Representation of Programming Constructs with the Kell-m Calculus

ROLANDO BLANCO, PAULO ALENCAR
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Technical Report CS-2009-11

Abstract

Kell-m is a new asynchronous, higher-order process calculus with localities, developed for modelling and verifying distributed event-based systems and applications. Although simple, due to the low level nature of kell-m, considerable effort is required when modelling complex systems. In this report we illustrate how common programming constructs such as variables, procedures, modules and lists can be represented using kell-m. These constructs facilitate the modelling of systems and applications using kell-m.

1 Introduction

Kell-m is a new process calculus part of the Kell family of process calculi (Stefani, 2003). Kell-m was developed for modelling distributed event-based systems (DEBSs) and applications. Kell-m is also the modelling language in a model checker tool we have developed for the representation and verification of DEBSs, and in general, distributed communicating systems.

Systems are modelled with kell-m as processes executing in parallel. Processes can, optionally, be localized within kells. Kells can be dynamically created and destroyed, and kells can be contained within other kells. Similarly to other process calculi, processes interact by communicating using channels (Milner, 1980, 1999; Parrow, 2001). Only two processes can participate on a communication on a specific channel at a given time. Processes and names, representing channels or kells, can be transmitted as part of a communication. Hence, kell-m is a higher-oder process calculus. Because a process writing on a channel cannot perform any actions after the write, kell-m is also an asynchronous process calculus.

As with most low level formalisms, using kell-m for modelling complex systems can be tedious and error prone. In this report we show how basic pro-

gramming constructs can be modelled in kell-m. Most of the constructs here presented match the functionality of constructs typically used in software development, or they can be used to represent other construct. Our goal is to facilitate the representation of systems using kell-m.

We start our presentation in Section 2, by introducing kell-m's syntax and operational semantics. Variables are represented in Section 3. Menus, a form of procedural abstraction commonly used in process calculi, are represented in Section 4. In Section 5 we represent conditional statements, and in Section 6 we introduce a simple modularization construct for encapsulating data and associated operations, similar to abstract data types. We continue with a representation for lists in Section 7, and conclude the report in Section 8.

2 Syntax and Operational Semantics

Processes in kell-m have the following syntax:

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid \mathbf{new}\ a\ P \mid P|P \mid K[P] \mid x \mid \overline{a}(\widetilde{w}) \mid \xi \triangleright P \\ \xi & ::= & a(\widetilde{v}) \mid K[x] \\ v & ::= & c \mid x \\ w & ::= & c \mid P \end{array}$$

Where P represents a process; $\mathbf{0}$ is the *null* process (a process with no actions); **new** $a\ P$ is a restriction, where a fresh channel a is created, and a is bound in process P; $|$ is parallel composition of processes; $K[P]$ is a kell with name K, and executing process P; w is a name or a process; $\overline{a}(\widetilde{w})$ specifies a write operation where a combination of zero or more names and processes are being written on channel a. Note the calculus is asynchronous: a write operation cannot be followed by any operation.

K represents the name of a kell; x is a process variable; a represents a channel; c represents a channel variable. $\xi \triangleright P$ represents a *trigger*; ξ is a pattern: $a(c) \triangleright P$ matches $\overline{a}(d)$. The channel c is bound in P, and after the match, occurrences of c are replaced with d: $P\{c/d\}$. Hence:

$$\overline{a}(d) \mid (a(c) \triangleright P)$$

reduces ($\rightarrow$) to:

$$\rightarrow P\{d/c\}$$

We write **new** $a,b,c\ P$ to represent **new** a **new** b **new** $c\ P$, and **new** $\widetilde{c}$ to represent **new** $c_1, c_2, ..., c_n$ when $\widetilde{c} = c_1, c_2, ..., c_n$.

When $\diamond$ is used instead of $\triangleright$, the pattern represents a *receptive trigger*, and it is not consumed when a match occurs:

$$(a(c) \diamond P) \mid \overline{a}(d) \rightarrow (a(c) \diamond P) \mid P\{d/c\}$$

Specifically, $\diamond$ is defined as the following *fixed point* (Stefani, 2003):

$$\xi \diamond P \equiv \mathbf{new}\ t\ (Y(P, \xi, t) \mid \overline{t}(Y(P, \xi, t)))$$

with

$$Y(P, \xi, t) \equiv t(y) \triangleright (\xi \triangleright (P \mid y \mid \overline{t}(y)))$$

Hence,

$$\begin{array}{lcl} a(c) \diamond P & \equiv & \textbf{new } t\ (Y(P, a(c), t) \mid \overline{t}(Y(p, a(c), t))) \\ & \equiv & t(y) \triangleright (a(c) \triangleright (P \mid y \mid \overline{t}(y))) \mid \overline{t}(Y(p, a(c), t)) \\ & \rightarrow & a(c) \triangleright (P \mid Y(P, a(c), t) \mid \overline{t}(Y(p, a(c), t))) \end{array}$$

if $\overline{a}(d)$, we obtain:

$$P\{d/c\} \mid Y(P, a(c), t) \mid \overline{t}(Y(p, a(c), t))) \equiv P\{d/c\} \mid (a(c) \diamond P)$$

Patterns can also specify kells. For example, $K[x] \triangleright R$ matches the kell $K[P]$. The process variable x is bound in R, and it is replaced with P after the match:

$$(K[x] \triangleright R) \mid K[P] \rightarrow R\{P/x\}$$

We will use uppercase letters for kell names and processes, and lowercase letters for channels and variables.

When a kell is matched by a pattern, we say that the kell has been *passivated*. For example, a process:

$$\mathit{stop}(K) \diamond (K[x] \triangleright \mathbf{0})$$

receives in its channel *stop* the name K of a kell; it matches the kell K's process to x, and reduces the kell to the null process $\mathbf{0}$. Such a process is useful to stop the execution of a kell:

$$\begin{array}{ll} & T[\overline{a}(b)] \mid \overline{\mathit{stop}}(T) \mid (\mathit{stop}(K) \diamond (K[x] \triangleright \mathbf{0})) \\ \rightarrow & T[\overline{a}(b)] \mid (\mathit{stop}(K) \diamond (K[x] \triangleright \mathbf{0})) \mid (T[x] \triangleright \mathbf{0}) \\ \rightarrow & (\mathit{stop}(K) \diamond (K[x] \triangleright \mathbf{0})) \mid \mathbf{0} \end{array}$$

In the previous example, the kell $T[\overline{a}(b)]$ is terminated by the process $\overline{\mathit{stop}}(T)$.

$\mid$ has higher precedence than, both, $\diamond$ and $\triangleright$. **new** has higher precedence than $\diamond$ and $\triangleright$, but lower than $\mid$. Associativity of $\mid$, $\diamond$, and $\triangleright$ is left-to-right. For example,

$$\textbf{new } e\ a(c) \triangleright c(d) \triangleright P \mid \overline{a}(d)$$

is equivalent to:

$$(\textbf{new } e\ (a(c) \triangleright (c(d) \triangleright P))) \mid \overline{a}(d)$$

We assume transparent membranes: communication can happen between any two processes independently of their kell location. This is in contrast with the regular kell calculus, where communication between processes is restricted by the location of the process. For example, in the calculus here proposed:

$$\overline{a}(c) \mid \ B[a(d) \triangleright P]$$

$\overline{a}(c)$ is matched with $a(d) \triangleright P$. Also, in:

$$K[\overline{a}(c)] \mid U[a(d) \triangleright P]$$

$\overline{a}(c)$ in kell K matches $a(d)$ in kell U. When transparent membranes are assumed, kells can communicate via the channels without the need of constructs $\uparrow$, $\downarrow$, used in traditional kell calculus to specify messages coming from a containing (similarly, contained) kell (Stefani, 2003; Bidinger et al., 2005).

The following functions determine the free (fn) and bound (bn) names in a kell-m process:

$$
\begin{array}{ll}
bn(\mathbf{0}) = \emptyset & fn(\mathbf{0}) = \emptyset \\
bn(x) = \emptyset & fn(x) = \{x\} \\
bn(\mathbf{new}\ a\ P) = \{a\} \cup bn(P) & fn(\mathbf{new}\ a\ P) = fn(P) \setminus \{a\} \\
bn(\overline{a}(\widetilde{w})) = bn(\widetilde{w}) & fn(\overline{a}(\widetilde{w})) = \{a\} \cup \{fn(\widetilde{w})\} \\
bn(\widetilde{w}) = \bigcup_{w_i \in \widetilde{w}} bn(w_i) & fn(\widetilde{w}) = \bigcup_{w_i \in \widetilde{w}} fn(w_i) \\
bn(a(\widetilde{c}) \triangleright P) = \{\widetilde{c}\} \cup bn(P) & fn(a(\widetilde{c}) \triangleright P) = fn(P) \setminus \{\widetilde{c}\} \\
bn(K[x] \triangleright P) = \{x\} \cup bn(P) & fn(K[x] \triangleright P) = fn(P) \setminus \{x\} \\
bn(P \mid Q) = bn(P) \cap bn(P) & fn(P \mid Q) = fn(P) \cup fn(Q)
\end{array}
$$

The following are the structural equivalences for the kell-m calculus:

$$\mathbf{new}\ a\ \ \mathbf{0} \equiv \mathbf{0} \qquad P \mid \mathbf{0} \equiv P \qquad P \equiv P\{\widetilde{c}/\widetilde{d}\}, \text{with } \widetilde{d} \in bn(P)$$

$$K[\mathbf{0}] \equiv \mathbf{0} \qquad \mathbf{new}\ a,\ b\ P \equiv \mathbf{new}\ b,\ a\ P$$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$\frac{a \notin fn(Q)}{(\mathbf{new}\ a\ P) \mid Q \equiv \mathbf{new}\ a\ (P \mid Q)}$$

2.1 LTS Semantics

As is traditional in process algebras, we use a labelled transition system (LTS) to give the operational semantics of the kell-m calculus. The LTS describes the possible evolution of a process. Actions performed during the transitions can be: $\overline{a}(\widetilde{w})$, $a(\widetilde{c})$, $\overline{K}[P]$, $K[x]$, and τ:

- $\overline{a}(\widetilde{w})$, represents an output action on channel a,
- $a(\widetilde{c})$ represents an input action (via a matching trigger) on channel a,
- $\overline{K}[P]$ represents the output of kell K's process,
- $K[x]$ represents the input of kell's K process, via a matching trigger.
- τ represents the matching of input and output channel, or kell actions.

The transitions for the calculus are determined as follows:

$$\frac{P \xrightarrow{\alpha} P',\ P \equiv Q}{Q \xrightarrow{\alpha} P'}\ \textsc{Struct}$$

$$\overline{a}(\widetilde{w}) \xrightarrow{\overline{a}(\widetilde{w})} \mathbf{0} \quad \textsc{Out} \qquad a(\widetilde{c}) \triangleright P \xrightarrow{a(\widetilde{c})} P \quad \textsc{In}$$

$$K[P] \xrightarrow{\overline{K}[P]} \mathbf{0} \quad \textsc{KellOut} \qquad K[x] \triangleright P \xrightarrow{K[x]} P \quad \textsc{KellIn}$$

$$\frac{P \xrightarrow{\alpha} Q,\ c \notin bn(\alpha)}{\mathbf{new}\ c\ P \xrightarrow{\alpha} \mathbf{new}\ c\ Q}\ \textsc{Restrict} \qquad \frac{P \xrightarrow{\alpha} Q,\ K \notin bn(\alpha)}{K[P] \xrightarrow{\alpha} K[Q]}\ \textsc{Advance}$$

$$\frac{P \xrightarrow{\alpha} Q,\ \alpha \notin fn(P')}{P|P' \xrightarrow{\alpha} Q|P'}\ \textsc{Par} \qquad \frac{P_d\{\widetilde{w}/\widetilde{x}\} \xrightarrow{\alpha} Q,\ P(\widetilde{x}) \stackrel{def}{=} P_d}{P(\widetilde{w}) \xrightarrow{\alpha} Q}\ \textsc{Proc}$$

$$\frac{P \xrightarrow{\overline{a}(\widetilde{w})} Q,\ c \in names(\widetilde{w}),\ c \neq a}{\mathbf{new}\ c\ P \xrightarrow{\overline{a}(\widetilde{w}')} Q,\ \text{with } \widetilde{w}' = \widetilde{w}\{\mathbf{new}\ c\ /c\}}\ \textsc{Open}$$

$$\frac{P \xrightarrow{a(\widetilde{c})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q'}{P \mid Q \xrightarrow{\tau} P'\{\widetilde{w}/\widetilde{c}\} \mid Q'}\ \text{L-}\textsc{React} \qquad \frac{P \xrightarrow{K[x]} P',\ Q \xrightarrow{\overline{K}[R]} Q'}{P \mid Q \xrightarrow{\tau} P'\{\widetilde{R}/x\} \mid Q'}\ \text{L-}\textsc{Suspend}$$

$$\frac{P \xrightarrow{a(\widetilde{d})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q',\ \widetilde{c} \subseteq \widetilde{w},\ (\mathbf{new}\ c\ ) \in \widetilde{w} \text{ if } c \in \widetilde{c}}{P \mid Q \xrightarrow{\tau} \mathbf{new}\ \widetilde{c}\ (P'\{\widetilde{w}/\widetilde{c}\} \mid Q')}\ \text{L-}\textsc{Close}$$

R-* transition rules can be trivially deduced by first using the Struct rule, and then the corresponding L-* rule.

When illustrating transitions for process expressions, we sometimes write the name of the channel involved in a communication action right after the τ, e.g., $P \xrightarrow{\tau,a} Q$.

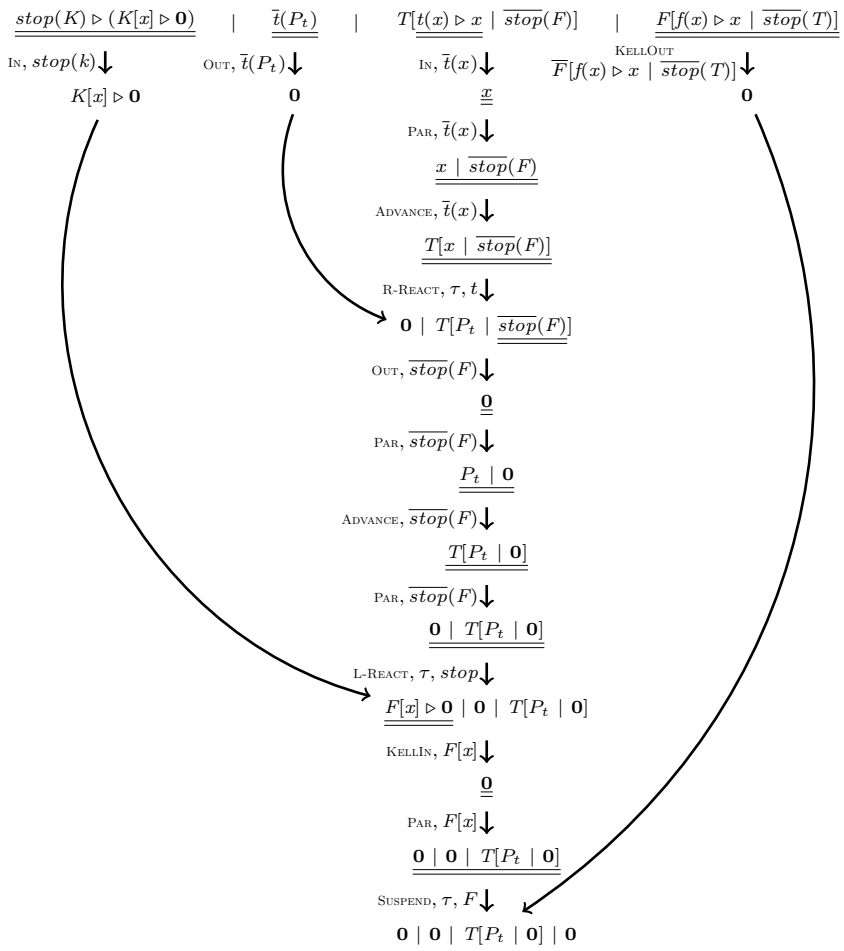
Kells are themselves channels. In terms of classical process theory (Milner, 1999), both $\overline{a}(\widetilde{w})$ and $K[P]$ correspond to *concretions*. Trigger matching expressions $a(\widetilde{c})$ and $K[x]$ in triggers $a(\widetilde{c}) \triangleright P$, and $K[x] \triangleright P$ correspond to *abstractions*. Because of our assumption of transparent membranes, communications between abstractions and concretions can occur at any kell-depth.

We refer to rules *-React, *-Suspend and *-Close as the *communication rules*. These are the rules where abstractions and concretions are matched. The transitions in these rules are decorated with τ.

To illustrate the use of the transition rules, consider the process P defined as:

$$P \stackrel{def}{=} stop(K) \triangleright (K[x] \triangleright \mathbf{0}) \mid T[t(x) \triangleright x \mid \overline{stop}(F)] \mid F[f(x) \triangleright x \mid \overline{stop}(T)]$$

Any process received on channel t or f is executed. If channel t is used, channel f is discarded, and vice versa. As we will see in Section 5, such a process is useful to represent conditionals. We will now show that $\mathbf{0} \mid \mathbf{0} \mid T[P_t|\mathbf{0}] \mid \mathbf{0}$ can be inferred from $\overline{t}(P_t) \mid P$ as follows (we have rearranged the process expressions to facilitate the drawing of an inference tree; process expressions involved in the transitions are double-underlined):



We use the notation $K^n[P]$ to specify a kell K and its process P when the kell is embedded within $n-1$ other kells:

$$
\begin{array}{ll}
K^1[P] & \text{when } K[P] \\
K^2[P] & \text{when } \exists K_1 : K_1[K[P] \cdots] \\
\cdots & \\
K^n[P] & \text{when } \exists K_1, ..., k_{n-1} : K_1[K_2[\cdots K_{n-1}[K[P] \cdots] \cdots] \cdots]
\end{array}
$$

We call n the *depth-level* of kell K, and write $K^*[P]$ to specify a kell-m process at any depth-level. In particular, we write $K^0[P]$ when P is not within a kell.

For notational convenience, we introduce generalized versions of the communication rules *-REACT, *-SUSPEND and *-CLOSE as follows (only the left-hand

versions are shown, right-hand versions are similarly defined):

$$\frac{P \xrightarrow{a(\widetilde{c})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q'}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} K^*[P'\{\widetilde{w}/\widetilde{c}\}] \mid M^*[Q']}\ \text{L-React}$$

$$\frac{P \xrightarrow{K[x]} P',\ Q \xrightarrow{\overline{K}[R]} Q'}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} K^*[P'\{\widetilde{R}/x\}] \mid M^*[Q']}\ \text{L-Suspend}$$

$$\frac{P \xrightarrow{a(\widetilde{d})} P',\ Q \xrightarrow{\overline{a}(\widetilde{w})} Q',\ \widetilde{c} \subseteq \widetilde{w},\ (\textbf{new}\ c\ ) \in \widetilde{w} \text{ if } c \in \widetilde{c}}{K^*[P] \mid M^*[Q] \xrightarrow{\tau} \textbf{new}\ \widetilde{c}\ (K^*[P'\{\widetilde{w}/\widetilde{c}\}] \mid M^*[Q'])}\ \text{L-Close}$$

The generalized rules are equivalent to applications of the Par and Advance rules, before using the communication rules.

2.2 Reduction Semantics

An alternative to using LTSs to describe the operational semantics of the kell-m calculus, is the use of reduction rules. When using reduction rules, communication between processes is inferred, directly, from the syntax of the processes, instead of from transitions where abstraction and concretion actions occur Parrow (2001).

We now introduce the reduction rules for the kell-m calculus:

$$\frac{}{a(\widetilde{c}) \triangleright P \mid \overline{a}(\widetilde{w}) \rightarrowtail P\{\widetilde{w}/\widetilde{c}\} \mid \mathbf{0}}\ \text{L-ReductReact}$$

$$\frac{}{M[x] \triangleright P \mid M[P_M] \rightarrowtail P\{P_M/x\} \mid \mathbf{0}}\ \text{L-ReductSuspend}$$

$$\frac{P \rightarrowtail P'}{\textbf{new}\ c\ P \rightarrowtail \textbf{new}\ c\ P'}\ \text{ReductRestrict}$$

$$\frac{P \rightarrowtail P'}{K[P] \rightarrowtail K[P']}\ \text{ReductInKell} \qquad \frac{P \rightarrowtail P'}{P|Q \rightarrowtail P'|Q}\ \text{ReductPar}$$

$$\frac{P' \equiv P,\ P \rightarrowtail Q,\ Q \equiv Q'}{P' \rightarrowtail Q'}\ \text{ReductStruct}$$

Reactions and kell passivations can occur within kells:

$$\frac{P \mid Q \rightarrowtail P' \mid Q'}{K^*[P \cdots] \mid M^*[Q \cdots] \rightarrowtail K^*[P' \cdots] \mid M^*[Q' \cdots]} \text{ ReductOutKell}$$

where $\cdots$ represents zero or more (bound name set, kell-m process)-pairs composed in parallel.

Again, in the reduction rules, we assume that there are no name conflicts for bound names.

The following reduction rule explicitly handles name extrusion from kells:

$$\frac{(\mathbf{new}\ \widetilde{c}\ P) \mid Q \rightarrowtail \mathbf{new}\ \widetilde{c}\ (P' \mid Q')}{K^*[\mathbf{new}\ \widetilde{c}\ P \cdots] \mid M^*[Q \cdots] \rightarrowtail \mathbf{new}\ \widetilde{c}\ (K^*[P' \cdots] \mid M^*[Q' \cdots])} \text{ L-ReductExtrusion}$$

When illustrating reductions of process expressions, we sometimes write the name of the channel involved in a communication action, e.g., $P \rightarrowtail {}^{a}Q$.

R-* reduction rules can be trivially deduced by first using the ReductStruct rule, and then the corresponding L-* rule.

For example, a process P defined as:

$$P \stackrel{def}{=} K[\mathbf{new}\ a,\ b\ (\overline{c}(\overline{a}(b)) \mid a(d) \triangleright Q)] \mid c(x) \triangleright x$$

can be reduced as follows:

$$\begin{array}{rll} P \rightarrowtail^{c} & \mathbf{new}\ a,\ b\ (K[\mathbf{0} \mid a(d) \triangleright Q] \mid \overline{a}(b)) & \text{L-ReductExtrusion} \\ \rightarrowtail^{a} & \mathbf{new}\ a,\ b\ (K[\mathbf{0} \mid Q\{b/d\}] \mid \mathbf{0}) & \text{ReductOutKell} \end{array}$$

When dealing with higher-order expressions, the scope extrusion occurs for any name in the expression that is bound when the expression is output via a channel ($\overline{c}(\overline{a}(b))$ in the example).

As observed by Parrow (2001), although reduction rules are simpler than LTS semantics, in the sense that there are no label on the transitions, there is loss of information when compared to LTS transitions. For example, consider the process $\overline{a}(w)$. This process cannot be reduced. It, nevertheless, has the potential to communicate with another process via the channel a. Such potential for communication is manifest in the Out LTS transition, but is lost when using reduction semantics.

3 Variables

A process for creating variables with names received on channel *vname* and values received on channel *value* can be represented as:

$$\begin{array}{l} var(vname,\ value) \diamond vname(r,\ u) \ \triangleright \mathbf{new}\ R,U\ (\\ \quad R[\ r(rc) \triangleright \overline{stop}(U) \mid \overline{rc}(value) \mid \overline{var}(vname,\ value)] \mid \\ \quad U[\ u(newval) \triangleright \overline{stop}(R) \mid \overline{var}(vname,\ newval)] \\ ) \end{array}$$

A variable is then a channel, that provided a read channel r and a write channel w, and based on which of these two channels is used, returns the variable's value, or sets a new value. In the previous process, every time a variable is used, a new kell R is produced. After multiple reads:

$$\textbf{new } R \;\; R[\textbf{new } R \;\; R[\textbf{new } R \; R \cdots [\textbf{new } R \; R[\overline{var}(vname,\ value)] \cdots]$$

A better way to construct variables is:

$$\begin{array}{l} var(vname,\ value) \diamond vname(r,\ u) \ \triangleright \textbf{new } R,U,c \ (\\ \quad R[\ r(rc) \triangleright \overline{stop}(U) \mid \overline{rc}(value) \mid \overline{c}(vname,\ value)] \mid \\ \quad U[\ u(newval) \triangleright \overline{stop}(R) \mid \overline{c}(vname,\ newval)] \mid \\ \quad c(n, val) \triangleright \overline{var}(n,val) \\ ) \end{array}$$

The process $\overline{var}(v,\ a)$, makes channel v a variable with value a.

Generic, setter and getter processes can be defined as:

$$set(vname,\ newval) \diamond \textbf{new } r,u \ (\overline{\overline{vname}}(r,u) \mid \overline{u}(newval))$$

$$get(vname,\ rc) \diamond \textbf{new } r,u \ (\overline{\overline{vname}}(r,u) \mid \overline{r}(rc))$$

To retrieve, on channel rc, the value of a variable $vname$:

$$\overline{get}(vname,\ rc) \mid rc(value) \triangleright \cdots$$

If we will be using a channel to read the value returned by another process, we typically name the channel rc, for *return channel*.

The following type of process invocation is frequently used:

$$\textbf{new } rc \ (\overline{c}(\widetilde{ps},\ rc) \mid rc(\widetilde{vs}) \triangleright \cdots)$$

where $\widetilde{ps}$ represents zero or more parameter *names* written to c, and $\widetilde{vs}$ are the *names* returned on channel rc. A *name* can be a channel, a process, or the name of a kell. For these invocations we write:

$$@c(\widetilde{ps})(\widetilde{vs}) \triangleright \cdots$$

For example,

$$\textbf{new } rc \ (\overline{get}(name,\ rc) \mid rc(val) \triangleright P)$$

can be written:

$$@get(name)(val) \triangleright P$$

Note that val is bound in P. We use the same syntax when $\diamond$ appears instead of $\triangleright$. Hence,

$$@get(name)(val) \diamond P$$

is the same as

$$\textbf{new } rc \ (\overline{get}(name,\ rc) \mid rc(val) \diamond P)$$

We write $@c(\widetilde{ps})$ for $\overline{c}(\widetilde{ps})$, when no channel in $\widetilde{ps}$ is used to return values.

When the values returned on a channel are immediately used as inputs for another channel, for example:

$$@get(v_2)(val) \triangleright \overline{set}(v_1,\ val)$$

we write instead:

$$@set(v_1, @get(v_2))$$

Here, we are setting the value of variable v_1 to the value stored in v_2. We further add syntactic sugar by writing this process as: $v_1 := *v_2$. In general, $@set(v,\ val)$ is written $v := val$, and $*v$ is syntactic sugar for $@get(v)(*v)$. The $*v$ is just a name. Hence, if the value of a variable v is a channel, $\overline{*v}(\widetilde{p})$ is valid, as well as $*v(\widetilde{p}) \triangleright P$ and $*v(\widetilde{p}) \diamond P$.

4 Menus

Menus are processes that receive as parameters multiple channels and, based on the channels used, execute an operation (Milner, 1999). Example:

$$\begin{aligned} casetf(t,\ f) \diamond \textbf{new}\ T,\ F\ (\ & T[t() \triangleright P_T \mid \overline{stop}(F)] \\ & |F[f() \triangleright P_F \mid \overline{stop}(T)]) \end{aligned}$$

where **new** $T,\ F\ (P)$ stands for **new** T (**new** $F\ P$). In the example, *casetf* receives two channels, t and f. If there is a write on channel t, then process P_T is executed. If the write is on channel f, process P_F is executed instead. The result is unpredicted if there are simultaneous writes on both t and f. A process wanting to execute P_T would be:

$$\textbf{new}\ t,f\ (\overline{casetf}(t,\ f) \mid \overline{t}() \triangleright Q)$$

Similarly, to execute P_F:

$$\textbf{new}\ t,f\ (\overline{casetf}(t,\ f) \mid \overline{f}() \triangleright Q)$$

For a channel c, when no names are passed in the channel, we sometimes write $\overline{c}$ for $\overline{c}()$, and $c(f)$or $c()$.

This menu construct, where only one channel is used, can be seen as a deterministic form of the the *sum* $P\ +\ Q$ operator of the π-calculus, when only one of the processes in the sum is intended to execute:

$$casetf(t,\ f).(t().P_T\ +\ f().P_F)$$

We generalize this type of menu construction:

$$
\begin{array}{l}
menu(name, c_1, P_1, c_2, P_2, ..., c_n, P_n) \diamond (\\
\quad name(c_1, ..., c_n) \diamond \textbf{new } C_1, ..., C_n \ (\\
\qquad C_1[c_1(\widetilde{ps_1}) \triangleright P_1 \mid \overline{stop}(C_2) \mid \overline{stop}(C_3) \mid \cdots \mid \overline{stop}(C_n)] \\
\qquad \mid C_2[c_2(\widetilde{ps_2}) \triangleright P_2 \mid \overline{stop}(C_1) \mid \overline{stop}(C_3) \mid \cdots \mid \overline{stop}(C_n)] \\
\qquad \mid \cdots \\
\qquad \mid C_n[c_n(\widetilde{ps_n}) \triangleright P_n \mid \overline{stop}(C_1) \mid \overline{stop}(C_2) \mid \cdots \mid \overline{stop}(C_{n-1})] \\
\quad) \\
)
\end{array}
$$

Hence, for $casetf$, we can write $menu(casetf, t, P_T, f, P_F)$.

When we want to execute the process activated by channel c_i in a menu m, we write:

$$@m.c_i(\widetilde{ps_i}) \mid Q \equiv \textbf{new } c_1, ..., c_n \ (\overline{m}(c_1, ..., c_n) \mid \overline{c_i}(\widetilde{ps_i}) \mid Q)$$

Note that if the activated process returns values via a return channel at the end of $\widetilde{ps_i}$ we can write:

$$@m.c_i(\widetilde{ps'_i})(\widetilde{vals})$$

where $\widetilde{ps_i} \equiv (\widetilde{ps'_i}, rc)$.

We write $m.c_i$ to identify the channel used to activate a menu process P_i. Hence, a process:

$$apply(c, \widetilde{ps'_i}, rc) \diamond \overline{rc}(@c(\widetilde{ps'_i}))$$

can be used to invoke functionality via channels:

$$@apply(m.c_i, \widetilde{ps'_i})(\widetilde{vals}) \equiv @m.c_i(\widetilde{ps'_i})(\widetilde{vals})$$

5 Conditional Statements

A process of the form:

$$@cond(\widetilde{ps})(t, f) \triangleright \overline{casetf}(t, P_T, f, P_F)$$

is written:

$$\textbf{if } @cond(\widetilde{ps}) \textbf{ then } P_T \textbf{ else } P_F \textbf{ fi}$$

If the P_F process is $\mathbf{0}$, we write:

$$\textbf{if } @cond(\widetilde{ps}) \textbf{ then } P_T \textbf{ fi}$$

An if/elsif statement:

$$\begin{array}{l}\textbf{if} \quad @cond_1(\widetilde{ps1}) \textbf{ then } P_1\\ \textbf{elsif } @cond_2(\widetilde{ps2}) \textbf{ then } P_2\\ \quad \cdots\\ \textbf{elsif } @cond_n(\widetilde{psn}) \textbf{ then } P_n\\ \textbf{else } P_d\\ \textbf{fi}\end{array}$$

is equivalent to:

$$\begin{array}{l}\textbf{if} \quad @cond_1(\widetilde{ps_1}) \textbf{ then } P_1\\ \textbf{else if } @cond_2(\widetilde{ps_2}) \textbf{ then } P_2\\ \cdots\\ \textbf{else if } @cond_n(\widetilde{ps_n}) \textbf{ then } P_n\\ \textbf{else } P_d\\ \textbf{fi} \cdots \textbf{fi}\end{array}$$

A parallel case with no default condition is written:

$$\begin{array}{l}\textbf{case}_p \textbf{ when } @cond_1(\widetilde{ps1}) \textbf{ then } P_1\\ \quad \textbf{when } @cond_2(\widetilde{ps2}) \textbf{ then } P_2\\ \quad \cdots\\ \quad \textbf{when } @cond_n(\widetilde{psn}) \textbf{ then } P_n\\ \textbf{esac}_p\end{array}$$

and is equivalent to:

$$\begin{array}{l}\textbf{if } @cond_1(\widetilde{ps_1}) \textbf{ then } P_1 \textbf{ else 0 fi} \mid\\ \textbf{if } @cond_2(\widetilde{ps_2}) \textbf{ then } P_2 \textbf{ else 0 fi} \mid\\ \cdots\\ \textbf{if } @cond_n(\widetilde{ps_n}) \textbf{ then } P_n \textbf{ else 0 fi}\end{array}$$

Negation, is implemented by a process at channel *not* that, given channels t', f', t, and f, writes on channel t if it can read from f', and writes to f if it can read from t':

$$not(t', f', rc) \diamond \textbf{new } t, f\ (\overline{rc}(t,f) \mid \overline{casetf}(t', \bar{f}(), f', \bar{t}()))$$

We write:

$$\textbf{not } @cond(\widetilde{ps})$$

for:

$$@not(@cond(\widetilde{ps}))(t, f)$$

6 Modularization

We encapsulate variables and menus into *modules*, a construct similar to OO classes but without inheritance and other OO features.

A module is defined as:

$$
\begin{array}{l}
\textbf{module } name \\
\quad \begin{array}{l}
\textbf{var } v_1, ..., v_m \\
c_1(\widetilde{ps_1}) \triangleright P_1 \\
c_2(\widetilde{ps_2}) \triangleright P_2 \\
\cdots \\
c_n(\widetilde{ps_n}) \triangleright P_n \\
init(\widetilde{ps}, self) \triangleright P_{init}
\end{array}
\end{array}
$$

which corresponds to:

$$
\begin{array}{l}
name(\widetilde{ps}, rc) \diamond \textbf{new } self, SELF, v_1, ..., v_m \ (\\
\quad \overline{rc}(self) \mid \overline{var}(v_1, \bot) \mid \cdots \mid \overline{var}(v_m, \bot) \mid \overline{init}(\widetilde{ps}, self) \mid \\
\quad init(\widetilde{ps}, self) \triangleright P_{init} \mid SELF[\overline{menu}(self, c_1, P_1, c_2, P_2, ..., c_n, P_n)] \\
)
\end{array}
$$

The module encapsulates variables $v_1, ..., v_m$, and implements operations at channels $c_1, ..., c_n$. We call these channels the *methods* of the module. An initialization operation *init* is executed when an instance of the module is created. *init* receives as parameters any values passed to the module, plus the newly created instance *self*, and executes the process P_{init}. This process P_{init} typically instantiates the module variables $v_1, ..., v_m$, and performs any other required initialization tasks.

Every time an instance of the module is created, a copy of the methods and variables are encapsulated within a kell *SELF*. To create an instance *inst* of a module *name*, and to execute the method c_i in the newly created instance, a process executes:

$$@name(\widetilde{ps})(inst) \triangleright @inst.c_i(\widetilde{ps_i})(vals)$$

For example, assume a module *temperature*, used to store the temperature

for a given latitude, longitude location:

module *temperature*

var $temp, lat, lon$

$gettemp(rc) \triangleright \overline{rc}(*temp)$

$settemp(ntemp) \triangleright temp := ntemp$

$getlat(rc) \triangleright \overline{rc}(*lat)$

$setlat(nlat) \triangleright lat := nlat$

$getlon(rc) \triangleright \overline{rc}(*lon)$

$setlon(nlon) \triangleright temp := nlon$

$destroy() \triangleright \overline{stop}(SELF)$

$init(itemp,\ ilat,\ ilon,\ self) \triangleright (temp := itemp \mid lat := ilat \mid lon := ilon)$

The *destroy* method discards an instance of the *temperature* module. We create a temperature t, by invoking:

$$@temperature(22,\ 43,\ 80)(t)$$

For simplicity, we assume native support for numbers in our calculus. For ways to represent numbers in process calculus see Milner (1999).

In general, we assume the existence of the getter and setter methods for a module and, in the temperature example, we write $*t.temp$ for $@t.gettemp()(val)$, and $t.temp := newtemp$ for $@t.settemp(newtemp)$.

7 Lists and Iterators

Inspired by the implementation of lists in Milner (1999) (also in Magee et al. (1995)), a list receives two channels x and y. If the list is empty, the list writes to x. If the list is not empty, it writes to y the list's header s and tail ss:

module *lists*

$empty(rc) \triangleright$ **new** $l\ (\overline{rc}(l) \mid l(x,y) \diamond \overline{x}())$

$cons(s,ss,rc) \triangleright$ **new** $l\ (\overline{rc}(l) \mid l(x,y) \diamond \overline{y}(s,ss))$

$init(self) \triangleright \mathbf{0}$

Hence, the *lists* module knows how to construct lists. In the rest of this document, we will assume the existence of a *list* instance of the *lists* module, defined as:

$$@lists()(list)$$

Using, this *list* instance, an empty list l is obtained by executing:

$$@list.empty()(l)$$

A list l with one element a is constructed by:

$$@list.cons(a, @list.empty())(l)$$

We now include in the *lists* menu, methods *car* and *cdr* with the usual meaning:

$$car(l,\ rc) \triangleright \textbf{new}\ x,y\ (\bar{l}(x,y)\ |\ y(s,ss) \triangleright \overline{rc}(s))$$
$$cdr(l,\ rc) \triangleright \textbf{new}\ x,y\ (\bar{l}(x,y)\ |\ y(s,ss) \triangleright \overline{rc}(ss))$$

Also, we introduce ::, and $[;\cdots;]$ as used in Ocaml (INRIA, 2008):

$$\begin{aligned}
[\ ] &\equiv @list.empty() \\
a :: [\ ] &\equiv [\ a\ ] \equiv @list.cons(a, @list.empty()) \\
a :: b :: [\ ] &\equiv [\ a; b\ ] \equiv @list.cons(a, @list.cons(b, @list.empty())) \\
[\ a; b; c\ ] &\equiv a :: b :: c :: [\ ]
\end{aligned}$$

:: is therefore a shorthand for *list.cons*. As well, we introduce:

$$\begin{array}{l}\textbf{match}\ l\ \textbf{with} \\ s :: ss\ \triangleright\ P_1 \\ \textbf{or}\ [\ ]\ \triangleright\ P_2\end{array}$$

to represent:

$$\textbf{if}\ @list.isempty(l)\ \textbf{then}\ P_2\ \textbf{else new}\ x,\ y\ (\bar{l}(x,y)\ |\ y(s,\ ss) \triangleright P_1)\ \textbf{fi}$$

where *isempty* is in the *lists* menu, defined as:

$$\begin{aligned}isempty(l,rc) \triangleright \textbf{new}\ t,\ f,\ x,\ y,\ T,\ F\ (&\overline{rc}(t,\ f)\ |\ \bar{l}(x,\ y)\ |\ T[\overline{x}() \triangleright \bar{t}()\ |\ \overline{stop}(F)] \\ &|\ F[\overline{y}(s,ss) \triangleright \bar{f}()\ |\ \overline{stop}(T)])\end{aligned}$$

We also add *ht* to the *lists* menu. *ht* returns the head and tail of a list:

$$ht(l,\ rc) \triangleright \textbf{new}\ x,y\ (\bar{l}(x,y)\ |\ y(s,ss) \triangleright \overline{rc}(s,ss))$$

This can also be written as:

$$ht(l,\ rc) \triangleright \textbf{new}\ x\ (\bar{l}(x,rc))$$

If l is empty, $\overline{x}()$ will be written, but no process will be waiting for input on x. Hence, *ht* should only be invoked on non empty lists.

Other list methods we add to the *lists* menu are:

$$\begin{aligned}&foldr(p,v,l,rc)\ \diamond\ \textbf{match}\ l\ \textbf{with} \\ &\qquad\qquad [\ ]\ \triangleright\ \overline{rc}(v) \\ &\qquad\quad \textbf{or}\ s :: ss\ \triangleright\ \overline{rc}(@p(s, @self.foldr(p,\ v,\ ss))) \\ &copy(l,\ rc) \triangleright \overline{rc}(@self.foldr(cpy, [\ ], l))\end{aligned}$$

where,

$$cpy(s,\ ss,\ rc) \diamond \overline{rc}(@list.cons(s, ss))$$

Which is equivalent to $cpy(s,\ ss,\ rc) \diamond \overline{rc}(s :: ss)$. Hence, we can write:

$$copy(l,\ rc) \triangleright \overline{rc}(@list.foldr(list.cons, [\,], l))$$

Recall that $menu.c_i$ specifies the channel for operation c_i in the menu $menu$, and can be passed as parameter for another process. For example:

$$apply(list.cons,\ s,\ ss,\ rc) \triangleright \overline{rc}(@list.cons(h,t))$$

In general, if $a(op_1, p_1, ..., op_n, p_n)$ is a menu, then $a.op_i$ is syntactic sugar for **new** $op_1, ..., op_n$ $(\overline{a}(op_1, ..., op_n) \mid op_i \cdots)$.

We now continue with more list methods in the *lists* menu. The *del* method deletes all occurrences of e in list l. We assume the existence of the $=$ operator, which is able to decide if two names are the same.

$$\begin{aligned}
&append(l_1, l_2, rc) \triangleright \overline{rc}(@self.foldr(list.cons,\ l2,\ l1))\\
&reverse(l,rc)\ \diamond\ \textbf{match}\ l\ \textbf{with}\\
&\qquad\qquad [\,]\ \triangleright\ \overline{rc}([\,])\\
&\qquad\quad \textbf{or}\ s :: ss\ \triangleright\ \overline{rc}(@self.append(@reverse(ss), [s]))\\
&del(l,e,rc)\ \triangleright\ \overline{rc}(@self.foldr(d, [\,], l))
\end{aligned}$$

where,

$$d(s,\ l,\ rc) \triangleright \textbf{if}\ s = e\ \textbf{then}\ \overline{rc}(l)\ \textbf{else}\ \overline{rc}(s :: l)\ \textbf{fi}$$

Sorted lists, assuming a channel c, that given two elements, decides if the first element should be before the second one in the sorted list, can be obtained with:

$$\begin{aligned}
cons_s(h,\ hs,\ c,\ rc)\ \diamond\ &\textbf{match}\ hs\ \textbf{with}\\
&[\,]\ \triangleright\ \overline{rc}(@list.cons(h, [\,]))\\
\textbf{or}\ s :: ss\ \triangleright\ &\textbf{if}\ @c(h,s)\ \textbf{then}\ \overline{rc}(h :: hs)\\
&\textbf{else}\ \overline{rc}(@self.cons(s, @self.cons_s(h,\ ss)))\ \textbf{fi}
\end{aligned}$$

A parallel *foreach* iterator is implemented by:

$$\begin{aligned}
foreach(l,\ p)\ \diamond\ &\textbf{match}\ l\ \textbf{with}\\
&[\,]\ \triangleright\ \mathbf{0}\\
\textbf{or}\ s :: ss\ \triangleright\ &@p(s)\ \mid\ @self.foreach(ss,\ p)
\end{aligned}$$

A sequential version of the iterator can be implemented if process p uses a *done* channel:

$$\begin{aligned}
foreach_s(l, p, done)\ \diamond\ &\textbf{match}\ l\ \textbf{with}\\
&[\,]\ \triangleright\ \overline{done}()\\
\textbf{or}\ s :: ss\ \triangleright\ &@p(s)(pdone)\ \triangleright\ @self.foreach_s(ss,\ p,\ done)
\end{aligned}$$

We write:

$$\textbf{foreach}\ t\ \textbf{in}\ ts\ \ @p(s)$$

for @*list.foreach*(*ts, p*), and

$$\textbf{foreach}_s\ t\ \textbf{in}\ ts\ \ @p(s)(pdone)\ \overline{done}()$$

for @*list.foreach*$_s$(*ts, p, done*). We also write:

$$\textbf{foreach}\ t\ \textbf{in}\ ts\ \ @t.foo(bar)$$

for @*list.foreach*(*ts, p*), where $p(t) \triangleright @t.foo(bar)$.

8 Conclusion

Computations are modelled in kell-m as parallel processes communicating via channels. This simple computational model allows for a straightforward formalization and it is the basis for a model checker we have developed for modelling and verifying DEBSs. Due to the low level nature of kell-m, modelling systems using kell-m requires a considerable effort and can be error prone. In this report we have illustrated how variables, conditionals, iterators, modules, and lists, can be represented using kell-m. These programming constructs can then be used to facilitate system modelling. We are currently working on a translator that, given a system representation using the constructs here presented, produces an equivalent kell-m process. Along with the model checker already developed, this translator will be part of a tool-set for the representation and verification of DEBSs.

References

Bidinger, P., Schmitt, A., & Stefani, J.-B. (2005). An abstract machine for the kell calculus. In M. Steffen & G. Zavattaro (Eds.), *7th ifip int. conf. formal methods for open object-based distributed systems (FMOODS)* (Vol. 3535, pp. 31–46). Springer Verlag.

INRIA. (2008). *Objective caml.* Available from `http://caml.inria.fr/ocaml/index.en.html`

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of the 5th european software engineering conference* (pp. 137–153). London, UK: Springer-Verlag.

Milner, R. (1980). *A calculus of communicating systems* (Vol. 92). Springer-Verlag.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus.* Computer Laboratory, University of Cambridge: Cambridge University Press.

Parrow, J. (2001). An introduction to the pi-calculus. In J. Bergstra, A. Ponse, & S. Smolka (Eds.), *Handbook of pocess algebra* (pp. 479–543). Elsevier.

Stefani, J.-B. (2003). A calculus of kells. In *Proceedings 2nd international workshop on foundations of global computing* (Vol. 85). Elsevier.