indexed relations are symmetric. Two located processes LP and LQ are translocating strongly bisimilar with respect to Γ, M, written $LP \dot{\sim}^M_\Gamma LQ$, if there exists a translocating strong bisimulation which when indexed by Γ and M, contains the pair (LP, LQ).

(2) Replacing $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\beta} LQ'$ in the final item of this definition with $\Gamma\delta \Vdash LQ \xRightarrow[\Delta]{\hat{\beta}} LQ'$ yields the *weak* version of translocating simulation. A located process LQ weak translocating bisimulates LP with respect to Γ, M, denoted $LP \dot{\approx}^M_\Gamma LQ$, if there exists a weak translocating bisimulation which when indexed by Γ, M, contains the pair (LP, LQ).

Some simple examples of translocating bisimulations are the following.

$$@_a\texttt{iflocal}\ \langle b\rangle c\,\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q\ \dot{\sim}^{M_1}_\Gamma\ @_a\texttt{iflocal}\ \langle b\rangle c\,\texttt{!}v\ \texttt{then}\ P\ \texttt{else}\ Q'$$
$$@_a\langle b@s\rangle c\,\texttt{!}v\ \dot{\approx}^{M_2}_\Gamma\ @_b c\,\texttt{!}v$$

where $M_1 \subseteq \mathsf{mov}(\Gamma)/\{a, b\}$ and $M_2 \subseteq \mathsf{mov}(\Gamma)/\{b\}$; we assume that the preceding processes are well-typed with respect to Γ, and that $\Gamma \vdash a@s$ and $\Gamma \vdash b@s$.

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

THEOREM 9.1 (TRANSLOCATING CONGRUENCE). *Given a closed located type context Γ, Θ with Θ extensible, if*

—$LP \dot{\sim}^{M_P}_{\Gamma,\Theta} LP'$ *and* $LQ \dot{\sim}^{M_Q}_{\Gamma,\Theta} LQ'$,
—$\mathsf{mayMove}(LQ, LQ') \subseteq M_P$,
—$\mathsf{mayMove}(LP, LP') \subseteq M_Q$, *and*
—$M \stackrel{def}{=} M_P \cap M_Q \cap \mathsf{agents}(\Gamma)$

then

$$\texttt{new}\ \Theta\ \texttt{in}\ (LP \mid LQ) \dot{\sim}^M_\Gamma\ \texttt{new}\ \Theta\ \texttt{in}\ (LP' \mid LQ').$$

PROOF (SKETCH). The proof deals with derivatives of **new** Θ **in** $LP \mid LQ$ with respect to Γ, which have the general form of

$$LR_k\ =\ \texttt{new}\ \Theta, \Theta_{comm}\ \texttt{in}\ (LP_k \mid LQ_k)$$

well-typed with respect to $\Gamma, \Theta_{in}, \Theta_{out}$. Here we classify new names bound in the derivative, and those extruded to or from the environment as follows.

—Θ_{comm} consists of names exchanged by communication between LP and LQ. This can be classified further as Θ^{LP}_{comm}, the private names of LP extruded by output actions to LQ, and vice versa for Θ^{LQ}_{comm}.

—Θ_{out} consists of names extruded by output actions to the environment. Again, this can be classified further as Θ_{out}^{LP}, for the names extruded by LP, and vice versa for Θ_{out}^{LQ}.

—Θ_{in} consists of names received from the environment.

Using this classification of names, the set $\mathsf{mov}(\Theta_{in})$ anticipates the movements of agents received from the environment (i.e., the context of LR_k), and the set $M_P \cup \mathsf{mov}(\Theta_{comm}^{LQ}, \Theta_{out}^{LQ})$ anticipates the movements of free agents in LQ_k. Since the environment of LP_k comprises LQ_k and the context of LR_k as a whole, the translocating index of the bisimulation relations between LP_k and LP'_k must include the following set.

$$M_{P_k} = M_P \cup \mathsf{mov}\left(\Theta_{comm}^{LQ}, \Theta_{out}^{LQ}, \Theta_{in}\right)$$

The premises of Theorem 9.1 can therefore be generalized in the coinduction as follows.

—$LP_k \dot{\sim}^{M_{P_k}}_{\Gamma,\Theta_{in},\Theta_{out},\Theta_{comm},\Theta} LP'_k$, and $LQ_k \dot{\sim}^{M_{Q_k}}_{\Gamma,\Theta_{in},\Theta_{out},\Theta_{comm},\Theta} LQ'_k$, where M_{Q_k} is defined in the similar way as M_{P_k};

—$\mathsf{mayMove}(LP_k, LP'_k) \subseteq M_Q \cup \mathsf{mov}(\Theta_{comm}^{LP}, \Theta_{out}^{LP})$; and

—$\mathsf{mayMove}(LQ_k, LQ'_k) \subseteq M_P \cup \mathsf{mov}(\Theta_{comm}^{LQ}, \Theta_{out}^{LQ})$.

The proof of this theorem relies on the invariance under labeled transitions of the previous premises. □

By using the techniques outlined in the beginning of Section 9, we may prove that

```
new Φ_aux, m : Map[Agent^s Site] in
  @_D(Daemon
      | 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)
  | @_a([[P]]_a | Deliverer | ...)
```

$$\dot{\approx}^{\{b_1,\ldots,b_n\}}_{\Gamma,\Delta}$$

```
new Φ_aux, m : Map[Agent^s Site] in
  @_D(Daemon | lock!m)
  | @_a([[P]]_a | Deliverer | ...)
```

where the processes above are well-typed with respect to Γ, Δ. Applying the congruence result, the fragment of code from the $\mathcal{C}$-encoding given in the beginning of this section can be proved to translocating weak bisimulate the following process.

```
new Φ_aux, m : Map[Agent^s Site], Δ in
  @_D(Daemon | lock!m)
  | @_a([[P]]_a | Deliverer | ...)
  | @_b1([[Q_1]]_b1 | ...) | ... | @_bn([[Q_n]]_bn | ...)
```

9.3 Expansion

To construct the coupled simulation, we use an *expansion* relation $\dot{\succeq}$ [Nestmann and Pierce 1996] and the "up to" technique of Sangiorgi and Milner [1992], adapted with translocation, to allow elimination of target processes that are in intermediate/house-keeping stages.

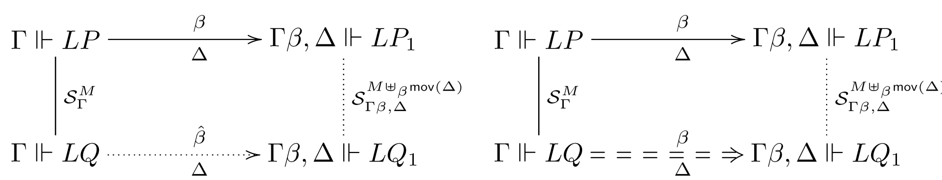
A definition of expansion uses two refinements of weak simulation: progressing and strict simulation. We adapt the definitions from Nestmann [1996], adding type contexts and translocation.

Definition 9.4 (*Progressing and Strict Simulation*). A weak translocating simulation $\mathcal{S}$ is called

—*strict* if, for all $(LP, LQ) \in \mathcal{S}_\Gamma^M$ and valid δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ implies there exists LQ' such that $\Gamma\delta \Vdash LQ \xrightarrow[\Delta]{\hat{\beta}} LQ'$ with $(LP', LQ') \in \mathcal{S}_{\Gamma\delta\beta,\Delta}^{M \uplus_\beta \mathsf{mov}(\Delta)}$;

—*progressing* if, for all $(LP, LQ) \in \mathcal{S}_\Gamma^M$ and valid δ for (Γ, M), $\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LP'$ implies there exists LQ' such that $\Gamma\delta \Vdash LQ \overset{\hat{\beta}}{\underset{\Delta}{\Longrightarrow}} LQ'$ with $(LP', LQ') \in \mathcal{S}_{\Gamma\delta\beta,\Delta}^{M \uplus_\beta \mathsf{mov}(\Delta)}$.

LQ is said to *progressing simulate* (or *strictly simulate*) LP with respect to Γ, M if there exists a progressing simulation $\mathcal{S}$ (or a strict simulation) such that $(LP, LQ) \in \mathcal{S}_\Gamma^M$.



(a) strict simulation (b) progressing simulation

The preceding diagrams show progressing and strict simulations. Informally, LQ strictly simulates LP means that LQ weakly simulates LP, but LQ never introduces more internal steps and may ignore the silent transitions of LP. On the other hand, LQ progressing simulates LP means that LQ weakly simulates LP, but LQ introduces more internal steps and never ignores a silent action, hence the absence of ˆ in the weak transition of LQ in the definition. The definition of expansion simply makes use of these two refinements.

Definition 9.5 (*Expansion*). An indexed binary relation $\mathcal{S}$ is a *translocating expansion* if $\mathcal{S}$ is a strict simulation and $\mathcal{S}^{-1}$ is a progressing simulation.

LP translocating expands LQ with respect to Γ under M, written $LP \dot{\succeq}_\Gamma^M LQ$, if there exists an expansion $\mathcal{S}$ with $(LP, LQ) \in \mathcal{S}_\Gamma^M$. Moreover, if $LP \dot{\succeq}_\Gamma^{\mathsf{mov}(\Gamma)} LQ$ then LP and LQ are said to be related by *expansion congruence*, written $LP \succeq_\Gamma LQ$.

We depend on a congruence result, analogous to that earlier, for expansion. The proof of this result is similar to that for translocating bisimulations.

9.4 Deterministic Reduction

A component in a system of concurrent processes may be deterministic, in the sense that its next computational step can be determined. An example of this is a location-dependent message $\langle b@s\rangle c\,\texttt{!}\,v$, executed in an agent a; if the agent b is static, and is located at s, then all the subsequent transitions are determined, eventually moving the output $c\,\texttt{!}\,v$ to b. We define deterministic reduction as follows.

Definition 9.6 (*Deterministic Reduction*). Given a closed located type context Γ and $M \subseteq \mathsf{mov}(\Gamma)$, a located processes LP is said to *deterministically reduce* to LQ with respect to (Γ, M), written $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$, if, for any valid δ for (Γ, M), the following hold:

—$\Gamma\delta \Vdash LP \xrightarrow{\tau} LQ$; and
—$\Gamma\delta \Vdash LP \xrightarrow[\Delta]{\beta} LQ'$ implies $\beta = \tau$, $\Delta = \bullet$ and $LQ' \dot{\sim}^M_{\Gamma\delta} LQ$.

We also define the relation $\overset{\mathsf{det}}{\underset{M}{\Longrightarrow}}$ to be the transitive closure of $\xrightarrow[M]{\mathsf{det}}$; that is $\Gamma \Vdash LP \overset{\mathsf{det}}{\underset{M}{\Longrightarrow}} LQ$ implies there exists $LP_1, \ldots, LP_n$ such that, letting $LP = LP_0$ and $LQ = LP_{n+1}$, we have

$$\Gamma \Vdash LP_i \xrightarrow[M]{\mathsf{det}} LP_{i+1} \qquad 0 \leq i \leq n$$

A process LP is said to be τ-deterministic with respect to Γ, M if there exists LQ such that $\Gamma \Vdash LP \overset{\mathsf{det}}{\underset{M}{\Longrightarrow}} LQ$.

The next lemma states the key property of τ-determinacy: that a deterministic reduction induces an expansion.

LEMMA 9.2 (DETERMINISTIC REDUCTION INDUCES EXPANSION). *If* $\Gamma \Vdash LP \xrightarrow[M]{\mathsf{det}} LQ$ *then* $LP \dot{\succeq}^M_\Gamma LQ$.

From the example fragment of code from $\mathcal{C}$-encoding, when the agent a received the message forwarded from the daemon, it sends an acknowledgment back to the daemon using $\langle D@SD\rangle\texttt{dack!}[]$. Since the location-dependent sugar output is τ-deterministic, we have:

$$@_a\langle D@SD\rangle\texttt{dack!}[] \;\dot{\succeq}^M_\Gamma\; @_D\texttt{dack!}[]$$

for any $M \subseteq \mathsf{mov}(\Gamma)$ such that $D \notin M$. However, since the daemon D is static, $@_a\langle D@SD\rangle\texttt{dack!}[]$ is related by expansion congruence to $@_D\texttt{dack!}[]$; and hence placing the location-dependent output in any program context yields an expansion.

9.5 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 acknowledgment 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 9.7 (*Translocating Path*). A *translocating path* of LP_0 with respect to (Γ, M) is a sequence

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

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

—δ_i is a valid relocator for $(\hat{\Gamma}, \hat{M})$, where

$$\begin{aligned}\hat{\Gamma} &\stackrel{\text{def}}{=} \Gamma, \Delta_1, \dots, \Delta_i \\ \hat{M} &\stackrel{\text{def}}{=} M \uplus_{\beta_1} \mathsf{mov}(\Delta_1) \dots \uplus_{\beta_i} \mathsf{mov}(\Delta_i), \text{ and}\end{aligned}$$

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

Definition 9.8 (*Temporary Immobility*). Given a closed located type context Γ, a located process LP with $\Gamma \vdash LP$, and a translocating index $M \subseteq \mathsf{agents}(\Gamma)$, LP is *temporarily immobile* under lock l with respect to (Γ, M) if, for all translocating paths

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

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

—$\beta_i = @_b c\,\mathtt{!}\,v$ implies $l \notin \mathsf{fv}\,(\beta_i)$; and
—$\beta_i \neq @_b$ migrate to s.

Consider, for example, the next process.

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

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, that is, with $a \notin M$. Assume further that a is at s and that the daemon is forwarding an LI message to a, that is, the preceding process is in parallel with

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

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

$$\begin{array}{l}\textbf{new}\ \texttt{lock} : \texttt{\^{}}^{\texttt{rw}}\texttt{Map[Agent}^{\texttt{s}}\ \texttt{Site]}\ \textbf{in} \\ \quad LQ \mid @_a\texttt{deliver!}[c\ v\ \texttt{ack}]\end{array}$$

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 9.3. *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*

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

Proofs of temporary immobility can be hard, since they involve quantification over derivatives. We formulate a coinductive definition of temporary immobility (which is equivalent to the one given here). A process is temporarily immobile if it belongs to a blocking set: a set which is closed under transitions that are not inputs on the lock channel, and in which no migration can occur. This alternative definition allows temporary immobility to be proved by analyzing single step transitions. Moreover, since temporary immobility is preserved by weak bisimulation, we may apply "up to" techniques [Sangiorgi and Milner 1992], so that we may work with sets which are a blocking set when closed up under weak bisimulation. Proving that the process LQ given before is temporarily immobile, for example, involves analyzing its transitions, which can be classified into two groups.

—*Local computation*, execution of the process P in a, which does not involve the daemon. The result of this type of transition is in the same form as LQ.
—*Daemon computation*, execution of the process P in a, which involves the daemon. The result of this type of transition expands a process which is of the same form as LQ. (Sending location-dependent message to the static daemon, for example, induces expansion.)

Temporary immobility is preserved by parallel composition and **new** binding. This can be used for proving that the following process is temporarily immobile.

$$\begin{array}{rcl} LR & = & \textbf{new}\ \Omega_{aux}\ \textbf{in} \\ & & \quad @_D Daemon \\ & & \quad \mid @_{b_1}(\llbracket P_1 \rrbracket_{b_1} \mid \texttt{currentloc!}s_1 \mid \texttt{Deliverer}) \mid \ldots \\ & & \quad \mid @_{b_n}(\llbracket P_n \rrbracket_{b_n} \mid \texttt{currentloc!}s_n \mid \texttt{Deliverer}) \end{array}$$

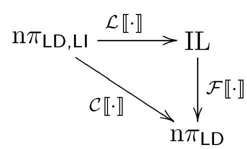
Since LR is strongly bisimilar to $LQ_1 \mid \ldots \mid LQ_n$, where LQ_i is obtained from LQ by replacing the name of the agent a by b_i and the process P by P_i. The proof of the strong bisimulation uses a result similar to a proposal of Milner [1993, p. 29].

10. CORRECTNESS: PROOF FOR THE CENTRAL FORWARDING SERVER

This section outlines the strategies taken in order to prove the correctness of the example CFS encoding $\mathcal{C}[\![\cdot]\!]$, defined in Section 3, using the techniques from Section 9.

10.1 Factoring the Proof

We simplify the construction of the main coupled simulation (between an arbitrary source program, in $\mathrm{n}\pi_{\mathsf{LD,LI}}$, and its encoding, in $\mathrm{n}\pi_{\mathsf{LD}}$) by factoring the encoding through an *intermediate language* IL, with states ranged over by *Sys*, that is specific to this encoding. The infrastructure encoding $\mathcal{C}[\![\cdot]\!]$ 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 $\mathcal{F}$, mapping systems in the intermediate language to their corresponding target terms.



In proving correctness of the loading encoding, we essentially deal with all the house-keeping steps, relating terms introduced by such steps to some normal forms. Such normal forms allow house-keeping steps to be abstracted away, so that in proving correctness of the unloading encoding, we can concentrate on relating partially committed terms to target-level terms. 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);
(2) dealing with states in which many agents may be partially committed simultaneously; and
(3) capturing some invariants, for example, that the daemon's site-map is correct, in a type system for IL.

The cost is that the typing and labeled transition rules for IL must be defined. For lack of space we only outline the essential points here, referring the reader again to Unyapoth [2001] for the full development.

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.

$$\mathrm{n}\pi_{\mathsf{LD,LI}} \underset{\mathcal{D}^\sharp[\![\cdot]\!]}{\overset{\mathcal{D}^\flat[\![\cdot]\!]}{\longleftarrow}} \mathrm{IL}$$

It suffices to have both functions commit creations and LI messages, as these are somewhat confluent.

We shall not define the loading, unloading, and decoding functions here. Instead we illustrate the correspondence between steps in the source, intermediate, and the target languages in the creation, migration, and location-independent messaging cases in Figure 8. In the figure, some τ communication steps are annotated with the command or the name of the channel involved. The figure also shows how partially committed states are mapped to terms in the source language by the decoding functions.

10.2 Intermediate Language

Each term of the intermediate language 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:

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

Each term $\mathsf{eProg}(\Delta; \boldsymbol{D}; \boldsymbol{A})$ is parameterized 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 as follows.

— 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 $c\,\texttt{!}\,v$ 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 next.

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

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 $\texttt{create}^Z \ b = P \ \texttt{in} \ Q$ (when its state is $\mathsf{RegA}(b \ Z \ s \ P \ Q)$) or $\texttt{migrate to} \ s \rightarrow P$ (when its state is $\mathsf{MtingA}(s \ P)$ or $\mathsf{MrdyA}(s \ P)$). 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 $\mathrm{n}\pi_{\mathsf{LD,LI}}$

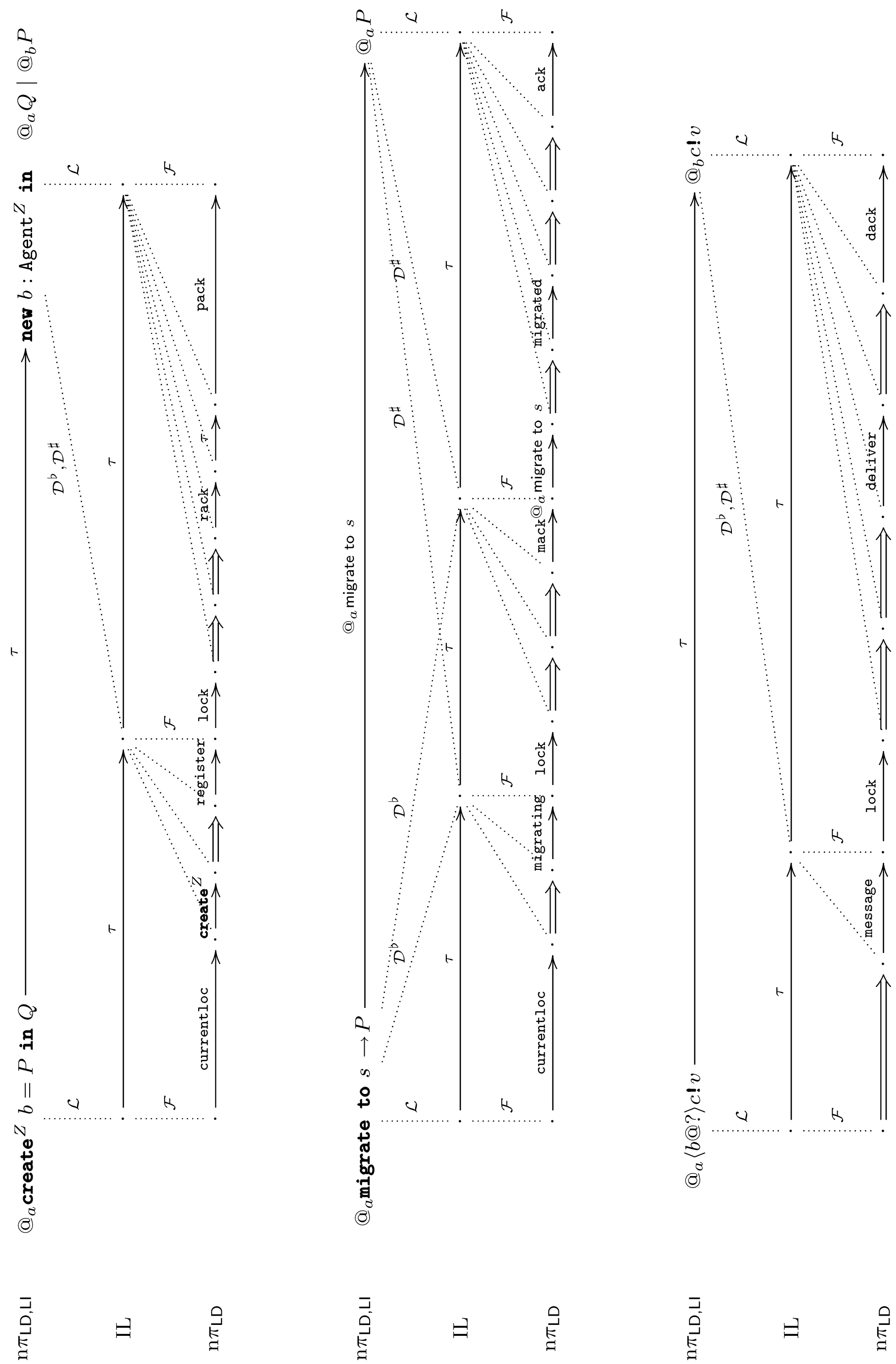


Fig. 8. Relationships between source, intermediate, and target.

LTS rules) is placed in parallel with other processes in the main body, except for execution of an LI output $\langle b \rangle c\,\texttt{!}v$, 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 $\texttt{create}^Z\ b = P\ \texttt{in}\ Q$ or $\texttt{migrate to}\ s \to P$ from the main body of a may proceed (in fact initiate) if the local lock is free, that is, 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 acknowledgment to b. After completion, the declaration $b : \texttt{Agent}^Z@s$ 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. The top third of Figure 8 gives the correspondences between steps in the source, intermediate, and 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 acknowledgment from b. After completion, the message $c\,\texttt{!}v$ 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, initializing the request and turning the pending state of a to $\mathsf{MrdyA}(s\ P)$. 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 acknowledgment, permitting a to migrate. The second step corresponds to a migrating to s and sending an acknowledgment back to the daemon, which updates its site map and then sends the final acknowledgment to a, allowing it to proceed.

10.3 Proof Outline

Note that our encoding is not *uniform* [Palamidessi 1997], as it introduces a centralized 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.

The main lemmas can now be stated.

LEMMA 10.1 (SYNTACTIC FACTORIZATION). *For any LP well-typed with respect to a valid system context* Φ

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

This follows from the definitions of the encoding and decoding functions.

LEMMA 10.2 (SEMANTIC CORRECTNESS OF IL). *For any Sys well-formed with respect to* Φ, $\mathcal{F}[\![Sys]\!] \dot{\succeq}^{\emptyset}_{\Phi} Sys$.

This lemma is the heart of the correctness argument. The proof uses expansion up to expansion to relate each well-formed term in the intermediate language with its corresponding target term. We use the techniques of Section 9; part of the reasoning for the LI message-delivery case was outlined there. In broad, we heavily employ the congruence properties of translocating expansion for factoring out program contexts which are not involved in house-keeping reductions of the target terms. Temporary immobility is used whenever we need to guarantee that location-dependent 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 10.3 ($\mathcal{D}^\flat$ IS A STRICT SIMULATION). *For any Sys well-formed for* 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 10.4 ($\mathcal{D}^{\sharp^{-1}}$ IS A PROGRESSING SIMULATION). *For any Sys well-formed with respect to* 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 two lemmas are proved by direct constructions of simulation relations. The analysis of possible transitions is made feasible by the factoring out of housekeeping steps.

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

THEOREM 10.5 (ENCODING CORRECTNESS). *For any LP well-formed with respect to a valid system context* Φ, $LP \fallingdotseq {}_\Phi\mathcal{C}_\Phi[\![LP]\!]$.

PROOF. The proof puts together the operational correspondence results developed earlier, as can be summarized in the next diagram.

$$
\begin{array}{llcl}
\mathrm{n}\pi_{\mathsf{LD,LI}} & LP & \xrightarrow[(10.1)]{\equiv} & \mathcal{D}\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!] \\
 & & & \fallingdotseq_\Phi \Big| \;(10.3,10.4) \\
\mathrm{IL} & \fallingdotseq_\Phi \;\vdots & & \mathcal{L}_\Phi\,[\![LP]\!] \\
 & & & \dot{\succeq}^{\emptyset}_{\Phi} \Big| \;(10.2) \\
\mathrm{n}\pi_{\mathsf{LD}} & \mathcal{C}_\Phi\,[\![LP]\!] & \xrightarrow[(10.1)]{\equiv} & \mathcal{F}\,[\![\mathcal{L}_\Phi\,[\![LP]\!]]\!]
\end{array}
$$

□

11. RELATED WORK

A range of work on mobility from different perspectives (process migration within a cluster, mobile computing, and wide-area migration in mobile agent languages) is surveyed in the collection edited by Milojičić et al. [1999].

The direct precursors of our work on Nomadic Pict were programming languages closely based on process calculi. The collection of Nielson [1997] describes CML, FACILE, LCS, and the Poly/ML concurrency primitives, all of which draw on channel-based communication as in Milner's CCS [1989]. With the exception of FACILE, these are focused on local concurrency, without support for distributed programming. Milner [1992] and Milner et al. [1992], generalized CCS to the π calculus, allowing channel names to be themselves sent over channels, and with an elegant operational semantics for fresh generation of new channel names. The π calculus is small but very expressive, allowing data structures, functions, objects, locks, and other constructs of sequential and concurrent programming to be encoded with asynchronous message-passing. This was demonstrated in the Pict language of [Pierce and Turner 2000; Turner 1996], which was an experiment in building a concurrent (but again not distributed) programming language based closely on the π calculus, by analogy to the development of functional programming languages such as ML and Haskell above the λ-calculus.

The distributed join-calculus of Fournet et al. [1996] aimed to redesign the π calculus to make a better foundation for distributed programming, as developed in the subsequent JoCaml programming language [Conchon and Le Fessant 1999]. The distributed join-calculus ensures syntactically that there is a unique receiver for each channel, and then regains expressivity by allowing receivers to synchronize on multiple messages. It also distributes processes over a hierarchical structure of abstract locations, which may be freshly generated and which may migrate to different points in the hierarchy. In implementations one can think of the first level of this hierarchy as physical machines, with lower levels as migratable running computations. Implementations had an elaborate overlay network built-in, with forwarding pointer chains (as in our algorithm of Section 5) and mechanisms to collapse those chains. The hidden complexity of this algorithm, and the fact that its behavior under failure had either to be exposed to the programmer or concealed by a high-level semantics in which reconnection was prohibited, was the immediate spur for our development of the lower level of abstraction of low-level Nomadic Pict, in which the semantics under failure is clear and in which one can see and analyze the design of such higher-level algorithms.

The π calculus is an attractive starting point for calculi for distributed computation, from its clear treatment of concurrency, the elegant treatment of names, and the similarity between π asynchronous message passing and asynchronous network communication.[1] This led to a wide variety of distributed process calculi, adding notions of distribution, locality, mobility, and security. Some parts of the rather large design space are surveyed in Sewell [2000] and Cardelli [1999], and we mention a few prominent examples.

—The early π_l calculus of Amadio and Prasad [1994], used for modeling the notions of locality and failure presented in the programming language FACILE [Thomsen et al. 1996].
—The dpi of Sewell [1998], used for studying a type system in which the input and output capabilities of channels may be either global or local.
—The Seal calculus of Vitek and Castagna [1998] and Castagna, Vitek, and Zappa Nardelli [2005] intended as a framework for writing secure distributed applications over large-scale open networks such as the Internet, and the Box-π calculus of Sewell and Vitek [2003], used for studying *wrappers*: secure environments that provide fine-grain control of the allowable interaction between them, and between components and other system resources.
—The various Dπ calculi of Riely and Hennessy [1998] and Hennessy [2007] used for studying *partially typed* semantics, designed for mobile agents in open distributed systems in which some sites may harbor malicious intentions. These typically address code mobility but not computation mobility, with a focus on type-based enforcement of desirable properties.
—The Ambient calculus of Cardelli and Gordon [1998], a calculus for describing the movement of processes and devices, including movement through administrative domains. This prompted further work on semantics, such as Merro and Zappa Nardelli [2005], and several variant calculi.
—The extension of TyCO with distribution and code mobility [Vasconcelos et al. 1998], a name-passing process calculus which allows asynchronous communication between concurrent objects via labeled messages carrying names.

These systems address a variety of distributed-systems problems with semantically well-founded approaches, generally focusing on the dynamics of interaction (as one would expect from their process-calculus origins) and in some cases on type systems. One can also take a more logical view, as in the P2 system [Loo et al. 2005] and SD3 [Jim 2001], both of which describe distributed algorithms declaratively. Murphy [2008] develops a language based on a Curry-Howard

[1]In practice, one would typically implement π-style asynchronous messaging above TCP connections, not UDP, as UDP does not provide retransmission and has a fixed upper-bound datagram size. In the absence of migration, such an implementation would provide a FIFO property that is not reflected in the π calculus semantics.

correspondence for a modal logic, focussing on type-theoretic guarantees that mobile code will never access resources that are not present at the current site.

There are many related programming languages, not based on a π calculus semantics but supporting some form of mobility, including Kali Scheme [Cejtin et al. 1995], Obliq [Cardelli 1995], and Mozart [Van Roy and Haridi 2004].

As for mobility at the virtual machine level, Xen live migration [Clark et al. 2005] deals with the special case of migration over a single switched LAN. In that setting, one can arrange for the migrating VM to carry its IP address with it, with an unsolicited ARP reply. This results in the loss of some in-flight IP packets, but (as higher-level protocols such as TCP must be resilient to such loss in any case) the migration is essentially transparent. Migration in the wide-area setting, without additional support from the IP layer, would presumably need overlay networks of the kind we describe, though perhaps a TCP-connection-based approach would be a better fit to applications than the asynchronous messages that we consider here.

Turning now to verification, Moreau [2002, 2001] develops a fault-tolerant directory service and message routing algorithm, based on forwarding pointers, and verifies the correctness of the abstract algorithm (mechanised in Coq). Verification of mobile communication infrastructures has also been considered in the Mobile UNITY setting, by McCann and Roman [1997]. There is, of course, a great deal of other work on verification of distributed algorithms in general, and on proof techniques for π calculi. Roughly speaking, the verification and proof techniques of process calculi can be classified as those based on types and those based on the dynamic behavior of processes. A type system for the π calculus was first proposed by Milner [1993], giving the notions of *sort* and *sortings*, which ensure uniformity of the kind of names that can be sent or received by channels. Many refinements on the type system have subsequently been proposed, including polymorphism [Turner 1996; Pierce and Sangiorgi 1997], subtyping [Pierce and Sangiorgi 1996], linear types [Kobayashi et al. 1996], objects [Walker 1995], and a generic type system [Igarashi and Kobayashi 2001]. Adding the notions of locality and distribution to the π calculus admits further refinements to be made. Sewell [1998] formulated *dpi* for studying a type system where each channel is located at an agent and can be given global/local usage capability as well as that for input/output. An approximation to the join-style of interaction, for example, can be obtained by giving them global-output and local-input capabilities. This type system retains the expressiveness of channel communication, yet admits optimization at compile time. Yoshida and Hennessy [1999] formulated a type system for $D\pi\lambda$ which emulates the join-style of interaction using input/output subtyping. The presence of higher-order processes makes this formulation challenging. The type system of $D\pi_1^r$ extends the concept of uniform receptiveness [Sangiorgi 1999] to ensure that each output (perhaps an inter-agent message) is guaranteed to react with a (unique) input process at its destination. The techniques of refining channel types are also used in ensuring security-related properties. For example, the partial typing of Riely and Hennessy [1999] ensures that resources of trusted sites are not abused by untrusted sites; Sewell and Vitek [2003] introduced causality types for reasoning about information flow between security domains; and Cardelli et al. [2000] introduced a notion of *groups* which can be used for ensuring that the boundary of an ambient may only be dissolved by trusted groups of ambients.

The behavioral theories of these distributed variants of process calculi are generally adapted from those of the π calculus, which are based around operational semantics and operational equivalences. A reduction semantics is given for all of the cited calculi. This, together with some notions of barbs, allows a definition of barbed bisimulation to be given, as is the case for the Distributed Join-calculus [Fournet and Gonthier 1996], the Seal calculus [Castagna and Vitek 1999], and the Ambient calculus [Gordon and Cardelli 1999]. A labeled transition semantics is also given for the π_{1l}, $\mathrm{D}\pi$, $\mathrm{D}\pi_1^{\mathrm{r}}$, Ambient, Seal, and Box π calculi, allowing some notions of bisimilarity to be given. These definitions of labeled transition semantics often involve refining that of the standard π with location annotation ($@_l$ for $\mathrm{D}\pi_{\mathrm{r}}^1$ and "relative location" tags for Seal and Box-π). The labelled transition semantics of $\mathrm{D}\pi$ [Riely and Hennessy 1998] extends the standard π input and output actions with labels that indicate movements and failures of locations. The style of transition systems for the Ambient calculus [Gordon and Cardelli 1999; Merro and Nardelli 2005] is quite different from those for π calculus, as they involve relative locations of ambients. The definitions of labeled transition semantics and operational equivalences of distributed CCS [Riely and Hennessy 1997] and π_{1l} [Amadio 1997] also take location failures into account.

Several authors have used process-calculus proof techniques to verify the correctness of implementations or abstract machines for various ambient calculi. Fournet et al. [2000] give a translation of Ambients into the Join calculus. As in our central forwarding server proof, they build an intermediate language to capture intermediate states of the translation and use coupled simulations, though they work in a barbed reduction-semantics setting rather than the labeled transition setting we adopt. They also describe an implementation in JoCaml based on this translation, though with some significant differences. Giannini et al. [2006] give an abstract machine (PAN) for Safe Ambients [Levi and Sangiorgi 2000], a restricted calculus in which ambient movement depends on agreement between both parties, and ambients are either single-threaded or immobile. This is rather different from Nomadic π, in which an agent can migrate to another site at any time. They prove the abstract machine has the same barbs as the source. Hirschkoff et al. [2007] refine this abstract machine, optimizing the treatment of forwarders, and prove it weakly bisimilar to PAN. They also describe an OCaml implementation loosely based on their abstract machine.

The work on Nomadic Pict described in the current article led to two substantial subsequent lines of research. Firstly, in a production language, one would like to express high-level abstractions such as that of high-level Nomadic Pict using a general-purpose module system rather than the special-purpose encodings of the Nomadic Pict implementation. An ML-style module system [Milner et al. 1997] is a good fit for this: one can express the high- and low-level abstractions as signatures, with abstract types of site name, agent name, etc., and operations to send messages, migrate, etc., and express a particular overlay network implementation as a functor from one to the other. However, when one imagines this in a wide-area setting, it quickly becomes obvious that one will need multiple different overlay network implementations, and that they will inevitably exist in multiple simultaneous versions. This observation prompted work on type equality for abstract types in the distributed setting [Sewell 2001; Leifer et al. 2003; Sewell et al. 2005, 2007; Billings et al. 2006; Deniélou and Leifer 2006], and the Acute and HashCaml prototype languages. The former provides a slightly lower level of abstraction than low-level Nomadic Pict: instead of migration, it has a primitive for freezing a group of threads into a thunk (together with support for modules, versions, etc.). This makes it possible to implement low-level Nomadic Pict itself as an Acute module [Sewell et al. 2007, Section 11], and high-level Nomadic Pict overlays could be implemented as further modules above that.

Secondly, recall that the low-level Nomadic Pict abstraction was designed to be implementable with a clear semantics in the presence of failure (site failure, message loss, or disconnection): each low-level reduction step is implementable with at most one asynchronous inter-site message. Later work took this further, characterizing the exact semantics (including failure cases) not for simple asynchronous messages, but instead for the communication primitives provided by the Sockets API to the UDP and TCP protocols [Serjantov et al. 2001; Wansbrough et al. 2002; Bishop et al. 2005, 2006; Ridge et al. 2008. Work by Compton [2005] (above that UDP model) and Ridge [2009] (above a simplified TCP model) demonstrates that it is feasible to verify, fully formally, executable distributed code above such models.

12. CONCLUSION

We have studied the overlay networks required for communication between mobile computations. By expressing such distributed algorithms as Nomadic Pict encodings, between carefully chosen (and well-defined) levels of abstraction, we have descriptions of them that are

- —executable: one can rapidly prototype the algorithms, and applications written above them in the high-level language;
- —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.

We discussed the design space of possible algorithms, and implemented a programming language that lets the algorithms (and applications above them) be executed. We developed semantics and proof techniques for proving correctness of such algorithms. The techniques were illustrated by a proof that an example algorithm is correct with respect to coupled simulation. This algorithm, though nontrivial, is relatively simple, but we believe that more sophisticated algorithms could be dealt with using the same techniques (albeit with new intermediate languages, tailored to particular algorithms).

More generally, the work is a step towards semantically-founded engineering of wide-area distributed systems. Here we dealt with the combination of migration and communication, and for a complete treatment one must also simultaneously address failure and malicious attack.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The initial work on the Nomadic Pict project was in collaboration with Benjamin C. Pierce [Sewell et al. 1998, 1999]. We also owe special thanks to Benjamin and to David N. Turner for allowing us to use the source-code of Pict, and thank Robin Milner and Ken Moody for their support.

REFERENCES

AMADIO, R. M. 1997. An asynchronous model of locality, failure, and process mobility. In *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*. Lecture Notes in Computer Science, vol. 1282. Springer, 374–391.

AMADIO, R. M. AND PRASAD, S. 1994. Localities and failures (extended abstract). In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'94)*. Lecture Notes in Computer Science, vol. 880. Springer, 205–216.

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press.

ARNOLD, K., WOLLRATH, A., O'SULLIVAN, B., SCHEIFLER, R., AND WALDO, J. 1999. *The Jini Specification*. Addison-Wesley, Reading, MA.

AWERBUCH, B. AND PELEG, D. 1995. Online tracking of mobile users. *J. ACM 42*, 5, 1021–1058.

BALLINTIJN, G., VAN STEEN, M., AND TANENBAUM, A. 1999. Simple crash recovery in a wide-area location service. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing Systems (PDCS'99)*. IASTED, 87–93.

BILLINGS, J., SEWELL, P., SHINWELL, M., AND STRNIŠA, R. 2006. Type-Safe distributed programming for OCaml. In *Proceedings of the ACM SIGPLAN Workshop on ML (ML'06)*. ACM, New York, 20–31.

BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'05)*. ACM, New York, 265–276.

BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. 2006. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. ACM, New York, 55–66.

CARDELLI, L. 1995. A language with distributed scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM, New York, 286–297.

CARDELLI, L. 1999. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Lecture Notes in Computer Science, vol. 1603, J. Vitek and C. D. Jensen, Eds. State-of-the-Art Survey. Springer, 51–94.

CARDELLI, L., GHELLI, G., AND GORDON, A. D. 2000. Ambient groups and mobility types. In *Proceedings of the 16th IFIP International Conference on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics (TCS'00)*. Lecture Notes in Computer Science, vol. 1872. Springer, 333–347.

CARDELLI, L. AND GORDON, A. D. 1998. Mobile ambients. In *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structure (FoSSaCS'98)*. Lecture Notes in Computer Science, vol. 1378. Springer, 140–155.

CARDELLI, L., GORDON, A. D., AND GHELLI, G. 1999. Mobility types for mobile ambients. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*. Lecture Notes in Computer Science, vol. 1644. Springer, 230–239.

CASTAGNA, G. AND VITEK, J. 1999. Commitment and confinement for the Seal calculus. Trusted objects, Centre Universitaire d'Informatique, University of Geneva.

CASTAGNA, G., VITEK, J., AND ZAPPA NARDELLI, F. 2005. The Seal calculus. *Inform. Comput. 201*, 1, 1–54.

CEJTIN, H., JAGANNATHAN, S., AND KELSEY, R. 1995. Higher-Order distributed objects. *ACM Trans. Program. Lang. Syst. 17*, 5, 704–739.

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2, 225–267.

CHERITON, D. 1988. The V distributed system. *Comm. ACM 31*, 3, 314–333.

CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. 2005. Live migration of virtual machines. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'05)*. 273–286.

COMPTON, M. 2005. Stenning's protocol implemented in UDP and verified in Isabelle. In *Proceedings of the Australasian Symposium on Theory of Computing (CATS'05)*. Australian Computer Society, Darlinghurst, Australia, 21–30.

CONCHON, S. AND LE FESSANT, F. 1999. Jocaml: Mobile agents for Objective-Caml. In *Proceedings of the 1st International Symposium on Agent Systems and Applications/3rd International Symposium on Mobile Agents (ASA/MA'99)*. IEEE Computer Society, 22–29.

DE NICOLA, R. AND HENNESSY, M. C. B. 1984. Testing equivalences for processes. *Theor. Comput. Sci. 34*, 1-2, 83–133.

DEMMER, M. J. AND HERLIHY, M. P. 1998. The arrow distributed directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98)*. Lecture Notes in Computer Science, vol. 1499. Springer, 119–133.

DENIÉLOU, P.-M. AND LEIFER, J. J. 2006. Abstraction preservation and subtyping in distributed languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM, New York, 286–297.

DOUGLIS, F. AND OUSTERHOUT, J. 1991. Transparent process migration: Design alternatives and the Sprite implementation. *Softw. Pract. Exper. 21*, 8, 757–785.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty processor. *J. ACM 32*, 2, 374–382.

FOURNET, C. AND GONTHIER, G. 1996. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM Press, New York, 372–385.

FOURNET, C., GONTHIER, G., LÉVY, J.-J., MARANGET, L., AND RÉMY, D. 1996. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*. Lecture Notes in Computer Science, vol. 1119. Springer, 406–421.

FOURNET, C., LÉVY, J.-J., AND SCHMITT, A. 2000. An asynchronous, distributed implementation of mobile ambients. In *Proceedings of the 16th IFIP International Conference on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics (TCS'00)*. Lecture Notes in Computer Science, vol. 1872. Springer, 348–364.

GIANNINI, P., SANGIORGI, D., AND VALENTE, A. 2006. Safe Ambients: Abstract machine and distributed implementation. *Sci. Comput. Program. 59*, 3, 209–249.

GORDON, A. D. AND CARDELLI, L. 1999. Equational properties of mobile ambients. In *Proceedings of the 2nd International Conference on Foundations of Software Science and Computation Structure (FoSSaCS'99)*. Lecture Notes in Computer Science, vol. 1578. Springer, 212–226.

GUERRAOUI, R. AND SCHIPER, A. 1996. Fault-Tolerance by replication in distributed systems. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'96)*. Lecture Notes in Computer Science, vol. 1088. Springer, 38–57.

HENNESSY, M. 2007. *A Distributed Pi-Calculus*. Cambridge University Press.

Hirschkoff, D., Pous, D., and Sangiorgi, D. 2007. An efficient abstract machine for Safe Ambients. *J. Logic Algebr. Program. 71*, 2, 114–149.

Igarashi, A. and Kobayashi, N. 2001. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01).* ACM, New York, 128–141.

Jim, T. 2001. SD3: A trust management system with certified evaluation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'01).* IEEE Computer Society, 106–115.

Jul, E., Levy, H., Hutchinson, N., and Black, A. 1988. Fine-Grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1, 109–133.

Kobayashi, N., Pierce, B. C., and Turner, D. N. 1996. Linearity and the pi-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96).* ACM, New York, 358–371.

Lange, D. B. and Aridor, Y. 1997. *Agent Transfer Protocol—ATP/0.1*. IBM Tokyo Research Laboratory.

Leifer, J. J., Peskine, G., Sewell, P., and Wansbrough, K. 2003. Global abstraction-safe marshalling with hash types. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03).* ACM, New York, 87–98.

Leroy, X. 1995. Le système Caml Special Light: Modules et compilation efficace en Caml. Tech. rep. RR-2721, INRIA, Institut National de Recherche en Informatique et en Automatique.

Levi, F. and Sangiorgi, D. 2000. Controlling interference in Ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*. ACM, New York, 352–364.

Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. 2005. Implementing declarative overlays. *ACM SIGOPS Oper. Syst. Rev. 39* 5, 75–90.

McCann, P. J. and Roman, G.-C. 1997. Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts. In *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97).* Lecture Notes in Computer Science, vol. 1282. Springer, 338–354.

Merro, M. and Zappa Nardelli, F. 2005. Behavioral theory for mobile ambients. *J. ACM 52*, 6, 961–1023.

Milner, R. 1989. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall.

Milner, R. 1992. Functions as processes. *J. Math. Struct. Comput. Sci. 2*, 2, 119–141.

Milner, R. 1993. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification,* F. L. Bauer, W. Brauer, and H. Schwichtenberg, Eds. Series F: Computer and System Sciences, vol. 94. NATO Advanced Study Institute, Springer.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, Parts I and II. *Inform. Comput. 100*, 1, 1–77.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (revised)*. The MIT Press.

Milojičić, D., Douglis, F., and Wheeler, R., Eds. 1999. *Mobility: Processes, Computers, and Agents.* Addison-Wesley, Reading, MA.

Moreau, L. 2001. Distributed directory service and message router for mobile agents. *Sci. Comput. Program. 39*, 2-3, 249–272.

Moreau, L. 2002. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC'02).* ACM, New York, 93–100.

Mullender, S. J. and Vitányi, P. M. B. 1988. Distributed match-making. *Algorithmica 3*, 367–391.

Murphy, VII, T. 2008. Modal types for mobile code. Ph.D. thesis, Tech. rep. CMU-CS-08-126, Carnegie Mellon University.

Needham, R. M. 1989. Names. In *Distributed Systems*, S. Mullender, Ed. Addison-Wesley, 89–101.

Nestmann, U. 1996. On determinacy and nondeterminacy in concurrent programming. Ph.D. thesis, Technische Fakultät, Universität Erlangen.

NESTMANN, U. AND PIERCE, B. C. 1996. Decoding choice encodings. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96).* Lecture Notes in Computer Science, vol. 1119. Springer, 179–194.

NIELSON, F., ED. 1997. *ML with Concurrency: Design, Analysis, Implementation, and Application*. Monographs in Computer Science. Springer.

PALAMIDESSI, C. 1997. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97).* ACM, New York, 256–265.

PARROW, J. AND SJÖDIN, P. 1992. Multiway synchronization verified with coupled simulation. In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR'92)*. Lecture Notes in Computer Science, vol. 630. Springer, 518–533.

PIERCE, B. C. AND SANGIORGI, D. 1996. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci. 6*, 5, 409–454.

PIERCE, B. C. AND SANGIORGI, D. 1997. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97).* ACM, New York, 242–255.

PIERCE, B. C. AND TURNER, D. N. 1995. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming: Proceedings of the International Workshop (TPPP'94),* T. Ito and A. Yonezawa, Eds. Lecture Notes in Computer Science, vol. 907. Springer, 187–215.

PIERCE, B. C. AND TURNER, D. N. 1997. *Pict Language Definition*. Available electronically as part of the Pict distribution. www.cis.upenn.edu/∼bcpirce/papers/pict/Html/Pict.html. (3/28/10).

PIERCE, B. C. AND TURNER, D. N. 2000. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. Foundations of Computing. MIT Press, 455–494.

POPEK, G. J. AND WALKER, B. J. 1986. *The LOCUS Distributed System Architecture*. Computer Systems Series. MIT Press, Cambridge, MA.

RIDGE, T. 2009. Verifying distributed systems: The operational approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL'09)*. ACM, New York, 429–440.

RIDGE, T., NORRISH, M., AND SEWELL, P. 2008. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*. Lecture Notes in Computer Science, vol. 5014. Springer, 294–309.

RIELY, J. AND HENNESSY, M. 1997. Distributed processes and location failures (extended abstract). In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97).* Lecture Notes in Computer Science, vol. 1256. Springer, 471–481.

RIELY, J. AND HENNESSY, M. 1998. A typed language for distributed mobile processes (extended abstract). In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98).* ACM, New York, 378–390.

RIELY, J. AND HENNESSY, M. 1999. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99).* ACM, New York, 93–104.

SANGIORGI, D. 1999. The name discipline of uniform receptiveness. *Theor. Comput. Sci. 221*, 1-2, 457–493.

SANGIORGI, D. AND MILNER, R. 1992. The problem of "weak bisimulation up to". In *Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR'92)*. Lecture Notes in Computer Science, vol. 630. Springer, 32–46.

SERJANTOV, A., SEWELL, P., AND WANSBROUGH, K. 2001. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (TACS'01).* Lecture Notes in Computer Science, vol. 2215. Springer, 535–559.

SEWELL, P. 1997. On implementations and semantics of a concurrent programming language. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*. Lecture Notes in Computer Science, vol. 1243. Springer, 391–405.

SEWELL, P. 1998. Global/Local subtyping and capability inference for a distributed pi-calculus. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*. Lecture Notes in Computer Science, vol. 1443. Springer, 695–706.

SEWELL, P. 2000. A brief introduction to applied π. Tech. rep. 498, Computer Laboratory, University of Cambridge, Cambridge, UK.

SEWELL, P. 2001. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, New York, 236–247.

SEWELL, P., LEIFER, J. J., WANSBROUGH, K., ZAPPA NARDELLI, F., ALLEN-WILLIAMS, M., HABOUZIT, P., AND VAFEIADIS, V. 2005. Acute: High-Level programming language design for distributed computation. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*. ACM, New York, 15–26.

SEWELL, P., LEIFER, J. J., WANSBROUGH, K., ZAPPA NARDELLI, F., ALLEN-WILLIAMS, M., HABOUZIT, P., AND VAFEIADIS, V. 2007. Acute: High-level programming language design for distributed computation. *J. Funct. Program. 17*, 4-5, 547–612.

SEWELL, P. AND VITEK, J. 2003. Secure composition of untrusted code: Box-π, wrappers and causality types. *J. Comput. Secur. 11*, 2, 135–188.

SEWELL, P. AND WOJCIECHOWSKI, P. T. 2008. Verifying overlay networks for relocatable computations (or: Nomadic Pict, relocated). In *Proceedings of the Joint HP-MSR Research Workshop on The Rise and Rise of the Declarative Datacentre*.
http: //research.microsoft.com/riseandrise (2/13/10).

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1998. Location independence for mobile agents. In *Proceedings of the Workshop on Internet Programming Languages (IFL'98), in conjunction with IEEE ICCL'98*. 1–6.

SEWELL, P., WOJCIECHOWSKI, P. T., AND PIERCE, B. C. 1999. Location-Independent communication for mobile agents: A two-level architecture. *Internet Programming Languages*. Lecture Notes in Computer Science, vol. 1686. Springer, 1–31.

THOMSEN, B., LETH, L., AND KUO, T.-M. 1996. A Facile tutorial. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*. Lecture Notes in Computer Science, vol. 1119. Springer, 278–298.

TURNER, D. N. 1996. The polymorphic pi-calculus: Theory and implementation. Ph.D. thesis, University of Edinburgh.

UNYAPOTH, A. 2001. Nomadic π-calculi: Expressing and verifying communication infrastructure for mobile computation. Ph.D. thesis, University of Cambridge. Also Tech. rep. UCAM-CL-TR-514, Computer Laboratory, University of Cambridge.

UNYAPOTH, A. AND SEWELL, P. 2001. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. ACM, New York, 116–127.

VAN ROY, P. AND HARIDI, S. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT Press.

VAN STEEN, M., HAUCK, F. J., BALLINTIJN, G., AND TANENBAUM, A. S. 1998. Algorithmic design of the Globe wide-area location service. *Comput. J. 41*, 5, 297–310.

VASCONCELOS, V. T., LOPES, L., AND SILVA, F. 1998. Distribution and mobility with lexical scoping in process calculi. In *Proceedings of the 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*. Electronic Notes in Theoretical Computer Science, vol. 16.3. Elsevier Science Publishers.

VITEK, J. AND CASTAGNA, G. 1998. Towards a calculus of secure mobile computations. In *Proceedings of the Workshop on Internet Programming Languages, in Conjunction with IEEE ICCL'98*.

WALKER, D. 1995. Objects in the π-calculus. *Inform. Comput. 116*, 2, 253–271.

WANSBROUGH, K., NORRISH, M., SEWELL, P., AND SERJANTOV, A. 2002. Timing UDP: Mechanized semantics for sockets, threads, and failures. In *Proceedings of the 11th European Symposium on Programming Languages and Systems (ESOP'02)*. Lecture Notes in Computer Science, vol. 2305. Springer, 278–294.

WOJCIECHOWSKI, P. T. 2000a. *Nomadic Pict. Documentation and User's Manual*. (2/13/10)

WOJCIECHOWSKI, P. T. 2000b. Nomadic Pict: Language and infrastructure design for mobile computation. Ph.D. thesis, University of Cambridge. Also Tech. rep. UCAM-CL-TR-492, Computer Laboratory, University of Cambridge.

WOJCIECHOWSKI, P. T. 2010. *The Nomadic Pict System*. http://www.cs.put.poznan.pl/pawelw/npict (2/13/10).

WOJCIECHOWSKI, P. T. 2001. Algorithms for location-independent communication between mobile agents. In *Proceedings of the AISB Symposium on Software Mobility and Adaptive Behaviour*.

WOJCIECHOWSKI, P. T. 2006. Scalable message routing for mobile software assistants. In *Proceedings of the 4th IFIP International Conference on Embedded and Ubiquitous Computing (EUC'06)*. Lecture Notes in Computer Science, vol. 4096. Springer, 355–364.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 1999. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of the 1st International Symposium on Agent Systems and Applications / 3rd International Symposium on Mobile Agents (ASA / MA'99)*. IEEE Computer Society.

WOJCIECHOWSKI, P. T. AND SEWELL, P. 2000. Nomadic Pict: Language and infrastructure design for mobile agents. IEEE Concurr. 8, 2, 42–52. *The 1st International Symposium on Agent Systems and Applications / 3rd International Symposium on Mobile Agents (ASA / MA'99)*.

YOSHIDA, N. AND HENNESSY, M. 1999. Subtyping and locality in distributed higher order processes (extended abstract). In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*. Lecture Notes in Computer Science, vol. 1664. Springer, 557–572.

Received December 2008; revised June 2009; accepted September 2009