A Calculus of Mobile Agents

Cédric Fournet, Georges Gonthier,
Jean-Jacques Lévy, Luc Maranget, Didier Rémy

INRIA Rocquencourt *
78153 Le Chesnay Cedex, FRANCE
e-mail Cedric.Fournet@inria.fr

Abstract. We introduce a calculus for mobile agents and give its chemical semantics, with a precise definition for migration, failure, and failure detection. Various examples written in our calculus illustrate how to express remote executions, dynamic loading of remote resources and protocols with mobile agents. We give the encoding of our distributed calculus into the join-calculus.

1 Introduction

It is not easy to match concurrency and distribution. Suppose, for instance, that we want to implement a concurrent calculus with CCS-like communication channels and with processes running on different physical sites. If we do not locate channels, we quickly face a global consensus problem for nearly every communication which uses the interconnection network. In a previous work [6], we introduced the join-calculus, an asynchronous variant of Milner's π-calculus with better locality and better static scoping rules. It avoids global consensus and thus may be implemented in a realistic distributed environment. Furthermore, it is shown to have the same expressive power as the π-calculus. In this paper, we extend the join-calculus with explicit locations and primitives for mobility. The new calculus, the Distributed Join-Calculus, allows to express mobile agents moving between physical sites. Agents are not only programs but core images of running processes with their communication capabilities.

The novelty of the distributed join-calculus is the introduction of locations. Intuitively, a location resides on a physical site, and contains a group of processes. We can move atomically a location to another site. We represent mobile agents by locations. Agents can contain mobile sub-agents, this is represented by nested locations. Locations move as a whole with all their sublocations. For these reasons, we organize locations in a tree.

Our calculus also provides a simple model of failure. The crash of a physical site causes the permanent failure of all its locations. More generally, any location can halt, with all its sublocations. The failure of a location can be detected at any other running location, allowing error recovery.

* This work is partly supported by the ESPRIT Basic Research Action 6454 - CONFER.

Our aim is to use this calculus as the core of a distributed programming language. In particular, our operational semantics is easily implementable in a distributed setting with failures. The specification of atomic reduction steps becomes critical, since it defines the balance between abstract features and realistic concerns.

In the spirit of the π-calculus, our calculus treats channel names and location names as first class values with lexical scopes. A location controls its own moves, and can only move towards a location whose name it has received. This provides a sound basis for static analysis and for secure mobility. Our calculus is complete for expressing distributed configurations. In the absence of failure, however, the execution of processes is independent of distribution. This location transparency is essential for the design of mobiles agents, and very helpful for checking their properties.

We present classical examples of distribution and mobility. The basic example is remote procedure call with timeouts. Dynamic loading of remote applications is our second example. Unlike Java applets, we download a process with its active communications, simply by moving its location. The third example, remote execution of a local agent, is the dual case. The last example is a combination of the second and third. The client creates an agent that moves to a server to perform some task; when this task is completed, the agent comes back to the client to report the result. We take this example, which we dub the client-agent-server architecture (CASA), as our paradigm for mobility. We show that causal error recovery can be integrated into this CASA with minimal implementation assumptions.

In section 2, we review related work. In section 3, we give a brief presentation of the join-calculus and recall the basics of the reflexive chemical machine framework. In section 4 and 5, we gradually extend the join-calculus. In section 4, we introduce our location model as a refinement of the reflexive chemical model and present a first set of new primitives aimed at expressing location management and migration. In section 5, we give our final calculus that copes with failure and recovery, discussing various semantical models for failure. In parallel, we develop our main example of the client-agent-server architecture. In section 6, we suggest techniques for formal proofs, we provide an encoding of the distributed calculus into the join-calculus, and we state a full abstraction theorem. Finally, we give directions for future work.

2 Related work

Migration has been investigated mostly for object-oriented languages. Initially used in distributed systems to achieve a better load-balancing, migration evolves to a language feature in Emerald [9] : objects can be *moved* from one machine to another; they can also be *attached* to one another, an object carrying its attached objects as it moves. At the language level, numerous calling conventions such as call-by-move reflect these capabilities, and the use of migration for safety purposes is advocated.

More recently, several languages have been proposed for large-scale distributed programming, with some support for the mobile agent paradigm. For instance, Obliq [5] encodes migration as a combination of remote cloning and aliasing, in a language with a global distributed scope. Examples of applications with large-grain mobility in Obliq can be found in [3]. However, little support is provided for failure recovery. In a functional setting, FACILE [7] provides process mobility from site to site, as the communication of higher-order values. As in this paper, the design choices are discussed in a chemical framework [10].

Mobility and locality already have other meanings in process calculi. Mobility in the π-calculus refers to the communication of channel names on channels [11], whereas locality has been used as a tool to capture spatial dependencies among processes in non-interleaving semantics [4, 14].

The formal model developed for core FACILE [1] is more closely related to our work. In the π_l-calculus, the authors extend the syntax of the π-calculus with locations. Channels are statically located; a location can fail, preventing further communication on its channels; location status can be tested in the language. Due to the properties of the π-calculus, observation with failures become very different from the usual observation, but an encoding of the π_l-calculus in the π-calculus is given and proved adequate. In this paper, we also introduce a distributed calculus as a refinement of a core calculus – the join-calculus –. However, the join-calculus was specifically designed for this purpose, which leads to simpler formal developments, even though our extensions capture both migration and failure.

3 Chemical frameworks

In this section, we introduce key notions for the syntax and semantics of our distributed calculi, we briefly present the join-calculus, and we define observational equivalence. The join-calculus is our basic process calculus. Later in this paper, we extend it by introducing locations, migration, and failure.

3.1 General setting

Our calculus is a name-passing calculus. We assume given an infinite set of port names with arities $\mathcal{N}$ (ports are also called channels). We use lowercase variables x, y, foo, bar, ... to denote the elements of $\mathcal{N}$. Names obey lexical scoping and can be sent in messages. At present, we only have port names. Later in this paper, we will introduce other values (location names, integers, booleans) and letters u, v, ... will denote values in general.

We assume that names are used consistently in processes, respecting their arities. This could be made precise by using a recursive sort discipline as in the polyadic π-calculus [11, 12]. We assume that all processes are well-sorted.

Notations: We use the following conventions: $\widetilde{v}$ is the tuple $v_1, v_2, \cdots v_n$, $(n \geq 0)$; RR' is the composition of the relations R and R'; R^* is the transitive closure of relation R.

Chemical rules: We present our operational semantics in the chemical abstract machine style of Berry and Boudol [2]. The CHAM provides a precise and convenient way to specify reduction modulo equivalence. It also conveys some intuition about implementation schemes and implementation costs, especially in distinguishing between local and global operations.

As usual, we use two families of chemical rules that operate on multisets of terms (the so-called chemical soups, or chemical solutions): *Structural rules* $\rightleftharpoons$ are reversible ($\rightharpoonup$ is *heating*, $\leftharpoondown$ is *cooling*); they represent the syntactical rearrangements of terms in solution. *Reduction rules* $\longrightarrow$ consume some specific terms in the soup, replacing them by some other terms; they correspond to the basic computation steps.

3.2 The join-calculus and the reflexive chemical machine (RCHAM)

Our starting point is the join-calculus as described in [6]. The join-calculus is as expressive as the asynchronous π-calculus. Furthermore, our calculus is closer to a programming language than the π-calculus. In particular, it can be seen as a concurrent extension of functional programming.

Syntax: Terms of the calculus are processes and definitions:

$$\begin{aligned} P &\stackrel{\text{def}}{=} x\langle\widetilde{v}\rangle \quad | \quad \mathbf{def}\ D\ \mathbf{in}\ P \quad | \quad P|P \quad | \quad \mathbf{0} \\ D &\stackrel{\text{def}}{=} J \triangleright P \quad | \quad D \wedge D \quad | \quad \mathbf{T} \\ J &\stackrel{\text{def}}{=} x\langle\widetilde{v}\rangle \quad | \quad J|J \end{aligned}$$

A process P is the asynchronous emission of a message $x\langle\widetilde{v}\rangle$, the definition of port names, the parallel composition of processes, or the null process. A definition D is made of a few reaction rules $J \triangleright P$ connected by the $\wedge$ operator. Such rules match join-patterns of messages J to trigger their guarded processes. They can be considered as an extension of named functions with synchronization, and obey similar lexical scoping rules:

- The formal parameters $v_1, v_2, \ldots v_n$ received in join-patterns are bound in (each instance of) the corresponding guarded process. They are pairwise distinct.
- Defined port names are recursively bound in the whole defining process **def** D **in** P, that is, in the main process P and in the guarded processes inside definition D.

A name is fresh with regards to a process or a solution when it is not free in them. We write $\{^x/_y\}$ for the substitution of name x for name variable y, and σ for an arbitrary substitution. We assume implicit α-conversion on bound variables to avoid clashes. Received variables $rv[J]$, defined variables $dv[J]$ and $dv[D]$, and free variables $fv[D]$ and $fv[P]$ are formally defined for the full calculus in Figure 2.

Local chemistry A *reflexive solution* $\mathcal{D} \vdash \mathcal{P}$ consists of two parts: $\mathcal{P}$ is a multiset of running processes; $\mathcal{D}$ is a multiset of active rules. Such reaction rules define the possible reductions of processes, while processes can introduce new names and reaction rules. The chemical rules are:

$$
\begin{array}{llcll}
\textbf{str-join} & \vdash P_1|P_2 & \rightleftharpoons & \vdash P_1, P_2 & \\
\textbf{str-null} & \vdash \mathbf{0} & \rightleftharpoons & \vdash & \\
\textbf{str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash & \\
\textbf{str-nodef} & \mathbf{T} \vdash & \rightleftharpoons & \vdash & \\
\textbf{str-def} & \vdash \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}} & (range(\sigma_{\mathrm{dv}})\ \text{fresh}) \\
\\
\textbf{red} & J \triangleright P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\mathrm{rv}} &
\end{array}
$$

The first four structural rules state that $|$ and $\wedge$ are associative and commutative, with units $\mathbf{0}$ and $\mathbf{T}$. The **str-def** rule provides *reflection*, with a static scoping discipline: a defining process can activate its reaction rules, substituting fresh names for its defined variables. Conversely, rules can be frozen on a process, as long as their names are local to that process. The single reduction rule **red** describes the use of active reactions ($J \triangleright P$) to consume join-messages present in the soup and produce a new instance of their guarded process.

In this paper, *the presentation of every chemical rule assumes an implicit context*. In other words, we omit the parts of multisets in chemical solutions that do not change by the effect of the presented rule. For instance, the verbose **str-def** rule is

$$\mathcal{D} \vdash \mathcal{P} \cup \{\mathbf{def}\ D\ \mathbf{in}\ P\} \rightleftharpoons \mathcal{D} \cup \{D\sigma_{\mathrm{dv}}\} \vdash \mathcal{P} \cup \{P\sigma_{\mathrm{dv}}\}$$

with the side-condition $\sigma_{\mathrm{dv}} : dv[D] \mapsto (\mathcal{N} - fv[\mathcal{P}] - fv[\mathcal{D}] - fv[\mathbf{def}\ D\ \mathbf{in}\ P])$.

Example 1. The simplest process is written $x\langle y\rangle$; it sends a name y on some other name x. In examples, we shall assume the existence of basic values, such as integers, strings, etc. For instance, assuming a printing service has been defined on name *print*, we would write $print\langle 3\rangle$. A program would be of the form

$$\mathbf{def}\ print\langle x\rangle \triangleright \ldots\ \mathbf{in}\ print\langle 3\rangle$$

To print several integers in order, we would need the printer to send back some message upon completion. For that purpose, the printer should be given a *return channel* κ together with every job.

$$\mathbf{def}\ print\langle x, \kappa\rangle \triangleright \ldots \kappa\langle\rangle \ldots\ \mathbf{in}\ \mathbf{def}\ \kappa\langle\rangle \triangleright print\langle 4, \kappa'\rangle\ \mathbf{in}\ print\langle 3, \kappa\rangle$$

In practice, sequential control is so common that it deserves some syntactic sugar to make continuations implicit, as in the language PICT [13]. We write:

$$\mathbf{def}\ \mathrm{print}(x) \triangleright \ldots \mathbf{reply\ \ to}\ \mathrm{print} \ldots\ \mathbf{in}\ \mathrm{print}(3); \mathrm{print}(4)$$

Synchronous names are written "x" and "print" instead of "x" and "$print$" to remind that they also carry an implicit continuation channel κ_{x}. In their definitions, we use fresh names κ_{x}, and we translate:

$$\begin{aligned} \mathrm{x}(\widetilde{v}) &\stackrel{\text{def}}{=} x\langle \widetilde{v}, \kappa_{\mathrm{x}} \rangle \text{ (in join-patterns J)} \\ \textbf{reply } \widetilde{V} \textbf{ to } \mathrm{x} &\stackrel{\text{def}}{=} \kappa_{\mathrm{x}}\langle \widetilde{V} \rangle \quad \text{(in guarded processes P)} \end{aligned}$$

On the caller's side, we introduce let-bindings, sequences, and nested calls. We use a reserved name κ, and we translate top-down, left-to-right:

$$\begin{aligned} x\langle \widetilde{V} \rangle &\stackrel{\text{def}}{=} \textbf{let } \widetilde{v} = \widetilde{V} \textbf{ in } x\langle \widetilde{v} \rangle \\ \textbf{let } \widetilde{u} = \mathrm{x}(\widetilde{V}) \textbf{ in } P &\stackrel{\text{def}}{=} \textbf{def } \kappa\langle \widetilde{u} \rangle \triangleright P \textbf{ in } x\langle \widetilde{V}, \kappa \rangle \\ \textbf{let } u = v \textbf{ in } P &\stackrel{\text{def}}{=} P\{^{u}/_{v}\} \\ \textbf{let } \widetilde{u} = \widetilde{V} \textbf{ in } P &\stackrel{\text{def}}{=} \textbf{let } u_1 = V_1 \textbf{ in let } u_2 = \ldots \textbf{ in } P \quad \text{(otherwise)} \\ \mathrm{x}(\widetilde{V}); P &\stackrel{\text{def}}{=} \textbf{def } \kappa\langle\rangle \triangleright P \textbf{ in } x\langle \widetilde{V}, \kappa \rangle \end{aligned}$$

3.3 Observation

We choose the observational equivalence framework as a formal basis for reasoning about processes [8, 6]. A first step is to define a reduction relation on processes, as a combination of heating, chemical reduction and cooling:

$$P \to P' \stackrel{\text{def}}{=} \emptyset \vdash \{P\} \quad (\rightleftharpoons^{*} \longrightarrow \rightleftharpoons^{*}) \quad \emptyset \vdash \{P'\}$$

In the definition above, the notation $\emptyset \vdash \{P\}$ stands for a chemical solution that contains no definitions and only one running process P.

Then, our idea of observation is to characterize processes by their capabilities to emit on certain names. Testing one particular name is enough: let "$test$" be that name. We define the *testing predicate* $\Downarrow$ as follows:

$$P \Downarrow \stackrel{\text{def}}{=} \exists P', \quad P \to^{*} (P' \mid test\langle\rangle)$$

Hence, the test succeeds when output on the name $test$ is enabled, possibly after some internal reductions took place.

The *observational congruence* is the largest equivalence relation $\approx$ that meets the following requirements:

- $\approx$ is a refinement of $\Downarrow$;
- $\approx$ is a congruence;
- $\approx$ is a weak bisimulation. That is, for all processes P and Q such that $P \approx Q$ holds, we have the following implication:

$$P \to^{*} P' \text{ implies } \exists Q', Q \to^{*} Q' \text{ and } P' \approx Q'$$

This equivalence is as discriminating as the barbed bisimulation congruence, which would test emission on every name x. We refer to [6] for discussion, examples and proof methods.

The above definition of observational congruence is parametric in the reduction relation and in the context syntax. As we refine the calculus, we will apply the same definition to yield refined equivalences.

4 Computing with locations

We now refine the reflexive CHAM to model distributed systems. First, we partition processes and definitions into several *local solutions*. This flat model suffices for representing both local computation on different sites and global communication between them. Then, we introduce some more structure to account for creation and migration of local solutions: we attach *location names* to solutions, and we organize them as a tree of nested locations

4.1 Distributed solutions

A distributed reflexive chemical machine (DRCHAM) is a multiset of CHAMs; we write its global state as several solutions $\mathcal{R}_i \vdash \mathcal{P}_i$ separated by $\|$; our chemical rules do not mention the solutions that are left unchanged. Using this convention, the local solutions evolve internally by the same rules as before. They can also interact with one another by the new reduction:

$$\textbf{comm} \;\vdash x\langle\widetilde{v}\rangle \;\; \| \;\; J \triangleright P \vdash \;\longrightarrow\; \vdash \; \| \;\; J \triangleright P \vdash x\langle\widetilde{v}\rangle \qquad (x \in dv[J])$$

This rule states that a message emitted in a given solution on a port name x that is remotely defined can be forwarded to the solution of its definition. Later on, this message can be consumed there using the **red** rule. This two-step decomposition of global communication reflects what happens at run-time in actual implementations, where message transport and message treatment are distinct operations. We only consider *well-formed* DRCHAMs, where every name is defined in at most one solution. Hence, the transport is deterministic, static, and point-to-point, and synchronization is only done locally on the receiving site during message treatment. As a distributed model of computation, the DRCHAM hides the details of message routing, but not those of synchronization.

4.2 The location tree

In order to compute with locations, we view them both as syntactic definitions and local chemical solutions; we use *location names* to relate the two. The set of location names is denoted by $\mathcal{L}$; we use the letters $a, b, \ldots \in \mathcal{L}$ for location names, and $\varphi, \psi \ldots \in \mathcal{L}^*$ for finite strings of location names.

Running locations are local labeled solutions $\mathcal{R} \vdash_\varphi \mathcal{P}$. We define the sublocation relation as: $\vdash_\varphi$ is a sublocation of $\vdash_\psi$ when ψ is a prefix of φ. In the following, DRCHAMs are multisets of labeled solutions whose labels φ are distinct, prefix-closed, and uniquely identified by their rightmost location name, if any. These conditions ensures that solutions ordered by the sublocation relation form a tree.

Location names are first-class values that statically identify a location. Like port names, they can be created locally, sent and received in messages, and they obey the lexical scoping discipline. To introduce new locations, we extend the syntax of definitions with a new location constructor:

$$D \stackrel{\text{def}}{=} \ \ldots \ | \ a\,[D : P]$$

In the heating direction, the semantics of this new construct is to create a sublocation of the current location containing the unique definition D and the unique running process P. More precisely, we have a new structural rule:

$$\textbf{str-loc} \ \ a\,[D : P] \vdash_{\varphi} \rightleftharpoons \vdash_{\varphi} \,\|\, \ \{D\} \vdash_{\varphi a} \{P\} \quad (a \text{ frozen})$$

The side condition means that there are no solutions of the form $\vdash_{\varphi a \psi}$ where ψ is a non-empty label. As the definition D could contain sublocation definitions, this side condition guarantees that D syntactically captures the whole subtree of a sublocations. Such a complete cooling has a "freezing effect" on locations and will be useful later for controlling migration.

All previous chemical rules apply unchanged, except for the explicit labeling of solutions. However, it is worth noticing that **str-def** also applies to defined location names, introducing fresh locations in running processes. In well-formed DRCHAMs, all reaction rules defining one name belong to a single location. To maintain this invariant when we dilute definitions, we constrain the syntax accordingly: in a multiple definition $D \wedge D'$, $dv[D] \cap dv[D']$ contains only port names that are not defined under a sublocation of D or D'.

Example 2. The simplest example of distribution is to send a value to a remote name. For instance, we may assume that the printer is running at location s (the server), while the print request is sent from another location c (the client):

$$\mathrm{print}(x) \triangleright \ldots \vdash_s \quad \| \quad \vdash_c \mathrm{print}(3); \ldots$$

The definition of print at location s is in the solution. In particular, it can be used from the client c.

Example 3. Remote procedure call is an abstraction of the previous example: it sends a value x to a remote service f and waits for a result.

$$\begin{array}{l} \quad \mathrm{f}(y) \triangleright \textbf{reply} \ \mathrm{computation}(y) \ \textbf{to} \ \mathrm{f} \vdash_s \\ \| \vdash_c \textbf{def} \ \mathrm{rpc}(\mathrm{g}, x) \triangleright \textbf{reply} \ \mathrm{g}(x) \ \textbf{to} \ \mathrm{rpc} \ \textbf{in} \ \ldots \mathrm{rpc}(\mathrm{f}, 3) \ldots \end{array}$$

As above, f is visible from both solutions. By contrast, rpc is local to c, and can be considered as part of its communication library. We can also use a more elaborate definition of rpc that handles timeouts:

$$\begin{array}{l} \textbf{def} \ \mathrm{rpc}(\mathrm{f}, x, \mathit{error}) \triangleright \\ \quad\quad \textbf{def} \ \mathit{incall}\langle\rangle \mid \mathit{done}\langle r \rangle \triangleright \textbf{reply} \ r \ \textbf{to} \ \mathrm{rpc} \\ \quad\quad\ \ \wedge \ \mathit{incall}\langle\rangle \mid \mathit{timeout}\langle\rangle \triangleright \mathit{error}\langle\rangle \\ \quad\quad\ \ \textbf{in} \ \ \mathit{incall}\langle\rangle \mid \mathit{done}\langle \mathrm{f}(x) \rangle \mid \mathit{start\_timer}\langle \mathit{timeout}, 3 \rangle \\ \ \textbf{in} \ \ldots \mathrm{rpc}(\mathrm{f}, 3, \mathit{error\_handler}) \ldots \end{array}$$

The *incall* message guarantees mutual exclusion between the normal return from the remote call and the timeout error message.

4.3 Migration

We are now ready to extend the syntax of processes with a new primitive for migration, along with a new chemical reduction:

$$P \stackrel{\text{def}}{=} \ \ldots \ | \ go\langle b, \kappa\rangle$$

$$\textbf{move} \ \ a\,[D : P|go\langle b,\kappa\rangle] \vdash_{\varphi} \,\|\, \vdash_{\psi b} \longrightarrow \vdash_{\varphi} \,\|\, \ a\,[D : P|\kappa\langle\rangle] \vdash_{\psi b}$$

Informally, the location a moves from its current position φa in the tree, to a new position ψba just under b. The destination solution $\vdash_{\psi b}$ is identified by its relative name b. Once a arrives, the continuation $\kappa\langle\rangle$ can trigger other computations. In case the rule **str-loc** has been used beforehand to cool down location a into a definition, its side-condition (a frozen) forces all the sublocations of a to migrate at the same time. As a consequence, migration to a sublocation is ruled out, and nested migrations in parallel are confluent.

In the paper, we use the same notation for port names and for primitives like $go\langle\cdot,\cdot\rangle$. We extend the synchronous call convention accordingly for $\mathrm{go}(\cdot)$. Notice, however, that primitives are not first-class names: they cannot be sent as values in messages.

Example 4. Another example of distribution is to download code from a code server *à la Java* for the computation to take place on the local site.

$$\begin{array}{ll} & \mathrm{load\_applet}(a) \triangleright \textbf{def}\ b[\mathrm{applet}(y) \triangleright \textbf{reply}\ \ldots\ \textbf{to}\ \mathrm{applet} \\ & \qquad\qquad\qquad\qquad\qquad\qquad : \mathrm{go}(a); \textbf{reply}\ \mathrm{applet}\ \textbf{to}\ \mathrm{load\_applet}]\ \textbf{in}\ \mathbf{0}\ \vdash_s \\ \| & \vdash_c \textbf{let}\ \mathrm{f} = \mathrm{load\_applet}(c)\ \textbf{in}\ \ldots \mathrm{f}(3) \ldots \end{array}$$

This reduces to the same server, and a local copy of the applet:

$$\begin{array}{ll} & \mathrm{load\_applet}(a) \triangleright \textbf{def}\ b[\mathrm{applet}(y) \triangleright \textbf{reply}\ \ldots\ \textbf{to}\ \mathrm{applet} \\ & \qquad\qquad\qquad\qquad\qquad\qquad : \mathrm{go}(a); \textbf{reply}\ \mathrm{applet}\ \textbf{to}\ \mathrm{load\_applet}]\ \textbf{in}\ \mathbf{0}\ \vdash_s \\ \| & b'[\mathrm{applet}(y) \triangleright \textbf{reply}\ \ldots\ \textbf{to}\ \mathrm{applet}] \vdash_c \ldots \mathrm{applet}(3) \ldots \end{array}$$

Assuming that the applet does not include another *go* primitive, b' remains attached to c and the program behaves as if a fresh copy of the applet had been defined at location c.

4.4 Building our CASA

The opposite of retrieving code is sending computation to a remote server. The client defines the request; the request moves to the server, runs there, and sends the result back to the client:

$$\textbf{def}\ \mathrm{f}(x,s) \triangleright a[\mathrm{go}(s); \textbf{reply}\ \ldots\ \textbf{to}\ \mathrm{f} : \mathbf{0}]\ \textbf{in}\ \ldots \mathrm{f}(3, \mathit{server}) \ldots$$

In the code above, the remote computation returns a tuple of basic values. In general however, the result might contain arbitrary data allocated during the computation, or even active data (processes with internal state). In the generic CASA, the server cannot just return a pointer to the data; it must also move the data and the code back to the client location. To illustrate this, we consider an agent that allocates and uses a reference cell; new_cell creates a fresh cell and returns its two methods, set for updates and get for access.

$$
\begin{array}{l}
\textbf{def } c[\mathrm{f}(x,s) \triangleright \\
\quad \textbf{def } a[\mathbf{T} : \mathrm{go}(s); \\
\quad\quad \textbf{let } \mathrm{set}, \mathrm{get} = \mathrm{new\_cell}(a) \\
\quad\quad \textbf{ in } \mathrm{set}(\mathrm{computation}(x)); \mathrm{go}(c); \textbf{reply } \mathrm{get} \textbf{ to } \mathrm{f}] \\
\quad \textbf{ in } \ldots : \mathbf{0}] \\
\textbf{ in } \ldots \mathrm{f}(3, \mathit{server}) \ldots
\end{array}
$$

The data is allocated within the agent at location a, upon arrival on the server. It does not need to be pre-allocated, and grows on demand during the computation. Eventually, the agent is repatriated to the client by the $\mathrm{go}(c)$ primitive call.

5 Failure and recovery

Modeling failures is the litmus test for a distributed computation formalism. In the absence of failures, locations have only pragmatic significance, and no semantic importance. In fact, it was our incapacity to come up with a simple failure model for the π-calculus that spawned the join-calculus.

In this section we present our failure model, we introduce our two failure management primitives, we show their use in examples, and finally we discuss the choice of our failure model.

5.1 Representing failures

We use a marker $\Omega \notin \mathcal{L}$ to tag failed locations. For every $a \in \mathcal{L}$, εa denotes either a or Ωa, and φ, ψ denote strings of such εa. In the DRCHAM, Ω appears in the location string φ of failed locations $\vdash_\varphi$. We say φ is *dead* if it contains Ω, and *alive* otherwise; the position of the tag indicates where the failure was triggered. In the process syntax, failed locations are frozen as tagged definitions $\Omega a\,[D : P]$; thus the general shape of a location definition is $\varepsilon a\,[D : P]$.

In order to preserve scopes, structural rules are allowed in failed locations, hence the structural rules in Figure 3 are almost unchanged from sections 3–4, except for the obvious generalization of **str-loc** to the failed location syntax.

We model failure by prohibiting reactions inside a failed location or any of its sublocations. More precisely, in Figure 3 we add a side condition to **red**, **comm**, and **move**, that prevents these rules from taking messages (or $go\langle\cdot,\cdot\rangle$ primitives) in a solution with a dead label. Note however that we do not prevent messages or even locations from moving to a failed location, as such deadly moves are unavoidable in an asynchronous distributed setting.

Because failure can only occur in a named location, the top solution $\vdash$ provides a "safe haven" where pervasive definitions, such as the behavior of integers, may be put. Because of this we need to consider two equivalences for the calculus with failures: a "static equivalence" that is a congruence for all but the

$$
\begin{array}{rll}
P \stackrel{\text{def}}{=} & x\langle\widetilde{v}\rangle & \text{message} \\
& \mathbf{def}\ D\ \mathbf{in}\ P & \text{definition} \\
& \mathbf{0} & \text{inert process} \\
& P|P & \text{composition} \\
& go\langle a,\kappa\rangle & \text{migration} \\
& halt\langle\rangle & \text{termination} \\
& fail\langle a,\kappa\rangle & \text{failure detection}
\end{array}
\qquad
\begin{array}{rll}
D \stackrel{\text{def}}{=} & J \triangleright P & \text{local rule} \\
& \mathbf{T} & \text{inert definition} \\
& D \wedge D & \text{co-definition} \\
& a\,[D:P] & \text{sub-location} \\
& \Omega a\,[D:P] & \text{dead sub-location} \\
\\
J \stackrel{\text{def}}{=} & x\langle\widetilde{v}\rangle & \text{message pattern} \\
& J|J & \text{join-pattern}
\end{array}
$$

Fig. 1. Syntax for the distributed-join-calculus

$$
\begin{array}{lll@{\qquad}lll}
J: & dv[x\langle\widetilde{v}\rangle] & \stackrel{\text{def}}{=} \{x\} & & rv[x\langle\widetilde{v}\rangle] & \stackrel{\text{def}}{=} \{u \in \widetilde{v}\} \\
& dv[J \mid J'] & \stackrel{\text{def}}{=} dv[J] \cup dv[J'] & & rv[J \mid J'] & \stackrel{\text{def}}{=} rv[J] \uplus rv[J'] \\
\\
D: & dv[J \triangleright P] & \stackrel{\text{def}}{=} dv[J] & & fv[J \triangleright P] & \stackrel{\text{def}}{=} dv[J] \cup (fv[P] - rv[J]) \\
& dv[\mathbf{T}] & \stackrel{\text{def}}{=} dv[\emptyset] & & fv[\mathbf{T}] & \stackrel{\text{def}}{=} \emptyset \\
& dv[D \wedge D'] & \stackrel{\text{def}}{=} dv[D] \cup dv[D'] & & fv[D \wedge D'] & \stackrel{\text{def}}{=} fv[D] \cup fv[D'] \\
& dv[a\,[D:P]] & \stackrel{\text{def}}{=} \{a\} \uplus dv[D] & & fv[a\,[D:P]] & \stackrel{\text{def}}{=} \{a\} \cup fv[D] \cup fv[P] \\
\\
P: & fv[x\langle\widetilde{v}\rangle] & \stackrel{\text{def}}{=} \{x\} \cup \{u \in \widetilde{v}\} & & fv[go\langle a,\kappa\rangle] & \stackrel{\text{def}}{=} \{a,\kappa\} \\
& fv[\mathbf{0}] & \stackrel{\text{def}}{=} \emptyset & & fv[halt\langle\rangle] & \stackrel{\text{def}}{=} \emptyset \\
& fv[P \mid P'] & \stackrel{\text{def}}{=} fv[P] \cup fv[P'] & & fv[fail\langle a,\kappa\rangle] & \stackrel{\text{def}}{=} \{a,\kappa\} \\
& fv[\mathbf{def}\ D\ \mathbf{in}\ P] & \stackrel{\text{def}}{=} (fv[P] \cup fv[D]) - dv[D] & & &
\end{array}
$$

Well-formed conditions for D: In a scope, location variables can be defined only once; port variables can only appear in the join-patterns of one location (cf. 3.2, 4.2)

Fig. 2. Scopes for the distributed-join-calculus

$$
\begin{array}{lrcll}
\textbf{str-join} & \vdash P_1|P_2 & \rightleftharpoons & \vdash P_1, P_2 & \\
\textbf{str-null} & \vdash \mathbf{0} & \rightleftharpoons & \vdash & \\
\textbf{str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash & \\
\textbf{str-nodef} & \mathbf{T} \vdash & \rightleftharpoons & \vdash & \\
\textbf{str-def} & \vdash \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D\sigma_{\text{dv}} \vdash P\sigma_{\text{dv}} & (range(\sigma_{\text{dv}})\text{ fresh}) \\
\textbf{str-loc} & \varepsilon a\,[D:P] \vdash_\varphi & \rightleftharpoons & \vdash_\varphi \parallel \{D\} \vdash_{\varphi\varepsilon a} \{P\} & (a\text{ frozen}) \\
\\
\textbf{red} & J \triangleright P \vdash_\varphi J\sigma_{\text{rv}} & \longrightarrow & J \triangleright P \vdash_\varphi P\sigma_{\text{rv}} & (\varphi\text{ alive}) \\
\textbf{comm} & \vdash_\varphi x\langle\widetilde{v}\rangle \parallel J \triangleright P \vdash & \longrightarrow & \vdash_\varphi \parallel J \triangleright P \vdash x\langle\widetilde{v}\rangle & (x \in dv[J],\ \varphi\text{ alive}) \\
\textbf{move} & a\,[D:P|go\langle b,\kappa\rangle] \vdash_\varphi \parallel \vdash_{\psi\varepsilon b} & \longrightarrow & \vdash_\varphi \parallel a\,[D:P|\kappa\langle\rangle] \vdash_{\psi\varepsilon b} & (\varphi\text{ alive}) \\
\textbf{halt} & a\,[D:P|halt\langle\rangle] \vdash_\varphi & \longrightarrow & \Omega a\,[D:P] \vdash_\varphi & (\varphi\text{ alive}) \\
\textbf{detect} & \vdash_\varphi fail\langle a,\kappa\rangle \parallel \vdash_{\psi\varepsilon a} & \longrightarrow & \vdash_\varphi \kappa\langle\rangle \parallel \vdash_{\psi\varepsilon a} & (\psi\varepsilon a\text{ dead},\ \varphi\text{ alive})
\end{array}
$$

Side conditions: in **str-def**, σ_{dv} instantiates the port variables $dv[D]$ to distinct, fresh names; in **red**, σ_{rv} substitutes the transmitted names for the received variables $rv[J]$; "a frozen" means that a has no sublocations in solution; φ is dead if it contains Ω, and alive otherwise.

Fig. 3. The distributed reflexive chemical machine

$\varepsilon a\,[\cdot : \cdot]$ constructor, and a "mobile equivalence" that is a congruence for the full calculus. The two notions coincide for processes that do not export agents.

5.2 Primitives for failure and recovery

We introduce two new primitives $halt\langle\rangle$ and $fail\langle\cdot,\cdot\rangle$. A $halt\langle\rangle$ at location a can make this location permanently inert (rule **halt** in Figure 3), while $fail\langle a,\kappa\rangle$ triggers $\kappa\langle\rangle$ after it detects that a has failed, i.e. that a or one of its parent locations has halted (rule **detect**). Note that the (φ alive) side condition in rules **move** and **comm** are sufficient to prevent all output from a dead location; it is attached to rules **red**, **halt**, and **detect** only for consistency.

In conjunction with the static equivalence, the $halt\langle\rangle$ primitive allows us to use the calculus to express the site failure patterns under which we prove an equivalence: a top-level location that does not move can only fail if it executes a $halt\langle\rangle$. In addition, $halt\langle\rangle$ can be used to encode a "kill" operation, as in

$$\begin{array}{l}\textbf{def } b[\mathit{kill}\langle\rangle \triangleright \mathit{halt}\langle\rangle : \mathit{start\_timer}\langle \mathit{kill}, 5\rangle] \\ \ \textbf{in let } \mathrm{f} = \mathrm{load\_applet}(b) \textbf{ in } \ldots \mathrm{f}(3) \ldots\end{array}$$

The $\mathit{fail}\langle\cdot,\cdot\rangle$ primitive provides a natural guard for error recovery. For example, we can make the CASA more secure as follows:

$$\mathrm{f}(x,s) \triangleright \textbf{def } a[\ldots] \textbf{ in } (\mathrm{fail}(a); \textbf{reply } \mathrm{f}(x,s') \textbf{ to } \mathrm{f})$$

If no error occurs, the agent returns permanently to the client, hence the fail is permanently disabled. Conversely, if the fail triggers then the server must have failed while hosting agent a. As this agent cannot return to the server, a new agent is created and sent to another server. Anyway, we are assured that there is at most one agent at large, and that its action is only completed once (which might be quite important, say if the action is "get a plane ticket"). This would still be true if the client did not know the server location, and the agent moved through several intermediate sites before reaching the server location.

This uniqueness property is difficult to obtain with timeouts only. The $\mathit{fail}\langle\cdot,\cdot\rangle$ primitive provides more information than timeouts do. However, timeouts are easier to implement and to model (they are just silent transitions in any bisimulation-based process calculus), so they are a natural complement of *fail*s. Indeed, for RPC-like interactions that are asynchronous and without side effects, there is little practical use for the uniqueness property, so a simpler timeout is preferable to a *fail* check.

5.3 Failure models

What does "failure" mean? The most conservative answer, in our message-passing setting, is that when a location fails, *some* messages to, at, or from the location are lost. However it is very hard to do sensible error recovery in such a weak model: it is impossible to issue a replacement b for a failed agent a without running the risk of having a and b interfere through side effects.

Assuming that all messages from a failed agent are lost would solve this. Unfortunately this strong model is not consistent with the **comm** rule and our asynchronous, distributed setting. It would require that the system track and delete all messages issued by a failing location.

A more reasonable requirement would be that a failed location a cannot respond to messages; this can be enforced by blocking output to a from all locations detecting an a failure (or having received messages triggered by that failure-detection). This "weak asynchronous" model can easily be seen to be testing-equivalent to our "strong asynchronous" model (simply delay the failure until all the required output from a leave a); hence we are justified in using the stricter, simpler model in the calculus, but only implementing the weaker one. However, the models do give different interpretations to $a\,[\mathbf{T} : \mathit{halt}\langle\rangle \mid x\langle\rangle \mid x\langle\rangle]$ under bisimulation congruence.

6 Proofs for mobile protocols

The primary purpose of our calculus is to found a core language with enough expressivity for distributed and mobile programming. But locations with their primitives can also be used to model fallible distributed environments, as specific contexts within the calculus. As a result, we can use our observational equivalence to relate precisely the distributed implementations with their specification (i.e. simpler programs and contexts without failures or distribution). In combination with the usual proof methods developed for other process calculi, this should provide a setting for the design and the proof of distributed programs under realistic assumptions.

In this section, we explore this setting through a few simple examples and an internal encoding of locations. The equivalence relation $\approx$ is the observational congruence defined in section 3, applied to the distributed join-calculus of section 5. Due to lack of space, proofs are omitted.

6.1 A sample of equational laws

First, we state several "garbage collection" laws which are useful for simplifying terms in proofs: we have $P \approx \mathbf{0}$ when P resides in a failed location, when it is guarded by patterns of messages that cannot be assembled, or when it has neither free port names nor $\mathit{halt}\langle\rangle$, $\mathit{go}\langle\cdot,\cdot\rangle$ primitives.

Second, some basic laws hold for the $\mathit{go}\langle\cdot,\cdot\rangle$, $\mathit{fail}\langle\cdot,\cdot\rangle$, and $\mathit{halt}\langle\rangle$ primitives. For instance, we have $\mathrm{fail}(a); \mathrm{fail}(b); P \approx \mathrm{fail}(b); \mathrm{fail}(a); P$. Because these primitives are strictly static, the analysis of their local usage yields simplifications of the location tree. The following laws show how to get rid of location b once it has reached its final destination a : when D, P contain neither $\mathit{go}\langle\cdot,\cdot\rangle$ nor $\mathit{halt}\langle\rangle$, the b boundary is irrelevant:

$$\mathbf{def}\ a\,[b\,[D : P] \wedge D' : P']\ \mathbf{in}\ P'' \approx \mathbf{def}\ a\,[b\,[\mathbf{T} : \mathbf{0}] \wedge D \wedge D' : P \mid P']\ \mathbf{in}\ P''$$

When a location is empty, migrations and failure-detections using its name b or its parent's name a cannot be distinguished:

$$\mathbf{def}\ a\,[b\,[\mathbf{T} : \mathbf{0}] \wedge D : P]\ \mathbf{in}\ P' \approx (\mathbf{def}\ a\,[D : P]\ \mathbf{in}\ P')\,\{^a/_b\}$$

6.2 Internal encoding

We present a translation from the distributed join-calculus with all the features introduced in section 4 and section 5, into the simpler join-calculus of section 3. In combination with the encoding of the join-calculus into the π-calculus [6], this provides an alternative definition of migration and failure in the usual setting of process calculi. This also suggests that our distributed extension does not unduly add semantic complexity.

The basic idea is to replace every location construct by a definition that supports an equivalent protocol, and every use of locality information by a message call for this protocol. Once this is done, the structural translation $[\![P]\!]$ of the distributed process P simply makes explicit the side-conditions of the DRCHAM. For instance, we have $[\![x\langle 1\rangle]\!] = \mathrm{ping}(); x\langle 1\rangle$, where ping() checks that the current location is alive before returning, thus mimicking the **comm** rule.

The interface to the encoding of location a consists of two port names. a stands for the location value; h_a provides internal access to the current location. They are sent to the encoding of location primitives (ping, fail, *halt*, go, subloc). The corresponding implementation $\mathcal{E}(\cdot)$ defines these primitives and the top-level location:

$$\begin{array}{l}
\mathbf{def}\ \mathrm{subloc}(\mathrm{h}_0) \triangleright \\
\quad \mathbf{def}\ \mathit{live}\langle \mathrm{p}\rangle \mid \mathrm{poll}() \triangleright \mathbf{let}\ \mathrm{r} = \mathrm{p}()\ \mathbf{in}\ (\mathit{live}\langle \mathrm{p}\rangle \mid \mathbf{reply}\ \mathrm{r}\ \mathbf{to}\ \mathrm{poll}) \\
\quad\ \ \wedge\ \mathit{live}\langle \mathrm{p}\rangle \mid \mathit{kill}\langle\rangle \triangleright \mathit{dead}\langle \mathrm{p}\rangle \\
\quad\ \ \wedge\ \mathit{dead}\langle \mathrm{p}\rangle \mid \mathrm{poll}() \triangleright \mathbf{let}\ \mathrm{r} = \mathrm{p}()\ \mathbf{in} \\
\qquad\qquad \mathit{dead}\langle \mathrm{p}\rangle \mid \mathbf{reply}\ (\mathbf{if}\ \mathrm{r} = \mathrm{alive}\ \mathbf{then}\ \mathrm{failed}\ \mathbf{else}\ \mathrm{r})\ \mathbf{to}\ \mathrm{poll} \\
\quad\ \ \wedge\ \mathit{live}\langle \mathrm{p}\rangle \mid \mathrm{get}() \triangleright \mathit{lock}\langle\rangle \mid \mathbf{reply}\ \mathrm{p}\ \mathbf{to}\ \mathrm{get} \\
\quad\ \ \wedge\ \mathit{lock}\langle\rangle \mid \mathrm{poll}() \triangleright \mathit{lock}\langle\rangle \mid \mathbf{reply}\ \mathrm{retry}\ \mathbf{to}\ \mathrm{poll} \\
\quad\ \ \wedge\ \mathit{lock}\langle\rangle \mid \mathit{set}\langle \mathrm{p}\rangle \triangleright \mathit{live}\langle \mathrm{p}\rangle\ \mathbf{in} \\
\quad \mathbf{def}\ \mathrm{here}() \triangleright \mathbf{reply}\ \mathrm{poll}, \mathit{kill}, \mathrm{get}, \mathit{set}\ \mathbf{to}\ \mathrm{here}\ \mathbf{in} \\
\quad \mathbf{let}\ \mathrm{poll}_0, _, _, _ = \mathrm{h}_0()\ \mathbf{in}\ \mathit{live}\langle \mathrm{poll}_0\rangle \mid \mathbf{reply}\ \mathrm{poll}, \mathrm{here}\ \mathbf{to}\ \mathrm{subloc}\ \mathbf{in} \\
\mathbf{def}\ \mathrm{ping}(\mathrm{h}) \triangleright \mathbf{let}\ \mathrm{p}, _, _, _ = \mathrm{h}()\ \mathbf{in}\ \mathbf{repeat}\ \mathrm{p}()\ \mathbf{until}\ \mathrm{alive}; \mathbf{reply}\ \ \mathbf{to}\ \mathrm{ping}\ \mathbf{in} \\
\mathbf{def}\ \mathrm{fail}(\mathrm{p}) \triangleright \mathbf{repeat}\ \mathrm{p}()\ \mathbf{until}\ \mathrm{failed}; \mathbf{reply}\ \ \mathbf{to}\ \mathrm{fail}\ \mathbf{in} \\
\mathbf{def}\ \mathit{halt}\langle \mathrm{h}\rangle \triangleright \mathbf{let}\ _, \mathit{kill}, _, _ = \mathrm{h}()\ \mathbf{in}\ \mathit{kill}\langle\rangle\ \mathbf{in} \\
\mathbf{def}\ \mathrm{go}(\mathrm{h}, \mathrm{p}') \triangleright \mathbf{let}\ \mathrm{p}, _, \mathrm{get}, \mathrm{set} = \mathrm{h}()\ \mathbf{in} \\
\quad \mathbf{def}\ \mathrm{attempt}() \triangleright \mathbf{if}\ (\mathbf{if}\ \mathrm{p}'() = \mathrm{retry}\ \mathbf{then}\ \mathrm{failed}\ \mathbf{else}\ \mathrm{p}()) = \mathrm{alive} \\
\qquad\quad \mathbf{then}\ \mathit{set}\langle \mathrm{p}'\rangle \mid \mathbf{reply}\ \mathrm{done}\ \mathbf{to}\ \mathrm{attempt} \\
\qquad\quad \mathbf{else}\ \mathit{set}\langle \mathrm{p}\rangle \mid \mathbf{reply}\ \mathrm{retry}\ \mathbf{to}\ \mathrm{attempt}\ \mathbf{in} \\
\quad \mathbf{repeat}\ \mathrm{attempt}(\mathrm{get}())\ \mathbf{until}\ \mathrm{done}\ \mathbf{in} \\
\mathbf{def}\ \mathrm{here}() \triangleright \\
\quad \mathbf{def}\ \mathrm{top}() \triangleright \mathbf{reply}\ \mathrm{alive}\ \mathbf{to}\ \mathrm{top}\ \ \wedge\ \ \mathrm{top}() \triangleright \mathbf{reply}\ \mathrm{retry}\ \mathbf{to}\ \mathrm{top} \\
\quad\ \ \wedge\ \mathit{kill}\langle\rangle \mid \mathrm{get}() \mid \mathit{set}\langle \mathrm{p}\rangle \triangleright \mathbf{0}\ \mathbf{in} \\
\quad \mathbf{reply}\ \mathrm{top}, \mathit{kill}, \mathrm{get}, \mathit{set}\ \mathbf{to}\ \mathrm{here}\ \mathbf{in} \\
\mathbf{def}\ \mathit{start}\langle \mathrm{h}_s, \widetilde{a}\rangle \triangleright (\cdot)\ \mathbf{in}\ \mathit{init}\langle \mathit{start}, \mathrm{here}, \mathrm{ping}, \mathrm{fail}, \mathit{halt}, \mathrm{go}, \mathrm{subloc}\rangle
\end{array}$$

$$
\begin{aligned}
[\![\mathbf{0}]\!]_a &\stackrel{\text{def}}{=} \mathbf{0} \\
[\![x\langle\widetilde{v}\rangle]\!]_a &\stackrel{\text{def}}{=} \mathrm{ping}(\mathrm{h}_a); x\langle\widetilde{v}\rangle \\
[\![\mathit{fail}\langle a,\kappa\rangle]\!]_a &\stackrel{\text{def}}{=} \mathrm{fail}(a); [\![\kappa\langle\rangle]\!] \\
[\![\mathit{halt}\langle\rangle]\!]_a &\stackrel{\text{def}}{=} \mathit{halt}\langle\mathrm{h}_a\rangle \\
[\![\mathit{go}\langle b,\kappa\rangle]\!]_a &\stackrel{\text{def}}{=} \mathrm{go}(\mathrm{h}_a, b); [\![\kappa\langle\rangle]\!] \\
[\![P \mid P']\!]_a &\stackrel{\text{def}}{=} [\![P]\!]_a \mid [\![P']\!]_a \\
[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!]_a &\stackrel{\text{def}}{=} [\![D]\!]_a^L \left(\mathbf{def}\ [\![D]\!]_a^D\ \mathbf{in}\ ([\![D]\!]_a^P \mid [\![P]\!]_a)\right)
\end{aligned}
$$

D	$[\![D]\!]_a^D$	$[\![D]\!]_a^P$	$[\![D]\!]_a^L$
$J \triangleright P$	$[\![J]\!]_a \triangleright [\![P]\!]_a$	$\mathbf{0}$	$(\cdot)$
$b\,[D : P]$	$[\![D]\!]_b^D$	$[\![D]\!]_b^P \mid [\![P]\!]_b$	$\mathbf{let}\ b, \mathrm{h}_b = \mathrm{subloc}(\mathrm{h}_a)\ \mathbf{in}\ [\![D]\!]_b^L(\cdot)$
$D \wedge D'$	$[\![D]\!]_a^D \wedge [\![D']\!]_a^D$	$[\![D]\!]_a^P \mid [\![D']\!]_a^P$	$[\![D]\!]_a^L([\![D']\!]_a^L(\cdot))$
$\mathbf{T}$	$\mathbf{T}$	$\mathbf{0}$	$(\cdot)$

In the translation above, we assume that location names in P are pairwise distinct. We omit the formal translation of the syntactic sugar we use for control (symbolic constants, **if then else**, **repeat until**).

When placed in an arbitrary context, the encoding $\mathcal{E}([\![P]\!]_s)$ exports the *init* message. The context can set up an arbitrary location tree using the location primitives, then starts the translation in some location by providing some valid interface $\mathrm{h}, \widetilde{a}$. To keep things simple, we use a refined sort discipline for the target calculus; the port names h_a and a are given special sorts; $\approx_{local}$ is the restricted congruence over contexts that do not define or sends messages to names of these sorts. In particular, this prevents contexts from accessing our internal representation or otherwise meddling with our protocol. We believe that this limitation can be enforced using "firewall" techniques as in [6].

Theorem 1 *The encoding $\mathcal{E}([\![\cdot]\!]_s)$ is fully-abstract up-to observational congruences $\approx$ in the distributed join-calculus and $\approx_{local}$ in the join-calculus:*

$$\forall P, P', \forall \widetilde{a} \supset (fv[P, P'] \cap \mathcal{L}), \quad P \approx P' \iff \mathcal{E}([\![P]\!]_s) \approx_{local} \mathcal{E}([\![P']\!]_s)$$

As a special case, contexts of the simple join-calculus have the same discriminating power than distributed ones, as long as there is no exchange of location names. This condition automatically holds for simple processes considered as distributed processes, meaning that simple and distributed observation coincide. This is in sharp contrast with the π-calculus with locality [1], where the distributed congruence is strictly finer than the local one, even for local processes.

7 Future work

In this paper, we laid the groundwork for a calculus of distributed processes with mobility and failure, and we investigated the use of process-calculus techniques for proving distributed protocols. In complement, more specific tools are needed (weaker equivalences, fairness). In order to validate our approach, we plan to apply the distributed join-calculus to asynchronous protocols in an unreliable setting, or with security requirements; to this end, we currently experiment with the design and implementation of a high-level programming language founded on our calculus.

Acknowledgments

This work benefited from numerous discussions with Roberto Amadio, Gérard Boudol, Damien Doligez, Florent Guillaume, Benjamin Pierce, Peter Sewell, and David Turner.

References

1. R. Amadio and S. Prasad. Localities and failures. In *14th Foundations of Software Technology and Theoretical Computer Science Conference*. Springer-Verlag, 1994. LNCS 880.
2. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
3. K. A. Bharat and L. Cardelli. Migratory applications. Technical Report 138, DEC-SRC, February 1996.
4. G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. A theory of processes with localities. *Formal Aspects of Computing*, 6:165–200, 1994.
5. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995.
6. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages*, Jan. 1996.
7. A. Giacalone, P. Mishra, and S. Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
8. K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151:437–486, 1995.
9. E. Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, University of Washington, Computer Science Department, Dec. 1988.
10. L. Leth and B. Thomsen. Some facile chemistry. Technical Report ECRC-92-14, European Computer-Industry Research Centre, Munich, May 1992.
11. R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification.* Springer Verlag, 1993.
12. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 1995. To appear. A summary was presented at LICS '93.
13. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming, Sendai, Japan*, Apr. 1995. LNCS 907.
14. D. Sangiorgi. Localities and non-interleaving semantics in calculi for mobile processes. Technical Report ECS–LFCS–94–282, University of Edinburgh, 94. to appear in TCS.