The Kell calculus : operational semantics and type system

Philippe Bidinger and Jean-Bernard Stefani

INRIA Rhône-Alpes, 38334 St Ismier, France
{ philippe.bidinger, jean-bernard.stefani}@inrialpes.fr

Abstract. This paper[1] presents the Kell calculus, a new distributed process calculus that retains the original insights of the Seal calculus (local actions, process replication) and of the M-calculus (higher-order processes and programmable membranes), although in a much simpler setting than the latter. The calculus is equipped with a type system that enforces a unicity property for location names that is crucial for the efficient implementation of the calculus.

1 Introduction

Numerous distributed process calculi have been introduced in the past ten years. One of the calculi that has received the most attention has been Mobile Ambients [5], as witnessed by the numerous variants that have been proposed to overcome some of its perceived deficiencies: Safe Ambients (SA) [11], Safe Ambients with passwords [12], Boxed Ambients (BA) [3], Controlled Ambients (CA) [16], New Boxed Ambients (NBA) [4], Ambients with process migration ($\mathbf{M}^3$) [7].

Mobile Ambients are unfortunately costly to implement in a distributed setting (i.e. with ambients representing potentially widely separated sites), in particular because of the synchronization implied in its migration primitives. Consider the reduction rule associated with the *in* primitive of Mobile Ambients:

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

This rule mandates a rendez-vous between ambient n and ambient m. Thus, if ambient n and ambient m are taken to represent two remote sites, a faithful implementation of this rule would require some form of distributed synchronization.

The difficulty of implementing Mobile Ambients in a distributed setting and the need for two and even three-way-synchronization between ambients to implement Ambient migration primitives, has been made clear by two implementation attempts. The first one, reported in [9], implements the original Mobile Ambients calculus using (an implementation of) the Distributed Join calculus. The second one, reported in [13], describes a Safe Ambients abstract machine, called PAN, that alleviates some of the difficulty inherent in Mobile Ambients implementation by implementing a variant of the original calculus with co-capabilities and single-threadedness [11], but where ambients no longer correspond to physical loci of computations.

[1] This work has been supported in part by the Mikado project – IST-2001-32222

Recent variants of Ambients, such as Boxed Ambients (BA) and New Boxed Ambients (NBA) propose a model which combines local communication across location boundaries (inspired by the Seal calculus [17]), and the Ambient migration primitives `in` and `out`. In a model such as NBA, communication can be implemented efficiently while migration primitives still imply in general a distributed rendez-vous. This is much preferrable to the original Mobile Ambients, but still raises a number of questions.

First, one can think of turning the Ambient migration primitives into asynchronous ones. This would be useful to take into account the possibility of failure for migration, especially in wide-area settings. 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):

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

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

In so doing, note that migration primitives now look very much like higher-order communication across location boundaries. Second, one may envisage further extensions allowing more sophisticated authentication schemes, or dynamic security checks (e.g. additional parameters for proof-carrying code schemes). This in turn would further strengthen the similarity between migration primitives and higher-order communication. Third, there are still pending questions 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 [10] or Nomadic Pict [18], 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 ?

Our answer to these questions is to move away from the Ambient primitives altogether, and instead to follow the lead of higher-order process calculi such as $D\lambda\pi$ [19] and the M-calculus [14], 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 with respect to Mobile Ambients. The reduction relation of the calculus, which defines its operational semantics, contains several so-called routing rules that govern the crossing of location boundaries. Clearly it would be interesting to explain these different rules as instances of basic primitive "boundary crossing" cases.

The calculus we introduce in this paper is an attempt to define a calculus with process migration and hierarchical localities, 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 membranes for localities. 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).

Avoiding primitives with implied distributed synchronization is not the only requirement for an efficient implementation in a distributed setting. Today's wide-area communication is predicated upon the existence of globally unique identifiers and addresses (e.g. IP addresses). It is therefore important, in a calculus intended as a foundation for wide-area distributed programming, to be able to enforce such a constraint, both for modelling and implementation purposes. In the Kell calculus, the unicity of addresses translates into the unicity of locality (kell) names. We show in the paper how to enforce the constraint by means of a polymorphic type system, inspired by the type system defined for the M-calculus.

The Kell calculus and its reduction semantics has already been introduced in [15], together with faithful encodings of Mobile Ambients and of the Distributed Join calculus. We present in this paper a variant of the Kell calculus with join patterns which is a more natural fit for the type system, together with a new reduction semantics.

The 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 defines a type system for the Kell calculus that enforces the unicity of kell name property. Section 5 concludes the paper with a discussion of related work and of directions for future 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 variant we use in this paper.

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, $\mathbf{0}$; names a, x, which also play the roles of (name and process) variables;
- 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, $a\langle \widetilde{w} \rangle$, where a is a name, and where $\widetilde{w}$ is a (possibly empty) vector of elements w that can be either names or processes.
- *triggers*, or receivers, of the form $\xi \triangleright P$, where ξ is a *receipt pattern* and P is an arbitrary kell calculus process.

The patterns used in this paper correspond to an extension of the Join patterns, i.e. patterns of messages used in the Join calculus:

$$\xi ::= J \quad | \quad J \mid a[x] \qquad J ::= a\langle \widetilde{u} \rangle \quad | \quad a\langle \widetilde{u} \rangle^{\uparrow} \quad | \quad a\langle \widetilde{u} \rangle^{\downarrow} \quad | \quad J \mid J$$

where $\widetilde{u}$ is a vector of elements u. Each element u can be either a (bound) variable x, or a free name, which we note (x). Variables are bound in patterns and their scope extends to the process of the right-hand side of the trigger sign $\triangleright$. Free names (x) are not bound in the pattern.

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 four simple forms, illustrated below:

1. 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 $a\langle x\rangle \triangleright P$ (notice that triggers, as in the Join calculus, are replicated, i.e. they persist after a reaction):
$$a\langle Q\rangle \mid (a\langle x\rangle \triangleright P) \;\rightarrow\; (a\langle x\rangle \triangleright P) \mid P\{Q/x\}$$
2. 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 $a\langle x\rangle \triangleright P$, located in kell b (the pattern $a\langle x\rangle^{\uparrow}$ indicates that a message is expected from *outside* the local kell):
$$a\langle Q\rangle \mid b[a\langle x\rangle^{\uparrow} \triangleright P] \;\rightarrow\; b[(a\langle x\rangle^{\uparrow} \triangleright P) \mid P\{Q/x\}]$$
3. 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 $a\langle x\rangle \triangleright P$, located in the parent kell of kell b (the pattern $a\langle x\rangle^{\downarrow}$ indicates that a message is expected from a kell *inside* the local kell):
$$(a\langle x\rangle^{\downarrow} \triangleright P) \mid b[a\langle Q\rangle \mid R] \;\rightarrow\; (a\langle x\rangle^{\downarrow} \triangleright P) \mid P\{Q/x\} \mid b[R]$$
4. 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:
$$a[Q] \mid (a[x] \triangleright b\langle x\rangle) \;\rightarrow\; (a[x] \triangleright b\langle x\rangle) \mid b\langle Q\rangle$$

Actions of the form 1 above are standard π-calculus actions. Actions of the form 2 and 3 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 Boxed Ambients or in the Seal calculus [6].

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 form:

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

Actions of the form 4 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. They can be compared to the migrate and replicate construct of the Seal calculus, but note that they provide more control over the execution of processes. Consider for instance the processes P and R defined as:

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

We have the following reductions:

$$\begin{aligned}\texttt{resume}\langle a\rangle \mid \texttt{suspend}\langle a\rangle \mid P \mid R \mid a[Q] &\to \texttt{resume}\langle a\rangle \mid P \mid R \mid a\langle Q\rangle \\ &\to P \mid R \mid a[Q]\end{aligned}$$

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 higher-order nature of the calculus, together with the above control capability, allows the definition of different forms of programmable "membranes" around kells. For instance, a membrane around $a[K]$ can take the form: $c[M(a) \mid a[K]]$, in which case its behavior is defined by the process $M(a)$. Here are some simple examples of membranes (we assume that all messages *to* kell a have the form $\texttt{rcv}\langle a, op, args\rangle$ and that all messages *from* kell a have the form $\texttt{snd}\langle b, op, args\rangle$):

Transparent membrane This is a membrane that does nothing (it just allows messages destined to, or emitted by, a to be transmitted without any control):

$$M_{trans} \triangleq (\texttt{rcv}\langle(a), b, x\rangle^{\uparrow} \triangleright \texttt{rcv}\langle a, b, x\rangle) \mid (\texttt{snd}\langle b, c, x\rangle^{\downarrow} \triangleright \texttt{snd}\langle b, c, x\rangle)$$

Intercepting membrane This is a membrane 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 $\texttt{m}\langle b, y\rangle$ seeks to leave kell a. Notice how this allows the definition of wrappers with pre and post-handling of messages:

$$M_{int} \triangleq (\texttt{rcv}\langle(a), b, x\rangle^{\uparrow} \triangleright \texttt{Pre}\langle b, x\rangle) \mid (\texttt{snd}\langle b, c, x\rangle^{\downarrow} \triangleright \texttt{Post}\langle b, c, x\rangle)$$

Migration membrane This is a membrane 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 asynchronous Ambient migration primitives `enter` and `move` given in Section 1, and the `go` primitive of the Distributed Join calculus:

$$\begin{aligned}M_{mig} \triangleq M_{trans} \;\mid\; & (\texttt{rcv}\langle(a), (\texttt{enter}), x\rangle^{\uparrow} \triangleright (a[y] \triangleright a[x \mid y])) \\ \mid\; & (\texttt{go}\langle b\rangle^{\downarrow} \triangleright (a[y] \triangleright \texttt{snd}\langle b, \texttt{enter}, a[y]\rangle))\end{aligned}$$

Membrane with fail-stop failures This is a membrane 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, model of failures:

$$\begin{aligned}M_{fails} \triangleq \nu c\, f.M_{trans} \mid c \;\mid\; & (\texttt{stop}\langle(a)\rangle^{\uparrow} \mid c \triangleright (a[y] \triangleright f)) \\ \mid\; & (\texttt{rcv}\langle(a), (\texttt{ping}), r\rangle^{\uparrow} \mid c \triangleright \texttt{snd}\langle r, \texttt{up}, a\rangle) \\ \mid\; & (\texttt{rcv}\langle(a), (\texttt{ping}), r\rangle^{\uparrow} \mid f \triangleright \texttt{snd}\langle r, \texttt{down}, a\rangle)\end{aligned}$$

Membrane with fail-stop failures and recovery This is a membrane that extends the previous one with the possibility of recovery:

$$\begin{aligned}M_{failr} \triangleq \nu c\, f.M_{trans} \mid c \;\mid\; & (\texttt{stop}\langle(a)\rangle^{\uparrow} \mid c \triangleright (a[y] \triangleright f\langle y\rangle)) \\ \mid\; & (\texttt{rcv}\langle(a), (\texttt{ping}), r\rangle^{\uparrow} \mid c \triangleright \texttt{snd}\langle r, \texttt{up}, a\rangle) \\ \mid\; & (\texttt{rcv}\langle(a), (\texttt{ping}), r\rangle^{\uparrow} \mid f\langle y\rangle \triangleright \texttt{snd}\langle r, \texttt{down}, a\rangle) \\ \mid\; & (\texttt{rcv}\langle(a), (\texttt{recover}), r\rangle^{\uparrow} \mid f\langle y\rangle \triangleright a[y] \mid c \mid \texttt{snd}\langle r, \texttt{rcvd}, a\rangle)\end{aligned}$$

3 The Kell calculus: syntax and operational semantics

3.1 Syntax

The syntax of the Kell calculus, together with the syntax of evaluation contexts, is given below:

$$
\begin{aligned}
P &::= \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid P \mid P \mid a[P] \mid a\langle \widetilde{P} \rangle \\
\xi &::= \bot \mid J \mid J \mid a[x] \\
J &::= a\langle \widetilde{u} \rangle^{*} \mid J \mid J \\
u &::= x \mid (x) \\
* &::= - \mid \uparrow \mid \downarrow \\
\mathbf{E} &::= \cdot \mid \nu a.\mathbf{E} \mid a[\mathbf{E}] \mid P \mid \mathbf{E}
\end{aligned}
$$

Filling the hole $\cdot$ in an evaluation context $\mathbf{E}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{E}\{Q\}$.

We assume an infinite set N of *names*. We let a, b, x, y and their decorated variants range over N. Note that names in the kell calculus act both as name constants and as (name or process) variables. We use $\widetilde{V}$ to denote finite vectors $(V_1, \ldots, V_q)$. Abusing the notation, we equate $\widetilde{V} = (V_1, \ldots, V_n)$ with the word $V_1 \ldots V_n$ and the set $\{V_1, \ldots, V_n\}$. We note $|\widetilde{V}|$ the length n of a vector $\widetilde{V} = (V_1, \ldots, V_n)$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q, R, S, T and their decorated variants range over processes. We call *message* a process of the form $a\langle \widetilde{P} \rangle$. We let M, N and their decorated variants range over messages and parallel composition of messages. We abbreviate a a message of the form $a\langle\rangle$ (i.e. a message with an empty vector of arguments). We call *kell* a process of the form $a[P]$. The name a in a kell $a[P]$ is called the name of the kell. In a kell of the form $a[\ldots \mid a_j[P_j] \mid \ldots \mid Q_k \mid \ldots]$ we call *subkells* the processes $a_j[P_j]$. 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. $J = \bot$). The empty join pattern, $\bot$, cannot match any message. We note $a\langle \widetilde{u} \rangle$ for $a\langle \widetilde{u} \rangle^{-}$.

In a term $\nu a.P$, the scope extends as far to the right as possible. In a term $\xi \triangleright P$, the scope of $\triangleright$ extends as far to the left and to the right as possible. Thus, $a\langle c \rangle \mid b[y] \triangleright P \mid Q$ stands for $(a\langle c \rangle \mid b[y]) \triangleright (P \mid Q)$. 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 empty, then $\nu\widetilde{a}.P \triangleq P$. We also note $\prod_{i \in I} P_i$, $I = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $I = \emptyset$, then $\prod_{i \in I} P_i \triangleq \mathbf{0}$.

A pattern ξ acts as a binder in the calculus. All names x that do not occur within parenthesis () in a pattern ξ are bound by the pattern. We call *pattern variables* (or *variables*, for short) such bound names in a pattern. Variables occurring in a pattern are supposed to be linear, i.e. there is only one occurrence of each variable in a given pattern. Names occurring in a pattern ξ under parenthesis (i.e. occurrences of the form (x) in ξ) are *not* bound in the pattern. We call free pattern names (or free names, for short), names occurring under $()$ in a pattern. The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Free names ($\mathtt{fn}$), receiver names ($\mathtt{rn}$), bound pattern variables ($\mathtt{bn}$) and free pattern names ($\mathtt{mn}$) are defined below:

$$\mathtt{fn}(\mathbf{0}) = \emptyset \qquad \mathtt{fn}(x) = \{x\}$$
$$\mathtt{fn}(\nu x.P) = \mathtt{fn}(P) \setminus \{x\} \qquad \mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q)$$
$$\mathtt{fn}(x[P]) = \mathtt{fn}(P) \cup \{x\} \qquad \mathtt{fn}(J) = \mathtt{rn}(J) \cup \mathtt{mn}(J)$$
$$\mathtt{fn}(J \triangleright P) = (\mathtt{fn}(P) \setminus \mathtt{bn}(J)) \cup \mathtt{fn}(J)$$
$$\mathtt{fn}(x\langle P_1, \ldots, P_n\rangle) = \mathtt{fn}(P_1) \cup \ldots \cup \mathtt{fn}(P_n) \cup \{x\}$$
$$\mathtt{fn}(J \mid y[x] \triangleright P) = (\mathtt{fn}(P) \setminus \mathtt{bn}(J) \setminus \{x\}) \cup \{y\} \cup \mathtt{fn}(J)$$
$$\mathtt{rn}(a\langle\widetilde{u}\rangle) = a \qquad \mathtt{rn}(J \mid J') = \mathtt{rn}(J) \cup \mathtt{rn}(J')$$
$$\mathtt{bn}(a\langle\widetilde{u}\rangle) = \mathsf{N} \cap \widetilde{u} \qquad \mathtt{bn}(J \mid J') = \mathtt{bn}(J) \cup \mathtt{bn}(J')$$
$$\mathtt{mn}(a\langle\widetilde{u}\rangle) = \{x \in \mathsf{N} \mid (x) \in \widetilde{u}\} \qquad \mathtt{mn}(J \mid J') = \mathtt{mn}(J) \cup \mathtt{mn}(J')$$

We call *substitution* a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{K}$ from names to names and Kell calculus processes that is the identity except on a finite set of names. 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. $\mathtt{supp}(\theta) = \{i \in \mathsf{N} \mid \theta(i) \neq i\}$).

Let J be a join pattern, and θ be a substitution such that $\mathtt{bn}(J) \subseteq \mathtt{supp}(\theta)$. 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:

$$\mathtt{cj}(a) = a \qquad \mathtt{cj}((a)) = a \qquad \mathtt{cj}(\perp) = \mathbf{0}$$
$$\mathtt{cj}(a\langle\widetilde{w}\rangle) = a\langle\widetilde{\mathtt{cj}(w)}\rangle \qquad \mathtt{cj}(a\langle\widetilde{w}\rangle^{\downarrow}) = a\langle\widetilde{\mathtt{cj}(w)}\rangle$$
$$\mathtt{cj}(a\langle\widetilde{w}\rangle^{\uparrow}) = a\langle\widetilde{\mathtt{cj}(w)}\rangle \qquad \mathtt{cj}(J \mid J') = \mathtt{cj}(J) \mid \mathtt{cj}(J')$$

We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

Formally, with the syntax presented, the reduction rules in section 3.2 could yield terms of the form $P[Q]$, which are not legal Kell calculus terms (i.e. the syntax does not distinguish between names playing the role of name variables, and names playing the role of process variables). The type system presented in Section 4 rules out such illegal terms.

3.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [2], via a structural equivalence relation and a reduction relation. The structural equivalence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 1 and that makes the parallel operator $\mid$ associative and commutative, with $\mathbf{0}$ as a neutral element.

Notice that we do not have structural equivalence rules that deal with scope extrusion beyond a kell boundary (i.e we do not have the Mobile Ambient rule $a[\nu b.P] \equiv \nu b.a[P]$, provided $b \neq a$). As in the Seal calculus, this is to avoid phenomena as illustrated below:

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

$$\nu a.\mathbf{0} \equiv \mathbf{0} \ [\text{S.NU.NIL}] \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P \ [\text{S.NU.COMM}]$$

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \ [\text{S.NU.PAR}] \qquad \frac{P =_\alpha Q}{P \equiv Q} \ [\text{S.}\alpha] \qquad \frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}} \ [\text{S.CONTEXT}]$$

Fig. 1. Structural equivalence

$$\frac{J = J_1 \mid J_2 \qquad J_1 = \prod_{j \in J} a_j \langle \widetilde{w}_j \rangle^{\uparrow} \neq \bot \qquad \mathtt{fn}(J_1\theta) \cap \widetilde{c} = \emptyset}{J_1\theta \mid b[\nu\widetilde{c}.R \mid J_2\theta \mid (J \triangleright Q)] \to b[\nu\widetilde{c}.R \mid Q\theta \mid (J \triangleright Q)]} \ [\text{R.IN}]$$

$$\frac{\begin{array}{c}\widetilde{d} = \widetilde{c} \setminus \widetilde{e} \qquad \widetilde{e} = \widetilde{c} \cap \mathtt{fn}(J_1\theta) \\ \widetilde{e} \cap \mathtt{fn}(J \triangleright Q, J_2\theta, b) = \emptyset \qquad J = J_1 \mid J_2 \qquad J_1 = \prod_{j \in J} a_j \langle \widetilde{w}_j \rangle^{\downarrow} \neq \bot\end{array}}{J_2\theta \mid (J \triangleright Q) \mid b[\nu\widetilde{c}.R \mid J_1\theta] \to \nu\widetilde{e}.Q\theta \mid (J \triangleright Q) \mid b[\nu\widetilde{d}.R]} \ [\text{R.OUT}]$$

$$J\theta \mid a[P] \mid (J \mid a[x] \triangleright Q) \to Q\theta\{P/x\} \mid (J \mid a[x] \triangleright Q) \ [\text{R.PASS}]$$

$$\frac{J \neq \bot}{J\theta \mid (J \triangleright Q) \to Q\theta \mid (J \triangleright Q)} \ [\text{R.LOCAL}] \qquad \frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}} \ [\text{R.CONTEXT}]$$

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

Fig. 2. Reduction Relation

The reduction relation $\to$ is the smallest binary relation on K^2 that satisfies the rules given in Figure 2, where we assume that $\mathtt{bn}(J) = \mathtt{supp}(\theta)$. 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 equivalence. 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. Rule R.PASS allows the passivation of a subkell, possibly upon receipt of messages from the local environment. In rules R.IN and R.OUT, note that the join pattern J_2 may be empty. Likewise, in rule R.PASS, the join pattern J may be empty.

4 Type system

As pointed out in the introduction, the unicity of kell names is an important property to enforce in order to ensure an efficient implementation of the calculus. For instance, a kell a modelling a computing site interconnected via a wide-area network such as the Internet would have triggers of the form $\mathtt{rcv}\langle(a),(b),\widetilde{x}\rangle \mid \ldots \triangleright P$ with a corresponding e.g. to a wide-area network address. In this setting, the name a must be unique, at least within the context of the enclosing environment (which models the behavior of the network). Enforcing the unicity of kell names, however, is difficult in presence of higher-order communication and kell passivation. For instance, assume that a trigger $\mathtt{twice}\langle x\rangle \triangleright x \mid x$ is defined. Then a trigger of the form $a[x] \triangleright \mathtt{twice}\langle a[x]\rangle$ would lead to the illicit duplication of kell a.

We present in this section a type system for the kell calculus that enforces the unicity of kell names. More precisely, the type system enforces the unicity of *active* kells. A kell $a[Q]$ is said to be active in P (and P is said to contain the active kell a) if $P = \mathbf{E}\{a[Q]\}$ and $a[Q]$ is not under a scope restriction for a. The general idea, borrowed from the M-calculus type system, is to define the type of a process P as a multiset Δ that represents an upper bound on the multiset of names of kells that may be or may become active in P. Intuitively, a process will therefore be well-typed if its type Δ ends up being a set.

The syntax of types is given below:

$$
\begin{aligned}
\sigma &::= \mathtt{kell}(w)_{\Delta\to\Delta'} \ \mid\ \langle\widetilde{\sigma}\rangle_{\Delta} \ \mid\ \langle\widetilde{\sigma}\rangle^{+}_{\Delta} \ \mid\ \Delta \\
\Delta &::= \emptyset \ \mid\ \rho \ \mid\ \delta \ \mid\ a \ \mid\ \Delta,\, \Delta \\
w &::= a \ \mid\ \delta \ \mid\ \emptyset \\
s &::= \forall\widetilde{\rho}\widetilde{\delta}.\sigma
\end{aligned}
$$

A type σ can be a *process type* Δ, a *kell name type* $\mathtt{kell}(w)_{\Delta\to\Delta'}$, a *channel type* $\langle\widetilde{\sigma}\rangle_{\Delta}$, or a *sendable channel type* $\langle\widetilde{\sigma}\rangle^{+}_{\Delta}$. A channel of type $\langle\widetilde{\sigma}\rangle_{\Delta}$ can receive messages with arguments of types $\widetilde{\sigma}$, and the receipt of messages on this channel leads to the creation of kells with names in Δ. A sendable channel type types a receiver variable that can be instantiated to a just received name. We note $\langle\widetilde{\sigma}\rangle^{(+)}_{\Delta}$ to denote a type that can be a channel type $\langle\widetilde{\sigma}\rangle_{\Delta}$ or a sendable channel type $\langle\widetilde{\sigma}\rangle^{+}_{\Delta}$.

An active kell name a, which hosts subkells whose names are in Δ, and whose passivation leads to the creation of kells whose names are in Δ', has type $\mathtt{kell}(a)_{\Delta\to\Delta'}$. This is because a kell name can be used both as the name of an active kell and as the name of a special channel used to passivate the kell of the same name (via rule R.PASS). Since kell names are also variables, one must allow name type variables δ as argument of kell name types. No kell name may have type $\mathtt{kell}(\emptyset)_{\Delta\to\Delta'}$ (these types are introduced for technical reasons).

We use $\forall\widetilde{\rho}\widetilde{\delta}.\sigma$ to denote a type scheme in which name type variables $\widetilde{\delta}$ and multiset variables $\widetilde{\rho}$ are generalized. To define the type system, we consider an extended syntax for the calculus where new names are annotated with their type scheme. Thus we write $\nu a : s.P$ instead of $\nu a.P$, where s is a type scheme. The notion of free names is modified to take into account the new syntax: $\mathtt{fn}(\nu y : s.P) = \mathtt{fn}(P, s) \setminus \{y\}$. The structural congruence rules S.NU.NIL, S.NU.COMM and S.NU.PAR are modified thus:

S.NU.NIL	$\nu a : s.\mathbf{0} \equiv \mathbf{0}$
S.NU.COMM	$\nu a : s.\nu b : s'.P \equiv \nu b : s'.\nu a : s.P$ if $a \notin \mathtt{fn}(s')$ and $b \notin \mathtt{fn}(s)$
S.NU.PAR	$\nu a : s.P \mid Q \equiv (\nu a : s.P) \mid Q$ if $a \notin \mathtt{fn}(Q)$

Multisets Δ can include names a, name type variables δ, and multiset variables ρ. We use several operations on multisets. Relation $\subseteq$ is the standard multiset inclusion.

Δ, Δ' is the union of multisets Δ and Δ'. Multiset $\Delta \setminus a$ is the multiset Δ minus a single occurrence of name a. $\Delta \sqcup \Delta'$ is the smallest multiset (in terms of inclusion) that contains both Δ and Δ'.

We define the subtyping relation $\leq$ (where $\widetilde{\sigma}$ and $\widetilde{\sigma'}$ are vectors of the same length n), which is the smallest reflexive relation obeying the rules below:

$$\begin{aligned}
\Delta \leq \Delta' &\Leftarrow \Delta \subseteq \Delta' \\
\langle\widetilde{\sigma}\rangle_{\Delta} \leq \langle\widetilde{\sigma'}\rangle_{\Delta'} &\Leftarrow \forall i \in \{1, \ldots, n\}, \sigma'_i \leq \sigma_i \ \wedge\ \Delta \subseteq \Delta' \\
\langle\widetilde{\sigma}\rangle^{+}_{\Delta} \leq \langle\widetilde{\sigma'}\rangle_{\Delta'} &\Leftarrow \forall i \in \{1, \ldots, n\}, \sigma'_i \leq \sigma_i \ \wedge\ \Delta \subseteq \Delta' \\
\mathtt{kell}(w_1)_{\Delta_1 \to \Delta_2} \leq \mathtt{kell}(w_2)_{\Delta'_1 \to \Delta'_2} &\Leftarrow w_1 = w_2 \ \wedge\ \Delta'_1 \subseteq \Delta_1 \ \wedge\ \Delta_2 \subseteq \Delta'_2
\end{aligned}$$

The intuition behind the subtyping relation is that it is safe (with respect to the unicity of kell names) to replace a process with a process that contains fewer active kells.

We use Γ and its decorated variants to denote type environments, i.e. finite mappings between names and type schemes. We define the set of free names and of free type variables below:

$\mathtt{fn}(\emptyset) = \emptyset$	$\mathtt{fv}(\emptyset) = \emptyset$
$\mathtt{fn}(\rho) = \emptyset$	$\mathtt{fv}(\rho) = \{\rho\}$
$\mathtt{fn}(\delta) = \emptyset$	$\mathtt{fv}(\delta) = \{\delta\}$
$\mathtt{fn}(a) = a$	$\mathtt{fv}(a) = \emptyset$
$\mathtt{fn}(\Delta, \Delta') = \mathtt{fn}(\Delta) \cup \mathtt{fn}(\Delta')$	$\mathtt{fv}(\Delta, \Delta') = \mathtt{fv}(\Delta) \cup \mathtt{fv}(\Delta')$
$\mathtt{fn}(\mathtt{kell}(w)_{\Delta \to \Delta'}) = \mathtt{fn}(w) \cup \mathtt{fn}(\Delta, \Delta')$	$\mathtt{fv}(\mathtt{kell}(w)_{\Delta \to \Delta'}) = \mathtt{fv}(w) \cup \mathtt{fv}(\Delta, \Delta')$
$\mathtt{fn}(\langle\widetilde{\sigma}\rangle^{(+)}_{\Delta}) = \mathtt{fn}(\sigma_1) \cup \ldots \cup \mathtt{fn}(\sigma_n) \cup \mathtt{fn}(\Delta)$	$\mathtt{fv}(\langle\widetilde{\sigma}\rangle^{(+)}_{\Delta}) = \mathtt{fv}(\sigma_1) \cup \ldots \cup \mathtt{fv}(\sigma_n) \cup \mathtt{fv}(\Delta)$
$\mathtt{fn}(\forall\widetilde{\beta}.\sigma) = \mathtt{fn}(\sigma)$	$\mathtt{fv}(\forall\widetilde{\beta}\sigma) = \mathtt{fv}(\sigma) \setminus \widetilde{\beta}$
$\mathtt{fn}(\Gamma) = \cup_{x \in dom(\Gamma)} \mathtt{fn}(\Gamma(x))$	$\mathtt{fv}(\Gamma) = \cup_{x \in dom(\Gamma)} \mathtt{fv}(\Gamma(x))$

Type judgments take the following form: $\Gamma \vdash P : \sigma$, where Γ is an environment, P is a process, and σ is a type. The type system is defined by the rules in Figure 3. They make use of the `Inst` operator, that takes a type scheme and returns a type where the generalized name type variables and multiset variables have been instantiated to names and multisets, respectively. A type environment Γ is said to a be *good* if $\mathtt{fn}(\Gamma) = \{x \in dom(\Gamma) \mid \Gamma(x) = \forall\widetilde{\beta}.\mathtt{kell}(x)_{\Delta \to \Delta'}\}$.

The typing rules use auxiliary functions which we now define. To deal with sendable receivers, we introduce a partition of the set of names, N: we define a set V such that $\mathsf{V} \subseteq \mathsf{N}$. If $a \in \mathsf{V}$, then a must be of type sendable. This is formalized in the definition of predicate `Pred` below. Assume that $(a_i, \forall\widetilde{\beta_i}.\langle\sigma_i^{1..m_i}\rangle_{\Delta_i}) \in \Gamma$ and $J = a_1\langle u_{1j}^{1..m_1}\rangle \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle$. We define the function `Extract` by:

$$\mathtt{Extract}(\Gamma, J) = \{x_{ij} : \sigma_{ij} \mid x_{ij} \in \mathtt{bn}(J)\}$$

We define the predicate $\mathtt{Pred}$ by:

$$\begin{aligned}
\mathtt{Pred}(\Gamma, J) = {} & \forall i, i' \in \{1..n\}.\forall j \in \{1..m_i\}.\forall j' \in \{1..m_{i'}\} \\
& (x_{ij}, x_{i'j'} \in \mathtt{mn}(J) \wedge (x_{ij} = x_{i'j'})) \implies \sigma_{ij} = \sigma_{i'j'} \\
& \wedge\ \forall i \in \{1..n\}.\forall j \in \{1..m_i\}.x_{ij} \in \mathtt{mn}(J) \implies (\sigma_{ij} \neq \Delta) \\
& \wedge\ \forall i \in \{1..n\}.fv(\Gamma) \cap \widetilde{\beta}_i = \emptyset \\
& \wedge\ \forall i, j \in \{1..n\}.i \neq j \implies \widetilde{\beta}_i \cap \widetilde{\beta}_j = \emptyset \\
& \wedge\ \forall x \in \mathtt{bn}(J), x \in \mathsf{V} \\
& \wedge\ a_i \in \mathsf{V} \implies (a_i, \langle \sigma_i^{1..m_i} \rangle^+_{\Delta_i}) \in \Gamma
\end{aligned}$$

Assume now that $(a, \forall \widetilde{\beta}.\mathtt{kell}(w)_{\Delta \to \Delta_0}) \in \Gamma$. We define the predicate $\mathtt{Pred}'$ by:

$$\mathtt{Pred}'(\Gamma, J, a) = \mathtt{Pred}(\Gamma, J)\ \wedge\ (fv(\Gamma) \cap \widetilde{\beta} = \emptyset)\ \wedge\ (a \notin \mathsf{V})$$

Some comments on these typing rules are in order. In rule T.MSG, the type of channel a is in fact a type scheme. The condition $\Gamma \vdash a : \langle \widetilde{\sigma} \rangle_\Delta$ provides an instance of this type scheme. Rule T.MSG requires arguments P_i of a message on channel a to have types which are subtypes of the expected argument types. Because of rules T.TRIG.MSG and T.TRIG.PASS, a trigger always has type $\emptyset$ (a trigger does not exhibit any active kell). The premises in both rules deal with the two sorts of names in a join pattern (variables and free names). In the case of a free name a (which occurs as (a) in the pattern), one must ensure that it is identically typed in all of its occurrences (first clause in the definition of $\mathtt{Pred}$), and that its type is not a process type (condition $\sigma_{ij} \neq \Delta$ in the second clause of the definition of $\mathtt{Pred}$). Note that the part of the premises that deals with free names is actually similar to the premises in rule T.MSG, since T.MSG only deals with free names and processes. Typing rules T.TRIG.MSG and T.TRIG.PASS may seem complex but they are in fact very close to the typing rule for Join calculus definitions: the guarded process is typed in an environment extended with formal parameters, and the result is checked to create fewer kells than advertised by the channel types. Every defined channel name that is a variable is checked to have a sendable channel type in the environment. The additional hypotheses check that the type schemes associated with channels (and the passivated kell name in T.TRIG.PASS) are consistent with the typing environment: no generalized variable may occur free in the environment, nor be shared by two channels (or a channel and the passivated kell name in T.TRIG.PASS).

The soundness of the type system is characterized by the following definitions and theorems, where a good type environment Γ is said to be *well-formed* if all pairs in Γ are of one of the following forms: $x : \langle \widetilde{\sigma} \rangle^+_\Delta$, $x : \forall \widetilde{\beta}.\langle \widetilde{\sigma} \rangle_\Delta$, or $x : \forall \widetilde{\beta}.\mathtt{kell}(x)_{\rho \to \Delta}$ with $\rho \notin \Delta - \rho$. A process P is said to have *failed* if $P = \mathbf{E}\{Q\}$, with Q containing two active localities bearing the same name. A process P is said to be *faulty* if $P \to^* Q$ with Q failed.

Theorem 1. *If $\Gamma \vdash P : \sigma$ with Γ well-formed, and $P \equiv Q$, then $\Gamma \vdash Q : \sigma$.*

Theorem 2 (Subject reduction). *If $\Gamma \vdash P : \sigma$ with Γ well-formed, and $P \to Q$, then there exists σ' such that $\sigma' \leq \sigma$ and $\Gamma \vdash Q : \sigma'$.*

Theorem 3 (Progress). *If $\Gamma \vdash P : \Delta$ with Γ well-formed, and Δ is a set containing only kell names, then the process P is not faulty.*

We now discuss some features and limitations of the type system. Note first that, because of the constraint $a \notin \mathsf{V}$ in the definition of predicate $\texttt{Pred}'$, an expression such as $a\langle y\rangle \triangleright (y[x] \triangleright P)$ is not typable. In other terms, one cannot instantiate a kell name with a received name. Like the type system of the M-calculus defined in [14] from which it is inspired, our type system simulates dependent types using polymorphism and name type variables (type variables that represent kell names), since kell names may occur in types. Consider now some simple examples. The process $(a\langle y\rangle \triangleright b[y]) \mid a\langle \mathbf{0}\rangle \mid a\langle \mathbf{0}\rangle$ is faulty, since in the environment $\Gamma = a : \forall\rho.\langle\rho\rangle_{b,\rho},\ b : \forall\rho'.dom(b)_{\rho'\to\emptyset}$, we get the type judgment $\Gamma \vdash (a\langle y\rangle \triangleright b[y]) \mid a\langle \mathbf{0}\rangle \mid a\langle \mathbf{0}\rangle : b, b$. This is an example of a process which is correctly flagged as faulty by our type system. As another example, the process $a\langle y\rangle \mid b\langle z\rangle \triangleright y \mid z$ is correct (non-faulty) and is indeed typable: with the environment $\Gamma = a : \forall\rho.\langle\rho\rangle_\rho,\ b : \forall\rho'.\langle\rho'\rangle_{\rho'}$, we get $\Gamma \vdash a\langle y\rangle \mid b\langle z\rangle \triangleright y \mid z : \emptyset$. On the other hand, consider the process $T = a\langle y\rangle \triangleright (\nu t : \langle\rangle_\emptyset.(t \mid b\langle z\rangle \triangleright y \mid z) \mid t)$. This process is correct since it does not duplicate kells it receives (it just instantiates a process received on a once). However this process is not typable with our type system. Indeed, the type system is too coarse in that respect since it deals with process T in the same way than with process $S = a\langle y\rangle \triangleright (b\langle z\rangle \triangleright y \mid z)$, which is indeed faulty since it may lead to an indefinite replication of active kell names received in the y argument. It is not clear how this limitation can be lifted.

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.

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 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].

The type system we introduced for the Kell calculus is directly inspired by the M-calculus type system [14]. Because the calculus is simpler, with less constructs, the resulting type system is also simpler. Whether the two type systems are comparable (notably with respect to the amount of correct processes they fail to type) is unclear, however. In particular, it is not clear whether a typed encoding of the M-calculus in the Kell calculus would yield a similar type system for the M-calculus as the original one.

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 a located process, is new. The examples provided in this paper, together with the encodings of Mobile Ambients and of the Distributed Join calculus given in [15], show that a single (higher-order) objective passivation 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 different forms of failures, including fail-stop failures with recovery, 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. The following issues seem worth considering:

- Developing a bisimulation theory for the Kell calculus. A characterization of contextual equivalence by means of a higher-order bisimulation seems highly nontrivial because of the passivation construct.
- 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 to the Kell calculus (in particular the ones dealing with resource and security constraints).
- 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. 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, 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.
10. 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.
11. 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.
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. 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.
14. 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.
15. J.B. Stefani. A Calculus of Kells. In *Proceedings 2nd International Workshop on Foundations of Global Computing*, 2003.
16. D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings CONCUR 02*, 2002.
17. J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA, Lecture Notes in Computer Science 1686, Springer*, 1998.
18. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.
19. N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

$$\frac{\Gamma \text{ good} \qquad (x, s = \forall\widetilde{\beta}.\sigma) \in \Gamma, \qquad \sigma\theta = \mathtt{Inst}(s) \qquad \mathtt{fn}(ran(\theta)) \subseteq \mathtt{fn}(\Gamma)}{\Gamma \vdash x : \sigma\theta} \text{ [T.VAR]}$$

$$\frac{\Gamma \text{ good}}{\Gamma \vdash \mathbf{0} : \emptyset} \text{ [T.NIL]} \qquad\qquad \frac{\Gamma \vdash P_1 : \Delta_1 \qquad \Gamma \vdash P_2 : \Delta_2}{\Gamma \vdash P_1 \mid P_2 : \Delta_1, \Delta_2} \text{ [T.PAR]}$$

$$\frac{\begin{array}{c} s = \forall\widetilde{\beta}.\langle\widetilde{\sigma}\rangle_\Delta \vee s = \langle\widetilde{\sigma}\rangle_\Delta^+ \\ fv(s) = \emptyset \qquad \Gamma,\ r : s \vdash P : \Delta' \qquad \mathtt{fn}(s) \subseteq \mathtt{fn}(\Gamma) \end{array}}{\Gamma \vdash \nu r : s.P : \Delta'} \text{ [T.CHAN.KELL]}$$

$$\frac{\begin{array}{c} s = \forall\widetilde{\beta}.\mathtt{kell}(a)_{\rho\to\Delta'} \qquad fv(s) = \emptyset \qquad \rho \notin \Delta' - \rho \\ \Gamma,\ a : s \vdash P : \Delta, \qquad a \notin \Delta - a \qquad \mathtt{fn}(\Delta') \subseteq \mathtt{fn}(\Gamma) \cup \{a\} \qquad a \notin \mathtt{fn}(\Gamma) \end{array}}{\Gamma \vdash \nu a : s.P : \Delta - a} \text{ [T.CHAN]}$$

$$\frac{\Gamma \vdash P : \Delta_0 \qquad \Gamma \vdash a : \mathtt{kell}(w)_{\Delta\to\Delta'} \qquad \Delta_0 \leq \Delta}{\Gamma \vdash a[P] : (w,\ \Delta_0) \sqcup \Delta'} \text{ [T.KELL]}$$

$$\frac{\Gamma \vdash a : \langle\widetilde{\sigma}\rangle_\Delta^{(+)} \qquad \Gamma \vdash P_i : \sigma_i' \qquad \sigma_i' \leq \sigma_i}{\Gamma \vdash a\langle\widetilde{P}\rangle : \Delta} \text{ [T.MSG]}$$

$$\frac{\begin{array}{c} J = a_1\langle u_{1j}^{1..m_1}\rangle^* \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle^* \\ \Gamma' = \mathtt{Extract}(\Gamma, J) \qquad \mathtt{Pred}(\Gamma, J) \\ (a_i, \forall\widetilde{\beta}_i.\langle\sigma_i^{1..m_i}\rangle_{\Delta_i}^{(+)}) \in \Gamma \qquad \Gamma,\ \Gamma' \vdash P : \Delta \qquad \Delta \leq \Delta_1, \ldots, \Delta_n \\ \Gamma \vdash a_i : \langle\tau_i^{1..m_i}\rangle_{\Delta_i'}^{(+)} \qquad \forall x_{ij} \in \mathrm{mn}(J).\Gamma \vdash x_{ij} : \tau_{ij}' \qquad \tau_{ij}' \leq \tau_{ij} \end{array}}{\Gamma \vdash J \triangleright P : \emptyset} \text{ [T.TRIG.MSG]}$$

$$\frac{\begin{array}{c} J = a_1\langle u_{1j}^{1..m_1}\rangle^* \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle^* \\ \Gamma' = \mathtt{Extract}(\Gamma, J) \qquad \mathtt{Pred}'(\Gamma, J, a) \qquad (a, \forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta\to\Delta_0}) \in \Gamma \\ (a_i, \forall\widetilde{\beta}_i.\langle\sigma_i^{1..m_i}\rangle_{\Delta_i}^{(+)}) \in \Gamma \qquad \Gamma,\ \Gamma', x : \Delta \vdash P : \Delta' \qquad \Delta' \leq \Delta_0, \ldots, \Delta_n \\ \Gamma \vdash a_i : \langle\tau_i^{1..m_i}\rangle_{\Delta_i'}^{(+)} \qquad \forall x_{ij} \in \mathrm{mn}(J).\Gamma \vdash x_{ij} : \tau_{ij}' \qquad \tau_{ij}' \leq \tau_{ij} \end{array}}{\Gamma \vdash J \mid a[x] \triangleright P : \emptyset} \text{ [T.TRIG.PASS]}$$

Fig. 3. Typing rules