

Seal: A Framework for Secure Mobile Computations

Jan Vitek¹ and Giuseppe Castagna²

¹ Object Systems Group, CUI, Université de Genève, Suisse

² C.N.R.S, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Paris, France

Abstract. The Seal calculus is a distributed process calculus with localities and mobility of computational entities called seals. Seal is also a framework for writing secure distributed applications over large scale open networks such as the Internet. This paper motivates our design choices, presents the syntax and reduction semantics of the calculus, and demonstrates its expressiveness by examples focused on security and management distributed systems.

1 Introduction

Advances in computer communications and computer hardware are changing the landscape of computing. Networking is now cheap and pervasive. The Internet has become a platform for large scale distributed programming. What is needed now is programming languages that support the development of Internet applications.

In the last couple of years a number of process calculi have been designed to model programming large scale distributed systems over open networks. Several of these calculi [12, 19, 34, 21] advocate programming models based on the notion of mobile computations. These mobile calculi give a semantics to programs structured as systems of communicating mobile computations also referred to as mobile agents.

We present a low level language called Seal which has been designed for distributed computing over large scale open networks such as the Internet. The language is based on a model of computing in which program mobility and resource access control are essential mechanisms. We introduce Seal as a fairly simple language, an extension of Milner's π -calculus, with the goal of being able to express the essential properties of Internet programs. We view Seal as a framework for exploring the design space of security and mobility features. For this reason rather than striving for minimality, Seal tries to express key features of Internet programming directly.

1.1 Internet design principles

The Seal language adheres to five design principles that are particularly suited to Internet programming. These principles are the following.

1.1.1 *No reliance on global state*

The physical size and number of hosts require solutions that scale to wide area networks of millions of nodes. At this size, the language in which programs are expressed can not afford to assume any shared state. Thus, algorithms that require synchronization over sets of machines or up-to-date information about the state of groups of processors are prohibited. For example, automatic memory management on the Internet would require synchronization of the whole system to collect all unreferenced objects, this means stopping the Internet for “a couple of instants” [29].

1.1.2 *Explicit localities*

Fluctuations in bandwidth, latency and reliability are so common that they can not be papered over in the language semantics [38]. The location where computation occurs and the location of its resources are essential to the efficiency and the fault tolerance of a distributed program. Locations should thus appear at the programming level and be under the program’s control.

1.1.3 *Restricted connectivity*

Failures of machines and communication links can occur without warning or detection [31]. Some of these failures may be temporary as machines may be restarted and connection reestablished, but others may be permanent. Furthermore, firewalls impose purposeful restriction on communication abilities of programs. This means that, at a given time, a computation may be able to communicate with only a subsets of the other entities on the network. At the extreme, a computation may have to operate disconnected.

1.1.4 *Dynamic configuration*

In an open network new hosts and communication links are added with no advance notice, hosts may disappear and even reappear under a new name and address. The topology both in the physical sense and in the logical sense of services and resources available is thus changing over time. Programming the Internet thus requires working in dynamically evolving environment. Ideally, the location of all elements of a distributed computation should be controlled by the computation itself, allowing it to adapt to changes in its environment.

1.1.5 *Access control*

Finally, security is the *conditio sine qua non* of every discussion of the Internet. Security requirements vary from application to application, policies must be tailored to the specific requirements of applications or application domains. At best, a language can provide mechanisms for protecting resources to facilitate the enforcement of policies.

1.2 **Models of Internet Programming**

These principles are a basis for designing a distributed programming language. Unfortunately, full-fledged computer languages are usually cluttered with seman-

tic noise coming from the many features not directly pertinent to distribution. We focus on a model of distributed programming as a miniature programming language composed of only the core features needed for mobile computations.

Distributed programming is an inherently concurrent activity, quite naturally concurrent language and distributed ones have common roots. The advantage of extending an existing concurrent model for distribution are that some of the theory can be reused as well. We consider three models of concurrency: *Linda*, *Actors*, and the π -calculus.

Linda is a coordination model; it specifies only how a community of concurrent, and possibly distributed, processes can communicate and synchronize their activities using a shared data structure called a tuple space [20]. Linda has been used successfully in small-scale configurations, but its scalability is limited by the synchronous nature of tuple space operations. In essence, the presence of single shared data structure contradicts our first design principle as it is a form of global state.

The Actor model described by Agha [3] presents a distributed computation as a group of named entities, actors, which communicate by asynchronous message passing. Communication is tied to knowledge of names; knowing the name of another actor suffices to be allowed to communicate with it. As names are values, name-passing models evolving patterns of cooperation amongst groups of actors. From our point of view, Actors are a step in the right direction as they forego the single shared data structure of Linda in favor of a message passing model which abides by our first design principle. Nevertheless, the calculus lacks a notion of locality: there are no distances between actors, nor are there means to restrict communication abilities.

The π -calculus [24], our last candidate, models concurrent computation by processes that exchange messages over named channels. Channel names play almost exactly the same role as actor names. Thus the π -calculus is also adequate to model evolving systems, but just as Actors, it lacks a suitable notion of location and has no direct means to model restricted connectivity. Two further features complicate a distributed interpretation of the π -calculus: synchronous message passing and choice, both imply synchronization consensus. Nevertheless, the π -calculus is attractive because of the solid theoretical basis developed over the years.

Further these three calculi lack resource access control mechanisms. They do not allow to model features like firewalls or sandbox commonly found in networks and programming languages.

Structure This paper is structured as follows. Section 2 presents our design choices and gives an informal introduction to the main features of the calculus. Section 3 details both syntax and operational semantics of the calculus. Section 4 presents some programming examples. Section 5 discusses related work. Finally, section 6 states some conclusions and outlines future directions of investigation.

2 The Seal Framework

This section outlines the main features of the Seal calculus, a distributed model of computing for large scale open networks. We briefly present our design choices, discuss the objectives of the work and present the abstractions of the framework.

Seal can be roughly described as the π -calculus with hierarchical locations, mobility, and resource access control. Unlike many distributed programming languages, our goal is not to provide a high-level programming model that eases distributed programming by hiding localities, but rather its goal is to expose the network and hand over control of localities and low-level protection to the system programmer. The means to this end are powerful mobility and protection primitives. Another view of Seal is as a substrate for implementing higher level languages and advanced distributed services. In this light the Seal takes on the role of lowest common denominator between various Internet applications. Sophisticated services that require higher degrees of coherence and synchronization can be built on top of it. Examples of such services are distributed memory management, location independent secure messaging, and channels with quality of service guarantees.

What Seal does provide is a model of mobility which subsumes message passing, remote evaluation as well as process migration and which models user mobility and hardware mobility. Furthermore, the framework provides a hierarchical protection model, in which each level of the hierarchy can implement security policies by mediation — actions performed at a lower level in the hierarchy are scrutinized and controlled by higher levels. The hierarchical model guarantees that a policy defined at some level will not be bypassed or amended at lower levels. When coupled with mediation this gives rise to programming styles that emphasize interposition techniques [26, 17, 33].

2.1 Abstractions

Seal unifies several concepts from distributed programming into three abstractions: **locations**, **processes** and **resources**. Locations are meant to stand for physical places such as those delimited by the boundaries of address spaces, host machines, routers, firewalls, local area networks or wide area networks. Locations also embody logical boundaries such as protection domains, sandboxes and applications. The process abstraction stands for any flow of control such as a thread or operating system process. Finally, resources unify physical resources such as memory locations and peripheral device interfaces with services such as those offered by other applications, the operating system or a runtime system.

We now review the main features of the model and at the same time introduce the concrete syntax of the Seal language by examples.

2.1.1 Names

Names denote two kinds of computational entities, seals and channels; they are also values and as such can be exchanged in communication. New names are

created by the restriction operator ($\nu _$), they are considered distinct from any other name. Thus the expression $(\nu x)P$ creates a fresh name x which can be used within the process P without fear of name clash with any other name.

2.1.2 Processes

In a process calculus every expression denotes a process — a computation — running in parallel with other processes. The simplest Seal expression is the inert process 0 , a process with no behavior. A process $\alpha . P$ is composed of an action α and a continuation process P ; this expression denotes a process waiting to perform α and then to behave like P . Actions consist of communication and moves and are explained later on. $P \mid Q$ denotes a process composed of two subprocesses, P and Q running in parallel. The replicated process $!P$ behaves like $P \mid !P$; it can be equivalently considered as a process that creates an infinite number of copies of P running in parallel. Finally a process can also be a location with a process body, that is a *seal*, as we will see next.

2.1.3 Locations

Seals are named, hierarchically-structured, locations. The expression $n[P]$ denotes a seal named n running process P . Since a seal encapsulate a process and a seal is also a process, then a seal can contain (the parallel composition of) several seals yielding a hierarchy of *subseals* of arbitrary depth. If P contains one or more seals, say $\vec{m} = m_1 \dots m_n$, then n is the parent of \vec{m} and \vec{m} are the children of n . The transitive closure of the set of children is the set of *subseals* of n . A configuration is depicted graphically in Figure 1 which shows an outermost seal that represents the network its children are hosts, and their subseals are instances of application programs. Of course in an implementation we would not program the network — we do not plan to actually control the flow of bits along

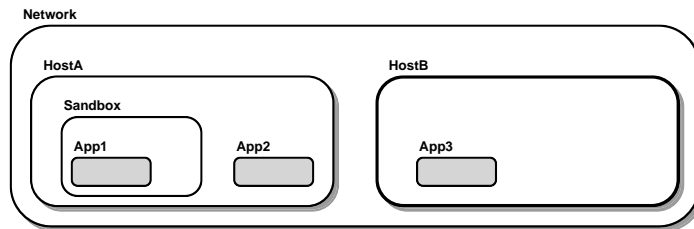


Fig. 1. Seal configuration.

wires — rather it would program the routers and such. Nevertheless it is elegant to be able to model the behavior of the network and of the programs that inhabit it within a single formalism. The configuration of Figure 1 corresponds to the expression appearing in Figure 2, where the processes P, P' and P'' denote behaviors of the sandbox and two hosts, and Q, Q' and Q'' denote the behaviors

of the applications. An alternate graphical representation is the configuration tree in the same figure. In a configuration tree process-labeled vertices represent seals while edges represent seal inclusion. The position of seal names (on edges) emphasizes the weak association between names and seals, they are merely tags used by parents to tell their children apart.

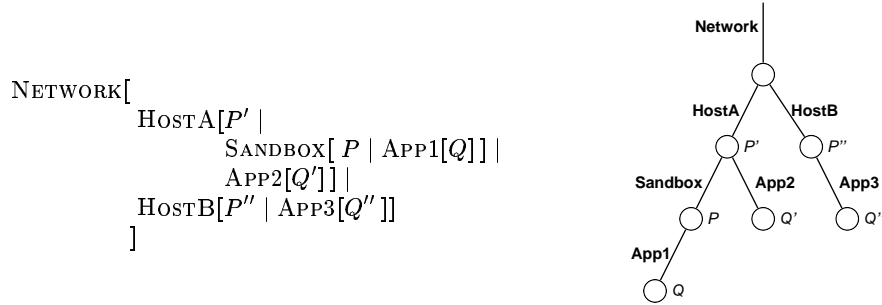


Fig. 2. Seal term and configuration tree.

2.1.4 Resources

The only resources in Seal are *channels*. Channels are named computational structures used to synchronize concurrent processes. As it happens for processes, channels are located. Channel denotations specify where channels are located. Thus, a channel x is denoted by x^η , where η is \star when the channel is local, is \uparrow when the channel is in the parent, and is the name of the seal when the channel is in a child. For example, consider the following process

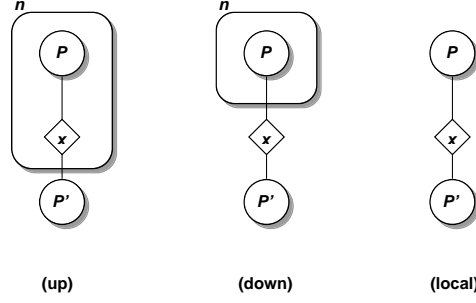
$$n_1 [P_1 \mid n_2 [P_2 \mid n_3 [P_3]]]$$

where the seal n_1 running the process P_1 has a child seal n_2 ; this child seal runs the process P_2 and has its own child the seal n_3 that runs the process P_3 . A channel x located in n_2 would be denoted as x^{n_2} by P_1 , as x^\star by P_2 , and as x^\uparrow by P_3 . Processes are allowed to use only the channels whose names they are aware of: it is not possible to guess names. In that respect names are communication capabilities, without propagation control and revocation, and provide a degree of protection and secrecy [1].

2.1.5 Communication

Seal allows only three distinct patterns of interaction shown in the figure below. We extend our graphical convention and use lozenges to denote channels and circles to denote processes.

There are two forms of *remote* interaction: a process located in the parent synchronizes with a process located in a child on a channel of the child (*down-sync*); a process in the parent synchronizes with a process in a child on a channel of



the parent (*up-sync*). And there is one form of *local* interaction: two co-located processes synchronize over a local channel. Channel synchronization is used both for communication (the channel is used to pass a name) and for mobility (the channel is used to move a seal). Each kind of interaction is specified by different actions.

There are two matching communication actions: $\bar{x}^n(y) \cdot P$ is a process that outputs the name y on channel x (located in η), and then behaves like P ; $x^n(\lambda y) \cdot P$ is a process that waits for input on channel x (located in η) and binds this input to all occurrences of y in P .

When two processes running in parallel wait for matching actions on the same channel, the processes synchronizes and their actions are consumed. Thus a local communication is performed by processes of the form $x^*(\lambda y) \cdot P \mid \bar{x}^*(z) \cdot Q$. After the synchronization we obtain $P' \mid Q$ where P' is obtained from P by replacing all unbound occurrences of y by z (a standard notation for P' is $P\{z/y\}$).

When a communication action acts on a non-local (respectively, local) channel we call it a *remote* (respectively, *local*) action. Remote actions produce *remote interactions*. So for remote communications we have, say, $x^*(\lambda y) \cdot P$ which tries to read a value emitted along channel x located in child seal n , or $\bar{x}^\dagger(y) \cdot P$ which tries to output a value along channel x located in the parent. The matching action of a remote action is always a local action. Consider the following expression (note that this expression does not synchronize, the missing ingredient will be explained soon):

$$n[x^*(\lambda y) \cdot P] \mid \bar{x}^n(z) \cdot P'$$

$x^*(\lambda y) \cdot P$ is a process waiting to read a value from local channel x . $\bar{x}^n(z) \cdot P'$ is a process that wants to emit a value along the channel x located in n . The remote action is matched by a local action.

Another way to view this form of interaction is that a local action is an “open” offer, that is, it does not prescribe which seal should provide a matching offer (it can also be matched by another local action). On the other hand, a remote action is a “closed” offer in the sense that the target seal is named explicitly. All combination of remote interaction in $P' \mid n[P]$ are summarized by the following table

	P	P'
(down-sync)	read $x^*(\lambda y) . Q$	$\bar{x}^n(z) . Q'$
	write $\bar{x}^*(y) . Q$	$x^n(\lambda z) . Q'$
(up-sync)	read $x^\dagger(\lambda y) . Q$	$\bar{x}^*(z) . Q'$
	write $\bar{x}^\dagger(y) . Q$	$x^*(\lambda z) . Q'$

These interaction patterns are restrictive. For example, they do not allow processes located in sibling seals (and even less those located in arbitrary seals) to communicate directly. Communication across a seal configuration must be encoded; in other words every distributed interaction up to packet routing must be programmed.

2.2 Mobility

Seals may be moved over channels. The expression $\bar{x}^*\{y\} . P$ denotes a process that is waiting to send a child seal y along channel x and then behave like P . The expression $x^*\{z\} . P$ denotes a process waiting to receive a seal along channel x and name it z .

Seal mobility is said *objective*, since a seal is moved by (a process in) the parent. The antithesis is *subjective* mobility in which a computational entity may move itself [12]. The computation encapsulated within a seal's boundary is not affected by the seal's mobility. Notification may be part of a mutually agreed upon mobility protocol, but in certain cases transparent mobility is desirable. For example, if the move occurred for load balancing reasons, the new parent will provide exactly the same services as the previous parent, and it is thus legitimate to hide mobility from seals.

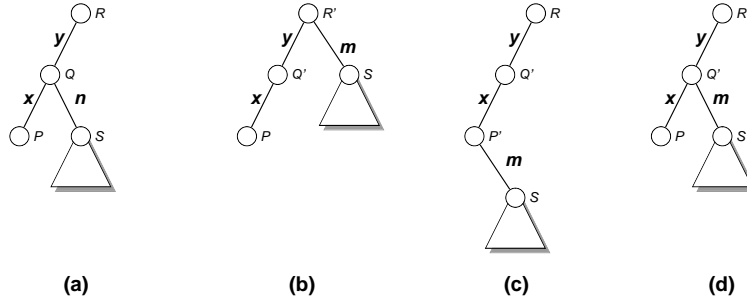
A seal move action requires that a seal bearing the requested name is present otherwise the operations blocks. If several seals bear the same name, the send will select one of them randomly. Move actions can only send a single seal along a channel and the name of the seal may not be preserved by mobility. The full form of a receive action is shown by the expression $x^n\{\bar{y}\} . P$ (where $\bar{y} = y_1 \dots y_n$) that denotes a process waiting to receive *one* seal and create n identical copies of that seal under the names \bar{y} . The complementary action is the send action $\bar{x}^n\{n\}$ which sends a seal denoted by name n over channel x . A first example of the use of mobility is a copy operation — typically used for fault tolerance to dynamically replicate running applications as new servers come online, or to improve availability by replicating computations. Let y be a fresh name, we encode a copy operator as follows (we use small capitals to denote defined operators, which, for simplicity, may be infix and composed of several keywords):

$$\text{COPY } x \text{ AS } z . P \quad \equiv \quad (\nu y)(\bar{y}^*\{x\} . \mathbf{0} \mid y^*\{xz\} . P)$$

Intuitively, $\text{COPY } n \text{ AS } m . \mathbf{0} \mid n[P]$ reduces to $m[P] \mid n[P]$, that is, it creates a seal m that is a copy of n . This is obtained by sending the seal denoted by n on a local channel y and reactivating two copies of it, one named n the other named

m . More precisely, the operation first creates a brand new channel name y (the purpose of this new name is to prevent any other process running in parallel from interfering with the protocol). Then, the subprocess on the right tries to move n on the local channel, while the one on the left waits to receive the seal and instantiate two copies of it, one named n and the other named m .

On the configuration tree, mobility corresponds to a tree rewriting operation. A move disconnects a subtree rooted at some seal y and grafts it either onto the parent of y , or onto one of y children, or back onto y itself. The rewriting operation relabels the edge associated to the moved seal, and can create a finite number of copies of the subtree rooted at the moved seal. The diagrams below show an initial configuration (a) and all three possible configurations obtained after a move. (b) is obtained by moving n into the parent and renaming it to m . (c) is obtained by moving n in x and renaming it to m . (d) is obtained by renaming n to m (local move). Each of these moves can also introduce copies.



For example, the initial configuration (a) depicted above corresponds to the expression $R \mid y[Q \mid x[P] \mid n[S]]$, while the final configuration (b) is described by the expression $R' \mid m[S] \mid y[Q' \mid x[P]]$. The processes involved in the move of n are Q (the sender) and R (the receiver) and use a channel located either in y or in its environment. If the channel, say w , is located in y then Q must perform the action $\bar{w}^* \langle n \rangle$ and R the action $w^n \langle m \rangle$, otherwise Q must perform the action $\bar{w}^\dagger \langle n \rangle$ and R the action $w^* \langle m \rangle$.

2.3 Protection

Use of non-local channels implies a threat to security, as an unknown process that may have migrated from an untrusted location has a way to access resources of its host. We propose several mechanisms to assist in the task of writing secure systems.

As a first line of defense, seals are not allowed to move about arbitrarily. Migration is always under the control of a seal's environment which decides when it occurs. Furthermore a seal must actually perform a receive action in order to allow a new seal to migrate from the outside and since it chooses the

name for the new seal it can choose this name fresh and thus arbitrarily isolate the newcomer from the other processes. Nevertheless, accepting a seal is a risk and further protection mechanisms are required.

The second line of defense is keeping tight control of names of channels. That is, by not giving out a name of a service, we can guarantee that the service can not be called by migrant seals, of course this also somewhat reduces the usefulness of the service. Once a name has been given out, it may be difficult to control its propagation in the system, it is safe to assume that it quickly becomes public. One could envision a notion of trust playing a role here. For instance a trusted seal, would not willfully reveal a secret name. Of course, we would still have to prove that it does not do it by mistake.

The third security ingredient is tight control over local resources. We have already said that a local action is an “open” offer since it may synchronize either with another local action or with a remote action. The latter case constitutes an external accesses to a local resource. Therefore we want to strictly monitor these access by allowing the synchronization with a remote action only in presence of an explicit permission.

The protection mechanism we propose to control inter-seal communication is called *portal*. The idea is that if a seal A wants to use seal B 's channel x , then B must *open* a portal for A at x . A portal is best viewed as an linear channel access permission. As soon as synchronization takes place, the portal is closed again. In the calculus the action to open a portal is $\mathbf{open}_n \mathbf{x}$ where \mathbf{x} is either x (the action allows the seal n to read once the local channel x) or \bar{x} (the action allows the seal n to write once on the local channel x). In Figure 3 we have a seal n containing the process P' and running in parallel with two processes P and S . The latter processes interact on local channels x' and y' , thus no portal is needed. S interacts with P' via a local channel x monitored by a portal controlled by S , and via a channel y in n and monitored by a portal controlled by P' . Imagine that P' wants to communicate to S the name of its channel y (possibly unknown

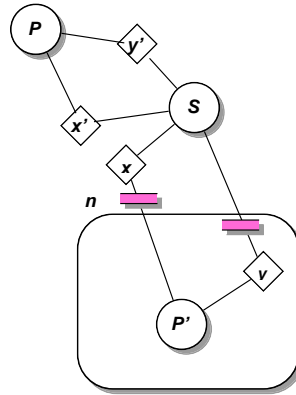


Fig. 3. Total mediation.

to S and then wait an acknowledgment on y). Then P' must send y to S via x and open the portal to allow S to write the acknowledgment on y (the channel y is used only for synchronization; to stress it we omit parameters and arguments of actions on it, and write $y^n()$ or $\bar{y}^n()$).

$$P' = (\bar{x}^\dagger(y) . \mathbf{open}_\uparrow \bar{y} . P_1) \mid (y^*() . P_2)$$

The process S opens the portal to allow n to write on local channel x , then it reads a name from a channel located in n and, finally sends the acknowledgment along the name it just read:

$$S = (\mathbf{open}_n \bar{x} . S_1) \mid (x^*(\lambda z) . \bar{z}^n() . S_2)$$

Thus we start by

$$\mathbf{open}_n \bar{x} . S_1 \mid x^*(\lambda z) . \bar{z}^n() . S_2 \mid n[\bar{x}^\dagger(y) . \mathbf{open}_\uparrow \bar{y} . P_1 \mid y^*() . P_2] \quad (1)$$

the processes synchronize on x and the first opening action is consumed:

$$S_1 \mid \bar{y}^n() . S_2 \mid n[\mathbf{open}_\uparrow \bar{y} . P_1 \mid y^*() . P_2] \quad (2)$$

then the processes synchronize on y and also the second opening is consumed:

$$S_1 \mid S_2 \mid n[P_1 \mid P_2]$$

If the local actions in (1) and (2) had not been in parallel with the corresponding opening actions, remote interaction would have been forbidden.

Now we can outline how a parent may implement total mediation. Total mediation implies that all observable actions (up-syncs and down-syncs) that involve one of its children are controlled by a security policy and that all the values exchanged are inspected. The mediation policy redirects all communications to the target seal passing them by a filter process, so that all values can be checked. Portals are opened only for channels that are allowed by the security policy. Note that if we allowed siblings to synchronize on a parent's channel, some interactions would not be subject to mediation. In Figure 3 process S is implementing an interposition policy, that provides channels x' and y' to P . The policy is in charge of opening the portal for x (the portal for y is opened within n) and of checking the values sent along both channels.

2.4 Discussion

Let us return to our five designing principles and discuss how the calculus addresses them. First, the seal model clearly does not rely on any global state. A seal knows only the names of its direct children and can communicate with them and its parent. This means synchronization involves at most two nested locations. Second, localities are explicit and the model clearly differentiates between local resources, and remote resources. Third, communication is restricted to local and neighbor (parent or children) interaction. Any further form of communication

must be explicitly programmed. This mean that we can easily model disconnected operations or the effects of a firewall. Fourth, dynamic reconfiguration is obtained by three ingredients: mobility, name passing, and dynamic binding. Mobility supports seal migration and duplication and can model topological reconfiguration of locations. Name passing can be used to explicitly reconfigure a migrated seal for the new environment while implicit configuration is obtained by dynamic binding of the \uparrow identifier. This tag denotes the parent environment and it is dynamically bound to the current parent. This allows a dynamic reconfiguration of the seal after migration, and the possibility to transparently update the services provided by the environment to its seals. Finally, portals allow to control access to local resources at a fine granularity.

3 The Seal Language

In this section the syntax and operational semantics of Seal are given. The language is an extension of Milner's synchronous polyadic π -calculus without matching and choice operators.

3.1 Syntax

We assume infinite sets \mathcal{N} of *names* and $\overline{\mathcal{N}}$ of *co-names* disjoint and in bijection via $(\bar{\cdot})$; we declare $\overline{\overline{x}} = x$. The set of location denotations extends names with two symbols (\star, \uparrow). Bold font variables denote either a name or the corresponding co-name, thus \mathbf{x} may be either x or \overline{x} .

\mathcal{N}	m, n, \dots, x, y, z	names	\mathcal{L}	$\eta ::= x \mid \uparrow \mid \star$	locations
$\overline{\mathcal{N}}$	$\overline{m}, \overline{n}, \dots, \overline{x}, \overline{y}, \overline{z}$	co-names	\mathcal{X}	X	process variables
	$\mathbf{x} ::= x \mid \overline{x}$				

The set of *processes*, ranged over by P, Q, R, S and *actions*, ranged over by α , are defined by the following grammars:

Table 1: Processes and Actions

Processes	Actions
$P ::= \mathbf{0}$	inactivity
$P \mid Q$	composition
$(\nu x)P$	restriction
$\alpha . P$	action
$!P$	replication
$x[P]$	seal
$x[X]$	abstract seal
$\alpha ::= \overline{x}^n(\overline{y})$	name output
$x^n(\lambda \overline{y})$	name input
$\overline{x}^n\{y\}$	seal send
$x^n\{\overline{y}\}$	seal receive
$\mathbf{open}_\eta \mathbf{x}$	portal open

$\mathbf{0}$ denotes the inert process. $P \mid Q$ denotes parallel composition. $(\nu x)P$ denotes restriction. $\alpha.P$ denotes an action α and a continuation P . $!P$ denotes replication. Finally, $x[P]$ and $x[X]$ denote, respectively, a seal named x with body process P and a seal x with body a process variable X .

Definition 1. *A process P is well-formed if and only if it contains no abstract seal $x[X]$ and no action of the form $\mathbf{open}_* \mathbf{x}$ (portal local open).*

Subsequently, we shall deal only with well-formed processes.

Following accepted terminology, the polarity of actions on names is positive and the polarity of actions on co-names is negative. $\bar{x}^\eta(\bar{y}).P$ denotes a process offering \bar{y} at channel x located in seal η with a continuation P . The process $x^\eta(\lambda\bar{y}).P$ denotes a process ready to input distinct names \bar{y} at x in η . The λ is a visual cue to remind the reader the \bar{y} are bound in P . $\bar{x}^\eta\langle y \rangle.P$ denotes the sender process offering at x in η a seal named y . The process $x^\eta\langle \bar{y} \rangle.P$ denotes the receiver process waiting to read a seal at x in η and start n copies of it under names $y_1 \dots y_n$. Note that this action is not binding. Finally, $\mathbf{open}_\eta x.P$ (respectively $\mathbf{open}_\eta \bar{x}.P$) denotes a process offering to open a portal for seal η to perform a positive (respectively negative) action on local channel x and then behave as P .

$\{y/x\}$ and $\{Q/X\}$ are meta-notations for substitutions. Thus $P\{\bar{y}/\bar{x}\}$ denotes the term obtained from P by simultaneous substitution of $y_1 \dots y_n$ for the free occurrences of distinct names $x_1 \dots x_n$, and $P\{Q/X\}$ denotes the term obtained from P by substituting process Q for all free occurrences of process variable X . Substitutions are ranged over by σ .

Location denotations \star, \uparrow and n denote respectively the current seal, the parent seal and a sub-seal bearing name n . The location denotations refer to the seal in which synchronization occurs. The simple case, which reduces to the π -calculus, is local synchronization, thus $P = \bar{x}^\star(y). \mathbf{0}$ is willing to emit name y along x and then become inert. $Q = x^\star(\lambda z). \bar{x}^\star(z). \mathbf{0}$ is a repeater which reads a name from x and emits it on the same channel. Local communication is always allowed, so the composition of the above mentioned processes reduces in one step:

$$P \mid Q \quad \rightarrow \quad \mathbf{0} \mid \bar{x}^\star(y). \mathbf{0}$$

Consider now the processes $P = \bar{x}^\uparrow(y). P'$, $Q = x^\star(\lambda z). Q'$ and $S = \mathbf{open}_n \bar{x}. S'$. The following configuration also reduces in one step:

$$n[P] \mid Q \mid S \quad \rightarrow \quad n[P'] \mid Q'\{y/z\} \mid S'$$

The case above involves a sub-seal trying to use a resource located in its environment; the symmetric case occurs when a process tries to access resources located in a sub-seal. Here for example, let $P = x^n(\lambda y). P'$, $Q = \bar{x}^\star(z). Q'$ and $S = \mathbf{open}_\uparrow x. S'$. The following configuration reduces in one step:

$$P \mid n[Q \mid S] \quad \rightarrow \quad P'\{z/y\} \mid n[Q' \mid S']$$

Notation We often omit the \star at the index of local communication and trailing $\mathbf{0}$ processes are elided, thus $x^\star(\lambda y) \cdot \mathbf{0}$ becomes $x(\lambda y)$. When communication is used purely to synchronize processes, we abbreviate $\bar{x}^n(y)$ to $\bar{x}^n()$ and input $x^n(\lambda y)$ to $x^n()$. Actions bind tighter than composition and composition bind tighter than restrictions, so that for instance $(\nu x)x() \cdot \bar{y}() \mid \bar{x}()$ means $(\nu x)((x() \cdot \bar{y}()) \mid \bar{x}())$.

3.2 Reduction semantics

The *reduction relation* \rightarrow is defined over processes and represents one step of computation. Reduction is defined by means of two auxiliary notions: *structural congruence* and *heating*.

3.2.1 Structural congruence

Structural congruence, \equiv , is the least congruence on processes satisfying the following axioms and rules:

Table 2: Structural congruence.

$P \mid \mathbf{0} \equiv P$	(Struct Dead Par)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu x)\mathbf{0} \equiv \mathbf{0}$	(Struct Dead Res)
$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	(Struct Res Res)
$b(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ if $x \notin fn(P)$	(Struct Res Par)

The set $fn(P)$ of *free names* of a process P is defined in a standard way (e.g., see [24]).

Intuitively, this relation does not correspond to any step of computation, instead it allows processes to be rearranged so that reduction can take place.

For example, we already saw that local synchronization is enabled when two complementary actions on a channel appear at the same level:

$$x(\lambda y) \cdot \bar{y}() \mid \bar{x}(z) \rightarrow \bar{y}() \mid \mathbf{0} \quad (3)$$

But, if the emitting process were $(\nu z)\bar{x}(z)$, then the ν -abstraction would hide the output action and thus prevent synchronization. Structural congruence rearranges the term so that can reduce:

$$x(\lambda y) \cdot \bar{y}() \mid (\nu z)\bar{x}(z) \equiv (\nu z)(x(\lambda y) \cdot \bar{y}() \mid \bar{x}(z)) \rightarrow (\nu z)(\bar{x}() \mid \mathbf{0}) \quad (4)$$

This result is obtained by alternating \rightarrow and \equiv rewritings, that is, by allowing replacement of terms by structurally equivalent terms. Structural congruence also handles the semantics of replicated actions and performs some house-keeping by sweeping out dead processes.

Structural congruence does not handle renaming of bound variables. Instead, we consider that alpha-conversions are silently performed whenever needed.

3.2.2 Heating

Although structural congruence provides a convenient way of rearranging terms to enable local synchronization, it does not suffice for non-local synchronization. Communication across seal boundaries requires special treatment. To illustrate this point let's modify example (3) so that the emitting process is located in a sub-seal:

$$\mathbf{open}_n \bar{x} \mid x^*(\lambda y) . \bar{y}() \mid n[\bar{x}^\dagger(z)] \quad \rightarrow \quad \mathbf{0} \mid \bar{z}() \mid n[\mathbf{0}]$$

Now, if like in (4) we ν -abstract the argument of the output, we expect the ν -abstraction to extrude the seal boundary and wrap around the input process, that is, informally the following should hold:

$$\mathbf{open}_n \bar{x} \mid x^*(\lambda y) . \bar{y}() \mid n[(\nu z)\bar{x}^\dagger(z)] \quad \rightarrow \quad (\nu z)(\mathbf{0} \mid \bar{z}() \mid n[\mathbf{0}])$$

This would require an equivalence such as $n[(\nu x)P] \equiv (\nu x)n[P]$. However, it would be an error to define these terms as equivalent, because we allow seal duplication¹. Indeed, if we compose both terms with the copier process defined earlier $Q = \text{COPY } n \text{ AS } m$ we obtain:

$$n[(\nu x)P] \mid Q \rightarrow n[(\nu x)P] \mid m[(\nu x)P] \text{ and } (\nu x)n[P] \mid Q \rightarrow (\nu x)n[P] \mid m[P]$$

The first term yields a configuration where seals n and m have each a private channel x , while in the other case, they share a common channel x . Our solution is to forego structural congruence at this point and perform the extrusion together with synchronization. To this end we define a *heating* relation on terms (we borrow the terminology of Berry and Boudol's Chemical Abstract Machine [7]). A term is "heated" to allow synchronization. Heating singles out all the ν -abstractions that must be extruded, that is, those that bind arguments of the negative action about to be performed. Heating will extrude as few ν -abstractions as possible. So for example the term

$$\mathbf{open}_n \bar{x} \mid x^*(\lambda y) . \bar{y}() \mid n[(\nu w)(\nu z)\bar{x}^\dagger(z)]$$

reduces to $\mathbf{0} \mid (\nu z)(\bar{z}() \mid n[(\nu w)\mathbf{0}])$ rather than to $\mathbf{0} \mid (\nu w)(\nu z)(\bar{z}() \mid n[\mathbf{0}])$.

More precisely, consider the term $(\nu w)(\nu z)\bar{x}^\dagger(z)$. Heating will tell us that to synchronize on \bar{x}^\dagger it is necessary to extrude (νz) . This is expressed by the following heating relation pair:

$$\begin{array}{ccc}
 (\nu w)(\nu z)\bar{x}^\dagger(z) & \prec & \bar{x}^\dagger.(\nu z)\langle z \rangle((\nu w)\mathbf{0}) \\
 \begin{array}{c} \text{channel} \quad \uparrow \\ \text{names to extrude} \quad \text{---} \end{array} & & \begin{array}{c} \uparrow \quad \uparrow \\ \text{arguments} \quad \text{---} \end{array} \\
 & & \text{residual}
 \end{array}$$

The heated form resembles Milner's *agents* [23]. The channel name comes first, it is followed by a list of extruded names, (νx) in this case, the arguments, z ,

¹ The Ambient Calculus [12] does not allow duplication and thus the equivalence holds.

and the residual process, $(\nu w)\mathbf{0}$. The argument values include both names and processes.

A term in heated form is called an *agent*. Agents are written ωP where ω is an agent prefix and P is a process. The set of *agent prefixes* ranged over by ω is defined by the following grammar:

Table 3: Agent prefixes

$\omega ::= \epsilon$	empty prefix
$(\nu \vec{x})\langle \vec{y} \rangle$	name concretion
$(\nu \vec{x})\langle P \rangle$	process concretion
$\langle \lambda \vec{y} \rangle$	name abstraction
$\langle \lambda X \rangle$	process abstraction

The sets $fn(\omega)$ of *free names* of an agent prefix and $bn(\omega)$ *bound names* of an agent prefix have standard definitions.

In order to simplify the presentation of the reduction rules we introduce the set $\overline{\mathcal{L}}$ of *co-locations* (that is in bijection with \mathcal{L} via $(\bar{\cdot})$) and the set of sync-locations. Their use is explained later on.

$\overline{\mathcal{L}} \bar{\eta} ::= \bar{x} \mid \bar{\uparrow} \mid \bar{x}$ co-locations	$\eta ::= \eta \mid \bar{\eta} \mid \mathbf{x}[]$ sync-locations
---	--

The heating relation \prec relates a well-formed process to a term of the form $\mathbf{x}^\eta.\omega P$ and is defined as the least relation respecting the following axioms and rules (where η denotes either η or \bar{x}):

Table 4: Heating.

$\bar{x}^\eta(\vec{y}).P \prec \bar{x}^\eta.\langle \vec{y} \rangle P$	(Heat Out)
$x^\eta(\lambda \vec{y}).P \prec x^\eta.\langle \lambda \vec{y} \rangle P$	(Heat In)
$\bar{x}^\eta\{y\}.P \mid y[Q] \prec \bar{x}^\eta.\langle Q \rangle P$	(Heat Send)
$x^\eta\{y_1, \dots, y_n\}.P \prec x^\eta.\langle \lambda X \rangle(P \mid y_1[X] \mid \dots \mid y_n[X])$	(Heat Recv)
$y \notin fn(\omega), y \notin \{x, \eta\}, P \prec \mathbf{x}^\eta.\omega P' \Rightarrow (\nu y)P \prec \mathbf{x}^\eta.\omega(\nu y)P'$	(Heat Res-1)
$y \in fn(\omega), y \notin \{x, \eta\}, P \prec \mathbf{x}^\eta.\omega P' \Rightarrow (\nu y)P \prec \mathbf{x}^\eta.(\nu y)\omega P'$	(Heat Res-2)
$bn(\omega) \cap fn(Q) = \emptyset, P \prec \mathbf{x}^\eta.\omega P' \Rightarrow P \mid Q \prec \mathbf{x}^\eta.\omega(P' \mid Q)$	(Heat Par)
$bn(\omega) \cap fn(Q) = \emptyset, P \prec \mathbf{x}^*.\omega P' \Rightarrow P \mid \mathbf{open}_\eta \bar{x}.Q \prec \mathbf{x}^\eta.\omega(P' \mid Q)$	(Heat Open)
$y \notin bn(\omega), P \prec \mathbf{x}^\uparrow.\omega P' \Rightarrow y[P] \prec \mathbf{x}^{y[]}.\omega y[P']$	(Heat Seal-1)
$y \notin bn(\omega), P \prec \mathbf{x}^\uparrow.\omega P' \Rightarrow y[P] \prec \mathbf{x}^{\bar{y}[]}.\omega y[P']$	(Heat Seal-2)

The first two axioms handle synchronization for communication. In particular the first axiom states that an output process does not need to extrude any name. The (Heat Send) and (Heat Recv) axioms deal with synchronization for mobility. (Heat Send) states that a negative action offers as argument the body of a seal. The fourth axiom says that a positive action heats into an abstraction where the process variable X stands for the body of the seals specified by \vec{y} , after synchronization the residual consists of the continuation P in parallel with the seals where X has been substituted by some process Q .

The following two rules select the names that will be extruded. If a ν -abstracted name does not occur free in the agent prefix ω then (Heat Res-1) applies and the name is not extruded. Instead, if a ν -abstracted name does occur free in ω then (Heat Res-2) applies and the name is extruded. The rule (Heat Par) simply propagates restrictions taking care of name conflicts. Note that it is always possible to alpha-convert *bound* variables so that name clashes are avoided.

The rule (Heat Open) combines a local action on some channel x and a permission to interact with a matching action from a process located in seal η . This is represented by changing the action label from x^* to $x^{\bar{\eta}}$.

Finally the last two rules allow actions originating from a seal y to synchronize with matching actions in the parent. When they flow through the boundaries of seal y the action labels are changed from x^\dagger to $x^{y\Box}$ and from x^\ddagger to $x^{\bar{y}\Box}$ in the process to prevent accidental matches and further propagation. In summary, $x^{y\Box}$ means that the seal y is ready to synchronize on its own channel x with the environment (it opened the channel to the environment and *committed* — see [13]— to perform an action on it), while $x^{\bar{y}}$ means that the environment is ready to synchronize y on the local channel x with the seal y (it opened the channel to y and committed to perform an action on it).

3.2.3 Reduction

We define the reduction relation \rightarrow as the least relation on well-formed processes that satisfies:

Table 5: Reduction.		
$\frac{\text{(Red Res)} \quad P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q}$	$\frac{\text{(Red Par)} \quad P \rightarrow Q}{P \mid R \rightarrow Q \mid R}$	$\frac{\text{(Red Seal)} \quad P \rightarrow Q}{x[P] \rightarrow x[Q]}$
$\frac{\text{(Red } \equiv)}{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q}$		
$\frac{\text{(Red Local)} \quad P \prec x^*.\omega_1 P' \quad Q \prec \bar{x}^*.\omega_2 Q'}{P \mid Q \rightarrow (\omega_1 P') \bullet (\omega_2 Q')}$	$\frac{\text{(Red Remote)} \quad P \prec x^{\bar{y}}.\omega_1 P' \quad Q \prec \bar{x}^{y\Box}.\omega_2 Q'}{P \mid Q \rightarrow (\omega_1 P') \bullet (\omega_2 Q')}$	

The *pseudoapplication* relation $(_)\bullet(_)$ used in the definition of synchronization is a partial *commutative* binary function from agents to processes. Let \vec{y}, \vec{x} be vectors of the same arity and $\vec{x} \notin fn(P)$, then we define pseudoapplication as:

<ol style="list-style-type: none"> 1. $(\langle \lambda \vec{y} \rangle P) \bullet ((\nu \vec{x}) \langle \vec{z} \rangle Q) = (\nu \vec{x})(P\{^{\vec{z}}/\vec{y}\} \mid Q)$ 2. $(\langle \lambda X \rangle P) \bullet ((\nu \vec{x}) \langle R \rangle Q) = (\nu \vec{x})(P\{^R/X\} \mid Q)$ 3. Undefined otherwise.

Of course, (Red Local) and (Red Remote) apply only provided that the inferred pseudoapplication is defined.

The first three rules perform reduction within restrictions, seals and parallel composition. The rule (Red \equiv) allows structural rearrangements to take place around a step of reduction.

The core of the semantics is given by the last two rules. They describe synchronization on a channel that is respectively local or remote. The combined use of heating and pseudoapplication allows us to compact several rules into a single one. For example, every single rule describes the synchronization in case of both communication and mobility. Let us show how the rules work by a couple of examples, starting with (Red Local).

Example 1. Consider the process $x(\lambda y).P \mid \bar{x}(z).Q$. By definition of heating we have $x(\lambda y).P \prec x.\langle \lambda y \rangle P$ and $\bar{x}(z).Q \prec \bar{x}.\langle z \rangle Q$, thus by (Red Local) the process reduces to $(\langle \lambda y \rangle P) \bullet (\langle z \rangle Q)$, that is $P\{^z/y\} \mid Q$. In summary, $x(\lambda y).P \mid \bar{x}(z).Q \rightarrow P\{^z/y\} \mid Q$. \square

In case of local synchronization the notational additions collapse two rules, communication and mobility, into the single (Red Local) rule. In the case of (Red Remote) the advantage is greater since in the absence of such notation we would be obliged to specify different rules for mobility and communication and also rule for positive and negative actions.

In order to understand the (Red Remote) rule let us show a second example of communication.

Example 2. Consider the process $x^*(\lambda y).P \mid \mathbf{open}_n \bar{x}.R \mid n[(\nu w)(\nu z)\bar{x}^\dagger(z).Q]$. By silent alpha conversion, we can consider, without loss of generality, that $y \notin fn(R)$, and $z \notin fn(P \mid R)$. By (Heat In), we have $x^*(\lambda y).P \prec x^*.\langle \lambda y \rangle P$. By (Heat Open) and the fact that $y \notin fn(R)$, we obtain $x^*(\lambda y).P \mid \mathbf{open}_n \bar{x}.R \prec x^{\bar{n}}.\langle \lambda y \rangle (P \mid R)$. Turning now to the seal, (Heat Out), (Heat Res-2) and (Heat Res-1) give us $(\nu w)(\nu z)\bar{x}^\dagger(z).Q \prec \bar{x}^\dagger.(\nu z)\langle z \rangle (\nu w)Q$. Therefore, by (Heat Seal), we get

$$n[(\nu w)(\nu z)\bar{x}^\dagger(z).Q] \prec \bar{x}^{\bar{n}}.(\nu z)\langle z \rangle n[(\nu w)Q]$$

The side conditions of pseudoapplication in (Red Remote) being satisfied we obtain

$$(\langle \lambda y \rangle (P \mid R)) \bullet ((\nu z)\langle z \rangle n[(\nu w)Q]) = (\nu z)((P \mid R)\{^z/y\} \mid n[(\nu w)Q])$$

Finally, since $y \notin fn(R)$ we have $(P \mid R)\{z/y\} = P\{z/y\} \mid R$. In summary we have

$$x^*(\lambda y).P \mid \mathbf{open}_n \bar{x}.R \mid n[(\nu w)(\nu z)\bar{x}^\dagger(z).Q] \rightarrow (\nu z)(P\{z/y\} \mid R \mid n[(\nu w)Q])$$

□

We conclude this section on reduction semantics by two remarks. First, as the second example shows, it is always possible to make terms satisfy the side conditions of pseudoapplication and heating rules. In fact, these conditions are on bound variables, that can be always alpha-converted to match the constraints. Secondly, we can show that reduction preserves well-formedness:

Lemma 1. *If P is well-formed and $P \prec \mathbf{x}^\eta.\omega Q$ then either Q is well formed or $\omega = \langle \lambda X \rangle$ and $Q\{R/X\}$ is well-formed for any well-formed R .*

Proof. By induction on structure of the derivation of $P \prec \mathbf{x}^\eta.\omega Q$.

Lemma 2. *Given a well-formed term P , if $P \rightarrow Q$ then Q is well formed.*

Proof. By induction on the structure of the derivation of $P \rightarrow Q$. For the rules of communication simply note that by (HeatSend) and (Heat Res-2) if P' is well-formed and $P' \prec \bar{x}^\eta.(\nu \bar{x})\langle Q' \rangle P''$, then also Q' is well-formed; use then Lemma 1.

4 Programming examples

We consider a distributed application management example in which the license of an application is to be extended automatically. The license string is a function of the host serial number and an expiration date. To extend the duration of the license it is necessary to regenerate a new license string for each instance of the application. If a customer decides to renew the license for a set of machines (designated by their IP numbers), the software manufacturer can generate a small mobile computation that will go around this set of machines, stopping at each machine long enough to obtain its serial number and run the password generation function on it. The overall configuration is shown in Figure 4. We will now present a Seal solution to this problem, outlining some of the necessary security properties. We start with the definition of two operators that will be used in the example.

4.1 Implementing the upgrade protocol

Renaming Seal renaming, $x \text{ BE } y$, atomically renames a seal bearing name x to y . This operator is defined as follows:

$$x \text{ BE } y.P = (\nu n)\bar{\pi}\langle x \rangle \mid n\langle y \rangle.P$$

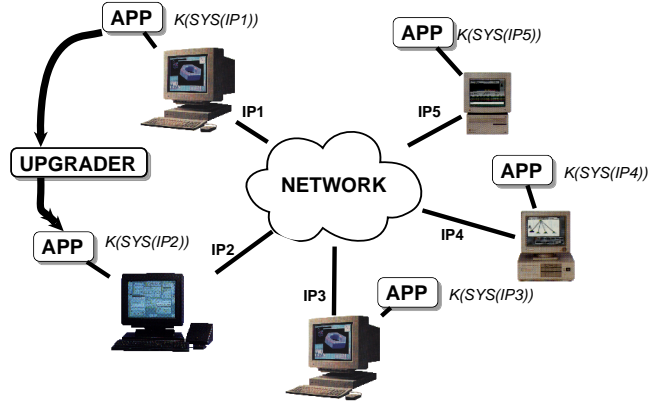


Fig. 4. Mobile upgrade agent.

Renaming is implemented by a local move. The action $\bar{n}\{x\}$ moves seal x along channel n . $n\{y\}$ receives a seal and names it y . In order to avoid interferences, the temporary fresh name n is used for the local channel.²

Linking A communication link allows two siblings to interact:

$$\text{LINK } n \ m \ \text{VIA } c . P = c^n(\lambda x). \bar{c}^m(x). P$$

The process reads x on channel c located in seal n and transmits it on a channel with the same name located in m . An example of linking is $n[\bar{x}(y) \mid \mathbf{open}_\dagger x] \mid \text{LINK } n \ m \ \text{VIA } x \mid m[x(\lambda z).P \mid \mathbf{open}_\dagger x]$ which first reduces in one step $x^n(\lambda z). \bar{x}^m(z) \mid n[\bar{x}(y) \mid \mathbf{open}_\dagger x] \mid m[x(\lambda z).P \mid \mathbf{open}_\dagger x] \rightarrow \bar{x}^m(y) \mid n[] \mid m[x(\lambda z).P \mid \mathbf{open}_\dagger x] \equiv \bar{x}^m(y) \mid m[x(\lambda z).P \mid \mathbf{open}_\dagger x] \mid n[]$ to finally reduce to $m[P\{y/z\}] \mid n[]$.

Network A very simplified network can be modeled by a single seal, with as many sub-seals as there are machines. Machines are designated by their IP number and for each machine a process takes care of routing packets sent to other machines on the net. The overall configuration is therefore:

$$\text{MACH } ip_1 \ P_1 \ \mid \dots \ \mid \text{MACH } ip_n \ P_n$$

More precisely, MACH is the description of three things: the actual machine (a seal whose name is the machine's IP number); the protocol that allows the machine to send out packets; the portion of the network transport layer that takes care of routing a packet to a given machine and port.

² Contrast this to renaming in the Ambient Calculus [12]: in our formalism renaming is performed by a process in the parent seal while in the Ambient Calculus ambients (that is, seals) rename themselves.

$$\begin{aligned} \text{MACH } ip P = & \\ & !(\nu pk) \text{ out}^{ip}\{pk\} \\ & \quad .\overline{hdr}^{pk}(\lambda ipdest \text{ port}) \\ & \quad \quad .\overline{port}^{ipdest}\{pk\} \\ & | ip[!\text{out}\{leave\} | !\text{open}_{\uparrow} \text{ out} | P] \end{aligned}$$

The network waits for the machine ip to send a new packet on channel out , it generates a new name pk for it, asks it on the header port hdr its destination machine and port, and routes it there. Besides running some generic process P , the machine ip moves out every packet whose name is $leave$.

Packets A packet is characterized by a seal n that represents the encapsulated information, and the IP number and port of destination that it publishes on its header port hdr . The seal n is encapsulated in and extracted from the packet through channels in and out respectively. We do not give a general description of how incoming packets are handled by machines, since this would be too specific. In general every process P on a machine will run a suite of protocols that listen to given ports and accordingly perform some operations. However, it possible to define two general operators that machines will use to handle packets. These operators named PACK and UNPACK are defined as follows:

$$\begin{aligned} \text{PACK } n \text{ ip port AS } p.P = & \\ & (\nu x) x[!\text{in}\{k\}] \\ & \quad .\overline{hdr}(ip \text{ port}) \\ & \quad \quad .\overline{out}\{k\} \\ & \quad | \text{open}_{\uparrow} \overline{in}.\text{open}_{\uparrow} \text{hdr}.\text{open}_{\uparrow} \text{out}] \\ & | \overline{in}^x\{n\}.x \text{ BE } p.P \end{aligned}$$

The action $\text{PACK } n \text{ ip port AS } p$ creates a packet named p , encapsulating the seal n , and with destination ip and $port$. Creation uses a temporary seal x to avoid external interferences during construction. Thus $\text{PACK } n \text{ ip1 pt2 AS } p | n[P]$ reduces to $p[\overline{hdr}(ip1 \text{ pt2}).\overline{out}\{k\} | k[P] | \text{open}_{\uparrow} \text{hdr}.\text{open}_{\uparrow} \text{out}]$.

$$\begin{aligned} \text{UNPACK } p \text{ AS } q.P = & \\ & (\nu x) \\ & \quad p \text{ BE } x \\ & \quad \quad .\overline{out}^x\{q\}.P \end{aligned}$$

The action $\text{UNPACK } p \text{ AS } q$ extracts the seal encapsulated in packet p and names it q ; the renaming is performed so that after the extraction p has become $(\nu x)x[!]$, which is equivalent to the inactive process. Thus $\text{UNPACK } p \text{ AS } n | p[\overline{out}\{k\} | k[P] | \text{open}_{\uparrow} \text{out}]$ reduces to $n[P] | (\nu x)x[!]$.

Upgrade protocol The only application specific protocol we describe in detail is the `UPGRADE_PROTOCOL` that listens on the *upgrade* port for mobile programs that upgrade the licence password of an application. This protocol must be run

(possibly inside a replication) by every machine that wants to allow licence upgrades. For the sake of simplicity we do not handle the cases in which the upgrade fails. The definition of the protocol is:

```
UPGRADE_PROTOCOL.P =
  open↑ upgrade |
  (ν upg pkt) upgrade{pkt}
    .UNPACK pkt AS upg
    .requestupg(λapp chn)
    .sysupg(vv)
    .LINK upg app VIA chn
    .nextupg(λip)
    .PACK upg ip upgrade AS leave.P
```

Let us describe it in detail. The protocol waits on the port *upgrade* for a packet that contains an upgrade application. The protocol extracts the upgrader application from the packet and names it with the fresh name *upg*. It then waits for *upg* to request a communication with a seal *app* (the application to upgrade) on channel *chn*. The protocol sends the machine serial number *vv* (used to generate the new license) to *upg* and allows a single communication from *upg* to *app* to take place. Finally the protocol asks *upg* its next destination and sends it out within a new packet.

Mobile upgrader application The upgrader application sequentially upgrades the application *app* in the various machines:

$$\text{upgrader}[\text{UPGRADE } ip_1.\text{UPGRADE } ip_2.\dots]$$

For simplicity we do not consider the cases in which one or more machines are down, or unreachable or fail during the upgrading.

The steps performed to upgrade a single machine *ip* are:

```
UPGRADE ip.P =
  next(ip)
  .request(app chn)
  .sysinfo(λvv)
  .chn(NEW_LICENSE vv) |
  open↑ request.open↑ sysinfo.open↑ chn.P
```

First the upgrader broadcasts the *ip* it wants to visit. Then it requests a communication channel *chn* to the application *app*. This action is performed only after that the upgrader has arrived at its destination. It also expects to receive some system information, before communicating the new license password (its computation not described here) to *app* via *chn*.

Upgrading the machines in a random order is obtained by running several upgrade processes in parallel within the same upgrader, $\text{upgrader}[\text{UPGRADE } ip_1 \mid \text{UPGRADE } ip_2 \mid \dots]$.

4.2 Security issues

The development of the Seal calculus emphasizes security issues. Indeed, we look for a calculus that allows context independent proofs of security, that is, our goal is to be able to localize security code in small, well-delineated portions of a system and to be able to reason about security properties without having to resort to whole program analysis.

The first property that we want to obtain is what [12] calls the *perfect firewall equation*: this says that an arbitrary process can be completely prevented from any communication, and is formally written³

$$(\nu x) x[P] \simeq \mathbf{0}$$

Here \simeq denotes some action-based observational equivalence on process. A formal example of definition of \simeq can be found in [13], but for the purposes of this section it suffices to consider as equivalent two processes that perform at the top level the same sequences of channel writings.

Intuitively, the perfect firewall equation holds in the Seal calculus because if a seal is given a fresh name that is not known by any other process, then there can be no portal open for that name (such as $\mathbf{open}_x \mathbf{y}$), nor can there be a located communication (such as $\mathbf{y}^x \{ \dots \}$). This property is preserved by reduction. (Note that the formal proof that this equation holds is far more complicated than the simple reasoning in the lines above: see [13]). It is a useful property as it guarantees that once a seal is in a firewall it cannot divulge any information, nor perform any externally visible action for that matter.

One difference between our approach and ambients is that here it is possible to force a seal into a firewall: the following operation will eventually trap a seal x

$$\mathbf{TRAP} x = (\nu ct) \bar{c}\{x\} \mid c\{t\}$$

If portals are viewed as capabilities, the TRAP operator is a non reversible form of revocation while renaming is a reversible revocation.

More generally, we would like to guarantee that some code or protocol has given security properties. For example, in the case of the upgrade application one wants to guarantee, by simple examination of the upgrade protocol, that every mobile upgrader, however it is defined, can only interact with the application *app* that it requested explicitly upon arrival, and that every interaction with the rest of the machine is mediated by the protocol.

More precisely, let \mathbf{P} denote the part of the update protocol that does not handle packet reception and dispatching:

$$\begin{aligned} \mathbf{P} = & \mathit{request}^{upg}(\lambda app, chn) \\ & .\overline{\mathit{sys}}^{upg}(vv) \\ & .\mathbf{LINK} \mathit{upg}, \mathit{app} \text{ VIA } chn \\ & .\mathit{next}^{upg}(\lambda ip) \end{aligned}$$

³ In [12] there is the extra requirement that x does not appear free in P .

If U is the body of the upgrader then the situation after migration is:

$$ip[((\nu \text{ upg})\mathbf{P} \mid \text{ upg}[U]) \mid Q]$$

Note that by the move (more precisely by the definition of substitution) upg cannot appear free in U . Then, for every process U in which upg is not free we can reason as follows:

1. Since upg never appears free in the argument of a negative action in \mathbf{P} , then:
 - (a) upg cannot move, and
 - (b) upg cannot appear free in any reductum of Q .
2. The two points above imply that there cannot be any interaction between $\text{ upg}[U]$ and Q (if upg could move then it could be renamed and escape the restriction on upg). Therefore every remote interaction of $\text{ upg}[U]$ is with \mathbf{P} .
3. From the previous point we have that the only process that may access resources located in upg is \mathbf{P} , and that upg can only access the external resources granted by \mathbf{P} (actually, none). Furthermore, since \mathbf{P} is a sequential process (there is no fork) we can also determine the exact order in which remote interactions (may) happen.
4. In conclusion whatever the upgrade program is, it can only send a single name (via the total mediation of the protocol) to the seal app specified on the *request* channel, and read the serial number vv provided by the \mathbf{P} protocol.

These security results can be declined in some more general properties:

1. *Bound communication*: The entering seal cannot communicate with anyone but with the seal requested on the channel *request*. In particular, this is true even in the case that two seals agreed on some communication protocol and entered independently the machine: only the hosting machine can allow them to communicate.
2. *No hitchhiking*: Unwanted seals cannot use a seal entering by the *upgrade* port as a Trojan horse to sneak into the machine, nor once the upgrading seal entered the machine can it be used to surreptitiously carry away some (partner) seal.⁴

The above mentioned security properties relate to protection of the host from the actions of mobile computations. To present a comprehensive solution we should also devise a protocol that provides some guarantees to mobile computations. Although the host may not modify the internal behavior of the upgrader, a malicious host may (1) lie about its serial number, (2) learn the itinerary of an upgrader, (3) trap an upgrader, (4) impersonate an upgrader, (5) listen in on the conversation between an upgrader and the upgraded application.

⁴ Hitchhiking is allowed in the Ambient calculus [12]. In the train example (see [11]) ambients representing passengers enter train stations, and then board trains. In the untyped calculus, there is no way of limiting the number of passengers that board a train and nothing prevents a passenger from hiding a potentially infinite number of hitchhikers that will come out as soon as the passenger is on the train.

In this section we reasoned about some security properties. The deductions we hinted cannot be considered as “proofs” of security. They are intentionally informal and *ad hoc*. For Seal to be used to state and prove security properties it is necessary to define a theory of proof, to develop some techniques of proof, and to define suitable notions of “observation”, “test”, and “specification” (see *e.g.* [1]). We are working in this direction [13, 33].

5 Related work

Ambients The Ambient calculus of Cardelli and Gordon [12] has been one of the inspirations of this work. Ambients resemble seals in the sense that they are named places with a hierarchical structure. The main difference between the models is that in the Ambient setting, mobility is triggered and controlled by the ambient itself and mobility control is based on capabilities. An ambient that has been given a capability to enter another ambient may do so at any time. Our model gives full control of mobility to the environment, thus it is always the environment that decides when a move may occur. In an ambient system trapping a migrating ambient is not entirely straightforward, while in Seal, the environment can enforce confinement on any seal running within it. Another important difference is that the boundary around an ambient can be dissolved, thus releasing the ambient’s content in the current environment. Such an operation is quite dangerous as the ambient being opened may contain any kind of code. Seal does not allow boundaries to be dissolved. Finally, the Ambient calculus is more minimal than Seal, with ambients computation is mobility. In Seal computation can be carried out in the π -calculus core.

Distributed calculi The difficulties of modeling some key aspects of distributed computing, in particular failures, within the π -calculus have driven a number of researchers to specifying distributed extensions [5, 28].⁵ The Linda model was extended with explicit localities and the ability to evaluate (dynamically scoped) processes at a given locality [15]. The distributed join-calculus of Fournet and Gonthier is a calculus specially designed for a distributed implementation [18] in which every channel is rooted at a given location. All of these calculi adopt a higher level view than Seal, allowing direct communication with remote processes. In particular, the perfect firewall equation typically does not hold. In programming terms this means that a mobile entity may always communicate with its creator and thus leak any information that it gleaned along the way. So policies such as the strong sandbox of Java can not be straightforwardly implemented. More subtly, the lack of syntactic difference between local and remote resources promotes a programming style in which computations are spread over a number of different nodes, thus increasing the degree of interdependency and making computation much more sensitive to failures. Our goal with mobility is to emphasize local interaction.

⁵ In the Seal, failures are modeled by trap operator which eventually entraps its target and prevents it from interacting with the environment.

Local vs. global The α -conversion involved in extrusion has unpleasant computational implications if the scope of a name is extended over a number of hosts. It is therefore desirable to control tightly the scope of names and be prepared to promote a locally unique name to global uniqueness. One approach for avoiding α -conversion is to use a location-aware naming scheme, such as the one described in the work of Bodei, Degano and Priami [9] who proposed that names be tied to their creation point in the syntax tree and that relative paths in the syntax tree be used to differentiate between equal names. In our setting this approach would fail as processes may move requiring tracking or forwarding services. A less elegant implementation techniques that avoids α -conversion is to generate name randomly and rely on the small likelihood of collisions. Nevertheless, generating and computing with such names may degrade efficiency of an application. Ideally, one would prefer to generate globally unique names only when they are needed. Peter Sewell proposed to use a type system to capture locality of names [32]. This approach would permit optimizations such as the use of simpler representations for local names.

Types for security Several researchers have proposed to rely on types for resource access control [21, 14, 32]. The work of Hennessy and Riely is innovative as it deals with open networks where a subset of hosts may be malicious. This raises very interesting problems: for instance, handing out an ill-typed value to a mobile application can not be detected right away if the value is a non-local channel name but may break the application later on. Their type system detects these kinds of error before the value is used, but this remains a genuine attack (if the goal is to prevent the mobile agent from carrying out its task). In Seal we did not choose types for controlling access to resources as we feel that resource allocation in real systems is very dynamic, typically the set of resources (memory, communication channels, cpu time, etc.) available to an entity will evolve over time. Types appear too rigid to model this aspect well. For our part, we plan to study the use of type systems for constraining other characteristics of the behavior of seals.

Cryptography The spi-calculus is an elegant extension of the π -calculus with cryptographic primitives developed by Abadi and Gordon [1]. In essence, they add public-key encryption to the π -calculus, interestingly it is possible to express this extension in the Seal calculus. But, the main contributions of this work are proof techniques for security protocols which we plan to adapt to our setting.

From Seal to π Sangiorgi devised a technique for translating a higher order π -calculus in the plain π [30] (see also [4]). In his technique, process sending is encoded by triggers. Instead of sending a whole process P , just a fresh name x is sent. The process P is guarded by an input on x and placed within a replication: $!x.P$. Thus, any output on x will release a copy of P . The same technique cannot be used here because seal mobility is defined on “running” processes. Consider the term $\bar{x}\{y\} \mid y[P]$. The rules of our calculus allow P to

reduce to some P' before the seal is captured by the move action. In general, the set of $\{P' \mid P \rightarrow^* P'\}$ is not finite, and we would need as many triggers as there are different P' s. Another translation from a mobile agent setting into the join-calculus was given in [19]. There, the translation relied on message routing. It can not be used for Seal because mobility also involves process copying.

Static and dynamic scoping We have already discussed that we do not allow the scope of a ν -abstraction to extrude outside a seal: $s[(\nu x)P] \not\equiv (\nu x)s[P]$. This complicated the definition of intra-seal interaction since the reduction must also handle extrusion of the restrictions over an outgoing argument. A different way to handle this problem would be to follow Bent Thomsen’s CHOCS and adopt dynamic scoping of ν -abstracted names [35]. In that case it would be possible to send a name outside of its static scope. However, this would clash with security, since dynamic binding would permit names to be “guessed”, *i.e.* our proof of security in the example of section 4 relied on the fact that restricted names were not known outside of their scope and that names would be alpha-converted to avoid conflicts. Without this, all security code becomes more complex and in general programs become more fragile. It is interesting to note that Thomsen himself later amended the definition of CHOCS to static scoping [36].

Inspection of our syntax for concretions and abstraction reveals that our calculus, like CHOCS, sends processes as values. But, for safety (and implementation) reasons, we restrict the occurrences of process variables to be encapsulated within seals. In fact our syntax ensure that a seal abstraction will always have the form: $x^n.[\lambda X] \dots (P \mid y_1[X] \mid \dots \mid y_n[X]) \dots$ where the only occurrences of a process variable are the bodies of seals y_1 through y_n . This guarantees that migrating processes will always be protected by boundaries and that their parent will not be able to compose them with arbitrary processes ($y[X \mid P]$ is forbidden as P may be a virus). One of the side effects of this restriction is that our calculus does not allow the open operation of Ambients.⁶

Asynchrony and synchrony Many recent works in concurrency theory and distribution have argued for asynchronous calculi. Boudol [10] gives a well motivated argument in favor of asynchrony in the framework of the π -calculus, while the Ambient Calculus adopts it in a distributed setting (see [11] for an extensive justification of this choice). The usual motivation for this is by a two-pronged argument: (1) synchronous communication does not sit well with a large scale distributed system as it requires global distributed consensus. (2) Synchronous communication can be implemented in an asynchronous setting, by means of a mutual inclusion protocol in which a sender waits for an acknowledgment. Boudol proved the adequacy of such an implementation w.r.t. a testing preorder [10].

⁶ An ambient can be opened, releasing its body in the environment. Without our syntactic restrictions, open could be coded in the seal calculus as: $\text{OPEN } n.P = \bar{x}\langle n \rangle \mid x.[\lambda X](X|P)$, the concretion releases the seal’s body process in environment.

While the first part of the argument is certainly true, the second part of the argument fails in a mobile environment. Mobility exposes the difference between asynchronous primitives and synchronous ones, differences become observable by processes. If a seal moves after a request has been issued and before the acknowledgment is sent, it will not receive the acknowledgment. This may mean, in programming terms, that the seal may try to request the same service a second time. One option would be to forward messages, but tracking mobile entities on a system such as the Internet would require an unrealistic amount of support and cooperation.

We consider that synchronous interactions are frequent and quite natural. Thus, the Seal calculus takes the opposite approach and all communications, including seal mobility, are synchronous. Note however that communication in our model is localized. Synchronization is either local to a seal or restricted to synchronization between parent and child (*e.g.* a machine and an application, a local area network and a machine). All other patterns of interaction must be implemented as sequences of synchronous exchanges and are thus asynchronous. Communications across machine boundaries in an implementation of seal will only require synchronization of adjacent seals. Mobility will likewise not require more than synchronization between adjacent seals.

So we are in presence of two different disciplines of communication. Our model supports only local and parent-child communication. This is synchronous, it requires synchronization consensus but this is limited in scope. Any other pattern of communication must be implemented in terms of our primitives and, in general, will be asynchronous. More powerful distributed communication mechanisms, *e.g.* incorporating distributed failure detection, are likely to be application specific and must be provided as derived operations.

Objects Many distributed programming languages have adopted the object-oriented model as a basis for structuring distributed services. At the outset this was also our view, as witnessed by the original title of this work: “a calculus of sealed objects”. The name was shortened to seals when we realized that the abstractions involved were more basic than those of the object paradigm. It also became clear early on that hierarchical protection would be very difficult to obtain in an object system, this mostly due to pervasive aliasing (the object spaghetti) [25]. Furthermore, we also came to understand that seals are coarser abstractions than objects. This point is echoed in our current implementation effort, JavaSeal, which represents seals by systems of objects [37].

6 Conclusion and Future work

The Seal calculus is a programming language suited for modeling Internet applications. Seal captures their inherent parallelism, the repartition of computational elements amongst multiple locations, the mobility of resources and locations, as well as the protection domains that arise from different security policies.

We already mentioned several points that are under investigation: the definition of suitable notions of observation, specification, and equivalence, in order to develop some general techniques and patterns of proofs of security properties.

The next step of this research will be the definition of leaner calculus. The most important criterion for its definition will be minimality. Therefore we will try to reduce the forms of synchronization, and simplify the semantics of portals. For example, if we separate channels for local and remote communications portals for up-sync would become useless. Further issues to consider include:

- Specification. Should/could the leaner calculus be used for specification? Is the formal system of this article a better candidate?
- Testing. Do standard definitions of testing for the process algebras suffice to test mobile software?
- Which properties could/must be captured by the calculus and which ones should instead be delegated to static analysis or dynamic checking?
- Since the synchronism of remote communications is cumbersome to implement, could/should it be weakened?

Moreover, the notion of resources must be extended to encompass memory and cpu-time, if denial of service attacks are to be considered. Furthermore, we are looking at types as means to provide security guarantees, in other words, types should not be just the means to check that channels transport well typed data, they should constrain the behavior of mobile computations so as to facilitate proofs of their security properties.

The long-term and much more ambitious goal is to provide a complete set of tools for software engineering. One essential component should be a specification language for conceptual design (analogous to UML for object-oriented languages) that can be automatically translated into the Seal calculus. The Seal calculus would then constitute a language in which to provide an intermediate representation of mobile software. This representation could be then be transformed either into a representation in a calculus in which test and formal proofs could be performed, or into a executable representations in Seal-based languages such as JavaSeal [37].

Acknowledgments

Our work benefited greatly from multiple discussion with Martín Abadi and Luca Cardelli, we are grateful for their insightful advice which prevented us for straying too far off the track. We also wish to thank Karen Bernstein, Walter Binder, Ciarán Bryce and Laurent Dami for comments and inspiration. The first author is funded by the Swiss PP project *ASAP: Agent System Architectures and Platforms* No 5003-45335.

References

1. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich, April 1997*. ACM Press, 1997. Long version as Technical Report 414, University of Cambridge.

2. ACM. *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)*, Jan. 1996.
3. G. Agha. *Actors – A model of concurrent computation in distributed systems*. The MIT Press, 1986.
4. R. M. Amadio. On the reduction of CHOCS bisimulation to π -calculus bisimulation. In Best [8], pages 112–126. Extended version as Rapport de Recherche, INRIA-Lorraine, 1993.
5. R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION '97*. Springer-Verlag, 1997. Full version as Rapport Interne, LIM Marseille, and Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.
6. F. M. auf der Heide and B. Monien, editors. *23rd Colloquium on Automata, Languages and Programming (ICALP) (Paderborn, Germany)*, volume 1099 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
8. E. Best, editor. *CONCUR'93, 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
9. C. Bodei, P. Degano, and C. Priami. Mobile processes with a distributed environment. In auf der Heide and Monien [6], pages 490–501. Full version as Università di Pisa Technical Report *Handling Locally Names of Mobile Agents*, 1996.
10. G. Boudol. Asynchrony and the π -calculus (Note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.
11. L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. Springer Verlag, 1999.
12. L. Cardelli and A. D. Gordon. Mobile Ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, number 1378 in LNCE, pages 140–155. Springer-Verlag, 1998.
13. G. Castagna and J. Vitek. Commitment and confinement for the seal calculus. Technical report, Laboratoire d'Informatique de l'École Normale Supérieure, 1999.
14. R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proceedings of COORDINATION'97*. Springer-Verlag, 1997.
15. R. De Nicola, G. Ferrari, and R. Pugliese. Locality based Linda: programming with explicit localities. In *Proceedings of FASE-TAPSOFT'97*. Springer-Verlag, 1997.
16. P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors. *24rd Colloquium on Automata, Languages and Programming (ICALP) (Bologna, Italy)*, volume 1256 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
17. B. Ford, B. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings Symposium on Operating Systems Design and Implementation(OSDI'96)*. ACM Press, Oct. 1996.
18. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In POPL'96 [2], pages 372–385.
19. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *CONCUR96*, pages 406–421, 1996.
20. D. Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, Jan. 1985.

21. M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Proceedings of the Workshop on Internet Programming Languages, (WIPL)*. Chicago, Ill., 1998.
22. K. G. Larsen, S. Skyum, and G. Winskel, editors. *25rd Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
23. R. Milner. The polyadic π -calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.
24. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.
25. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP'98 – Object-Oriented Programming, 12th European Conference Proceedings*. Brussels, Belgium, Springer-Verlag, July 1998.
26. P. Pardyak, S. Savage, and B. N. Bershad. Language and runtime support for dynamic interposition of system code. Nov. 1995.
27. *Proceedings of the Sixteenth Annual Symposium on Principles of Programming Languages (POPL) (Austin, TX)*. ACM Press, Austin Texas, January 1989.
28. J. Riely and M. Hennessy. Distributed processes and location failures. In Degano et al. [16], pages 471–481. Full version as Report 2/97, University of Sussex, Brighton.
29. H. Rodrigues and R. Jones. Cyclic distributed garbage collection with group merger. In *ECOOP'98 – Object-Oriented Programming, 12th European Conference Proceedings*. Brussels, Belgium, Springer-Verlag, July 1998.
30. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1993.
31. F. Schneider. What Good are Models and What Models are Good? In S. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Frontier Press, 1993.
32. P. Sewell. Global/local subtyping and capability inference for a distributed π -calculus. In Larsen et al. [22].
33. P. Sewell and J. Vitek. Secure composition of insecure components. In *Computer Security Foundations Workshop (CSFW-12)*. Mordano, Italy, June 1999.
34. P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In *Proceedings of the 1998 Workshop on Internet Programming Languages*, Chicago, Ill., May 1998.
35. B. Thomsen. A calculus of Higher Order Communicating Systems. In POPL'89 [27], pages 143–154.
36. B. Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.
37. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal: or How to make Java safe for agents. In D. Tschritzis, editor, *Electronic Commerce Objects*. University of Geneva, 1998.
38. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the programmable Internet*. Springer-Verlag, 1997.