Electronic Notes in Theoretical Computer Science 85 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume85.html 21 pages

A Calculus of Kells

Jean-Bernard Stefani [a]

[a] *INRIA Rhône-Alpes,*
655, Avenue de l'Europe, 38334 Saint-Ismier Cedex, France
E-mail: jean-bernard.stefani@inrialpes.fr

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) [12], Safe Ambients with passwords [14], Boxed Ambients (BA) [3], Controlled Ambients (CA) [20], New Boxed Ambients (NBA) [4], Ambients with process migration ($\mathbf{M}^3$) [7].

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 [10].

Part of the difficulty in implementing Mobile Ambients is related to the presence of "grave interferences", as explained in [12]. 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 [16], 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 [9] 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 [11] or Nomadic Pict [22], 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 $D\lambda\pi$ [23] and the M-calculus [18], 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 paper we present an element of this famly that enjoys a very simple pattern language. In this section, we present informally the different constructs of the calculus.

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, $a\langle \boldsymbol{u} \rangle$, where a is a name, $\boldsymbol{u}$ is a vector of elements u, and where each element u can be either a name or a process.
- *triggers*, or receivers, of the form $\xi \triangleright P$, where ξ is a message pattern (analogous to the join patterns in the Join calculus) and P is an arbitrary kell calculus process.

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 this paper, patterns are given by the following grammar:

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

where $\boldsymbol{w}$ 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 variables 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 $\triangleright$.

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

(i) 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 $a\langle x \rangle \triangleright P$:

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

(ii) 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 :

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

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.

(iii) 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:

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

(iv) Passivation 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 (i) above are standard π-calculus actions. Actions of the form (ii) and (iii) 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 Seal calculus [6] and in the Boxed Ambients calculus.

Actions of the form (iv) 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:

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

We have the following reductions:

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

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 calculus has no primitive for recursion, such as a replication operator $!P$. The reason is that, because of its higher-order character, it is possible to define *receptive triggers*, i.e. triggers that are preserved during a reduction (much like definitions in the Join calculus). Let $t \in \mathsf{N}$, ξ and P be such that t does not occur in ξ or P. In a manner reminiscent of the fixed point operator defined in CHOCS [21], we define $\xi \diamond P$ by:

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

It is easy to see with the rules of reduction given in Section 3.2 that if $M \mid (\xi \triangleright P) \rightarrow P\theta$, where θ is a substitution, then we have $M \mid (\xi \diamond P) \rightarrow (\xi \diamond P) \mid P\theta$.

The higher-order nature of the calculus, together with the passivation construct, 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 $M \triangleq (a\langle x\rangle^{\uparrow} \diamond a\langle x\rangle) \mid (\mathtt{m}\langle(b), x\rangle_{\downarrow} \diamond b\langle x\rangle)$. 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 $M \triangleq (a\langle x\rangle^{\uparrow} \diamond P(x)) \mid (\mathtt{m}\langle b, y\rangle_{\downarrow} \diamond Q(b, y))$. 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 $Q(b, y)$ when a message $\mathtt{m}\langle b, y\rangle$ seeks to leave kell a. Notice how this allows the definition of wrappers with pre and post-handling of messages.
- Migration membrane: Let

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

  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

  $$M \triangleq (\mathtt{stop}\langle a\rangle^{\uparrow} \mid a[y] \triangleright S) \mid (\mathtt{ping}\langle a,(r)\rangle^{\uparrow} \diamond \mathtt{m}\langle r, \mathtt{up}\rangle)$$
  $$S \triangleq \mathtt{ping}\langle a,(r)\rangle^{\uparrow} \diamond \mathtt{m}\langle r, \mathtt{down}\rangle$$

  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 [1] or the Distributed Join calculus models of failures.
- Localities with fail-stop failures and recovery: Let

  $$M \triangleq (\mathtt{stop}\langle a\rangle \mid a[y] \mid c \diamond b\langle y\rangle \mid f) \mid (\mathtt{ping}\langle a,(r)\rangle \mid c \diamond c \mid \mathtt{m}\langle r, \mathtt{up}\rangle)$$
  $$S \triangleq (\mathtt{ping}\langle a,(r)\rangle \mid f \diamond f \mid \mathtt{m}\langle r, \mathtt{down}\rangle) \mid (\mathtt{recover}\langle a\rangle \mid b\langle y\rangle \mid f \diamond c \mid a[y])$$

  Then, $e[\nu c\, f.\, c \mid M \mid S \mid a[K]]$ defines a membrane around kell a that models fail-stop failures of kell a together with a simple failure detector and the possibility of recovery.

These different examples can be coded very similarly in the M-calculus, illustrating the fact that the Kell calculus retains the ability of the M-calculus to define localities with different semantics. The first two examples, when involving only first-order communication, can be coded analogously in variants of Boxed Ambients and in the Seal calculus. The third example can be coded in the Seal calculus and in Ambients calculi (in the latter, by coding the objective move into a protocol

$$
\begin{aligned}
P &::= \mathbf{0} \quad | \quad x \quad | \quad \xi \triangleright P \quad | \quad \nu a.P \quad | \quad a\langle \boldsymbol{u} \rangle \quad | \quad P \mid P \quad | \quad a[P] \\
\xi &::= J \quad | \quad J \mid a[x] \\
J &::= \epsilon \quad | \quad a\langle \boldsymbol{w} \rangle \quad | \quad a\langle \boldsymbol{w} \rangle^{\uparrow} \quad | \quad a\langle \boldsymbol{w} \rangle_{\downarrow} \quad | \quad J \mid J \\
u &::= a \quad | \quad P \\
w &::= a \quad | \quad (a) \quad | \quad x \\
a &\in \mathsf{N} \\
x &\in \mathsf{V}
\end{aligned}
$$

Fig. 1. Syntax of the Kell Calculus

activating subjective migration). The fourth example can be coded in the Seal calculus. The fifth example can be only partially simulated in the Seal calculus since one can isolate, duplicate or destroy a seal, but one cannot freeze its execution.

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:

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

3 The Kell calculus: syntax and semantics

3.1 Syntax

The syntax of the Kell calculus is given in Figure 1. We assume an infinite set N of *names*, and an infinite set V of *process variables*, such 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. 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 $a\langle \boldsymbol{u} \rangle$. We let M, N and their decorated variants range over messages and parallel composition of messages. We call *trigger* a process of the form $\xi \triangleright P$, where ξ is a *receipt pattern* (or *pattern*, for short). A pattern can be a *join pattern* J, or a *control pattern* of the form $J \mid a[x]$, in which the join pattern J may be empty (i.e. $J = \epsilon$ – we set $J \mid \epsilon = \epsilon \mid J = J$).

In a term $\nu a.P$, the scope extends as far to the right as possible. We use $\boldsymbol{u}$ to denote finite vectors $(u_1, \ldots, u_q)$ (vectors can be empty; the empty vector is noted $\langle\rangle$). Abusing the notation, we equate $\boldsymbol{u} = (u_1, \ldots, u_n)$ with the word $u_1 \ldots u_n$ and the set $\{u_1, \ldots, u_n\}$. We note $|\boldsymbol{u}|$ the length n of a vector $\boldsymbol{u} = (u_1, \ldots, u_n)$. We

$$\mathbf{C} ::= \cdot \quad | \quad \xi \triangleright \mathbf{C} \quad | \quad \nu a.\mathbf{C} \quad | \quad (P \mid \mathbf{C}) \quad | \quad a[\mathbf{C}] \quad | \quad a\langle \mathbf{C} \rangle$$
$$\mathbf{E} ::= \cdot \quad | \quad \nu a.\mathbf{E} \quad | \quad a[\mathbf{E}] \quad | \quad P \mid \mathbf{E}$$

Fig. 2. Syntax of Contexts

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 $\nu \boldsymbol{a}.P$ if $\boldsymbol{a} = (a_1, \ldots, a_q)$. By convention, if the name vector $\boldsymbol{a}$ is empty, then $\nu \boldsymbol{a}.P \triangleq P$. 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 2. 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 ξ can bind *name variables*, of the form (a), where $a \in \mathsf{N}$, and *process variables*. All name and process variables that occur in a pattern ξ are bound in ξ. Name variables can only match names. Process variables can only match processes. Process variables can only occur linearly in a pattern ξ, i.e. if x occurs in ξ, then there is only one occurrence of x in ξ. 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 (process) variables ($\mathtt{fv}$) are defined in Figure 3. We note $\mathtt{fn}(\boldsymbol{u})$ to mean $\mathtt{fn}(u_1) \cup \mathtt{fn}(u_n)$, and likewise for $\mathtt{fv}(\boldsymbol{u})$, $\mathtt{bn}(\boldsymbol{u})$, $\mathtt{bv}(\boldsymbol{u})$. We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

We call *substitution* a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{V} \to \mathsf{K}$, from names to names and process variables to processes, that is the identity except on a finite set of names and variables. We write $P\theta$ the image under the substitution θ of process P. We note Θ the set of substitutions, and $\mathtt{supp}$ the support of a substitution (i.e. $\mathtt{supp}(\theta) = \{i \in \mathsf{L} \mid \theta(i) \neq i\}$).

Let J be a join pattern, and θ be a substitution such that $\mathtt{supp}(\theta) = \mathtt{bn}(J) \cup \mathtt{bv}(J)$. We define the image $J\theta$ of J under substitution θ as $\mathtt{cj}(J)\theta$, where $\mathtt{cj}$ is the function defined inductively as:

$$\mathtt{cj}(J \mid J') = \mathtt{cj}(J) \mid \mathtt{cj}(J') \qquad \mathtt{cj}(\epsilon) = \mathbf{0}$$
$$\mathtt{cj}(a\langle \boldsymbol{w} \rangle) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle \qquad \mathtt{cj}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle \qquad \mathtt{cj}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = a\langle \mathtt{cj}(\boldsymbol{w}) \rangle$$
$$\mathtt{cj}(a) = a \qquad \mathtt{cj}((a)) = a \qquad \mathtt{cj}(x) = x$$

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 4. The rules

$$
\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 \boldsymbol{u} \rangle) = \mathtt{fn}(\boldsymbol{u}) \cup \{a\} & \mathtt{fv}(a\langle \boldsymbol{u} \rangle) = \mathtt{fv}(\boldsymbol{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}(P) \setminus \mathtt{bv}(\xi) \\
\mathtt{fn}(J \mid a[x]) = \mathtt{fn}(J) \cup \{a\} & \mathtt{bn}(J \mid a[x]) = \mathtt{bn}(J) \\
\mathtt{fn}(J \mid J') = \mathtt{fn}(J) \cup \mathtt{fn}(J') & \mathtt{bn}(J \mid J') = \mathtt{bn}(J) \cup \mathtt{bn}(J') \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{fn}(a\langle \boldsymbol{w} \rangle) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{bn}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{fn}(a\langle \boldsymbol{w} \rangle) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{bn}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{fn}(a\langle \boldsymbol{w} \rangle) = \{a\} \cup \mathtt{fn}(w) & \mathtt{bn}(a\langle \boldsymbol{w} \rangle) = \mathtt{bn}(\boldsymbol{w}) \\
\mathtt{fn}((a)) = \emptyset & \mathtt{bn}((a)) = \{a\} \\
\mathtt{bn}(a) = \emptyset & \mathtt{bn}(x) = \emptyset \\
\mathtt{bv}(J \mid a[x]) = \mathtt{bv}(J) \cup \{x\} & \mathtt{bv}(J \mid J') = \mathtt{bv}(J) \cup \mathtt{bv}(J') \\
\mathtt{bv}(a\langle \boldsymbol{w} \rangle^{\uparrow}) = \mathtt{bv}(a\langle \boldsymbol{w} \rangle) & \mathtt{bv}(a\langle \boldsymbol{w} \rangle_{\downarrow}) = \mathtt{bv}(a\langle \boldsymbol{w} \rangle) \\
\mathtt{bv}(a\langle \boldsymbol{w} \rangle) = \mathtt{bv}(\boldsymbol{w}) & \mathtt{bv}(a) = \emptyset \\
\mathtt{bv}((a)) = \emptyset & \mathtt{bv}(x) = \{x\}
\end{array}
$$

Fig. 3. Free names and free variables

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \;[\text{S.PAR.ASSOC}] \qquad P \mid Q \equiv Q \mid P \;[\text{S.PAR.COMM}]$$

$$P \mid \mathbf{0} \equiv P \;[\text{S.PAR.NIL}] \qquad \nu a.\mathbf{0} \equiv \mathbf{0} \;[\text{S.NU.NIL}]$$

$$\nu a.\nu b.P \equiv \nu b.\nu a.P \;[\text{S.NU.COMM}] \qquad \frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \;[\text{S.NU.PAR}]$$

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

Fig. 4. Structural congruence

S.PAR.ASSOC, S.PAR.COMM, S.PAR.NIL state that the parallel operator $\mid$ is associative, commutative, and has **0** as a neutral element.

Notice that we do not have structural congruence rules that deal with scope extrusion outside a locality, as in Mobile Ambients. This is because, in presence of the possibility of passivating executing processes, the Ambient scope extrusion rule $a[\nu b.P] \equiv \nu b.a[P] \;\; (b \neq a)$ would give rise to behaviour which would violate the idea that structurally equivalent processes should behave similarly in the same

$$\frac{\begin{array}{c} J = J_1 \mid J_2 \\ J_1 = \prod_{i \in I_1} a_i \langle \boldsymbol{w_i} \rangle^{\uparrow} \neq \epsilon \qquad J_2 = \prod_{i \in I_2} a_i \langle \boldsymbol{w_i} \rangle \qquad \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) = \emptyset \end{array}}{J_1\theta \mid b[\nu \boldsymbol{c}.R \mid J_2\theta \mid (J \rhd P)] \to b[\nu \boldsymbol{c}.R \mid P\theta]} \text{ [R.IN]}$$

$$\frac{\begin{array}{c} J = J_1 \mid J_2 \mid a[x] \\ J_1 = \prod_{i \in I_1} a_i \langle \boldsymbol{w_i} \rangle^{\uparrow} \qquad J_2 = \prod_{i \in I_2} a_i \langle \boldsymbol{w_i} \rangle \qquad \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) = \emptyset \end{array}}{J_1\theta \mid b[\nu \boldsymbol{c}.R \mid J_2\theta \mid a[Q] \mid (J \rhd P)] \to b[\nu \boldsymbol{c}.R \mid P\theta\{Q/x\}]} \text{ [R.IN.PASS]}$$

$$\frac{\begin{array}{c} J = J_1 \mid J_2 \qquad J_1 = \prod_{i \in I_1} a_i \langle \boldsymbol{w_i} \rangle_{\downarrow} \neq \epsilon \qquad J_2 = \prod_{i \in I_2} a_i \langle \boldsymbol{w_i} \rangle \\ \boldsymbol{d} = \boldsymbol{c} \setminus \boldsymbol{e} \qquad \boldsymbol{e} = \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) \qquad \boldsymbol{e} \cap \mathtt{fn}(J \rhd P, J_2\theta, b) = \emptyset \end{array}}{J_2\theta \mid (J \rhd P) \mid b[\nu \boldsymbol{c}.R \mid J_1\theta] \to \nu \boldsymbol{e}.P\theta \mid b[\nu \boldsymbol{d}.R]} \text{ [R.OUT]}$$

$$\frac{\begin{array}{c} J = J_1 \mid J_2 \mid a[x] \qquad J_1 = \prod_{i \in I_1} a_i \langle \boldsymbol{w_i} \rangle_{\downarrow} \qquad J_2 = \prod_{i \in I_2} a_i \langle \boldsymbol{w_i} \rangle \\ \boldsymbol{d} = \boldsymbol{c} \setminus \boldsymbol{e} \qquad \boldsymbol{e} = \boldsymbol{c} \cap \mathtt{fn}(J_1\theta) \qquad \boldsymbol{e} \cap \mathtt{fn}(a, b, Q, J \rhd P, J_2\theta) = \emptyset \end{array}}{J_2\theta \mid a[Q] \mid (J \rhd P) \mid b[\nu \boldsymbol{c}.R \mid J_1\theta] \to \nu \boldsymbol{e}.P\theta\{Q/x\} \mid b[\nu \boldsymbol{d}.R]} \text{ [R.OUT.PASS]}$$

$$\frac{J = \prod_{i \in I} a_i \langle \boldsymbol{w_i} \rangle \neq \epsilon}{J\theta \mid (J \rhd P) \to P\theta} \text{ [R.LOCAL]} \qquad \frac{J = J_1 \mid a[x] \qquad J_1 = \prod_{i \in I} a_i \langle \boldsymbol{w_i} \rangle}{J_1\theta \mid a[Q] \mid (J \rhd P) \to P\theta\{Q/x\}} \text{ [R.LOCAL.PASS]}$$

$$\frac{P \to Q}{E[P] \to E[Q]} \text{ [R.CONTEXT]} \qquad \frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'} \text{ [R-STRUCT]}$$

Fig. 5. Reduction Relation

evaluation context, as illustrated by the following reductions:

$$(a[x] \rhd x \mid x) \mid a[\nu b.P] \;\to\; (\nu b.P) \mid (\nu b.P) \qquad (a[x] \rhd x \mid x) \mid \nu b.a[P] \;\to\; \nu b.P \mid P$$

The reduction relation $\to$ is the smallest binary relation on K^2 that satisfies the rules given in Figure 5, where we assume that $\mathtt{supp}(\theta) = \mathtt{bn}(J) \cup \mathtt{bv}(J)$.

Some comments are in order. Rules R.IN an R.OUT take into account the presence of restrictions inside kells, since restricted names cannot be automatically extruded out of kells through the structural congruence. Rule R.OUT explicitly extrudes restricted names that are communicated outside a kell boundary. Note that names that are not communicated are not extruded. Rules R.IN and R.OUT govern the crossing of kell boundaries. Note that only messages may cross a kell boundary. In rule R.IN, a trigger receives messages from the local environment as well as from the outside of the enclosing kell. In rule R.OUT, a trigger receives messages from the local environment as well as from a subkell. Rules R.IN.PASS, R.OUT.PASS and R.LOCAL.PASS allow the passivation of a subkell, possibly upon receipt of messages. In rules R.IN and R.OUT, note that the join pattern J_2 may be empty. Likewise, in rules R.IN.PASS, R.OUT.PASS and R.LOCAL.PASS, the join patterns J_1 and J_2 may be empty.

Rules R.IN, R.OUT, R.IN.PASS, R.OUT.PASS look rather involved, but only because they allow synchronizing the receipt of messages crossing a kell boundary with the receipt of local messages and the passivation of a subkell.

4 Encodings

We present in this section several encodings of process calculi to illustrate the versatility of the Kell calculus. In particular, we present faithful encodings of calculi with localities. By *faithful encoding* of a process calculus with a locality construct $a[P]$, we mean a function $[\![]\!]$ which is such that $[\![a[P]]\!] = a[M(a,c) \mid c[[\![P]\!] \mid \texttt{Aux}]] \mid \texttt{Env}$ or $[\![a[P]]\!] = c[M(a,c) \mid a[[\![P]\!] \mid \texttt{Aux}]] \mid \texttt{Env}$, or even $[\![a[P]]\!] = a[M(a) \mid [\![P]\!]] \mid \texttt{Env}$, where $\texttt{Env}$ and $\texttt{Aux}$ are stateless processes. In other terms, a faithful encoding translates a locality of name a into a kell of name a, possibly embedded in a controlling kell. The semantics of the locality is then captured by the membrane process $M(a,c)$ or $M(a)$.

To simplify the encodings, we use receptive triggers $\xi \diamond P$. We also use the abbreviations abstraction $(x)P$ and application PQ. The resulting extended calculus is defined by induction thus (notice the implicit typing to ensure well-formed processes):

$$[\![P]\!] \triangleq P \quad \text{if } P \in \mathsf{K}$$

$$[\![(x)P]\!]_f \triangleq f\langle x\rangle \triangleright [\![P]\!]$$

$$[\![PQ]\!] \triangleq \nu f.[\![P]\!]_f \mid f\langle [\![Q]\!]\rangle$$

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 input guarded sums and name matching (cf [17] for a definition) is given below, where we assume that the names $1, \ldots, n, \ldots$, and $\mathtt{k}$ do not appear free in P, P_j, Q, and where $\boldsymbol{b} = b_1 \ldots b_n$, $\boldsymbol{b^j} = b^j_1 \ldots b^j_{n_j}$, $j \in J$.

$$\begin{aligned}
[\![\mathbf{0}]\!] &= \mathbf{0}\\
[\![a\boldsymbol{b}.P]\!] &= a\langle [\![P]\!], b_1, \ldots, b_n\rangle\\
[\![\tau.P]\!] &= \nu\mathtt{k}.\mathtt{k} \mid (\mathtt{k} \triangleright [\![P]\!])\\
[\![[a=b]P]\!] &= \nu l.(l\langle a\rangle \triangleright [\![P]\!]) \mid l\langle b\rangle\\
[\![\nu a.P]\!] &= \nu a.[\![P]\!]\\
[\![P \mid Q]\!] &= [\![P]\!] \mid [\![Q]\!]\\
[\![!P]\!] &= \nu\mathtt{k}.\mathtt{k} \mid (\mathtt{k} \diamond [\![P]\!] \mid \mathtt{k})\\
[\![\sum_{j\in J} a_j(\boldsymbol{b}^j).P_j]\!] &= \nu\mathtt{k}.\mathtt{k} \mid \prod_{j\in J} \mathtt{k} \mid a_j\langle x_j, (b^j_1), \ldots, (b^j_{n_j})\rangle \triangleright [\![P_j]\!] \mid x_j
\end{aligned}$$

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

$$!P \rightarrow_\pi !P \mid P \qquad [a = a]P \rightarrow_\pi P$$

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 $[\![P]\!] \rightarrow^* P'$, then there exists Q, R such that $P \rightarrow_\pi Q$, and $P' \rightarrow^* \equiv [\![Q]\!] \mid R$.*

If we assume the existence of a function $f : \mathsf{N} \rightarrow \{0,1\}^*$, that, given a name a, returns a binary encoding $f(a)$ of a, we can strengthen the result into an encoding of the synchronous polyadic π-calculus with input guarded sums, name matching *and* name unmatching. The new encoding $[\![]\!]^+$ is defined as follows:

$$[\![P]\!]^+ = [\![P]\!]_s \mid \mathtt{GC}$$
$$GC = \mathtt{collect}\langle(c), x\rangle_\downarrow \diamond (c[z] \triangleright x)$$

where $[\![]\!]_s$ is just like function $[\![]\!]$ above, except that the definitions for name matching and guarded input have been replaced respectively by:

$$[\![[a = b]P, Q]\!]_s = \nu c\, r.\ c[\mathtt{check}(a, r) \mid \lceil b \rceil \mid \mathtt{ThenElse}(P, Q, r, c)]$$

$$[\![\sum_{j \in J} a_j(\boldsymbol{b}^j).P_j]\!]_s = \nu \mathtt{k}\, c.$$

$$c[\mathtt{k} \mid \prod_{j \in J} \mathtt{k} \mid a_j\langle x_j, (b^j_1), \ldots, (b^j_{n_j})\rangle \triangleright \mathtt{collect}\langle c, [\![P_j]\!]_s \mid x_j\rangle]$$

The auxiliary processes $\mathtt{check}$, $\lceil b \rceil$, and $\mathtt{ThenElse}$ are defined as follows (we assume that $0, 1 \in \mathsf{N}$, that $f(a) = v_1 \ldots v_n$ and $f(b) = w_1 \ldots w_m$, with $v_i, w_j \in \{0,1\}$, and we set $\overline{0} = 1, \overline{1} = 0$):

$$\begin{aligned}
\mathtt{check}(a, r) &= \prod_{i=1}^{n-1} \mathtt{lcheck}(v_i) \mid \mathtt{fcheck}(v_n) \mid \mathtt{return}(r) \\
\mathtt{lcheck}(v) &= (\mathtt{l}\langle v, x\rangle \triangleright x) \mid (\mathtt{l}\langle \overline{v}, x\rangle \triangleright \mathtt{nok}) \\
\mathtt{fcheck}(v) &= (\mathtt{l}\langle v, x\rangle \triangleright \mathtt{ok} \mid x) \mid (\mathtt{l}\langle \overline{v}, x\rangle \triangleright \mathtt{nok}) \\
\mathtt{return}(r) &= (\mathtt{ok} \mid \mathtt{end} \triangleright r\langle \mathtt{yes}\rangle) \\
&\quad \mid (\mathtt{ok} \mid \mathtt{l}\langle (v), x\rangle \triangleright r\langle \mathtt{no}\rangle) \mid (\mathtt{nok} \triangleright r\langle \mathtt{no}\rangle) \\
\lceil b \rceil &= \mathtt{l}\langle w_1, \mathtt{l}\langle w_2, \ldots \mathtt{l}\langle w_m, \mathtt{end}\rangle \ldots\rangle\rangle \\
\mathtt{ThenElse}(P, Q, r, c) &= (r\langle \mathtt{no}\rangle \triangleright \mathtt{collect}\langle c, [\![Q]\!]_s\rangle) \\
&\quad \mid (r\langle \mathtt{yes}\rangle \triangleright \mathtt{collect}\langle c, [\![P]\!]_s\rangle)
\end{aligned}$$

Intuitively, the process GC is a garbage collector that collects auxiliary kells which have been created by the translation and that can be safely discarded after they have served their purpose. The garbage collector is a stateless process that spawns a specific collector for kell c upon receipt of a message $\mathtt{collect}\langle c, P\rangle$. The c collector removes kell c and releases process P as a continuation. We will find variant of the garbage collector idea in other encodings. Process $\mathtt{check}(a, r)$ realizes a straightforward bitwise comparison against the binary encoding of a and returns the result on port r. The process $\lceil b \rceil$ encodes the bit string corresponding to the binary representation of b as a list. Finally, process `ThenElse` just triggers the collection of the auxiliary kell used during the match, passing the expected continuation as an argument of the `collect` message. It is important to realize that enclosing the matching process inside an auxiliary kell c has two purposes: the first one is to isolate the computation carried out during the matching process from the rest of the computation, to avoid interferences; the second one is to serve as a cell for future garbage collection of useless processes. With this encoding we get:

Proposition 4.2 *If $P \rightarrow_\pi Q$, then $[\![P]\!]^+ \rightarrow^* \equiv [\![Q]\!]^+$. Conversely, if $[\![P]\!]^+ \rightarrow^* P'$, then there exists Q such that $P \rightarrow_\pi Q$, and $P' \rightarrow^* \equiv [\![Q]\!]^+$.*

4.2 Encoding Klaim

We consider now an encoding of a Klaim-like language. Specifically, we consider an (untyped) version of the Klaim language defined in [15]. For simplicity, we do not consider Klaim process definitions, we consider empty Klaim environments (i.e. names of nodes have global significance in this version of Klaim), and that tuples have only a single element. The encoding is defined as follows (an evaluated tuple element u can be either a node name a or a Klaim process P; a tuple element w can be either a node name pattern $!a$, a process pattern $!x$, a node name a, or a Klaim process P; and we define $\overline{(a)} \triangleq a$, $\overline{a} = a$, $\overline{x} = x$, $\overline{P} = P$).

$$
\begin{aligned}
[\![\mathbf{0}]\!]_b &= \mathbf{0} \qquad [\![a]\!]_b = a \\
[\![!a]\!]_b &= (a) \qquad [\![x]\!]_b = x \\
[\![!x]\!]_b &= x \\
[\![P \mid Q]\!]_b &= [\![P]\!]_b \mid [\![Q]\!]_b \\
[\![\mathtt{out}(u)]\!]_b &= \mathtt{out}\langle [\![u]\!]_b \rangle \\
[\![\mathtt{out}(w)@a.P]\!]_b &= \nu c.\mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \mid \mathtt{m}\langle b, c\langle \mathtt{ok} \rangle \rangle \rangle \mid (c\langle \mathtt{ok} \rangle \triangleright [\![P]\!]_b) \\
[\![\mathtt{int}(w)@a.P]\!]_b &= \nu c.\mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \triangleright \mathtt{m}\langle b, c\langle [\![P]\!]_b \rangle \rangle \rangle \mid (c\langle x \rangle \triangleright x) \\
[\![\mathtt{read}(w)@a.P]\!]_b &= \nu c.\mathtt{m}\langle a, \mathtt{out}\langle [\![w]\!]_a \rangle \triangleright \mathtt{m}\langle b, c\langle [\![P]\!]_b \rangle \rangle \mid \mathtt{out}\langle \overline{[\![w]\!]_a} \rangle \rangle \mid (c\langle x \rangle \triangleright x) \\
[\![\mathtt{eval}(P)@a.Q]\!]_b &= \nu c.\mathtt{m}\langle a, [\![P]\!]_a \mid \mathtt{m}\langle b, c\langle \mathtt{ok} \rangle \rangle \rangle \mid (c\langle \mathtt{ok} \rangle \triangleright [\![Q]\!]_b) \\
[\![\mathtt{newloc}(a).P]\!]_b &= \nu c.\mathtt{newNode}\langle c \rangle \triangleright c\langle (a) \rangle^{\uparrow} \triangleright [\![P]\!]_b \\
[\![a :: P]\!] &= a[\mathtt{K}(a) \mid [\![P]\!]_a] \mid \mathtt{KEnv} \\
[\![N_1 \parallel N_2]\!] &= [\![N_1]\!] \mid [\![N_2]\!] \\
\mathtt{K}(a) &= a\langle x \rangle^{\uparrow} \diamond x \\
\mathtt{KEnv} &= (\mathtt{newNode}\langle (c) \rangle_{\downarrow} \diamond \nu a.a[\mathtt{K}(a)] \mid c\langle a \rangle) \mid (\mathtt{m}\langle (a), x \rangle_{\downarrow} \diamond a\langle x \rangle)
\end{aligned}
$$

The idea of the encoding is simple: each Klaim node $a :: P$ is modelled by a kell $a[\mathtt{K}(a) \mid [\![P]\!]]$, where $\mathtt{K}(a)$ constitutes the program of the membrane associated with a Klaim domain. In this case, we do not separate the content P of a Klaim domain from its membrane. Klaim instructions are then encoded as messages bearing some code that will be executed upon arrival at the target node. The environment `Kenv` plays the role of a simple router and of a Klaim node factory. Note that this encoding does not use the passivation construct of the Kell calculus: Klaim does not support strong mobility (i.e. migrating executing process). For this simplified variant of Klaim, we obtain

Proposition 4.3 *If $N \to_K N'$, then $[\![N]\!] \to^* \equiv [\![N']\!]$. Conversely, if $[\![N]\!] \to^* P$, then there exists N' such that $N \to_K N'$, and $P \to^* \equiv [\![N']\!]$.*

4.3 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 faithful and deadlock-free, but it relies on a simple locking scheme that reduces the parallelism inherent in ambient reductions. The encoding is not divergence-free (because of the definition of process $\mathtt{F}(a, \mathtt{t})$ below). 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 [10]) but it would be more complex. The encoding demonstrates that the passivation construct provides the basis for implementing the subjective moves of Mobile Ambients, as well as its objective `open` primitive.

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

$$[\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![\mathtt{in}\, a.P]\!] = \mathtt{in}\langle a, [\![P]\!]\rangle$$

$$[\![\nu n.P]\!] = \nu n.[\![P]\!] \qquad [\![\mathtt{out}\, a.P]\!] = \mathtt{out}\langle a, [\![P]\!]\rangle$$

$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!] \qquad [\![\mathtt{open}\, a.P]\!] = \mathtt{open}\langle a, [\![P]\!]\rangle$$

$$[\![!P]\!] = \nu \mathtt{k}.\, \mathtt{k} \mid (\mathtt{k} \diamond [\![P]\!] \mid \mathtt{k})$$

$$[\![a[P]]\!] = \nu c.\, c[\mathtt{A}(a, c) \mid a[[\![P]\!] \mid \mathtt{AmbEnv}]] \mid \mathtt{AmbEnv}$$

The `AmbEnv` process is defined below. It consists of three processes: an ambient factory `C`, a garbage collector `GC`, and a process `Aux`, explained later on.

$$\mathtt{AmbEnv} = \nu \mathtt{t}.\, (\mathtt{C}\ \mathtt{t}\ \mathtt{C}) \mid \mathtt{GC} \mid \mathtt{Aux}$$

The ambient factory is defined using abstraction and application abbreviations introduced at the beginning of Section 4. Its role is to create new ambients, upon receipt of a `make` message. The definition of this factory is made complicated by the fact that it must create a copy of itself to be placed alongside the newly created ambient. To this end, we resort to a construction that is very similar to that of the `Y` fixpoint constructor we used in the definition of receptive triggers.

$$\begin{aligned}
\mathtt{C} &= (\mathtt{t}\ x)\, \mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle x\rangle \\
\mathtt{Factory}(\mathtt{t}) &= (\mathtt{t}\langle x\rangle \mid \mathtt{make}\langle (n), p, (k)\rangle_{\downarrow} \diamond \mathtt{MK}(\mathtt{t}, x, n, p, k)) \\
&\quad \mid (\mathtt{t}\langle x\rangle \mid \mathtt{make}\langle (n), p, (k)\rangle \diamond \mathtt{MK}(\mathtt{t}, x, n, p, k)) \\
\mathtt{MK}(\mathtt{t}, x, n, p, k) &= \mathtt{t}\langle x\rangle \mid k\langle \mathtt{AE}(x) \mid \nu c.\, c[\mathtt{A}(n, c) \mid n[p \mid \mathtt{AE}(x)]]\rangle \\
\mathtt{AE}(x) &= \nu a.(x\ a\ x) \mid \mathtt{GC} \mid \mathtt{Aux}
\end{aligned}$$

The garbage collector process $\mathtt{GC}$ corresponds to the garbage collector we defined in Section 4.1 for the encoding of the π-calculus with unmatching.

The "membrane process" $\mathtt{A}$ is defined below. It consists in the parallel composition of three processes $\mathtt{S}$, $\mathtt{T}$, $\mathtt{F}$, together with a private lock, $\mathtt{t}$, which is used to avoid conflicts between concurrent moves.

$$\mathtt{A}(a,c) = \nu\mathtt{t}.\mathtt{t} \mid \mathtt{S}(a,c,\mathtt{t}) \mid \mathtt{T}(a,c,\mathtt{t}) \mid \mathtt{F}(a,c,\mathtt{t})$$

Process $\mathtt{S}$ is responsible for dealing with the $\mathtt{in}$ and $\mathtt{out}$ primitives (which are translated as higher-order messages on ports $\mathtt{in}$ and $\mathtt{out}$, respectively). In each case, the behavior is simple: passivate the ambient a, put it in a message ($\mathtt{to}$, in the case of $\mathtt{in}$; $\mathtt{up}$, in the case of $\mathtt{out}$), and asks the garbage collector to collect the now useless enclosing kell. The message bearing the passivated ambient will be released after the garbage collection has taken place.

$$\begin{aligned}\mathtt{S}(a,c,\mathtt{t}) = {} & (\mathtt{t} \mid \mathtt{in}\langle(m),p\rangle_{\downarrow} \triangleright \mathtt{a}[z] \triangleright \mathtt{collect}\langle c, \mathtt{to}\langle a,m,p,z\rangle\rangle) \\ & \mid (\mathtt{t} \mid \mathtt{out}\langle(m),p\rangle_{\downarrow} \triangleright \mathtt{a}[z] \triangleright \mathtt{collect}\langle c, \mathtt{up}\langle a,m,p,z\rangle\rangle)\end{aligned}$$

Process $\mathtt{T}$ is responsible for dealing with the $\mathtt{open}$ primitive, which is translated as a message on port $\mathtt{open}$, and with the $\mathtt{to}$ and $\mathtt{up}$ messages generated by the $\mathtt{S}$ process of some other ambients (siblings or subambients). For $\mathtt{open}$, the behavior is very simple: passivate the kell and ask the garbage collector to collect the enclosing, passing the content of the passivated kell as a continuation in the message to the garbage collector. For $\mathtt{to}$, the behavior is barely more complex: when receiving the message, passivate the local kell, ask the outside factory to create a new ambient with the required characteristics, and reactivate the kell, inserting the new kell the factory has returned. In the case of $\mathtt{up}$, the behavior is a bit more complex, since it requires the cooperation of the environment: forward the $\mathtt{up}$ message to the environment via an $\mathtt{upup}$ message; the $\mathtt{upup}$ message will be captured by the $\mathtt{Aux}$ process in the controlled part of the parent kell, that re-creates the ambient that has initiated the $\mathtt{out}$ and installs it as a sibling of the ambient that sent the $\mathtt{upup}$ message; upon receipt of the notification from the environment, unlock the ambient.

$$\begin{aligned}\mathtt{T}(a,c,\mathtt{t}) = {} & (\mathtt{t} \mid a[z] \mid \mathtt{open}\langle a,p\rangle^{\uparrow} \triangleright \mathtt{collect}\langle c,(z \mid p)\rangle) \\ & \mid \ (\mathtt{t} \mid \mathtt{to}\langle(n),a,p,x\rangle^{\uparrow} \mid a[z] \diamond \\ & \qquad (\nu k.\,\mathtt{make}\langle n,p \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y]))) \\ & \mid \ (\mathtt{t} \mid \mathtt{up}\langle(n),a,p,x\rangle_{\downarrow} \diamond \\ & \qquad (\nu k.\,\mathtt{upup}\langle n,p \mid x,k\rangle \mid (k^{\uparrow} \triangleright \mathtt{t}))) \\ \mathtt{Aux} = {} & \mathtt{upup}\langle(n),p,(k)\rangle_{\downarrow} \diamond (\nu h.\mathtt{make}\langle n,p,h\rangle \mid (h\langle y\rangle \triangleright k \mid y))\end{aligned}$$

With the protocol put in place above, we have captured the effects of the $\mathtt{in}$ and $\mathtt{out}$ primitives, by means of an asynchronous protocol. However, this protocol may fail because the target ambient is not present (in the case of $\mathtt{in}$), or because the enclosing ambient is not of the right name (in the case of $\mathtt{out}$). The process $\mathtt{F}$ below handles these two failure cases. It intercepts the $\mathtt{to}$ and $\mathtt{up}$ command messages and recreates the originating ambient in the exact state it was just prior to the beginning of the migration protocol.

$$\begin{array}{l}
\mathtt{F}(a,c,\mathtt{t}) = (\mathtt{t} \mid \mathtt{to}\langle(n),(m),p,x\rangle_{\downarrow} \diamond \\
\qquad\qquad \mathtt{a}[z] \triangleright \\
\qquad\qquad\quad (\nu k.\, \mathtt{make}\langle n, \mathtt{in}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y])) \\
\qquad\quad \mid (\mathtt{t} \mid \mathtt{up}\langle(n),(m),p,x\rangle_{\downarrow} \diamond \\
\qquad\qquad \mathtt{a}[z] \triangleright \\
\qquad\qquad\quad (\nu k.\, \mathtt{make}\langle n, \mathtt{out}\langle m,p\rangle \mid x,k\rangle \mid (k\langle y\rangle^{\uparrow} \triangleright \mathtt{t} \mid \mathtt{a}[z \mid y]))
\end{array}$$

A few comments on this encoding are in order. 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 $\mathtt{AmbEnv}$ is a helper process that characterizes the environment required by Mobile Ambients, and that provides garbage collection and factory facilities.

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

$$!P \to_{MA} !P \mid P$$

then we obtain the following (where $\equiv^c$ is the structural congruence of the Kell calculus augmented with the rule $\mathtt{AmbEnv} \mid \mathtt{AmbEnv} \equiv^c \mathtt{AmbEnv}$):

Proposition 4.4 *If $P \to_{MA} Q$, then we have $[\![P]\!] \to^* \equiv^c [\![Q']\!]$, with $Q' \equiv_{MA} Q$. Conversely, if $[\![P]\!] \to^* P'$, then there exists Q such that $P \to_{MA} Q$, and $P' \to^* \equiv^c [\![Q]\!]$.*

4.4 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. We also slightly extend the language of patterns : this provides for a straightforward encoding. One can avoid this extension by associating with each membrane in the translation the list of names of sub-locations, and by updating this list after each move. The router process $\mathtt{IR}$ below can be modified to check for the presence or absence of a particular name in the list (e.g. resorting to the encoding of names in Section 4.1 to implement the check), instead of relying on extended patterns. Extending the language of patterns makes for a simpler encoding and points at useful possible variants of the calculus.

The extension of the calculus is defined as follows. With a message pattern $a\langle \boldsymbol{w}\rangle^{\uparrow}$, $a\langle \boldsymbol{w}\rangle_{\downarrow}$, or $a\langle \boldsymbol{w}\rangle$, one can associate a predicate π of the following two forms $a \in \mathtt{C}$ and $a \notin \mathtt{C}$. Intuitively, the predicate $a \in \mathtt{C}$ indicates that an active kell of name a exists somewhere within the tree of subkells routed at the current kell. The join patterns are now defined by:

$$J ::= \epsilon \quad | \quad \mu \quad | \quad \mu :: \pi \quad | \quad J \mid J$$
$$\mu ::= a\langle \boldsymbol{w} \rangle \quad | \quad a\langle \boldsymbol{w} \rangle^{\uparrow} \quad | \quad a\langle \boldsymbol{w} \rangle$$

By convention, a pattern of the form μ is equivalent to a pattern of the form $\mu :: \top$, where $\top$ corresponds to the boolean `true`. The name a appearing in predicates $\pi = a \in \mathtt{C}$ or $\pi = a \notin \mathtt{C}$ is bound in $\mu :: \pi$, if there is an occurrence of (a) in μ, otherwise it is free. The definition of the application of a substitution θ to a join pattern J is modified by adding the clause $\mathtt{cj}(\mu :: \pi) = \mathtt{cj}(\mu)$ in the definition of function `cj` in Section 3.1. The reduction rules in Figure 5 are modified by introducing the condition $\mathtt{Cond}(J, R, \theta)$ in the premises of rules R.IN, R.OUT, IN.PASS, and OUT.PASS. Assume $J = \prod_{i \in I} \mu_i :: \pi_i$, then $\mathtt{Cond}(J, R)$ is defined as:

$$\mathtt{Cond}(J, R) \triangleq \bigwedge_{i \in I} \mathtt{H}(\pi_i \theta, R) \qquad \mathtt{H}(\top, R) \triangleq \top$$

$$\mathtt{H}(a \in \mathtt{C}, R) \triangleq (R \equiv \mathbf{E}\{a[P]\}) \qquad \mathtt{H}(a \notin \mathtt{C}, R) \triangleq \neg \mathtt{H}(a \in \mathtt{C}, R)$$

The encoding can now be defined as follows. 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 `m`, `mm`, `collect`, `query`, `make`, `va`, `enter` do not occur free in P, D:

$$[\![a]\!]_b = a \qquad [\![\mathbf{0}]\!]_b = \mathbf{0}$$

$$[\![\top]\!]_b = \mathbf{0} \qquad [\![P \mid Q]\!]_b = [\![P]\!]_b \mid [\![Q]\!]_b$$

$$[\![\mathtt{go}\, a; P]\!]_b = \mathtt{va}\langle a, [\![P]\!]_a \rangle \qquad [\![D, D']\!]_b = [\![D]\!]_b \mid [\![D']\!]_b$$

$$[\![a\langle n_1, \ldots, n_q \rangle]\!]_b = \mathtt{m}\langle b, a, n_1, \ldots, n_q \rangle \qquad [\![D \,\mathtt{in}\, P]\!]_b = \nu \boldsymbol{n}.\, [\![D]\!]_b \mid [\![P]\!]_b \qquad \boldsymbol{n} = \mathtt{df}(D)$$

$$[\![n_1 \boldsymbol{m_1} \mid \ldots \mid n_q \boldsymbol{m_q} \triangleright P]\!]_b = \mathtt{m}\langle b, n_1, (\boldsymbol{m_1}) \rangle \mid \ldots \mid \mathtt{m}\langle b, n_q, (\boldsymbol{m_q}) \rangle \diamond [\![P]\!]_b$$

$$[\![a[D : P]]\!]_b = \nu c.\, c[\mathtt{DJ}(a, c) \mid a[[\![D]\!]_a \mid [\![P]\!]_a]] \mid \mathtt{DJEnv}$$

together with the following auxiliary definitions:

$$
\begin{aligned}
\mathtt{DJ}(a, c) &= \nu \mathtt{t}.\, \mathtt{t} \mid \mathtt{IR}(a) \mid \mathtt{Go}(a, c, \mathtt{t}) \mid \mathtt{Enter}(a, \mathtt{t}) \\
\mathtt{IR}(a) &= \mathtt{m}\langle (b), x \rangle^{\uparrow} :: b \in \mathtt{C} \mid a[p] \diamond a[p \mid \mathtt{m}\langle b, x \rangle] \\
&\quad \mid \mathtt{m}\langle (b), x \rangle_{\downarrow} :: b \notin \mathtt{C} \diamond \mathtt{mm}\langle b, x \rangle \\
\mathtt{Go}(a, c, \mathtt{t}) &= \mathtt{t} \mid \mathtt{va}\langle (b), p \rangle_{\downarrow} :: b \notin \mathtt{C} \triangleright \\
&\qquad (a[q] \triangleright \mathtt{collect}\langle c, \mathtt{enter}\langle b, a, (q \mid p) \rangle \rangle) \\
\mathtt{Enter}(a, \mathtt{t}) &= \mathtt{t} \mid \mathtt{enter}\langle a, (b), x \rangle^{\uparrow} \mid a[p] \diamond \\
&\qquad \nu k.\mathtt{make}\langle b, x, k \rangle \triangleright (k\langle y \rangle^{\uparrow} \triangleright \mathtt{t} \mid a[p \mid y]) \\
\mathtt{DJEnv} &= \nu \mathtt{t}.(\mathtt{C}\,\mathtt{t}\,\mathtt{C}) \mid \mathtt{ER} \mid \mathtt{GC} \\
\mathtt{C} &= (\mathtt{t}\, x)\, \mathtt{Factory}(\mathtt{t}) \mid \mathtt{t}\langle x \rangle \\
\mathtt{Factory}(\mathtt{t}) &= \mathtt{t}\langle x \rangle \mid \mathtt{make}\langle (n), p, (k) \rangle_{\downarrow} \diamond \mathtt{MKJ}(x, n, p, k) \\
\mathtt{MKJ}(x, n, p, k) &= \mathtt{t}\langle x \rangle \mid k\langle \mathtt{DJE}(x) \mid c[\mathtt{DJ}(n, c) \mid n[p]] \rangle \\
\mathtt{DJE}(x) &= \nu a\, c.(x\, a\, x) \mid \mathtt{ER} \mid \mathtt{GC} \\
\mathtt{ER} &= \mathtt{mm}\langle x \rangle_{\downarrow} \diamond \mathtt{m}\langle x \rangle
\end{aligned}
$$

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 [1] . We obtain the following (where $\equiv^c$ is the structural congruence of the Kell calculus augmented with the rule $\mathtt{DJEnv} \mid \mathtt{DJEnv} \equiv^c \mathtt{DJEnv}$):

Proposition 4.5 *If $P \rightarrow_{DJ} Q$, then $[\![P]\!] \rightarrow^* \equiv^c [\![Q]\!]$. Conversely, if $[\![P]\!] \rightarrow^* P'$, then there exists Q such that $P \rightarrow_{DJ} Q$ and $P' \rightarrow^*_{DJ} \equiv^c Q$.*

5 Conclusion

We have introduced the Kell calculus, a new process calculus with hierarchical localities, strictly local actions, higher-order communication and locality passivation. Like the M-calculus, the Kell calculus allows an encoding of different forms of locality membranes, including localities with different forms of failures. The Kell calculus, however, appears simpler than the M-calculus, and does not rely on complex routing rules in contrast to the M-calculus.

We have shown by means of encodings of (a simplified form of) Klaim, of Mobile Ambients and of the Distributed Join calculus that the Kell calculus has considerable expressive power. The report [19] 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 $a[M(a) \mid [\![P]\!]] \mid \mathtt{Env}$ or $c[M(a,c) \mid c[[\![P]\!]]] \mid \mathtt{Env}$, 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 [11], Seal [6], and DiTyCo [13]. Obtaining such encodings would give strong evidence that the Kell calculus embodies very fundamental constructs for distributed programming.

The Kell calculus shares the local character of its actions with the Seal calculus [6]. Indeed, as in the Seal calculus, primitive actions in our calculus include local communications and communications across a single locality boundary. In contrast to Seal, however, our communications are higher-order, whereas Seal distinguishes between first-order communications on the one hand and migrating and replicating localities on the other hand. The choice in Seal to eschew higher-order communication was made primarily with a view to simplify its underlying theory. However, as the results in [6] reveal, the higher-order character of the migrate and replicate

[1] This is not the case of the encoding of the DJoin calculus in the M-calculus defined in [18], 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 [18].

primitives in Seal already poses some problems (e.g. with respect to a complete characterization of contextual equivalence). With the Kell calculus higher-order pirmitives, we gain the ability to handle directly passivated process states. This allows for instance a direct modelling of such failure behaviors as fail-stop with recovery, a behaviour which would be less straightforward to model in Seal (seals can be replicated and destroyed but they cannot be passivated and reactivated; it is possible to place Seals in opaque membranes to simulate passivation but this is not entirely satisfactory since one can allow observation of passivated states – e.g. in the form of checkpoints). Another perceived advantage of the higher-order character of the Kell calculus over Seal is the potential to extend the calculus with multi-stage programming along e.g. the lines of MetaKlaim [8].

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.

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] G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 2556 in LNCS. Springer, 2002.

[7] 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.

[8] G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A Type-Safe Multi-Stage Language for Global Computing. *to appear in Mathematical Structures in Computer Science*, 2003.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

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

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