An Abstract Machine for the Kell Calculus ⋆

Philippe Bidinger ⋆⋆, Alan Schmitt, and Jean-Bernard Stefani

INRIA Rhône-Alpes, 38334 St Ismier, France
firstname.lastname@inrialpes.fr

Abstract. The Kell Calculus is a family of process calculi intended as a basis for studying distributed component-based programming. This paper presents an abstract machine for an instance of this calculus, a proof of its correctness, and a prototype OCaml implementation. The main originality of our abstract machine is that it does not mandate a particular physical configuration (e.g. mapping of localities to physical sites), and it is independent of any supporting network services. This allows to separate the proof of correctness of the abstract machine per se, from the proof of correctness of higher-level communication and migration protocols which can be implemented on the machine.

1 Introduction

The Kell calculus [18,17] is a family of higher-order process calculi with hierarchical localities and locality passivation, which is indexed by the pattern language used in input constructs. It has been introduced to study programming models for wide-area distributed systems and component-based systems. A major assumption in a wide-area environment is the need for modular dynamicity, *i.e.* the ability to modify a running system by replacing some of its components, or by introducing new components (*e.g.* plug-ins). The Kell calculus can be seen as an attempt to understand the operational basis of *modular dynamicity*: localities in the Kell calculus model named components, and locality passivation provides the basis for dynamic reconfiguration operations.

Two of the main design principles for the calculus are to keep all the actions "local" in order to facilitate its distributed implementation, and to allow different forms of localities to coexist. A consequence of the locality principle is that the calculus allows different forms of networks to be modeled (by different processes). Thus, on the one hand, an implementation of the calculus should not need to consider atomic actions occurring across wide-area networks. On the other hand, an implementation of the calculus should not imply the use of purely asynchronous communications between localities: one can have legitimate implementations of the calculus that exploit and rely on the synchronous or quasi-synchronous properties of specific environments (e.g. a local machine with different processes, a high-performance, low-latency local area network for homogeneous PC clusters).

We present in this paper a distributed abstract machine for an instance of the Kell calculus, and its implementation in OCaml. The original feature of our abstract machine

⋆ This work has been supported by EU project MIKADO IST-2001-32222.
⋆⋆ Partly supported by EU IHP Marie Curie Training Site '**DisCo**: Semantic Foundations of Distributed Computation', contract HPMT-CT-2001-00290.

is that, in contrast to other works on abstract machines for distributed process calculi, it does not depend on a given network model, and can be used to implement the calculus in different physical configurations. Let us explain in more detail what this means. An implementation of our abstract machine typically comprises two distinct parts:

- An implementation of the abstract machine specification per se, that conforms to the reduction rules given in section 3 below.
- Libraries, in the chosen implementation language, that provides access to network services, and that conform to a Kell calculus model of these services (i.e. a Kell calculus process).

For instance, assume that one wants to realize a physical configuration consisting of a network N, that interconnects two computers m_1 and m_2, that each run an implementation of the Kell calculus abstract machine, and a Kell calculus program (P_1 and P_2, respectively). This configuration would be modelled in the Kell calculus by the process

$$C \triangleq N[\texttt{Net} \mid m_1[\texttt{NetLib} \mid P_1] \mid m_2[\texttt{NetLib} \mid P_2]]$$

where the process `Net` models the behavior of network N, and where the process `NetLib` models the presence, at each site, of a library providing access to the network services modeled by `Net`. From the point of view of the Kell calculus abstract machine, the library `NetLib` is just a standard Kell calculus process, but whose communications will have side-effects (i.e. accessing the actual network services modelled by `Net`) outside the abstract machine implementation.

The interesting aspect of our approach is the fact that we can thus provide implementations for different environments which all rely on the same abstract machine description and implementation. Consider for instance the physical configuration consisting of a network N, that interconnects two computers m_1 and m_2, that each run two separate processes, p_i^1 and p_i^2 ($i = 1, 2$). Each process p_i^j runs an implementation of the abstract machine, with a program Q_i^j. This configuration can be modelled by

$$C' \triangleq N[\texttt{Net} \mid M_1 \mid M_2]$$
$$M_1 \triangleq m_1[\texttt{NetOS} \mid \texttt{Ipc} \mid p_1^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^1] \mid p_1^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_1^2]]$$
$$M_2 \triangleq m_2[\texttt{NetOS} \mid \texttt{Ipc} \mid p_2^1[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^1] \mid p_2^2[\texttt{NetLib} \mid \texttt{IpcLib} \mid Q_2^2]]$$

where the process `NetOS` models the presence, at each site m_i, of some means (e.g. an operating system library) to access the network services modeled by `Net`, where the process `Ipc` models the presence, at each site, of a local communication library (e.g. an interprocess communication library provided by the local operating system), and where the processes `NetLib` and `IpcLib` model the presence, at each process p_i^j, of interfaces for accessing the different communication services provided, respectively, by the combination `Net` and `NetOS`, and by `Ipc`. Again, `NetLib` and `IpcLib` both appear as standard Kell calculus processes from the point of view of the abstract machine (i.e. they communicate with other processes by message exchange and can become passivated with their enclosing locality). However, the communication services they give access to can have very different semantics, if only in terms of reliability, latency, or security. The important point to note is that different communication services can coexist in the same implementation, and can be used selectively by application processes.

An important benefit of the independence of our abstract machine specification from any supporting network services, made possible by the local character of primitives in the Kell calculus, is the simplification of its proof of correctness. Indeed, the proof of correctness of our abstract machine does not involve the proof of a non-trivial distributed migration protocol, as is the case, for instance, with the JoCaml implementation of the distributed Join calculus [5], or with various abstract machines for ambient calculi [7,10,5,14] [1]. Furthermore, the correctness of the machine is ensured, regardless of the network services used for the actual implementation.

The abstract machine described in this paper constitutes a first step in a potential series of more and more refined abstract machines, getting us closer to a provably correct implementation of the calculus. Such a progressive approach aims at breaking up the proof of correctness of an abstract machine close to implementation into more tractable steps. For this reason, our abstract machine remains non-deterministic, and still has a number of high-level constructs such as variable substitution. Compared to the calculus, the abstract machine realizes three important functions: (1) it handles names and name restriction; (2) it "flattens" a Kell calculus process with nested localities into a configuration of non-nested localities with dependency pointers; (3) it makes explicit high-level process marshalling and unmarshalling functions which are involved in the implementation of the locality passivation construct of the Kell calculus.

The correctness of the abstract machine is stated, following [14], as barbed bisimilarity between a process of the calculus and its abstract machine interpretation. However, the results we obtain are in fact stronger than pure barbed bisimilarity as they involve some form of contextual equivalence. The results are stated using a strong form of bisimilarity, for we use a notion of sub-reduction to abstract away purely administrative reductions. Proofs can be found in the long version of this paper, available at [11].

The paper is organized as follows. Section 2 presents the instance of the Kell calculus we use in this paper. Section 3 specifies our abstract machine for the calculus. In Section 4 we give a correctness result for the abstract machine. In Section 5, we discuss an Ocaml prototype implementation of our abstract machine. In Section 6, we discuss related works. Section 7 concludes the paper with a discussion of future work.

2 The Kell calculus: syntax and operational semantics

2.1 Syntax

We now define the instance of the kell calculus we use in this paper. We allow five kinds of input patterns: *kell patterns*, that match a subkell, *local patterns*, that match a local message, *up patterns*, that match a message in the parent kell, and two kinds of *down patterns*, that match a message from a subkell. The syntax of the Kell calculus, together with the syntax of evaluation contexts, is given below:

[1] Note that the Channel Ambient abstract machine presented in [13] assumes that ambients may synchronize, for instance to run an `in` primitive. This assumption might be difficult to implement in an asynchronous distributed setting.

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

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 notation, we equate $\widetilde{V}$ with the word $V_1 \ldots V_n$ and the set $\{V_1, \ldots, V_n\}$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q, R and their decorated variants range over processes. We say that a process is in *normal form* when it does not contain any name restriction operator. We use P_*, Q_*, R_* and their decorated variants to denote these processes. We call *message* a process of the form $a\langle \widetilde{P} \rangle$. We call *kell* [2] a process of the form $a[P]$, with a 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 an *up pattern* $a\langle \widetilde{u} \rangle^{\uparrow}$, a *down pattern* $a\langle \widetilde{u} \rangle^{\downarrow^b}$ or $a\langle \widetilde{u} \rangle^{\downarrow}$, a *local pattern* $a\langle \widetilde{u} \rangle$, or a *control pattern* $a[x]$. A down pattern $a\langle \widetilde{u} \rangle^{\downarrow^b}$ matches a message on channel a coming from a subkell named b. A down pattern $a\langle \widetilde{u} \rangle^{\downarrow}$ matches a message on channel a coming from any subkell.

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 right as possible. Thus, $a\langle c \rangle \triangleright P \mid Q$ stands for $a\langle c \rangle \triangleright (P \mid Q)$. 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 them free pattern names (or free names, for short). We assumes that bound names of a pattern are disjoint from free names. The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Free names (fn), bound names (bn), free pattern variables (fpn), and bound pattern names (bpn) are defined as usual. We just point out the

[2] The work "kell" is intended to remind the word "cell", in a loose analogy with biological cells.

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

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

Fig. 1. Structural equivalence

handling of free pattern names:

$$\mathsf{fpn}(a\langle\widetilde{u}\rangle) = \{a\} \cup \{x \in \mathsf{N} \mid (x) \in \widetilde{u}\} \qquad \mathsf{bpn}(a\langle\widetilde{u}\rangle) = \{x \in \mathsf{N} \mid x \in \widetilde{u}\}$$

We call *substitution* a function $\phi : \mathsf{N} \to \mathsf{N} \uplus \mathsf{K}$ from names to names and Kell calculus processes that is the identity except on a finite set of names. We note supp the support of a substitution (*i.e.* $\mathsf{supp}(\phi) = \{i \in \mathsf{N} \mid \phi(i) \neq i\}$). We assume when writing $\xi\phi$ that $\mathsf{fpn}(\xi) \cap \mathsf{supp}(\phi) = \emptyset$ and that $\mathsf{supp}(\phi) \subseteq \mathsf{bpn}(\xi)$.

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

Formally, the reduction rules in section 2.2 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). However, a simple type system can be used to rule out such illegal terms.

2.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [1], 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.

The reduction relation $\to$ is the smallest binary relation on K that satisfies the rules given in Figure 2.

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 $b \neq a$). This is to avoid phenomena as illustrated below:

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

However, such name extrusion is still needed to allow communication across kell boundaries. The solution adopted here is to allow only scope *extrusion* across kell boundaries and to restrict passivation to processes without name restriction in evaluation context. Formally, this is achieved by requiring a process to be in normal form (P_*) in rule R.PASS and by adding a scope extrusion sub-reduction relation $\overset{\equiv}{\to}$.

Rules R.IN and R.OUT govern the crossing of kell boundaries. Only messages may cross a kell boundary. In rule R.IN, a trigger receives a message from the outside of the enclosing kell. In rule R.OUT, a trigger receives a message from a subkell.

$$\frac{a \neq b}{a[\nu b.P] \stackrel{\equiv}{\to} \nu b.a[P]}\ [\text{SR.NEW}] \qquad \frac{P \stackrel{\equiv}{\to} P'}{E[P] \stackrel{\equiv}{\to} E[P']}\ [\text{SR.CONTEXT}]$$

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

$$\frac{\widetilde{v} = \widetilde{u}\varphi}{c\langle\widetilde{v}\rangle \mid b[R \mid (c\langle\widetilde{u}\rangle^{\uparrow} \triangleright Q)] \to b[R \mid Q\varphi]}\ [\text{R.IN}] \qquad \frac{\widetilde{v} = \widetilde{u}\varphi}{c\langle\widetilde{v}\rangle \mid (c\langle\widetilde{u}\rangle \triangleright Q) \to Q\varphi}\ [\text{R.LOCAL}]$$

$$\frac{\widetilde{v} = \widetilde{u}\varphi \qquad \downarrow^{\bullet} = \downarrow^{b} \wedge \downarrow^{\bullet} = \downarrow}{b[c\langle\widetilde{v}\rangle \mid R] \mid (c\langle\widetilde{u}\rangle^{\downarrow^{\bullet}} \triangleright Q) \to b[R] \mid Q\varphi}\ [\text{R.OUT}]$$

$$\frac{}{a[P_*] \mid (a[x] \triangleright Q) \to Q\{P_*/x\}}\ [\text{R.PASS}] \qquad \frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}}\ [\text{R.CONTEXT}]$$

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

Fig. 2. Reduction Relation

3 Abstract Machine

3.1 Syntax

Following [14], our abstract machine is specified in the form of a process calculus whose terms, called *machine terms*, correspond to abstract machine states. Intuitively, a machine term consists in a set of localities, each executing a different program, organized in a tree by means of pointers between localities.

The syntax of the abstract machine calculus is given below:

$$\begin{array}{lll} M ::= \mathbf{0} \mid L \mid M \mid M & & M_* ::= \mathbf{0} \mid L_* \mid M_* \mid M_* \\ L ::= h : m[P]_{k,S} & & L_* ::= h : m[P_*]_{k,S} \\ S ::= \emptyset \mid h \mid S,\, S & & \end{array}$$

$$\begin{array}{l} P ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid \nu a.P \mid P \mid P \mid a[P] \mid a\langle\widetilde{P}\rangle \mid \mathbf{reify}(k, M_*) \\ P_* ::= \mathbf{0} \mid x \mid \xi \triangleright P \mid P_* \mid P_* \mid a\langle\widetilde{P}\rangle \\ \xi ::= a\langle\widetilde{u}\rangle \mid a\langle\widetilde{u}\rangle^{\downarrow} \mid a\langle\widetilde{u}\rangle^{\downarrow^{a}} \mid a\langle\widetilde{u}\rangle^{\uparrow} \mid a[x] \\ u ::= x \mid (x) \qquad x \in \mathsf{N} \qquad h, k, l \in \mathsf{MN} \qquad a, m \in \mathsf{N} \cup \mathsf{MN} \end{array}$$

Terms generated by the productions M, M_* in the abstract machine grammar are called *machine terms* (or *machines* for short, when no ambiguity arises), and are ranged over by M, N and their decorated variants. We designate their set by M. Machine terms make use of two sorts of names: the set N and a disjoint infinite set MN whose elements are called *machine names*. We call *locality* a machine term of the form $h : m[P]_{k,S}$. In a

$$\frac{M =_\alpha N}{M \equiv N}\ [\text{M.SE.}\alpha] \qquad \frac{P \equiv P' \qquad S \equiv S'}{l : h[P]_{k,S} \equiv l : h[P']_{k,S'}}\ [\text{M.SE.CTX}]$$

Fig. 3. Structural equivalence for machines

locality $h : m[P]_{k,S}$, m is the name of the kell the locality represents, h is the machine name of the locality, k is the machine name of its parent locality, S is the set of the machine names of its sublocalities, and P is the *machine process* being run at locality h. We use three particular machine names: $\mathbf{r}$, $\mathbf{rn}$ and $\mathbf{rp}$, which denote, respectively, the machine name of the root locality, the name of the root kell (associated with the root locality), and the machine name of the (virtual) root parent locality. Machine names appearing in a machine term are all unique (in contrast to kell names).

We call MK the set of machine processes (i.e. terms generated via the productions P, P_* in the abstract machine grammar), and we have $\mathsf{K} \subseteq \mathsf{MK}$. The machine processes are slightly different from Kell calculus processes. First a new term $\mathbf{reify}(k, M_*)$ is introduced to represent a passivated machine. The term M_* is a tree of machines encoded as a parallel composition of localities and k is the machine name of the root of this tree. Secondly, the names that can be used by a machine process belong to $\mathsf{N} \cup \mathsf{MN}$. This point will be made clear in the next subsection. A machine process is in normal form, written P_*, when it has no name restriction operator nor kells in evaluation context. A machine is in normal form when all machine processes in its localities are in normal form. We use the $._*$ suffix to denote machines and processes in normal form. The definitions and conventions given in section 2 extend to machine processes. Note that we use the same meta-variables to denote processes and machine processes. When it is not clear from the context, we will precise whether a variable denote a process or a machine process.

3.2 Reduction semantics

The reduction relation is defined as for the calculus via a structural congruence relation and a reduction relation.

First, we define two equivalence relations (both denoted by $\equiv$), on machine processes and sets of localities, respectively, as the smallest relations that make the parallel operator $|$ (resp. the $,$ operator) associative and commutative with $\mathbf{0}$ (resp. $\emptyset$) as a neutral element. Then, we define the structural congruence $\equiv$ on machines as the smallest equivalence relation that verifies the rules in figure 3 and that makes the parallel operator $|$ associative and commutative with $\mathbf{0}$ as a neutral element.

This structural equivalence, together with the rules M.S.CTX and M.S.STR, allows us to view machines as sets of localities and terms S as sets of machine names. Note that the equivalence relation on machine processes is different from the one on kell calculus processes as it does not contain rules dealing with restriction. This is because restriction is handled by the abstract machine as a name creation operator (rule M.S.NEW).

$$\frac{l\ fresh \in \mathsf{MN}}{h : n[(\nu a.P) \mid Q]_{k,S} \stackrel{\equiv}{\rightarrow} h : n[P\{l/a\} \mid Q]_{k,S}}\ [\text{M.S.NEW}]$$

$$\frac{h'\ fresh \in \mathsf{MN}}{h : n[m[P] \mid Q]_{k,S} \stackrel{\equiv}{\rightarrow} h : n[Q]_{k,(S,h')} \mid h' : m[P]_{h,\emptyset}}\ [\text{M.S.CELL}]$$

$$\frac{\begin{array}{c} M_* = l : n[R_*]_{l',S'} \mid M'_* \\ \mathbf{locnames}(M'_*) = \{l_i / i \in I\} \qquad k_i\ fresh \in \mathsf{MN},\ i \in I \end{array}}{h : m[\mathbf{reify}(l, M_*) \mid P]_{k,S} \stackrel{\equiv}{\rightarrow} h : m[R_* \mid P]_{k,(S,S'\{k_i/l_i\}_{i\in I}} \mid M'_*\{h/l\}\{k_i/l_i\}_{i\in I}}\ [\text{M.S.ACT}]$$

$$\frac{M \stackrel{\equiv}{\rightarrow} M'}{M \mid N \stackrel{\equiv}{\rightarrow} M' \mid N}\ [\text{M.S.CTX}] \qquad \frac{M \equiv M' \qquad M' \stackrel{\equiv}{\rightarrow} M'' \qquad M'' \equiv M'''}{M \stackrel{\equiv}{\rightarrow} M'''}\ [\text{M.S.STR}]$$

Fig. 4. Sub-reduction for machines

The reduction relation is defined as the smallest relation that satisfies the rules in Figures 4 and 5. It uses a subreduction relation $\stackrel{\equiv}{\rightarrow}$. The first subreduction rule, M.S.NEW, deals with restriction, which is interpreted as name creation. The reason the rule imposes the newly created name to be a machine name is related to the correctness proof, where we need to distinguish between restricted and free Kell names. Rule M.S.CELL creates a new locality when a kell is in the locality process. Rule M.S.ACT activates a passivated machine. Activation involves releasing the process held in the root locality of the passivated machine in the current locality, and releasing the sublocalities of the passivated machine as new sublocalities of the current locality.

The reduction rules M.IN, M.OUT, M.LOCAL, and M.PASS are the direct equivalent of the Kell calculus rules R.IN, R.OUT, R.LOCAL, and R.PASS, respectively. In rule M.PASS, the localities passivated are in normal form.

The reduction rules use the auxiliary function $\mathbf{locnames}$, the predicate $\mathbf{tree}$, and the notion of well-formed machine, which we now define.

The predicate $\mathbf{tree}(M, l, a, p)$ is defined as follows (where S may be empty):

$$\mathbf{tree}(M, l, a, p) = (M \equiv l : a[P]_{p,S} \mid \prod_{j\in S} M_j) \wedge_{j\in S} \mathbf{tree}(M_i, l_j, a_j, p_j))$$

with the additional condition that l, p, l_j, p_j are all distinct.

The function $\mathbf{locnames}(M)$ designates the set of locality names of all localities present in a machine M.

We say that a machine M is *well-formed* if we have $\mathbf{tree}(M, \mathbf{r}, \mathbf{rn}, \mathbf{rp})$. The set of well-formed machines is noted WFM. Finally, we will need the relation $\cong$ defined as follows: $M \cong N$ if and only if $\mathbf{tree}(M, l, m, p)$ and $M\sigma \equiv N\sigma'$ where σ and σ' are injective renaming of machine names such that $\sigma(l) = \sigma'(l) = l$ and $\sigma(p) = \sigma'(p) = p$ and if $m \in \mathsf{MN}$, $\sigma(m) = \sigma'(m) = m$.

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\ [\text{M.PAR}] \qquad \frac{\xi = c\langle \widetilde{u} \rangle}{\xi\varphi \mid (\xi \triangleright Q) \to Q\varphi}\ [\text{M.LOCAL}]$$

$$\frac{\xi = c\langle \widetilde{u} \rangle^{\uparrow}}{h : a[\xi\varphi \mid P]_{k,S} \mid h' : b[(\xi \triangleright Q) \mid R]_{h,S'} \mapsto h : a[P]_{k,S} \mid h' : b[Q\varphi \mid R]_{h,S'}}\ [\text{M.IN}]$$

$$\frac{\xi = c\langle \widetilde{u} \rangle^{\downarrow} \ \vee \ \xi = c\langle \widetilde{u} \rangle^{\downarrow^a}}{h : a[\xi\varphi \mid P]_{h',S} \mid h' : b[(\xi \triangleright Q) \mid R]_{k',S'} \mapsto h : a[P]_{h',S} \mid h' : b[Q\varphi \mid R]_{k',S'}}\ [\text{M.OUT}]$$

$$\frac{M_* = l : a[R_*]_{h,S'} \mid M'_* \qquad \mathbf{tree}(M_*, l, a, h)}{h : m[(a[x] \triangleright P) \mid Q]_{k,S} \mid M_* \mapsto h : m[P\{\mathbf{reify}(l, M_*)/x\} \mid Q]_{k,S \setminus l}}\ [\text{M.PASS}]$$

$$\frac{M \mapsto M'}{M \mid N \mapsto M' \mid N}\ [\text{M.CTX}] \qquad \frac{M \equiv M' \qquad M' \mapsto M'' \qquad M'' \equiv M'''}{M \mapsto M'''}\ [\text{M.STR}]$$

$$\frac{P \to P'}{h : m[P]_{k,S} \mapsto h : m[P']_{k,S}}\ [\text{M.RED}] \qquad \frac{M \stackrel{\equiv}{\to}^* M' \qquad M' \mapsto M''}{M \to M''}\ [\text{M.NORM}]$$

Fig. 5. Reduction for machines

4 Correctness

We establish the correctness of our abstract machine by establishing a strong bisimilarity result between Kell calculus processes and their interpretation by the abstract machine. The notion of equivalence we adopt is strong barbed bisimulation [15], which we denote by $\sim$. This notion of bisimulation can be used to compare different transition systems, provided that they are equipped with observability predicates and a reduction relation. An originality of our correctness result is that it relies on a strong form of barbed bisimilarity, instead of a weak one. This is possible because we abstract away administrative reduction rules through the subreduction relations in both the calculus and the abstract machine semantics. Our main result is the following:

Theorem 1 (Correctness). *For any Kell calculus process P, we have $[\![P]\!] \sim P$.*

This theorem asserts the equivalence of any Kell calculus process P with its translation in the abstract machine calculus. In the rest of this section we give the main definitions and intermediate results that intervene in the proof of Theorem 1.

We first define the translation of a Kell calculus process in the abstract machine calculus.

Definition 1. $[\![P]\!] = \mathbf{r} : \mathbf{rn}[P]_{\mathbf{rp},\emptyset}$

A first important property of our model is to ensure that the tree structure of the machine is preserved through reduction.

Proposition 1 (Well-Formedness). *If* $\mathbf{tree}(M, l, a, p)$ *and* $M \cong M'$, $M \stackrel{\equiv}{\rightarrow} M'$, $M \mapsto M'$, *or* $M \rightarrow M'$, *then* $\mathbf{tree}(M', l, a, p)$. *In particular, well-formedness is preserved by reduction. Moreover, for any process* P, $[\![P]\!]$ *is well-formed.*

From now on, unless otherwise stated, we only consider machine terms M such that $\mathbf{tree}(M, l, a, p)$ for some names l, a, p. The definitions of strong barbed bisimulation and strong barbed bisimilarity are classical [15]. We reproduce them below.

Definition 2 (Strong barbed bisimulation). *Let* TS_1 *and* TS_2 *be two sets of transition systems equipped with the same observability predicates* $\downarrow_a$, $a \in \mathsf{N}$. *A relation* $R \subseteq TS_1 \times TS_2$ *is a* strong barbed simulation *if whenever* $(A, B) \in R$, *we have*

- *If* $A \downarrow_a$ *then* $B \downarrow_a$
- *If* $A \rightarrow A'$ *then there exists* B' *such that* $B \rightarrow B'$ *and* $(A', B') \in R'$

A relation R *is a strong barbed bisimulation if* R *and* R^{-1} *are both strong barbed simulations.*

Definition 3 (Strong barbed bisimilarity). *Two transition systems* A *and* B *are said to be* strongly barbed bisimilar, *noted* $A \sim B$, *if there exists a strong barbed bisimulation* R *such that* $(A, B) \in R$.

To define strong bisimilarity for Kell calculus processes and machines we rely on the following observability predicates.

Definition 4 (Observability predicate for processes). *If* P *is a Kell calculus process,* $P \downarrow_a$ *holds if one of the following cases holds:*

1. $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.a\langle\widetilde{P}\rangle \mid R \mid P'$, *with* $a \notin \widetilde{c}$
2. $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.m[a\langle\widetilde{P}\rangle \mid R] \mid P'$, *with* $a \notin \widetilde{c}$
3. $P \equiv \stackrel{\equiv}{\rightarrow}^* \nu\widetilde{c}.a[P] \mid P'$, *with* $a \notin \widetilde{c}$

Definition 5 (Observability predicate for machines). *If* M *is a well-formed machine and* $a \in \mathsf{N}$, $M \downarrow_a$ *holds if one of the following cases holds:*

1. $M \equiv \stackrel{\equiv}{\rightarrow}^* \mathbf{r} : \mathbf{rn}[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{rp},S} \mid M'$
2. $M \equiv \stackrel{\equiv}{\rightarrow}^* h : m[a\langle\widetilde{P}\rangle \mid R]_{\mathbf{r},S} \mid M'$
3. $M \equiv \stackrel{\equiv}{\rightarrow}^* h : a[P]_{\mathbf{r},S} \mid M'$

Intuitively , a barb on a means that after an arbitrary number of administrative reductions, a process P (or machine M) can exhibit a local message (clause 1), a up message (clause 2), or a kell message (clause 3). These observations are similar to those find *e.g.* in Ambient calculi.

We now define two equivalence relations over machines that we use to state correctness properties. The first one identifies two machines that have the same normal form. The second one corresponds to a form of strong barbed congruence. Note that the second one is defined on well-formed machine only.

Lemma 1 (Normal form). *If M is a machine term, then there exists M'_* such that $M \stackrel{\equiv}{\rightarrow}^* M'_*$. Moreover, if $M \equiv \stackrel{\equiv}{\rightarrow}^* M''_*$ then $M'_* \cong M''_*$. Besides, $M \stackrel{\equiv}{\not\rightarrow}$ if and only if $M = M'_*$ for some M'_*.*

Definition 6 (Equivalence). *Two machines M and N are said to be* equivalent, *noted $M \doteq N$, if they have the same normal form (up to $\cong$).*

From now on, we will use the same notation M_* for a normal form of M (*i.e.* $M \stackrel{\equiv}{\rightarrow}^* M_* \stackrel{\equiv}{\not\rightarrow}$), and for an arbitrary term in normal form.

Definition 7. *Let $M = l : n[P]_{p,S} \mid M'$ be a machine such that $\mathbf{tree}(M, l, n, p)$ and h a fresh machine name. We define:*

$$\begin{aligned} M \mid Q &= l : n[P \mid Q]_{p,S} \mid M' \\ a[M] &= l : n[\mathbf{0}]_{p,h} \mid h : a[P]_{l,S} \mid M'\{h/l\} \\ \nu a.M &= M\{h/a\} \end{aligned}$$

We extend these definitions to any contexts of the following form:

$$\mathbf{E} ::= . \quad | \quad (R \mid \mathbf{E}) \quad | \quad a[\mathbf{E}] \quad | \quad \nu a.\mathbf{E}$$

Definition 8 (Contextual equivalence for machines). *Two well-formed machines M and N are contextually equivalent ($M \sim_c N$) if and only if $\forall \mathbf{E}, \mathbf{E}[M] \sim \mathbf{E}[N]$.*

We check easily that $\sim_c$ is the largest relation over machines included in strong barbed bisimilarity that is a preserved by $a[.]$, $\nu a.$. and $. \mid R$.

Lemma 2. *$\sim_c$, $\doteq$, $\cong$ and $\equiv$ are equivalence relations.*

Lemma 3. *We have $\equiv \subseteq \cong \subseteq \doteq$ and if we consider the restrictions of these relations to well-formed machines, they are all strong barbed bisimulation and $\doteq \subseteq \sim_c$.*

We now state two properties that relate machine reductions to process reductions (soundness), and process reductions to machine reductions (completeness).

Proposition 2 (Soundness). $[\![P]\!] \rightarrow M \implies P \rightarrow P'$ *with* $[\![P']\!] \sim_c M$.

Proof. For lack of space, we only give here a sketch of the proof. We first define by induction an inverse translation function $[\![.]\!]^{mac}$ from machines to processes. This function has three roles: to expand the "reified" processes, to rebuild the tree structure of the term, and to recreate restricted names from machine names.

The soundness proposition results from the following lemmas:

Lemma 4. *If M is well-formed and $M \stackrel{\equiv}{\rightarrow} N$ then $[\![M]\!]^{mac} \equiv \stackrel{\equiv}{\rightarrow}^* [\![N]\!]^{mac}$.*

Lemma 5. *If M is well-formed and $M \mapsto N$ then $[\![M]\!]^{mac} \mapsto [\![N]\!]^{mac}$.*

Lemma 6. *If M is a well-formed machine, then $[\![[\![M]\!]^{mac}]\!] \sim_c M$. If P is a process, then $[\![[\![P]\!]]\!]^{mac} \equiv P$.*

Proposition 3 (Completeness). $P \to P' \implies [\![P]\!] \to \sim_c [\![P']\!]$

Proof (Sketch).

The proof of this proposition is on induction on the derivation of $P \to P'$ and need the two following lemmas:

Lemma 7. *If $P \equiv P'$ then $[\![P]\!] \doteq [\![P']\!]$. If $P \xrightarrow{\equiv} P'$ then $[\![P]\!] \doteq [\![P']\!]$.*

Lemma 8. *Let P_* be a process and M_* a machine such that $\mathbf{tree}(M_*, p, a, r)$. If we have $p : a[P_*]_{p',\emptyset} \xrightarrow{\equiv}{}^* \cong M_*$ then for any machine N we have $N\{\mathbf{reify}(p, M_*)/x\} \sim_c N\{P_*/x\}$.*

The proof of Theorem 1 then results immediately from Propositions 2 and 3 by showing that the relation $\{\langle [\![P]\!], P\rangle \mid P \in \mathsf{K}\}$ is a strong barbed bisimulation up to $\sim_c$.

5 Implementation

We have implemented a prototype of our abstract machine in OCaml, which realizes a Kell calculus interpreter, and is available at [11]. The source language for the interpreter (called `kcl`) is essentially a typed extension of the calculus presented in this paper, with values. Values are either basic (integers, lists, strings), higher-order (process abstractions, passivated processes) or expressions built upon classical operators such as arithmetic operators or marshalling/unmarshalling primitives.

User programs are first parsed and typed-checked using a simple type inference algorithm. Then, they are executed by a runtime that follows closely the reductions of the abstract machine. Unlike the abstract machine, the runtime is deterministic (we do not detail here the particular reduction strategy we use). Moreover, we use environments in order to avoid the use of substitutions. The freshness conditions in the rules M.S.CELL, M.S.ACT and M.S.NEW are implemented either through the use of runtime pointers for locality names, or by a global fresh identifier generator for names created by a *new* instruction.

An independent part of the interpreter allows user programs to access various services as library functions, which may also be modeled as Kell Calculus processes. More precisely, we can see an interpreter as a context $\mathtt{vmid}[\mathtt{Lib} \mid u[\,\cdot\,]]$ executing a user program P (filling the hole) according to the rules of the abstract machine. The program P can use services specified in `Lib` that correspond to OCaml functions, but are accessed transparently from P like any other receiver. Similarly, these functions can generate messages in the `vmid` locality that can be received by P. In the implementation, messages sent from the top level of P are treated differently whether they are addressed to a receiver in `Lib` or not. A very simple library could be $\mathtt{Lib} = (\mathtt{echo}^{\downarrow}\langle x\rangle \diamond Q)$, where Q specifies the output of the string x on the standard output, and where $\diamond$ denotes to a replicated input construct (which can be encoded in the Kell calculus as shown in [17]).

A distributed configuration of interpreters can be specified as follows. If we run the programs $P_0, \ldots, P_n$ on different interpreters, the resulting behavior is specified by the following term

$$\mathtt{Net} \mid \mathtt{vmid}_0[\mathtt{Lib}(\mathtt{vmid}_0) \mid u[P_0]] \mid \ldots \mid \mathtt{vmid}_n[\mathtt{Lib}(\mathtt{vmid}_n) \mid u[P_n]]$$

where we assume `vmid` names to be distinct. The processes `Lib` model the local libraries and `Net` the network. In our implementation they are mainly defined as follows (omitting the type annotations):

$$\texttt{Lib}(\texttt{vmid}) = (\texttt{send}^{\downarrow}\langle x, y\rangle \diamond \texttt{send}\langle x, y\rangle \mid (\texttt{recv}^{\uparrow}\langle(\texttt{vmid}), y\rangle \diamond \texttt{msg}\langle x\rangle \mid (\texttt{echo}^{\downarrow}\langle x\rangle \diamond Q)$$

$$\texttt{Net} = \texttt{send}^{\downarrow}\langle x, y\rangle \diamond \texttt{rcv}\langle x, y\rangle$$

These processes specify an environment allowing the exchange of asynchronous messages between interpreters, and providing some output capability. The `vmid` name allows to send messages to uniquely deignated kells. In addition, marshalling and unmarshalling functions allow to send arbitrary values over the network.

We give in Figure 6 the code of a distributed application consisting of a client and a server that simply executes the code that it receives. `vm` is a constructor that builds an identifier for a virtual machine (typically to locate a name server) from an address and a port. `thisloc` is bound to the identifier of the machine in which it is evaluated. The construct `def in` corresponds to an input ($\xi \triangleright P$) and `rdef` to a replicated input. We use marshalling and unmarshalling functions that convert arbitrary values to string and conversely.

client.kcl

```
new a in new b in new c in
let serverid = vm ("plutonium.inrialpes.fr", 6000) in
let myid = thisloc in
  ( def a [ X ] in send < serverid, marshall(X) > )
| ( def b [ X ] in X )
| ( def c [ X ] in X | X)
| a [ send < myid, marshall ( echo <"good"> | b[c[echo <"bye">]] ) >
      | echo < "hello" > ]
|  rdef msg up < X > in unmarshall(X) as proc
```

server.kcl

```
rdef msg up < X > in X
```

Fig. 6. kcl example

The execution of the server and the client on two different machines gives the following result.

```
plutonium:~/kcl-0.1/bidinger$ kcl server.kcl -p 6000
hello

californium:~/kcl-0.1/bidinger$ kcl client.kcl -p 7000
good
bye
bye
```

6 Related work

There has been a number of recent papers devoted to the description and implementation of abstract machines for distributed process calculi. One can cite notably the Jocaml distributed implementation of the Join calculus [6,5], the Join calculus implementation of Mobile Ambients [7], Nomadic Pict [21,19], the abstract machine for the M-calculus [9], the Fusion Machine [8], the PAN and GCPAN abstract machines for Safe Ambients [14,10], the CAM abstract machine for Channel Ambients [13]. In addition, there have been also implementations of distributed calculi such as the Seal calculus [20], Klaim [2], or DiTyCO [12].

Our abstract machine specification has been designed to be independent from the actual implementation environment and the network services it provides. It thus can be used in widely different configurations. For instance, one is not limited to mapping top-level localities to physical sites as in [7,5,9], or does not need to introduce physical sites as a different locality abstractions than that of the supported calculus as in [10,14]. This separation between abstract machine behavior and network semantics is not present in other abstract machines for distributed process calculi.

The Seal calculus [4] and the M-calculus [16] are the only calculi that share with the Kell calculus a combination of local actions and hierarchical localities, and could thus achieve a similar independence between abstract machine and network services. No abstract machine is described for the Seal calculus, however (only an implementation is mentioned in [20]), and the M-calculus abstract machine described in [9] relies on a fixed network model and a mapping of top-level localities to physical sites. Calculi which rely on an explicit flat network model such as Nomadic Pict, DiTyCO, Klaim have abstract machines and implementations which presuppose a given physical configuration and its supporting network model.

The Fusion Machine implements the general fusion calculus, where no localities are present, but the abstract machine itself is based on a fixed asynchronous network model. Furthermore, because of the nature of communications in Fusion, the Fusion machine relies on a non-trivial migration protocol for achieving synchronization in presence of multiple sites. In contrast to our calculus and abstract machine, this prevents distributed Fusion programs to directly, and at no cost, exploit low-level network services such as a basic datagram service.

Abstract machines and implementations for distributed process calculi with hierarchical localities other than the Seal calculus and the M-calculus, namely the Join calculus and Ambient calculi, must implement migration primitives, which forces a dependence on a given network model. For instance, the JoCaml abstract machine for the distributed join calculus [5] depends on an asynchronous message passing network model and on a specific interpretation of the locality hierarchy (top level localities are interpreted as physical sites). The PAN [14] and GCPAN [10] abstract machines for Safe Mobile Ambients depend as well on an asynchronous message passing network model for specifying the migration of ambients between sites (corresponding to the interpretation of the Ambient primitive `open`), and on the introduction of a notion of execution site, not related to ambients. The Channel Ambient abstract machine [13] leaves in fact the realization of its `in` and `out` migration primitives unspecified.

7 Conclusion

We have presented an abstract machine for an instance of the Kell calculus, and discussed briefly its OCaml implementation. The originality of our abstract machine lies in the fact that it is independent from any network services that could be used for a distributed implementation. Indeed, as our simple OCaml implementation illustrates, we can isolate network services provided by a given environment in language libraries that can be reified as standard Kell calculus processes for use by Kell calculus programs. While this means that our abstract machine, just as the Kell calculus, does not embody any sophisticated abstraction for distributed programming, it demonstrates that the calculus and its associated machine provide a very flexible basis for developing these abstractions. Furthermore, this independence has the advantage of simplifying the proof of correctness of our abstract machine, as it does not depend on the correctness proof of a sophisticated distributed protocol.

Much work remains of course towards a provably correct implementation of the calculus. Our non-deterministic abstract machine remains too abstract in a number of dimensions to be the basis for an efficient implementation of the calculus. First, truly local actions can only be realized, and efficiency obtained, if there is some determinacy in routing messages to triggers (as it is enforced in our OCaml implementation). One can think of applying a type system similar to that reported in [3], which guarantees the unicity of kell names, to obtain linearity conditions ensuring the unicity of message destinations. Secondly, an efficient machine would require a more deterministic behavior. Here we face the prospect of a more difficult proof of correctness, and more difficulty in stating the correctness conditions, which must probably relate the non-determinism at the calculus level with the determinism of the abstract machine through some sort of fairness condition.

References

1. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science, vol. 96*, 1992.
2. L. Bettini, M. Loreti, and R. Pugliese. Srtuctured nets in klaim. In *Proceedings of the 2000 ACM Symposium on Applied Computing, ACM Press*, 2000.
3. P. Bidinger and J.B. Stefani. The Kell Calculus: Operational Semantics and Type System. In *Proc. 6th IFIP FMOODS International Conference*, volume 2884 of *LNCS*. Springer, 2003.
4. G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th FST-TCS*, number 2556 in LNCS. Springer, 2002.
5. Fabrice Le Fessant. *JoCaml: Conception et Implantation d'un Langage à Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2001.
6. 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.
7. 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.
8. Philippa Gardner, Cosimo Laneve, and Lucian Wischik. The fusion machine. In *CONCUR 2002*, volume 2421 of *LNCS*. Springer-Verlag, 2002.

9. F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.
10. D. Hirschkoff, D. Pous, and D. Sangiorgi. An Efficient Abstract Machine for Safe Ambients, 2004. Unpublished. Available at: http://www.cs.unibo.it/∼sangio/DOC_public/gcpan.ps.gz.
11. The Kell calculus page. http://sardes.inrialpes.fr/kells/.
12. 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.
13. A. Phillips, N. Yoshida, and S. Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proceedings of ESOP 2004*, LNCS. Springer-Verlag, April 2004.
14. D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th ICALP*, volume 2076 of *LNCS*. Springer-Verlag, 2001.
15. D. Sangiorgi and S. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
16. 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.
17. A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In P. Quaglia, editor, *Global Computing*, volume 3267 of *LNCS*. Springer, 2004.
18. J.B. Stefani. A Calculus of Kells. In *Proceedings 2nd International Workshop on Foundations of Global Computing*, 2003.
19. A. Unyapoth and P. Sewell. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, 2001.
20. 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.
21. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.