A Calculus of Kells

Jean-Bernard Stefani
INRIA
jean-bernard.stefani@inria.fr

June 12, 2003

Abstract

This paper introduces the Kell calculus, a new process calculus that retains the original insights of the M-calculus (local actions, higher-order processes and programmable membranes) in a much simpler setting. The calculus is shown expressive enough to provide a direct encoding of several recent distributed process calculi such as Mobile Ambients and the Distributed Join calculus.

1 Introduction

The calculus of Mobile Ambients [5] has received much attention in the past five years, as witnessed by the numerous variants that have been proposed to overcome some of its perceived deficiencies: Safe Ambients (SA) [10], Safe Ambients with passwords [12], Boxed Ambients (BA) [3], Controlled Ambients (CA) [19], Boxed Ambients with communication control (NBA) [4], Ambients with process migration ($\mathbf{M}^3$) [6].

Mobile Ambients, unfortunately, are difficult to implement in a distributed setting. Consider, for instance, the reduction rule associated with the `in` capability in the original Mobile Ambients:

$$n[\mathtt{in}\, n.P \mid Q] \mid m[R] \rightarrow m[R \mid n[P \mid Q]]$$

This rule essentially mandates a rendez-vous between ambient n and ambient m. Thus, a distributed implementation of this rule, i.e. one where ambient n and ambient m are located on different physical sites, would require a distributed synchronization between the two sites. The inherent complexity of the required distributed synchronization has been made clear in the Distributed Join calculus implementation of Mobile Ambients reported in [8].

Part of the difficulty in implementing Mobile Ambients is related to the presence of "grave interferences", as explained in [10]. However, even with variants of Mobile Ambients with co-capabilities and a type system ensuring ambient single-threadedness (i.e. ensuring that at any one time there is at most one process inside an ambient that carries a capability), realizing ambient migration as authorized e.g. by the Safe Ambients `in` primitive

$$n[\mathtt{in}\, m.P_1 \mid P_2] \mid m[\overline{\mathtt{in}}.Q_1 \mid Q_2] \rightarrow m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2]$$

still requires a rendez-vous between ambients.

This is illustrated by the Safe Ambients abstract machine, called PAN, described in [15], which requires a 2-phase protocol involving ambients n and m above, together with their parent ambient to implement the `in` and `out` moves.

Interestingly, the PAN abstract machine is further simplified by adopting an unconventional interpretation: ambients are considered to represent only logical loci of computation, and not physical locations. Each ambient is mapped to a physical location but the `in` and `out` primitives do not modify the physical location of ambients. Instead, the `open` primitive, as a side-effect, modifies the physical location of processes running inside the ambient to be dissolved. With this interpretation, of course, ambients cease to be meaningful abstractions for the control of the physical distribution of computations.

The problem with the Mobile Ambient primitives is not so much that they are difficult to implement in a distributed setting, but that they provide the only means for communication between remote ambients.

This, in turn, means that a simple message exchange between remote ambients must bear the cost of a distributed synchronization. This is clearly not acceptable: there are many useful applications that require only simple asynchronous point-to-point message exchanges, and relying on Ambient-like primitives for remote communication would result in a heavy performance loss for these applications. As a minimum, therefore, one should look for a programming model where costly migration primitives coexist with simple asynchronous message exchange for remote communications. Boxed Ambients (BA) and their NBA adopt this approach. Communication in BA or NBA is synchronous but one can argue that it is in fact a form of local communication since it only takes place between an ambient and a process located in its parent ambient. Thus, communication between two remote ambients, i.e. siblings located in a 'network ambient', necessarily involves two different communication events (i.e. an emitting event at the ambient that originates the communication and a receiving event at the ambient that receives the communication). The Distributed Join calculus [7] makes the same separation between remote communications and locality migration, which is provided by the `go` primitive and also involves some form of distributed rendez-vous to be faithfully implemented.

Still, the distributed synchronization implied by mobility primitives raises important issues. In a distributed setting, failures are inevitable, be they permanent or transient, network or site failures. Taking into account such failures would require, as a minimum, turning mobility primitives into abortable transactions, thus preserving their atomicity but making explicit their behavior in presence of failure. This, in turn, suggest that it would be useful to split up ambient migration primitives, especially the `in` primitive of Mobile Ambients or the `enter` primitive of NBA, into finer grained primitives whose implementation need not rely on some distributed synchronization. To illustrate, one could think of splitting the Mobile Ambients `in` primitive into a pair of primitives `move` and `enter` whose behavior would be given by the following reduction rules (we use co-capabilities and passwords, as in the NBA calculus):

$$n[\mathtt{move}\langle m, h\rangle.P \mid Q] \mid \overline{\mathtt{move}}(x, y).R \rightarrow \mathtt{enter}\langle n, m, h, P, Q\rangle \mid R\{m/x, h/y\}$$

$$\mathtt{enter}\langle n, m, h, P, Q\rangle \mid m[\overline{\mathtt{enter}}(x, h).S \mid T] \rightarrow m[S\{x/n\} \mid T \mid n[P \mid Q]]$$

However, we do not pursue that approach here for several reasons. First, one may envisage further extensions allowing for more sophisticated authentication schemes, or dynamic security checks (e.g. additional parameters allowing for proof-carrying code schemes). Second, several questions remain concerning migration primitives and their combination. For instance, should we go for communications à la Boxed Ambients or should we consider instead to split up the migration primitives such as `to` migration primitive in the $\mathbf{M}^3$ calculus, yielding a form of communication similar to Dπ [9] or Nomadic Pict [21], where communication is a side-effect of process migration ? Should we allow for more objective forms of migration to reflect control that ambients can exercize on their content ?

The possible variants seem endless. This is why we follow instead the lead of higher-order calculi such as $D\lambda\pi$ [22] and the M-calculus [17], where process migration is a side-effect of higher-order communication. Indeed, as demonstrated in the M-calculus, higher-order communication, coupled with programmable localities, provides the means to model different forms of migration protocols, and different forms of locality semantics. The M-calculus avoids embedding predefined choices concerning migration primitives and their interplay. Instead, these choices can be defined, within the calculus itself, by programming the appropriate behavior in locality "membranes" (the control part P of an M-calculus locality $a(P)[Q]$). The M-calculus, however, may appear as rather complex, especially compared to Mobile Ambients. In particular, its operational semantics features several so-called routing rules which it would be interesting to reduce to a few simple cases.

The calculus we introduce in this paper is an attempt to define a calculus with process migration that avoids the need for distributed synchronization, while preserving the simplicity of Mobile Ambients and retaining the basic insights of the M-calculus: migration as higher-order communication, programmable locality "membranes". We call this new calculus the Kell calculus (the word "kell" is a variation on the word "cell", and denotes a locality or locus of computation). This calculus constitutes a direct extension of the asynchronous higher-order π-calculus with hierarchical localities.

This paper is organized as follows. Section 2 informally introduces the main constructs of the Kell calculus, together with several examples. Section 3 gives the syntax and operational semantics of the calculus. Section 4 presents several encodings of known process calculi, thus demonstrating the expressive power of the Kell calculus. Section 5 concludes the paper with a discussion of related work and of directions for further research.

2 Introducing the Kell calculus

The Kell calculus is in fact a family of calculi that share the same constructs and that differ only in the language of message patterns used in triggers (see below). In this section, we present informally the different constructs of the Kell calculus using an element of this family, that enjoys a very simple pattern language.

The core of the calculus is the asynchronous higher-order π-calculus. Among the basic constructs of the calculus we thus find:

- the *null* process, $\mathbf{0}$; process *variables*, x;
- the *restriction*, $\nu a.P$, where a is a name, P is an arbitrary Kell calculus process, and ν is a binding operator;
- the *parallel* composition, $P \mid Q$;
- *messages* of the form, $a\langle\widetilde{u}\rangle$, where a is a name, and where $\widetilde{u}$ is a vector of elements u that can be either a name or a process.
- *triggers*, or receivers, of the form $\xi \triangleright P$, where ξ is a message pattern and P is an arbitrary kell calculus process.

In this section, patterns are given by the following grammar:

$$\xi ::= a\langle\widetilde{w}\rangle \quad | \quad a\langle\widetilde{w}\rangle^{\uparrow} \quad | \quad a\langle\widetilde{w}\rangle_{\downarrow} \quad | \quad a[x]$$

where $\widetilde{w}$ is a vector of elements w, which can be either a name, a name variable of the form (b), where b is a name, or a process variable x. Name and process variables are of course bound in patterns and their scope extend to the process of the right-hand side of the trigger sign $\triangleright$.

To this higher-order π-calculus core, we add just one construct, the kell construct, $a[P]$, which is used to localize the execution of a process P at location (we say "kell") a.

In the Kell calculus, computing actions can take five simple forms, illustrated below:

1. Creation of a (globally unique) new name as in the reduction below, where Δ is a global environment that records the names that have been created during a (top-level) computation, and where b is a fresh name that doe not appear in P or Δ:

$$\Delta \vdash \nu a.P \;\rightarrow\; P\{b/a\} \dashv \Delta \uplus \{b\}$$

2. Receipt of a local message, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x\rangle \triangleright P$:

$$a\langle Q\rangle \mid (a\langle x\rangle \triangleright P) \;\rightarrow\; P\{Q/x\}$$

3. Receipt of a message originated from the environment of a kell, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x\rangle^{\uparrow} \triangleright P$, located in kell b (by convention, we do not mention the name environment Δ in a reduction when it is left unchanged):

$$a\langle Q\rangle \mid b[a\langle x\rangle^{\uparrow} \triangleright P] \;\rightarrow\; b[P\{Q/x\}]$$

In pattern $a\langle x\rangle^{\uparrow}$, the up arrow $^{\uparrow}$ denotes a message that should come from the outside of the immediately enclosing kell.

4. Receipt of a message originated from a sub-kell, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, and coming from sub-kell b, is received by the trigger $a\langle x\rangle_{\downarrow} \triangleright P$, located in the parent kell of kell b:

$$(a\langle x\rangle_{\downarrow} \triangleright P) \mid b[a\langle Q\rangle \mid R] \;\rightarrow\; P\{Q/x\} \mid b[R]$$

5. Suspension of a kell, as in the reduction below, where the sub-kell named a is destroyed, and the process Q it contains is sent in a message on port b:

$$a[Q] \mid (a[x] \triangleright b\langle x\rangle) \;\rightarrow\; b\langle Q\rangle$$

Actions of the form 1 and 2 above are standard π-calculus actions. The handling of restriction is a bit unconventional but it will be motivated in section 3 below. Actions of the form 3 and 4 are just extensions of the message receipt action of the π-calculus to the case of triggers located inside a kell. They can be compared to the communication actions in the Boxed Ambients calculus.

Actions of the form 5 are characteristic of the Kell calculus. They allow the environment of a kell to exercize control over the execution of the process located inside a kell. Consider for instance the process P, defined as $P \triangleq \mathtt{stop}\langle(b)\rangle \triangleright (b[x] \triangleright \mathbf{0})$. We have the following reductions:

$$\mathtt{stop}\langle a\rangle \mid P \mid a[Q] \;\rightarrow\; (a[x] \triangleright \mathbf{0}) \mid a[Q] \;\rightarrow\; \mathbf{0}$$

In this example, the environment of kell a collects it, thus destroying it and the process Q that it holds. Other forms of control over process execution are possible. Consider the process P and R defined as:

$$P \triangleq \mathtt{suspend}\langle a\rangle \triangleright (a[x] \triangleright R \mid a\langle x\rangle) \quad R \triangleq \mathtt{resume}\langle a\rangle \triangleright (a\langle x\rangle \triangleright a[x])$$

We have the following reductions:

$$\begin{array}{rcl} \mathtt{resume}\langle a\rangle \mid \mathtt{suspend}\langle a\rangle \mid P \mid a[Q] & \rightarrow & \mathtt{resume}\langle a\rangle \mid (a[x] \triangleright R \mid a\langle x\rangle) \mid a[Q] \\ & \rightarrow & \mathtt{resume}\langle a\rangle \mid R \mid a\langle Q\rangle \\ & \rightarrow & (a\langle x\rangle \triangleright a[x]) \mid a\langle Q\rangle \\ & \rightarrow & a[Q] \end{array}$$

In this example, the environment of kell a first suspends its execution (there is no evaluation under a $a\langle.\rangle$ context), and then resumes it (processes can execute under a $a[.]$ context).

The higher-order nature of the calculus, together with the above control capability, allows the definition of different forms of programmable "membranes" around kells. Here are some simple examples. Assume that all triggers in process K are of the form $a\langle x\rangle \triangleright \ldots$, and that all messages emitted towards the environment of kell a are of the form $\mathtt{m}\langle b, \ldots\rangle$, where b is a target kell. We can define around kell a the following membranes:

- Transparent membrane: Let $M \triangleq a\langle x\rangle^{\uparrow} \triangleright a\langle x\rangle \quad \mid \quad \mathtt{m}\langle(b), x\rangle_{\downarrow} \triangleright b\langle x\rangle$. Then $c[M \mid a[K]]$ defines a membrane around kell a that does nothing (it just allows messages destined to, or emitted by, a to be transmitted without any control).
- Intercepting membrane: Let $M \triangleq a\langle x\rangle^{\uparrow} \triangleright P(x) \quad \mid \quad \mathtt{m}\langle b, y\rangle_{\downarrow} \triangleright Q(b, y)$. Then $c[M \mid a\langle K\rangle]$ defines a membrane around kell a that triggers behaviour $P(x)$ when a message $a\langle x\rangle$ seeks to enter kell a, and behaviour $Q(y)$ when a message $\mathtt{m}\langle y\rangle$ seeks to leave kell a. Notice how this allows the definition of wrappers with pre and post-handling of messages.
- Migration membrane: Let

  $$M \triangleq \mathtt{enter}\langle a, x\rangle^{\uparrow} \triangleright (a[y] \triangleright a[y \mid x]) \quad \mid \quad \mathtt{go}\langle(b)\rangle_{\downarrow} \triangleright (a[y] \triangleright \mathtt{enter}\langle b, a[y]\rangle)$$

  Then $c[M \mid a[K]]$ defines a membrane around kell a that allows new processes to enter kell a via the `enter` operation, and allows kell a to move to a different kell b via the `go` operation. Compare these operations with the migration primitives of Mobile Ambients, and the `go` primitive of the Distributed Join calculus.
- Localities with failures: Let

  $$\begin{array}{rcl} M & \triangleq & \mathtt{stop}\langle a\rangle^{\uparrow} \triangleright (a[y] \triangleright S) \quad \mid \quad \mathtt{ping}\langle a, (r)\rangle^{\uparrow} \triangleright r\langle \mathtt{up}\rangle \\ S & \triangleq & \mathtt{ping}\langle a, (r)\rangle^{\uparrow} \triangleright r\langle \mathtt{down}\rangle \end{array}$$

  Then, $c[M \mid a[K]]$ defines a membrane around kell a that allows to `stop` the execution of locality a (simulating a failure in a fail-stop model), and that implements a simple failure detector via the `ping` operation. Compare these operations with the π_{1l}-calculus model of failures [1].

Actions in the Kell calculus obey a locality principle that states that any computing action should involve only one locality at a time (and its environment, when considering crossing locality boundaries). In particular, notice that there are no reductions in the calculus that, similar to the Mobile Ambients `in` move, would involve two adjacent kells. In particular, we *do not* have reductions of the following forms:

$$a[\mathtt{in}\langle Q\rangle] \mid b[\mathtt{in}\langle x\rangle \triangleright x] \;\rightarrow\; a[\mathbf{0}] \mid b[Q] \qquad a[Q] \mid b[a[x] \triangleright x] \;\rightarrow\; b[a[Q]]$$

$$\begin{array}{rcl} P & ::= & \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid a\langle u\rangle \mid P \mid P \mid a[P] \\ u & ::= & a \mid P \end{array}$$

Figure 1: Syntax of the Kell Calculus

$$\begin{array}{rcl} K & ::= & \mathbf{0} \mid a[u] \mid K \mid K \\ M & ::= & \mathbf{0} \mid a\langle u\rangle \mid M \mid M \end{array}$$

Figure 2: Syntax of parallel compositions of kells and messages

3 The Kell calculus: syntax and semantics

3.1 Syntax

The syntax of the Kell calculus is given in Figure 1 It is parameterized by the pattern language used to define patterns ξ in triggers $\xi \triangleright P$. We assume an infinite set N of *names*, and an infinite set V of *process variables*. We assume that $\mathsf{N} \cap \mathsf{V} = \emptyset$. We let a, b, n, m and their decorated variants range over N; and p, q, x, y range over V. The set L of *identifiers* is defined as $\mathsf{L} = \mathsf{N} \cup \mathsf{V}$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q and their decorated variants range over processes. We call *kell* a process of the form $a[u]$. The name a in a kell $a[u]$ is called the name of the kell. We let K, L and their decorated variants range over kells and parallel composition of kells. In a kell of the form $a[\ldots \mid a_j[u_j] \mid \ldots \mid Q_k \mid \ldots]$ we call *subkells* the processes $a_j[u_j]$. We call *message* a process of the form $a\langle u\rangle$. We let M, N and their decorated variants range over messages and parallel composition of messages. The syntax of parallel compositions of kells and messages is given in Figure 2.

In a term $\nu a.P$, the scope extends as far to the right as possible. We use $\widetilde{u}$ to denote finite vectors $(u_1, \ldots, u_q)$. We use standard abbreviations from the the π-calculus: $\nu a_1 \ldots a_q.P$ for $\nu a_1. \ldots \nu a_q.P$, or $\nu\widetilde{a}.P$ if $\widetilde{a} = (a_1, \ldots, a_q)$. By convention, if the name vector $\widetilde{a}$ is null, then $\nu\widetilde{a}.P \triangleq P$. We abbreviate $a\langle P_1, \ldots, P_n\rangle$ a message of the form $a\langle 1\langle P_1\rangle \mid \ldots \mid n\langle P_n\rangle\rangle$, where $1, \ldots, n, \ldots$ belong to N. We abbreviate a a message of the form $a\langle \mathbf{0}\rangle$. We also note $\prod_{j \in J} P_j$, $J = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $J = \emptyset$, then $\prod_{j \in J} P_j \triangleq \mathbf{0}$.

A Kell calculus context is a term $\mathbf{C}$ built according to the grammar given in Figure 3. Filling the hole $\cdot$ in $\mathbf{C}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{C}\{Q\}$. We let $\mathbf{C}$ and its decorated variants range over Kell calculus contexts. We make use of a specific form of contexts, called evaluation contexts (noted $\mathbf{E}$), which are used to specify the operational semantics of the calculus.

A pattern ξ acts as a binder in the calculus. A pattern can bind *name markers*, of the form (a), where $a \in \mathsf{N}$, and *process markers*, of the form (x), where $x \in \mathsf{V}$. All markers appearing in a pattern ξ are bound by the pattern. Name markers can only match names. Process markers can only match processes. In a slight abuse of notation, we frequently dispense with the parenthesis $(.)$ around markers (especially process markers) when it is clear from the context which identifiers act as markers[1].

A process P matches a pattern ξ if there is a substitution θ (i.e. a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{V} \to \mathsf{K}$, from names to names and process variables to Kell calculus terms that is the identity except on a finite set of

[1]Pattern languages used in this paper do not make use of free process variables in trigger patterns. As a consequence, this convention can be employed systematically for process markers since there is no risk of confusing a process marker with a free process variable.

$$\begin{array}{rcl} \mathbf{C} & ::= & \cdot \mid \xi \triangleright \mathbf{C} \mid \nu a.\mathbf{C} \mid (P \mid \mathbf{C}) \mid a[\mathbf{C}] \mid a\langle \mathbf{C}\rangle \\ \mathbf{E} & ::= & \cdot \mid a[\mathbf{E}] \mid P \mid \mathbf{E} \end{array}$$

Figure 3: Syntax of Contexts

$$\begin{array}{ll}
\mathtt{fn}(\mathbf{0}) = \emptyset & \mathtt{fv}(\mathbf{0}) = \emptyset \\
\mathtt{fn}(a) = \{a\} & \mathtt{fv}(a) = \emptyset \\
\mathtt{fn}(x) = \emptyset & \mathtt{fv}(x) = \{x\} \\
\mathtt{fn}(\nu a.P) = \mathtt{fn}(P) \setminus \{a\} & \mathtt{fv}(\nu a.P) = \mathtt{fv}(P) \\
\mathtt{fn}(a[P]) = \mathtt{fn}(P) \cup \{a\} & \mathtt{fv}(a[P]) = \mathtt{fv}(P) \\
\mathtt{fn}(a\langle u\rangle) = \mathtt{fn}(u) \cup \{a\} & \mathtt{fv}(a\langle u\rangle) = \mathtt{fv}(u) \\
\mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q) & \mathtt{fv}(P \mid Q) = \mathtt{fv}(P) \cup \mathtt{fv}(Q) \\
\mathtt{fn}(\xi \triangleright P) = \mathtt{fn}(\xi) \cup (\mathtt{fn}(P) \setminus \mathtt{bn}(\xi)) & \mathtt{fv}(\xi \triangleright P) = \mathtt{fv}(\xi) \cup (\mathtt{fv}(P) \setminus \mathtt{bv}(\xi))
\end{array}$$

Figure 4: Free names and free variables

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \text{ [S.PAR.A]} \qquad P \mid Q \equiv Q \mid P \text{ [S.PAR.C]} \qquad P \mid \mathbf{0} \equiv P \text{ [S.PAR.N]}$$

$$\frac{\xi \equiv \zeta}{\xi \triangleright P \equiv \zeta \triangleright P} \text{ [S.TRIG]} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ [S.}\alpha\text{]} \qquad \frac{P \equiv Q}{\mathbf{C}\{P\} \equiv \mathbf{C}\{Q\}} \text{ [S.CONTEXT]}$$

Figure 5: Structural congruence

identifiers), that maps identifiers u appearing as markers in ξ (i.e. $u \in \mathtt{bn}(\xi) \cup \mathtt{bv}(\xi)$) to Kell calculus terms, such that $\xi\theta = P$ (i.e. such that the image of pattern ξ under substitution θ is the process P). We also make use of context-dependent patterns. Such patterns typically include a side condition or a guard that depends on the current evaluation context. Matching for these patterns is defined as for standard patterns, but using the notion of a context-dependent substitution. A context-dependent substitution θ is a function that maps pairs $\langle \mathbf{E}, u\rangle$ of Kell calculus execution contexts and identifiers onto names or Kell calculus terms. We note $P\theta_{\mathbf{E}}$ the image of the term P under substitution θ, given a context $\mathbf{E}$.

We make the following assumptions on pattern languages:

- One can decide whether a pattern matches a given term and the result of applying a substitution on markers to a pattern ξ is a Kell calculus process. Generally, given a context $\mathbf{E}$, we say that a pattern ξ matches a kell calculus term P in context $\mathbf{E}$, if there exists a context-dependent substitution θ such that $\xi\theta_{\mathbf{E}} = P$.
- A pattern language is compatible with the structural congruence defined below, i.e. if $P \equiv Q$ then there is no Kell calculus context that can distinguish between P and Q. Also, there are no ξ, θ such that $\xi\theta \equiv \mathbf{0}$.
- Pattern languages are equipped with a structural congruence relation, noted $\equiv$. Pattern languages are also equipped with a function $\mathtt{sk}$, which maps a pattern ξ to a parallel composition of actions (see section 3.3 below). Intuitively, $\xi.\mathtt{sk}$ corresponds to the set of ports on which pattern ξ expects messages or kells (we use a postfix notation for $\mathtt{sk}$).
- Pattern languages are equipped with four functions $\mathtt{fn}$, $\mathtt{bn}$, $\mathtt{fv}$, and $\mathtt{bv}$, that map a pattern ξ to its set of free names, bound names, free process variables and bound process variables, respectively.

The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Notions of free names ($\mathtt{fn}$) and free variables ($\mathtt{fv}$) are classical and are defined in Figure 4. We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

3.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [2], via a structural congruence and a reduction relation. The structural congruence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 5. The rules S.PAR.A, S.PAR.C, S.PAR.N state that the parallel operator $\mid$ is associative , commutative, and has $\mathbf{0}$ as a neutral element. Note that, in rule S.TRIG, we rely on the structural congruence relation on patterns, also noted $\equiv$.

Notice that we do not have structural congruence rules that deal with scope extrusion as in the π-calculus. This is because, in presence of the possibility of suspending executing processes, the standard

$$\frac{\mathbf{E} = \cdot \mid N \mid K \mid Q \qquad Q = \prod_{j \in J} a_j[N_j \mid Q_j] \qquad Q' = \prod_{j \in J} a_j[Q_j] \qquad \xi\theta_{\mathbf{E}} = N \mid K \mid \prod_{j \in J} N_j}{\xi \triangleright P \mid N \mid K \mid Q \rightarrow P\theta_{\mathbf{E}} \mid Q'} \text{ [R.RED.S]}$$

$$\frac{\begin{array}{c}\mathbf{E} = M \mid a[\cdot N \mid K \mid Q \mid R] \\ Q = \prod_{j \in J} a_j[N_j \mid Q_j] \qquad Q' = \prod_{j \in J} a_j[Q_j] \qquad \xi\theta_{\mathbf{E}} = M \mid N \mid K \mid \prod_{j \in J} N_j\end{array}}{a[\xi \triangleright P \mid N \mid K \mid Q \mid R] \mid M \rightarrow a[P\theta_{\mathbf{E}} \mid Q' \mid R]} \text{ [R.RED.H]}$$

$$\frac{\mathtt{fn}(\nu a.P) \subseteq \Delta \qquad b \notin \Delta}{\Delta \vdash \nu a.P \ \rightarrow \ P\{b/a\} \dashv \Delta \cup \{b\}} \text{ [R.NU]} \qquad \frac{\Delta \vdash P \ \rightarrow \ Q \dashv \Delta' \qquad \mathtt{fn}(\mathbf{E}\{P\}) \subseteq \Delta}{\Delta \vdash \mathbf{E}\{P\} \ \rightarrow \ \mathbf{E}\{Q\} \dashv \Delta'} \text{ [R.CTX]}$$

$$\frac{P \equiv P' \qquad \Delta \vdash P' \ \rightarrow \ Q' \dashv \Delta' \qquad Q' \equiv Q}{\Delta \vdash P \ \rightarrow \ Q \dashv \Delta'} \text{ [R.EQV]}$$

Figure 6: Reduction Relation

$$\frac{\mathbf{E} = N \mid a[\cdot \mid Q] \qquad \xi\theta_{\mathbf{E}} \equiv N}{N \mid a[\xi \triangleright P \mid Q] \rightarrow a[P\theta_{\mathbf{E}} \mid Q]} \text{ [IN]} \qquad \frac{\mathbf{E} = \cdot \mid a[N \mid Q] \qquad \xi\theta_{\mathbf{E}} \equiv N}{\xi \triangleright P \mid a[N \mid Q] \rightarrow P\theta_{\mathbf{E}} \mid a[Q]} \text{ [OUT]}$$

$$\frac{\mathbf{E} = \cdot \qquad \xi\theta_{\mathbf{E}} = N}{\xi \triangleright P \mid N \rightarrow P\theta_{\mathbf{E}}} \text{ [BETA]}$$

Figure 7: Derived rules

scope extrusion rule $a[\nu b.P] \equiv \nu b.a[P] \ \ (b \neq a)$ would give rise to strange behaviour, as illustrated by the following reductions:

$$(a[x] \triangleright x \mid x) \mid a[\nu b.P] \ \rightarrow \ (\nu b.P) \mid (\nu b.P) \qquad (a[x] \triangleright x \mid x) \mid \nu b.a[P] \ \rightarrow \ \nu b.P \mid P$$

which would violate the idea that structurally equivalent processes should behave similarly in the same evaluation context.

Let $\wp_{\mathsf{f}}(\mathsf{N})$ be the set of finite subsets of N. We let Δ and its decorated variants range over $\wp_{\mathsf{f}}(\mathsf{N})$. The reduction relation $\rightarrow$ is the smallest relation $\rightarrow \subseteq (\wp_{\mathsf{f}}(\mathsf{N}) \times \mathsf{K})^2$ that verifies

$$\rightarrow \subseteq \{(\langle\Delta, P\rangle, \langle\Delta', P'\rangle) \in (\wp_{\mathsf{f}}(\mathsf{N}) \times \mathsf{K})^2 \mid \mathtt{fn}(P) \subseteq \Delta\}$$

and that satisfies the rules given in Figure 6. We write $\Delta \vdash P \ \rightarrow \ P' \dashv \Delta'$ for $(\langle\Delta, P\rangle, \langle\Delta', P'\rangle) \in \rightarrow$. Intuitively, such a reduction means that, in a context where the current set of created names is Δ and is such that $\mathtt{fn}(P) \subseteq \Delta$, process P can evolve into process P', possibly creating new names in the process, which are recorded in Δ'. When the set Δ is unchanged during a reduction, we abbreviate $\Delta \vdash P \ \rightarrow \ P' \dashv \Delta$ to $P \rightarrow P'$.

Notice that we allow pattern matching on messages and subkells within the same kell, and only on messages external to the kell, or within a subkell. This means that only messages are allowed to cross the boundary of a kell.

The basic reduction rules of the calculus R.RED.S and R.RED.H in Figure 6 look rather involved but their nature can be easily revealed by considering the derived rules in Figure 7. Rules IN and OUT correspond, respectively, to the case of messages entering a kell, and to the case of messages leaving a kell. These rules are themselves just extensions of rule BETA, which is the standard reduction rule of the asynchronous π-calculus, extended to take filtering into account. Rules IN and OUT indicate that messages can cross kell boundaries, and that crossing a kell boundary requires the presence of a trigger on the other side of the boundary. Note that rules R.RED.S and R.RED.H take into account the possibility for patterns to match parallel composition of messages occurring at different levels (outside the receiving kell and within subkells of the receiving kell).

$$\begin{array}{rcl} C & ::= & \eta.P \quad | \quad C \mid C \\ \eta & ::= & \mathbf{0} \quad | \quad \overline{a}[u] \quad | \quad \overline{a}\langle u\rangle \quad | \quad a[C_p] \\ C_p & ::= & \overline{a}\langle u\rangle.P \quad | \quad C_p \mid C_p \end{array}$$

Figure 8: Syntax of Concretions

$$\begin{array}{rcl} F & ::= & F_p \quad | \quad a[F_p] \quad | \quad F@C \\ F_p & ::= & (\xi)P \quad | \quad F_p@C \end{array}$$

Figure 9: Syntax of Abstractions

The rules R.RED.S and R.RED.H identify the only forms of contexts that can be used for pattern matching.

3.3 Labelled transition system semantics

We define in this section a labelled transition system for Kell calculus processes. The labelled transition system is defined by means of a commitment relation in the style of the commitment rules for the π-calculus defined in [13]. We define first a notion of concretion. A concretion C is given by the grammar in Figure 8, where u and P are as in Figure 1.

We define the functions `red`, `pld`, and `ctx` on concretions (we use a postfix notation for these functions):

$$\begin{array}{lll} (\mathbf{0}.P).\mathtt{red} \triangleq P & (\mathbf{0}.P).\mathtt{pld} \triangleq \mathbf{0} & (\mathbf{0}.P).\mathtt{ctx} \triangleq P \\ (\overline{a}\langle u\rangle.P).\mathtt{red} \triangleq P & (\overline{a}\langle u\rangle.P).\mathtt{pld} \triangleq a\langle u\rangle & (\overline{a}\langle u\rangle.P).\mathtt{ctx} \triangleq a\langle u\rangle \mid P \\ (\overline{a}[u].P).\mathtt{red} \triangleq P & (\overline{a}[u].P).\mathtt{pld} \triangleq a[u] & (\overline{a}[u].P).\mathtt{ctx} \triangleq a[u] \mid P \\ (a[C_p].P).\mathtt{red} \triangleq a[C_p.\mathtt{red}] \mid P & (a[C_p].P).\mathtt{pld} \triangleq C_p.\mathtt{pld} & (a[C_p].P).\mathtt{ctx} \triangleq a[C_p.\mathtt{ctx}] \mid P \\ (C \mid C').\mathtt{red} \triangleq C.\mathtt{red} \mid C'.\mathtt{red} & (C \mid C').\mathtt{pld} \triangleq C.\mathtt{pld} \mid C'.\mathtt{pld} & (C \mid C').\mathtt{ctx} \triangleq C.\mathtt{ctx} \mid C'.\mathtt{ctx} \end{array}$$

We then define a notion of abstraction. An abstraction F is given by the grammar in Figure 9, where ξ, P are as in Figure 1, and C is a concretion.

An agent A is a Kell calculus process P, a concretion C or an abstraction F. We note A the set of agents. We let A, B and their decorated variants range over agents; F, G, and their decorated variants range over abstractions; C,D and their decorated variants range over concretions.

We define the effects of operators "|" and "@" on concretions and abstractions thus:

$$\begin{array}{ll} \eta.P \mid Q \triangleq \eta.(P \mid Q) & (C_1 \mid C_2) \mid Q \triangleq C_1 \mid (C_2 \mid Q) \\ F \mid P \triangleq F@(\mathbf{0}.P) & (F@C)@C' \triangleq F@(C \mid C') \\ (\xi)P@C \triangleq P\theta_{\mathbf{E}} \mid C.\mathtt{red} & \text{if } \exists\theta,\ \xi\theta_{\mathbf{E}} \equiv C.\mathtt{pld} \\ & \quad \mathbf{E} = \cdot \mid C.\mathtt{ctx} \\ a[(\xi)P@C]@C'_p \triangleq a[P\theta_{\mathbf{E}} \mid C.\mathtt{red}] \mid C'_p.\mathtt{red} & \text{if } \exists\theta,\ \xi\theta_{\mathbf{E}} \equiv C.\mathtt{pld} \mid C'_p.\mathtt{pld} \\ & \quad \mathbf{E} = a[\cdot \mid C.\mathtt{ctx}] \mid C'_p.\mathtt{pld} \end{array}$$

Note that, when one can apply the last two clauses above in the definition of operator @, we have $F@C \in \mathsf{K}$. Note also that, as a function $@ : \mathsf{A} \to \mathsf{K}$, operator @ is only partially defined.

The notions of free names, free variables, bound names and bound variables extend immediately to agents. We also extend the structural congruence relation over agents by defining it as the smallest equivalence relation (also noted $\equiv$), that contains the structural congruence relation on Kell calculus processes, and that satisfies the rules in Figure 10. In rule S.CONC, we set by definition $\eta \equiv \eta'$ if and only if $(\eta = \eta' = \mathbf{0}) \vee (\eta = \overline{a}\langle u\rangle \wedge \eta' = \overline{a}\langle v\rangle \wedge a\langle u\rangle \equiv a\langle v\rangle) \vee (\eta = \overline{a}[u] \wedge \eta' = \overline{a}[v] \wedge a[u] \equiv a[v])$.

Actions are given by the grammar in Figure 11, where $a \in \mathsf{N}$. We denote Λ the set of actions, and we let α, β and their decorated variant range over Λ. By definition, the parallel operator $\mid$ on actions is

$$A \mid \mathbf{0} \equiv A)\ [\text{S.A.NIL}] \qquad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3)\ [\text{S.A.ASSC}] \qquad C_1 \mid C_2 \equiv C_2 \mid C_1\ [\text{S.A.COMC}]$$

$$\eta.P \mid \mathbf{0}.Q \equiv \eta.(P \mid Q)\ [\text{S.A.NILC}] \qquad \frac{\eta \equiv \eta' \qquad P \equiv Q}{\eta.P \equiv \eta'.Q}\ [\text{S.A.CONC}] \qquad \frac{C_1 \equiv C_2}{C_1 \mid C \equiv C_2 \mid C}\ [\text{S.A.PARC}]$$

$$\frac{C_p \equiv C'_p}{a[C_p] \equiv a[C'_p]}\ [\text{S.A.KELLC}] \qquad \frac{\xi \equiv \zeta \qquad P \equiv Q}{(\xi)P \equiv (\zeta)Q}\ [\text{S.A.TRIG}] \qquad \frac{F_1 \equiv F_2 \qquad C_1 \equiv C_2}{F_1 @ C_1 \equiv F_2 @ C_2}\ [\text{S.A.APP}]$$

$$\frac{F_p \equiv F'_p}{a[F_p] \equiv a[F'_p]}\ [\text{S.A.KELLF}] \qquad \frac{A =_\alpha B}{A \equiv B}\ [\text{S.A.}\alpha]$$

Figure 10: Structural congruence on agents

$$\alpha \quad ::= \quad \epsilon \quad | \quad \tau \quad | \quad a \quad | \quad \overline{a} \quad | \quad \alpha \mid \alpha$$

Figure 11: Syntax of Actions

associative, commutative, and has ϵ has a neutral element. Also, for any name a we set $\overline{a} \mid a \triangleq \epsilon$, i.e. $\overline{a}$ and a are inverse of each other.

The commitment relation is the smallest relation R on $(\wp_f(N) \times \mathsf{K}) \times \Lambda \times (\wp_f(N) \times \mathsf{A})$, that verifies $(\langle \Delta, P \rangle, \alpha, \langle \Delta', A \rangle) \in R \Rightarrow \mathtt{fn}(P) \subseteq \Delta$, and that satisfies the rules in Figure 12. We write $\Delta \vdash P \xrightarrow{\alpha} A \dashv \Delta'$ for $(\langle \Delta, P \rangle, \alpha, \langle \Delta', A \rangle) \in R$, and $P \xrightarrow{\alpha} P'$ for $\Delta \vdash P \xrightarrow{\alpha} P' \dashv \Delta$ and $\mathtt{fn}(P) \subseteq \Delta$.

The correspondence between the reduction semantics and the labelled transition semantics is given by the following theorems.

Theorem 1 *If $P \equiv Q$ and $\Delta \vdash P \xrightarrow{\alpha} A \dashv \Delta'$, then there exists $B \equiv A$ such that $\Delta \vdash Q \xrightarrow{\alpha} B \dashv \Delta'$.*

Theorem 2 *For all P, P', $\Delta \vdash P \xrightarrow{\tau} P' \dashv \Delta'$ if and only if there exists P'' such that $P \to P''$ and $P'' \equiv P'$.*

4 Encodings

We illustrate in this section the expressive power of the Kell calculus by defining encodings of several process calculi with localities. In this section, we rely on a specific pattern language whose syntax is given in Figure 13. Intuitively, a pattern of the form $\rho :: \pi$ matches processes which match the pattern ρ, provided that, in the current evaluation context, the predicate π is satisfied (note that the scope of bound names or variables appearing in pattern ρ extends to predicate π). The intuition behind the predicates given in Figure 13 is as follows:

- Predicate $a \neq b$ is true when name a is different from name b.
- In a given context $\mathbf{C} \equiv c[\cdot \mid Q]$, predicate $a \in \mathsf{C}$ is true if $Q \equiv a[R] \mid S$, i.e. if there is a subkell of c named a.
- In a given context $\mathbf{C} \equiv c[\cdot \mid Q]$, predicate $a \in \mathsf{C}^*$ is true if $Q \equiv \mathbf{E}\{a[R]\}$, i.e. if there is a subkell of c named a, or if, recursively, one of the subkells of c verifies the predicate.
- In a given context $\mathbf{C} \equiv c[\cdot \mid Q]$, predicate $a \in b \bullet \mathsf{C}$ is true if there is a subkell b of c that has a subkell named a.
- In a given context $\mathbf{C} \equiv c[\cdot \mid Q]$, predicate $a \in b \bullet \mathsf{C}^*$ is true if there is a subkell b of c that has a subkell named a, or that is such that one of its subkells verifies the predicate $a \in \mathsf{C}^*$.

$$a\langle Q\rangle \xrightarrow{\overline{a}} \overline{a}\langle Q\rangle.\mathbf{0}\ \text{[T.RecM]} \qquad a[Q] \xrightarrow{\overline{a}} \overline{a}[Q].\mathbf{0}\ \text{[T.RecK]} \qquad \xi \triangleright P \xrightarrow{\xi.\mathtt{sk}} (\xi)P\ \text{[T.RecT]}$$

$$\frac{b \notin \Delta \qquad \mathtt{fn}(\nu a.P) \subseteq \Delta}{\Delta \vdash \nu a.P \xrightarrow{\tau} P\{b/a\} \dashv \Delta \cup \{b\}}\ \text{[T.New]} \qquad \frac{\Delta \vdash P \xrightarrow{\tau} Q \dashv \Delta' \qquad \mathtt{fn}(a[P]) \subseteq \Delta}{\Delta \vdash a[P] \xrightarrow{\tau} a[Q] \dashv \Delta'}\ \text{[T.KellP]}$$

$$\frac{\Delta \vdash P \xrightarrow{\alpha} F_p \dashv \Delta' \qquad \mathtt{fn}(a[P]) \subseteq \Delta}{\Delta \vdash a[P] \xrightarrow{\alpha} a[F_p] \dashv \Delta'}\ \text{[T.KellF]} \qquad \frac{\Delta \vdash P \xrightarrow{\alpha} C_p \dashv \Delta' \qquad \mathtt{fn}(a[P]) \subseteq \Delta}{\Delta \vdash a[P] \xrightarrow{\alpha} a[C_p].\mathbf{0} \dashv \Delta'}\ \text{[T.KellC]}$$

$$\frac{\Delta \vdash P \xrightarrow{\alpha} A \dashv \Delta' \qquad \mathtt{fn}(P \mid Q) \subseteq \Delta}{\Delta \vdash P \mid Q \xrightarrow{\alpha} A \mid Q \dashv \Delta'}\ \text{[T.ParL]} \qquad \frac{\Delta \vdash P \xrightarrow{\alpha} A \dashv \Delta' \qquad \mathtt{fn}(P \mid Q) \subseteq \Delta}{\Delta \vdash Q \mid P \xrightarrow{\alpha} A \mid Q \dashv \Delta'}\ \text{[T.ParR]}$$

$$\frac{\Delta \vdash P \xrightarrow{\alpha} C \dashv \Delta \qquad \Delta \vdash Q \xrightarrow{\beta} C' \dashv \Delta \qquad \mathtt{fn}(P \mid Q) \subseteq \Delta}{\Delta \vdash P \mid Q \xrightarrow{\alpha \mid \beta} C \mid C' \dashv \Delta}\ \text{[T.ParB]}$$

$$\frac{\Delta \vdash P \xrightarrow{\alpha} F \dashv \Delta \qquad \Delta \vdash Q \xrightarrow{\beta} C \dashv \Delta \qquad \mathtt{fn}(P \mid Q) \subseteq \Delta}{\Delta \vdash P \mid Q \xrightarrow{\alpha \mid \beta} F@C \dashv \Delta}\ \text{[T.ParBL]}$$

$$\frac{\Delta \vdash P \xrightarrow{\alpha} F \dashv \Delta \qquad \Delta \vdash Q \xrightarrow{\beta} C \dashv \Delta \qquad \mathtt{fn}(P \mid Q) \subseteq \Delta}{\Delta \vdash Q \mid P \xrightarrow{\alpha \mid \beta} F@C \dashv \Delta}\ \text{[T.ParBR]}$$

$$\frac{\Delta \vdash P \xrightarrow{\epsilon} Q \dashv \Delta \qquad \mathtt{fn}(P) \subseteq \Delta}{\Delta \vdash P \xrightarrow{\tau} Q \dashv \Delta}\ \text{[T.Red]} \qquad \frac{\Delta \vdash P \xrightarrow{\alpha} A \dashv \Delta' \qquad Q =_\alpha P \qquad \mathtt{fn}(P) \subseteq \Delta}{\Delta \vdash Q \xrightarrow{\alpha} A \dashv \Delta'}\ \text{[T.}\alpha\text{]}$$

Figure 12: Commitment Relation

The semantics of patterns is defined as follows. A context-dependent substitution θ is a standard substitution mapping markers to names or Kell calculus processes that is extended to work in contexts $\mathbf{C}$ as defined below:

$$(\rho' :: \pi)\theta_{\mathbf{C}} = M \iff \rho\theta = M \ \wedge\ \psi(M, \rho', \mathbf{C}) \ \wedge\ \phi(\pi\theta, \mathbf{C})$$

where ψ and ϕ are predicates defined by:

$$\begin{array}{rcl} \psi(M, \rho^{\uparrow}, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid M \mid a[\cdot \mid Q] \\ \psi(M, \rho_{\downarrow}, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid a[\cdot \mid b[M \mid R] \mid S] \ \vee\ \mathbf{C} \equiv \cdot \mid b[M \mid R] \mid S \\ \psi(M, \rho, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid a[\cdot \mid M \mid R] \ \vee\ \mathbf{C} \equiv \cdot \mid M \mid R \end{array}$$

$$
\begin{array}{rcl}
\xi & ::= & \rho' \quad | \quad \rho' :: \pi \quad | \quad \xi \mid \xi \\
\rho' & ::= & \rho \quad | \quad \rho^{\uparrow} \quad | \quad \rho_{\downarrow} \\
\rho & ::= & a\langle w\rangle \quad | \quad a[(x)] \\
w & ::= & a \quad | \quad (a) \quad | \quad (x) \quad | \quad \rho \quad | \quad \rho \mid \rho \\
\pi & ::= & a \neq b \quad | \quad b \in \mathtt{C} \quad | \quad b \in \mathtt{C}^* \quad | \quad b \in a \bullet \mathtt{C} \quad | \quad b \in a \bullet \mathtt{C}^* \quad | \quad \neg\pi
\end{array}
$$

Figure 13: Syntax of patterns

$$
\begin{array}{rcl}
\phi(b \in \mathtt{C}, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid c[\cdot \mid b[R] \mid S] \\
\phi(b \in \mathtt{C}^*, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid c[\cdot \mid \mathbf{E}\{b[R]\}] \\
\phi(b \in a \bullet \mathtt{C}, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid c[\cdot \mid a[b[R] \mid S] \mid T] \\
\phi(b \in a \bullet \mathtt{C}^*, \mathbf{C}) & \triangleq & \mathbf{C} \equiv N \mid c[\cdot \mid a[\mathbf{E}\{b[R]\}] \mid T] \\
\phi(a \neq b, \mathbf{C}) & \triangleq & (a \neq b) \\
\phi(\neg\pi, \mathbf{C}) & \triangleq & \neg\phi(\pi, \mathbf{C})
\end{array}
$$

The structural congruence relation on patterns is defined by the following rules: the parallel operator, $|$, is associative and commutative; if $\xi \equiv \zeta$ then $\xi \mid \eta \equiv \zeta \mid \eta$; if $\rho'_1 \equiv \rho'_2$, then $\rho'_1 :: \pi \equiv \rho'_2 :: \pi$; if $\rho \equiv \overline{\rho}$, then $\rho \mid \rho_0 \equiv \overline{\rho} \mid \rho_0$, $\rho^{\uparrow} \equiv \overline{\rho}^{\uparrow}$, $\rho_{\downarrow} \equiv \overline{\rho}_{\downarrow}$ and $a\langle\rho\rangle \equiv a\langle\overline{\rho}\rangle$; if two patterns differ only by α-conversion, then they are equivalent.

We also use several abbreviations in the encodings. We define first receptive triggers, i.e. triggers that are preserved during a reduction. Let $t \in \mathsf{N}$, ξ and P be such that $t \notin \mathtt{fn}(\xi) \cup \mathtt{fn}(P)$. In a construction reminiscent of the fixed point operator defined in CHOCS [20], we define $\xi \diamond P$ by:

$$
\begin{array}{rcl}
\xi \diamond P & \triangleq & \nu t.Y(P, \xi, t) \mid t\langle Y(P, \xi, t)\rangle \\
Y(P, \xi, t) & \triangleq & \xi \mid t\langle y\rangle \triangleright P \mid y \mid t\langle y\rangle
\end{array}
$$

Let $\mathbf{C}$, Q, Q', R, M, N, K and θ be as in the premises of rule R.RED.H. Then, it is easy to show that

$$
a[\xi \diamond P \mid N \mid K \mid Q \mid R] \mid M \to a[\xi \diamond P \mid P\theta_{\mathbf{C}} \mid Q' \mid R]
$$

We also use abstraction $(x)P$ and application PQ. The resulting extended calculus is defined by induction thus (notice the implicit typing to ensure well-formed processes):

$$
[\![P]\!] \triangleq P \quad \text{if } P \in \mathsf{K} \qquad [\![(x)P]\!]_f \triangleq f\langle x\rangle \triangleright [\![P]\!] \qquad [\![PQ]\!] \triangleq \nu f.[\![P]\!]_f \mid f\langle [\![Q]\!]\rangle
$$

4.1 Encoding the synchronous π-calculus

The asynchronous π-calculus is a direct subcalculus of the Kell calculus. Because of its higher-order character, the Kell calculus can also encode directly the synchronous π-calculus. An encoding of the synchronous (polyadic) π-calculus with name matching and input guarded sums (cf [16] for a definition) is given below, where we assume that the names $1, \ldots, n, \ldots$, and k do not appear free in P, P_j, Q, and where $\widetilde{b} = b_1 \ldots b_n$, $\widetilde{b^j} = b_1^j \ldots b_{n_j}^j$, $j \in J$.

$$\begin{array}{rcl}
[\![\mathbf{0}]\!] & = & \mathbf{0} \\
[\![a\widetilde{b}.P]\!] & = & a\langle [\![P]\!], b_1, \ldots, b_n \rangle \\
[\![\tau.P]\!] & = & \nu \mathsf{k}.\mathsf{k} \mid (\mathsf{k} \triangleright [\![P]\!]) \\
[\![[a = b]P, Q]\!] & = & \nu l.(l\langle a \rangle \triangleright [\![P]\!]) \mid (l\langle (b) \rangle :: b \neq a \triangleright Q) \mid l\langle b \rangle \\
[\![\nu a.P]\!] & = & \nu a.[\![P]\!] \\
[\![P \mid Q]\!] & = & [\![P]\!] \mid [\![Q]\!] \\
[\![!P]\!] & = & \nu \mathsf{k}.\mathsf{k} \mid (\mathsf{k} \diamond [\![P]\!] \mid \mathsf{k}) \\
[\![\sum_{j \in J} a_j(\widetilde{b^j}).P_j]\!] & = & \nu \mathsf{k}.\mathsf{k} \mid \prod_{j \in J} \mathsf{k} \mid a_j\langle x_j, (b_1^j), \ldots, (b_{n_j}^j) \rangle \triangleright [\![P_j]\!] \mid x_j
\end{array}$$

If we adopt the slightly unconventional semantics for the π-calculus that replaces the usual structural congruence rules for restriction, matching and replication by the following reduction rules:

$$\Delta \vdash \nu a.P \rightarrow_\pi P\{b/a\} \dashv \Delta \uplus b \qquad !P \rightarrow_\pi !P \mid P \qquad [a = a]P \rightarrow_\pi P$$

then we obtain

Proposition 4.1 *If $P \rightarrow_\pi Q$, then $[\![P]\!] \rightarrow\equiv [\![Q]\!] \mid R$, where R is a parallel composition of inert processes of the form $\nu a.a \mid \xi \triangleright P$ or $\nu a.a \mid \xi \diamond P$. Conversely, if $[\![P]\!] \rightarrow^* \equiv [\![Q]\!] \mid R$, then $P \rightarrow_\pi^* Q$.*

4.2 Encoding Mobile Ambients

For simplicity, we present in this section an encoding of Mobile Ambients without local anonymous communication. The encoding we define below could be easily amended to account for it. The encoding is deadlock-free, but it relies on a simple locking scheme that reduces the parallelism inherent in ambient reductions. The encoding is also divergence-free, but it relies on the use of patterns with predicates of the form $b \in a \bullet \mathsf{C}$. An encoding that does not suffer from these limitations is certainly possible (e.g. one could mimick the protocol employed in the Join calculus implementation of ambients described in [8]) but it would be more complex.

The encoding of Mobile Ambients in the Kell calculus is given below, where we assume that the names 1, ... , n, ... , `t`, `to`, `up`, `in`, `out`, `open`, `amb`, `make`, `query`, `collect`, and `k` do not appear free in P, Q.

$$\begin{array}{ll}
[\![\mathbf{0}]\!] = \mathbf{0} & [\![\mathtt{in}\, a.P]\!] = \mathtt{in}\langle a, [\![P]\!] \rangle \\
[\![\nu n.P]\!] = \nu n.[\![P]\!] & [\![\mathtt{out}\, a.P]\!] = \mathtt{out}\langle a, [\![P]\!] \rangle \\
[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] & [\![\mathtt{open}\, a.P]\!] = \mathtt{open}\langle a, [\![P]\!] \rangle \\
[\![!P]\!] = \nu \mathsf{k}.\,\mathsf{k} \mid (\mathsf{k} \diamond [\![P]\!] \mid \mathsf{k}) & \\
[\![a[P]]\!] = a[\mathsf{A}(a) \mid \mathtt{amb}[[\![P]\!]]] \mid \mathtt{AmbEnv} &
\end{array}$$

$$
\begin{array}{rcl}
\mathtt{A}(a) & = & \nu\mathtt{t}.\,\mathtt{t} \mid \mathtt{S}(a,\mathtt{t}) \mid \mathtt{T}(a,\mathtt{t}) \mid \mathtt{F}(a,\mathtt{t}) \mid \mathtt{NQ}(a,t) \\
\mathtt{S}(a,\mathtt{t}) & = & (\mathtt{t} \mid \mathtt{in}\langle m,p\rangle_{\downarrow} \triangleright \\
 & & \quad \mathtt{amb}[z] \triangleright \nu k.\,\mathtt{collect}\langle a,k,\mathtt{to}\langle a,\mathtt{in},m,p,z\rangle\rangle \mid \mathtt{YQ}(a,k)) \\
 & & \mid (\mathtt{t} \mid \mathtt{out}\langle m,p\rangle_{\downarrow} \triangleright \\
 & & \quad \mathtt{amb}[z] \triangleright \nu k.\,\mathtt{collect}\langle a,k,\mathtt{up}\circ\langle a,\mathtt{out},m,p,z\rangle\rangle \mid \mathtt{YQ}(a,k)) \\
\mathtt{T}(a,\mathtt{t}) & = & (\mathtt{t} \mid \mathtt{amb}[z] \mid \mathtt{open}\langle a,p\rangle^{\uparrow} \triangleright \\
 & & \quad \nu k.\,\mathtt{collect}\langle a,k,(z \mid p)\rangle \mid \mathtt{YQ}(a,k)) \\
 & & \mid\ (\mathtt{t} \mid \mathtt{to}\langle n,\mathtt{in},a,p,x\rangle^{\uparrow} \mid \mathtt{amb}[z] \diamond \\
 & & \quad\ (\nu k.\,\mathtt{make}\langle n,p \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright t \mid \mathtt{amb}[z \mid y])) \\
 & & \mid\ (\mathtt{t} \mid \mathtt{up}\langle n,\mathtt{out},a,p,x\rangle_{\downarrow} \diamond \\
 & & \quad\ \mathtt{amb}[z] \triangleright (\nu k.\,\mathtt{make}\langle n,p \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright t \mid \mathtt{amb}[z]))) \\
\mathtt{YQ}(a,k) & = & \mathtt{query}\langle a,k,(r)\rangle \triangleright r\langle a,k,\mathtt{yes}\rangle \\
\mathtt{NQ}(a,t) & = & \mathtt{t} \mid \mathtt{query}\langle a,(k),(r)\rangle \diamond \mathtt{t} \mid r\langle a,k,\mathtt{no}\rangle \\
\mathtt{F}(a,\mathtt{t}) & = & (\mathtt{t} \mid \mathtt{to}\langle n,\mathtt{in},m,p,x\rangle_{\downarrow} :: m \in \mathtt{amb}\bullet\mathtt{C} \diamond \\
 & & \quad \mathtt{amb}[z] \triangleright \\
 & & \quad\ (\nu k.\,\mathtt{make}\langle n,\mathtt{in}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright t \mid \mathtt{amb}[z \mid y])) \\
 & & \mid (\mathtt{t} \mid \mathtt{up}\langle n,\mathtt{out},m,p,x\rangle_{\downarrow} :: m \neq a \diamond \\
 & & \quad \mathtt{amb}[z] \triangleright \\
 & & \quad\ (\nu k.\,\mathtt{make}\langle n,\mathtt{out}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright t \mid \mathtt{amb}[z \mid y])) \\
\mathtt{AmbEnv} & = & \nu\mathtt{t}.\,(\mathtt{C}\ \mathtt{t}\ \mathtt{C}) \mid \mathtt{Collector} \\
\mathtt{C} & = & (\mathtt{t}\ f)\,\mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle f\rangle \\
\mathtt{Factory}(\mathtt{t}) & = & \mathtt{t}\langle f\rangle \mid \mathtt{make}\langle n,p,k\rangle_{\downarrow} \diamond \\
 & & \quad k\langle \nu a.(f\ a\ f) \mid \mathtt{Collector} \mid n[\mathtt{A}(n) \mid \mathtt{amb}[p]]\rangle
\end{array}
$$

A few comments on this encoding are in order. The encoding uses abstraction and application abbreviations defined above, as well as a garbage collector process `Collector` defined as follows:

$$
\begin{array}{rcl}
\mathtt{Collector} & = & \mathtt{collect}\langle (a),(k),p\rangle \diamond (a[x] \triangleright \\
 & & \quad \nu r\,b.\,\mathtt{GC}(a,k,b,r,x,p) \mid b[a[x] \mid \mathtt{query}\langle a,k,r\rangle \mid r\langle a,k,(b)\rangle_{\downarrow} \triangleright r\langle a,k,b\rangle]) \\
\mathtt{GC}(a,k,b,r,x,p) & = & (r\langle a,k,\mathtt{yes}\rangle_{\downarrow} \triangleright (b[y] \triangleright p)) \\
 & & \mid (r\langle a,k,\mathtt{no}\rangle_{\downarrow} \triangleright (b[y] \triangleright (\mathtt{collect}\langle a,k,p\rangle \mid a[x])))
\end{array}
$$

Since, in Mobile Ambients, there can be several ambients bearing the same name within the same ambient, authentication by `Collector` is required prior to collecting a given ambient.

The encoding of the ambient construct, $a[P]$, is typical of encoding of calculi with explicit locations. The process $\mathtt{A}(a)$ in the encoding can be understood as implementing the interaction protocol that is characteristics of Mobile Ambients. Encoding of other forms of ambient calculi would involve defining different variants of this process. Process `AmbEnv` is a helper process that characterizes the environment required by Mobile Ambients, and that provides garbage collection and factory facilities. The auxiliary processes $\mathtt{A}(a)$ and `AmbEnv` are defined below. We use a lock `t` per ambient to avoid conflicts between concurrent moves. Process $\mathtt{S}(a,\mathtt{t})$ starts the execution of `in` and `out` moves at the source ambient. Process $\mathtt{T}(a,\mathtt{t})$ implements the `open` primitive and terminates the execution of `in` and `out` moves originated at a source ambient within process `S`. Notice that all the ambient primitives lead to the the destruction of the source ambient, which is later recreated at the end of execution of the `in` and `out` primitives. Process $\mathtt{F}(a,\mathtt{t})$ aborts transactions implementing the `in` and `out` moves if they cannot complete successfully (i.e. if the required ambient is not present).

If we adopt the slightly unconventional semantics for Mobile Ambients that replaces the usual structural congruence rules for restriction and replication by the following reduction rules:

$$\Delta \vdash \nu a.P \rightarrow_{MA} P\{b/a\} \dashv \Delta \uplus b \qquad !P \rightarrow_{MA} !P \mid P$$

then we easily obtain

Proposition 4.2 *If $P \rightarrow_{MA} Q$, then $[\![P]\!] \rightarrow^{*} \equiv [\![Q]\!]$. Conversely, if $[\![P]\!] \rightarrow^{*} \equiv [\![Q]\!]$, then $P \rightarrow^{*}_{MA} Q$.*

4.3 Encoding the DJoin calculus

An encoding of the Distributed Join (DJoin) calculus can be obtained as follows. For simplicity, we consider only the DJoin calculus without failures. An encoding of the Djoin with fail-stop failures can be obtained by refining the encoding below with failure constructs similar to those introduced in section 2. For any DJoin definition D, we note $\mathtt{df}(D)$ the set of names (channels and locations) it defines. The DJoin encoding is a function of a name that keeps track of the current DJoin location. It is defined by induction as follows, where we assume that $\mathtt{m}$, $\mathtt{mm}$, $\mathtt{loc}$, $\mathtt{collect}$, $\mathtt{query}$, $\mathtt{make}$, $\mathtt{va}$, $\mathtt{enter}$ do not occur free in P, D:

$$
\begin{array}{lll}
[\![a]\!]_b = a & [\![\mathbf{0}]\!]_b = \mathbf{0} & \\
[\![\top]\!]_b = \mathbf{0} & [\![P \mid Q]\!]_b = [\![P]\!]_b \mid [\![Q]\!]_b & \\
[\![\mathtt{go}\, a; P]\!]_b = \mathtt{va}\langle a, [\![P]\!]_a \rangle & [\![D, D']\!]_b = [\![D]\!]_b \mid [\![D']\!]_b & \\
[\![a\langle n_1, \ldots, n_q \rangle]\!]_b = \mathtt{m}\langle b, a, n_1, \ldots, n_q \rangle & [\![D \,\mathtt{in}\, P]\!]_b = \nu\widetilde{n}.\, [\![D]\!]_b \mid [\![P]\!]_b & \widetilde{n} = \mathtt{df}(D)
\end{array}
$$

$$
\begin{array}{l}
[\![n_1 \widetilde{m_1} \mid \ldots \mid n_q \widetilde{m_q} \; \triangleright \; P]\!]_b = \mathtt{m}\langle b, n_1, \widetilde{(m_1)} \rangle \mid \ldots \mid \mathtt{m}\langle b, n_q, \widetilde{(m_q)} \rangle \; \diamond \; [\![P]\!]_b \\
[\![a[D : P]]\!]_b = a[\mathtt{DJ}(a) \mid \mathtt{loc}[[\![D]\!]_a \mid [\![P]\!]_a]] \mid \mathtt{DJEnv}
\end{array}
$$

together with the following auxiliary definitions:

$$
\begin{array}{rcl}
\mathtt{DJ}(a) & = & \nu\mathtt{t}.\, \mathtt{t} \mid \mathtt{IR} \mid \mathtt{Go}(a, \mathtt{t}) \mid \mathtt{Enter}(a, \mathtt{t}) \\
\mathtt{IR} & = & \mathtt{m}\langle (a), x \rangle^{\uparrow} :: a \in \mathtt{C}^{*} \mid \mathtt{loc}[p] \; \diamond \; \mathtt{loc}[p \mid \mathtt{m}\langle a, x \rangle] \\
 & & \mid \; \mathtt{m}\langle (a), x \rangle_{\downarrow} :: \neg(a \in \mathtt{C}^{*}) \; \diamond \; \mathtt{mm}\langle a, x \rangle \\
\mathtt{Go}(a, \mathtt{t}) & = & \mathtt{t} \mid \mathtt{va}\langle (c), p \rangle_{\downarrow} :: \neg(c \in \mathtt{C}^{*}) \; \triangleright \\
 & & \quad (\mathtt{loc}[q] \; \triangleright \\
 & & \quad\;\; \nu k.\, \mathtt{YQ}\langle a, k \rangle \mid \mathtt{collect}\langle a, k, \mathtt{enter}\langle c, a, (q \mid p) \rangle \rangle) \\
\mathtt{YQ}(a, k) & = & \mathtt{query}\langle a, k, (r) \rangle \; \triangleright \; r\langle a, k, \mathtt{yes} \rangle \\
\mathtt{Enter}(a, \mathtt{t}) & = & \mathtt{t} \mid \mathtt{enter}\langle a, (b), x \rangle^{\uparrow} \mid \mathtt{loc}[p] \; \diamond \\
 & & \nu k.\mathtt{make}\langle b, x, k \rangle \; \triangleright \; (k\langle y \rangle^{\uparrow} \; \triangleright \; \mathtt{t} \mid \mathtt{loc}[p \mid y]) \\
\mathtt{DJEnv} & = & \nu\mathtt{t}.(\mathtt{C}\,\mathtt{t}\,\mathtt{C}) \mid \mathtt{ER} \mid \mathtt{Collector} \\
\mathtt{C} & = & \lambda \mathtt{t}\, f.\, \mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle f \rangle \\
\mathtt{Factory}(\mathtt{t}) & = & \mathtt{t}\langle f \rangle \mid \mathtt{make}\langle n, p, k \rangle_{\downarrow} \; \diamond \; k\langle \nu a.(f\, a\, f) \mid \mathtt{ER} \mid \mathtt{GC} \mid n[\mathtt{DJ}(n) \mid \mathtt{loc}[p]] \rangle \\
\mathtt{ER} & = & \mathtt{mm}\langle x \rangle_{\downarrow} \; \diamond \; \mathtt{m}\langle x \rangle
\end{array}
$$

Some comments are in order. Note that the encoding of a DJoin locality takes the same general form as that of a Mobile Ambient: a locality a has a controlling process $\mathtt{DJ}(a)$, that implements the basic interaction protocol that governs a DJoin locality. The latter includes: routing messages on the basis of the target locality, implementing locality migration, by means of the $\mathtt{Go}(a)$ and $\mathtt{Enter}(a)$ processes. Note that the encoding given above is faithful to the DJoin semantics, since migration is only allowed if the target locality does not appear as a sublocality of the current locality[2]. We obtain

Proposition 4.3 *If $P \rightarrow_{DJ} Q$, then $[\![P]\!] \rightarrow^{*} \equiv [\![Q]\!]$. Conversely, if $[\![P]\!] \rightarrow^{*} \equiv [\![Q]\!]$, then $P \rightarrow^{*}_{DJ} Q$.*

[2]This is not the case of the encoding of the DJoin calculus in the M-calculus defined in [17], which does not test for the presence of the target locality as a sublocality of the locality to be migrated. It is possible to faithfully encode the Djoin calculus in the M-calculus but at the cost of a more complex translation than the one reported in [17].

5 Conclusion

We have introduced in this paper a family of new process calculi that we call collectively the Kell calculus. Calculi in this family share the same basic constructs and operational semantics rules. They differ only on the language of patterns used in trigger (or receiver) processes. A Kell calculus essentially consists in an extension of the asynchronous higher-order π-calculus with hierarchical localities. We have shown by means of encodings of Mobile Ambients and of the Distributed Join calculus that the Kell calculus has considerable expressive power. The report [18] shows how to encode the M-calculus in the Kell calculus used in section 4. All these encodings are locality-preserving, in the sense that they translate a locality $a[P]$ in one calculus into a kell of the form $a[M(a) \mid \mathtt{t}[[\![P]\!]] \mid \mathtt{Env}$, where $\mathtt{Env}$ is a stateless process. We believe such locality-preserving encodings can be derived for most process calculi with localities which have been proposed in the litterature, including the numerous variants of Mobile Ambients, Nomadic Pict, Dπ [9], Klaim [14], and DiTyCo [11]. Obtaining such encodings would give strong evidence that the Kell calculus embodies very fundamental constructs for distributed programming.

To the best of our knowledge, the dual use which is made in the Kell calculus of the locality construct $a[P]$, both as a locus for computation and as a handle for controlling the execution of located process, is new. The encodings provided in this paper show that a single (higher-order) objective control construct is sufficient to capture the variety of subjective migration primitives which have been proposed recently, in ambient calculi and other distributed process calculi. At the same time, this construct is powerful enough to model fail-stop failures, an important requirement for practical distributed programming.

Much work remains to be done, however, to assess the foundational character of the calculus with respect to distributed programming. Apart from the derivation of locality-preserving encodings mentioned above, the following issues are worth considering:

- Developing a bisimulation theory for the Kell calculus. Apart from the difficulties inherent with the higher-order character of the calculus, it would be interesting to obtain a theory parametric in the pattern language used.
- Developing type systems for the Kell calculus. Numerous type systems have been developed for mobile Ambients and their variants. It would be interesting to transfer these results (in particular the ones dealing with resource and security constraints) to the Kell calculus. Of particular interest would be the transfer of the type system developed for the M-calculus that guarantees the unicity of locality names, since this corresponds to a practical constraint in today's networks.
- Introducing the possibility to share processes among different kells. If one considers a kell (or locality) not only as a locus of computation but also as a component, sharing among kells appears as an important practical requirement. However, sharing raises considerable difficulties, which are very much related to the aliasing problem in object-oriented programming.

References

[1] R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science, vol. 96*, 1992.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.

[4] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST-TCS '02)*, volume LNCS 2556. Springer, 2002.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[6] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. $\mathbf{M}^3$: Mobility types for mobile processes in mobile ambients. In *CATS 2003)*, volume 78 of *ENTCS*, 2003.

[7] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.

[8] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.

[9] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical report, Technical Report 2/98 – School of Cognitive and Computer Sciences, University of Sussex, UK, 1998.

[10] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.

[11] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Mobile Object Systems Workshop (MOS'99)*, 1999.

[12] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.

[13] R. Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, 1999.

[14] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.

[15] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2001.

[16] D. Sangiorgi and S. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[18] J.B. Stefani. A calculus of higher-order distributed components. Technical report, INRIA RR-4692, 2003.

[19] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *to appear in Proceedings CONCUR 02*, 2002.

[20] B. Thomsen. A Theory of Higher Order Communicating Systems. *Information and Computation, Vol. 116, No 1*, 1995.

[21] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.

[22] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.